CHAPTER 1

Introduction

Supercomputers have been described as the fastest machines available at any specific time (e.g., see Lazou [86]). Such high-performance machines are in demand to solve a wide range of problems, including weather modeling, image processing, artificial intelligence, and seismic exploration (more extensive surveys of the problem domain can be found in Hwang [84], Paul [84], and Lazou [86]). Researchers (e.g., Peterson [84] and Ballhaus [84]) have already indicated that the next generation of supercomputers should be capable of gigaflop speeds. Which design parameters need to be examined to allow us to progress from today's to tomorrow's supercomputers? Enslow [77] identifies four system design areas that have contributed to improvements in machine performance:

(1) Devices and circuits: basic hardware speed.

(2) System architecture: the algorithms used to perform the basic functions such as arithmetic and logic operations.

(3) System organization: the topology for interconnecting the major units such as the control unit, memory modules, etc.

(4) System software: the speed and efficiency of the operating system and other supporting software.

While the speed of components has increased by approximately three orders of magnitude over the last thirty years (Hockney and Jesshope [81]), such continued performance improvements using current technology are not expected due to a number of physical limitations such as the speed of light and power dissipation (Noyce [79], Baer [84], Riganati and Schneck [84], and Denning [86]). This does not preclude, however, continued improvements from being made, especially in the last three areas. In particular, given a particular set of devices and circuits, performance may be raised by increasing the level of parallelism in both hardware and software, e.g., by using using multiple functional units or CPUs.

In fact, several past, present, and nearly all proposed supercomputer designs are based on the use of multiple processors to solve a single problem, where the processors execute either independently and asynchronously of one another (e.g., the NYU Ultracomputer (Gottlieb et al. [83a]), the Cosmic Cube (Seitz [85]), Cedar (Kuck et al. [87])) or in lock-step (e.g., the Illiac IV (Barnes et al. [68]), the ICL DAP (Parkinson and Liddell [83]), the MPP (Batcher [80]), the Connection Machine (Hillis [85])).

In this thesis we address operating system requirements for a particular multiprocessor design, one that has the following key characteristics: all processors are asynchronous; processors access a shared memory via a combining network, i.e., the processor-to-memory interconnection network permits concurrent memory references to the same location to be combined and satisfied in the time required for one memory reference; and, the only hardware coordination primitive is a ‘‘fetch-and-add’’ operation, which is essentially an indivisible add-to-memory. Ideally, operating system procedures and data structures on a shared-memory multiprocessor should be serialization-free and concurrently accessible to avoid (potential) performance-limiting bottlenecks.[1] The fetch-and-add primitive, in conjunction with combining interconnection networks, has been proposed as a means for achieving this goal (Gottlieb et al. [83a]), and indeed many highly parallel algorithms based on these two mechanisms have already been developed, such as concurrent queue insertion and deletion. Our thesis extends this work by presenting parallel fetch-and-add based algorithms for various components (e.g., memory allocation) of an operating system for our multiprocessor model.

This chapter is divided into three parts. In the first part we present a history of parallelism in hardware, ending with definitions of our machine model and the fetch-and-add primitive. In the second part we review a number of multiprocessor operating systems and examine various operating system problems (e.g., scheduling, memory allocation) for shared-memory multiprocessors and solutions that have been proposed. Finally, in the last part of the chapter, a detailed outline of the operating system issues studied in the thesis and an overview of our results are given.

---

[1] A bottleneck is a point of communication congestion that can retard a system's progress. A single path between two groups of hardware components, such as processors and memory, can become a *hardware* bottleneck. Similarly, concurrent calls upon a single serially accessible software module can lead to a *software* bottleneck.

1.1. History of Parallelism

The following evolution of parallelism in computer architecture is drawn from a number of sources, including Hockney and Jesshope [81], Haynes et al. [82], Hwang and Briggs [84], Riganati and Schneck [84], Baer [84], and Stone [87]. Our intention is not to present an exhaustive review of the area, but to cite major developments. While examples of machines that exhibit specified features will be given, they will not always represent the earliest implementation of a given technique.

We present the history of parallelism relative to Flynn's [66] machine taxonomy. Flynn divides machine architectures into four different categories based on the number of data and instruction streams that a machine processes concurrently while executing a single program. The classifications are:

- Single instruction steam, single data stream (SISD)
- Multiple instruction steam, single data stream (MISD)[2]
- Single instruction steam, multiple data stream (SIMD)
- Multiple instruction steam, multiple data stream (MIMD)

We note that a number of other taxonomies have also been proposed. For example, see Kuck [82], Schwartz [83], and Hwang and Briggs [84].

SISD machines are conventional uniprocessors. One of the first applications of parallelism in these machines, circa the late 1950s, was in the overlapping of CPU and I/O operations, as on the IBM 709 (Greenstadt [57]). This was followed by the introduction of interleaved memories and the overlapping of instruction decoding and execution, as exhibited by the IBM 7094 (IBM [59]). A major innovation implemented in the mid 1960s was functional parallelism, i.e., providing independent functional units (e.g., adders, multipliers) within a CPU and permitting them to operate concurrently on different data; it was a prominent feature of the CDC 6600 (Thornton [64]).

Additional parallelism was subsequently realized in the late 1960s by pipelining the functional and instruction units. This technique subdivides an operation into stages and permits

[2] As this category appears to lack any actual machine examples, it will not be mentioned further.

different stages to process different data at the same time; it is analogous to the operation of an assembly line (see the survey by Ramamoothy and Li [77] for further details). An example of a machine with pipelined functional and instruction units is the IBM 360/91 (Anderson et al. [67]).

From the mid to late 1970s pipelined functional units that operated on vectors of data were introduced; prime examples are the Cray-1 (Russel [78]) and the CDC Cyber 205 (Kozdrowicki and Theis [80]). We note that there is disagreement over whether vector machines should be classified as SISD (Hockney and Jesshope [81]) or SIMD (Flynn [72] and Stone [80]).

An SIMD machine consists of a control unit, a set of processing elements (PEs), a set of memory modules (MMs), and an interconnection network[3] that permits communication among the PEs and MMs. The control unit issues instructions to the PEs and all (enabled) PEs execute the same instruction at the same time on their own local data. Siegel [85] identifies two types of SIMD machine organizations. A PE-to-PE organization consists of PE-MM pairs, where the interconnection network supports communication among the PEs; the Illiac IV (Barnes et al. [68]) was an example of such a machine. A PE-to-memory organization has each PE (MM) connected to all or a subset of MMs (PEs), where the interconnection supports communication between PEs and MMs; one machine organized in this manner is the BSP (Kuck and Stokes [82]).

The Solomon machine (Gregory and McReynolds [63]), proposed in the early 1960s, was an SIMD machine design based on a 32×32 array of PEs. It has since been followed by a long line of other machines, including the Illiac IV, the ICL DAP (Parkinson and Liddell [83]), the MPP (Batcher [80]), the GF-11 (Beetem et al. [85]), and the Connection Machine (Hillis [85]).

MIMD machines, like SIMD machines, consist of a set of PEs, a set of MMs, and an interconnection network for communication among the PEs and MMs. Now, however, there is no central control unit; each PE independently executes its own instruction stream. As above, an MIMD machine may be classified as a PE-to-PE organization or as a PE-to-memory organization. These two organizations are also referred to, respectively, as message-passing and shared-memory architectures (Kruskal et al. [85]), or as ‘‘boudoir’’ and ‘‘dance hall’’ configurations (Keller et al. [84]).

---

[3] Networks are surveyed in the following section.

In the PE-to-PE organization, each PE is directly connected to a fixed number of other PEs; PEs communicate by passing messages to one another via the interconnection network. PE-to-memory machines share a central memory among the PEs and PEs communicate via this memory. Examples of the two organizations are, respectively, the Cosmic Cube (Seitz [85])) and the C.mmp (Wulf and Bell [72]). We note that it is also possible for an MIMD machine to support both the message-passing and shared-memory paradigms, e.g., the IBM RP3 (see Pfister et al. [85]).

In 1959, Holland [59] described a machine that was an early forerunner of an MIMD machine. It consisted of a two-dimensional grid of ‘‘modules,’’ where each module was an extremely primitive PE. During the 1960s and 1970s, (commercially produced) MIMD machines tended to be relatively modest in parallelism, containing two to approximately eight PEs. Examples include the Burroughs D-825 (ca. 1962), the Burroughs B7700 (ca. 1972), the IBM 370/168 MP (ca. 1974), the Univac 1100/80 (ca. 1976); see Enslow [77] for a more complete chronology. During this same period, moderate sized MIMD machines were also developed, such as the 16 PE C.mmp (Wulf and Bell [72]) and the 50 PE Cm* (Swan et al. [77]) both constructed at Carnegie-Mellon University. More recently, a multitude of MIMD machine architectures intended to support hundreds to thousands of PEs have been proposed. A few examples are the NYU Ultracomputer (Gottlieb et al. [83a]), the Cosmic Cube (Seitz [85]), and Cedar (Kuck et al. [87]); a survey of MIMD machines can be found in Hockney [85].

Other machines that are capable of operating in more than one mode have also been proposed. For example, PASM (Siegel et al. [87]), DADO (Stolfo and Miranker [86]), and NON-VON (Shaw [86]) can perform as SIMD, MSIMD (multiple SIMDs), or MIMD machines.

Architectures to support much finer levels of parallelism than supported by those mentioned above have also been developed. They include dataflow machines, systolic arrays, and very long instruction word (VLIW) machines.

Dataflow machines are designed explicitly to detect and support an algorithm’s inherent parallelism by being based on data (rather than control) flow. An algorithm is mapped into a data flow graph whose nodes and edges represent operators and operands respectively. Operations execute (by being assigned to functional units) when all of their operands are ready (see Dennis [80] for details). The Manchester Prototype DataFlow Computer (Gurd et al. [85])) is an example of this type of architecture; a survey of dataflow machine architecture can be found in Veen [86].

Systolic arrays are VLSI implementations of specific functions, such as matrix multiplication (Kung [82]). The array consists of a network of simple processing elements, called cells, each performing a simple operation. Data flows or is ''pulsed'' through the cells in a pipelined fashion, i.e., computations are overlapped to exploit temporal parallelism. These designs are simple, modular, and hardware efficient; their software requirements are minimal. Often, systolic arrays are incorporated into larger systems.

A VLIW machine consists of a set of PEs, a set of MMs, and an interconnection network. Each instruction executed on VLIW machine controls all of the PEs but operations executed by distinct PEs can be different (Fisher [84]). Instructions are like horizontal microcode, providing fine grained control. Since such a machine is extremely difficult to handcode, special compilation techniques to find and exploit the parallelism within a program have been developed.

Recall that SIMD and MIMD machines consist of PEs, MMs, and interconnection networks. A vast number of networks have been proposed for these machines. Hence, they are reviewed separately in the following section, where our discussion follows that of Feng [81].

1.2. Interconnection Networks

Our survey progresses from simple to complicated networks. A very brief description of each network is given, along with one or more machines that use it.

One of the simplest networks is the bus. It is a common or single path connecting all the functional units, i.e., PEs and MMs. Both the Alliant FX/8 (Perron and Mundie [86]) and the Elxsi System 6400 (Sanguinetti [86]) are bus-based architectures. The ring is another simple interconnection network. Conceptually, the functional units are arranged in a loop, where a unit is connected directly to its left and right neighbors; ZMOB (Weiser et al. [85]) supports such an interconnection structure. A two-dimensional grid connects functional units to their north, south, east and west neighbors; see for example the Illiac IV (Barnes et al. [68]), the ICL DAP (Parkinson and Liddell [83]), and the MPP (Batcher [80]). Still other interconnection structures include the tree and the N-dimensional hypercube, where the connections (edges) between the functional units (nodes) correspond to a tree and an N-dimensional cube, respectively. Examples of these networks can be found in the DADO (Stolfo and Miranker [86]) and the Cosmic Cube (Seitz [85]), respectively.

Other networks are best described by the number of switching stages needed for their implementation, where a network is composed of stages of switching elements and communication links, and interconnections are realized by the settings of the switches. PEs and MMs are situated at the input and output ports of the networks. Single-stage networks include the crossbar and the shuffle-exchange (Stone [71]). The S-1 (Widdoes and Correll [80]) and the C.mmp (Wulf and Bell [72]) use crossbars, while Schwartz [80] describes a shuffle-exchange based machine. Examples of multi-stage networks include the Omega (Lawrie [75]), the SW-Banyan (Goke and Lipovski [73]), the Delta (Patel [79]), and the Benes network (Benes [65]). The NYU Ultracomputer (Gottlieb et al. [83a]) and the IBM RP3 (Pfister et al. [85]) use networks with the topology of the Omega network, the TRAC (Sejnowski et al. [80]) employs a SW-Banyan network, and the GF-11 (Beetem et al. [85]) uses a Benes network.

Single and multi-stage networks can be classified as nonblocking, rearrangeable, or blocking. A nonblocking network, such as the crossbar, can connect an input port of the network to any free output port without blocking. Similarly, a rearrangeable network, e.g., the Benes network, can connect an input port of the network to any free output port, but existing internal connections may have to be rearranged. Finally, a blocking network, examples of which include all of the multi-stage switching networks cited above with the exception of the Benes network, can connect any input port of the network to any free port; however, connections of distinct pairs of input-output ports may conflict (causing one to be blocked) if both need to use a common path between two stages of the network.

We also note that some machines, such as CHiP (Snyder [82]), have dynamically reconfigurable networks.

As stated at the beginning of the chapter, our architectural model is a shared-memory MIMD machine. Therefore, we will now focus our attention on a number of issues concerning networks for such machines; the following review will be cited later when we describe how the model can be realized.

Three factors used in selecting a network for a shared-memory MIMD machine are bandwidth (i.e., the total number of memory requests processed per unit time), cost, and latency (i.e., the time to satisfy a PEs memory request). For example, the simplest and least expensive network is the bus, but the fact that bus bandwidth does not grow with the number of PEs limits the use of this type of network to relatively small machines. At the other end, an $N \times N$ crossbar has bandwidth $\theta(N)$, but cost $\theta(N^2)$. Between these two extremes lie $N \times N$ blocking multi-stage networks; in general they have bandwidth close to that of the crossbar, but their cost is only $\theta(N \log N)$.[4] We note that the minimum latency of an n-stage (blocking) multi-stage network (n is typically $\log N$) is $\theta(n)$ whereas for a crossbar it is $O(1)$.[5] Blocking multi-stage networks, in spite of this last property and because of their bandwidth and cost properties, are found in many shared-memory machines.

Several results regarding the bandwidth of blocking multi-stage networks are reviewed below. They are presented in the context of a shared-memory machine that consists of N PEs and N MMs interconnected via a blocking multi-stage network; the network in turn is composed of $\log N$ stages of 2×2 crossbar switches, where each switch determines to which one of its output ports it is to route a received message, i.e., network routing is distributed. Figure 1-1 is an example of such a network; it has the topology of an Omega network.

To achieve a maximum bandwidth of $O(N)$,

(1) the network must be pipelined; otherwise, the bandwidth is limited to $O(N/\log N)$ (Gottlieb et al. [83a]);

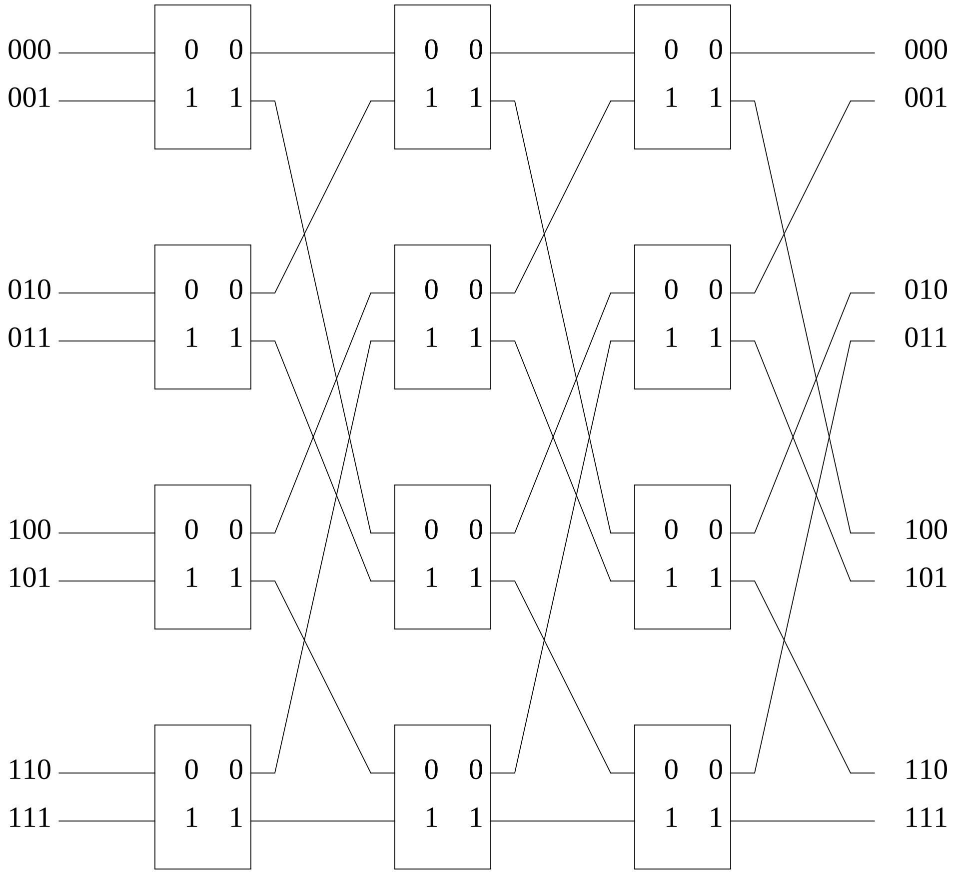
(2) the network must be packet-switched, i.e., data is passed through the network in packets and a physical path from an input port (e.g., a PE) to an output port (e.g., an MM) is not held; the alternative, circuit-switching is incompatible with pipelining (Gottlieb et al. [83a]);

(3) the network must be buffered, i.e., when two messages conflict by needing the same output port of a switch, one should be placed onto a queue associated with the port and the other one forwarded; Kruskal and Snir [83] have shown that if, instead, one of the messages were discarded (as in the Burroughs FMP [79]), the bandwidth would be limited to $O(N/\log N)$.

In their analysis of multi-stage networks, Kruskal and Snir assume that requests to memory modules are uniformly distributed. Pfister and Norton [85] have shown that if this is not so, i.e.,

[4] Nonblocking multi-stage networks tend not to be used due to their large set-up times, which can increase their latency.

[5] This is true (ignoring wire delays) *after* the crossbar is set up.



8×8 Omega network.

Figure 1-1.

if an MM is referenced more frequently than the other MMs, bandwidth can be significantly reduced: the ‘‘hot’’ MM degrades response not only to those PEs that are accessing it, but also to those PEs referencing other (non-hot) MMs. Pfister and Norton also show that if an MM is hot because requests are being directed to the same location within the MM, then by combining such requests when they meet at network switches, the hot spot can be eliminated. If an MM is hot because multiple locations within the MM are being referenced, then combining will not help. In this case, distributing the referenced locations over multiple MMs may reduce the hot spot traffic (Kumar and Pfister [86]).

Pfister and Norton studied combining only one pair of references per 2×2 switch, i.e., two-way combining. Lee et al. [86] suggest that for machines larger than those considered by Pfister and Norton ($N > 256$), 3-way combining may be necessary to avoid hot spots, or in general, greater than k-way combining when $k \times k$ switches are used.

A software solution to the hot spot problem has been proposed by Yew et al. [86] that uses an inexpensive ‘‘software combining tree’’ in place of an expensive hardware combining network. The basic idea is to replace a hot spot location with a tree of locations that are distributed over multiple MMs. Memory references, instead of being directed to a single hot location, are spread over the tree, reducing the negative effects of the hot spot.

1.3. Processor Coordination

Machines composed of multiple PEs must provide mechanisms to coordinate the PEs. For SIMD architectures (e.g., the ILLIAC IV), the central control unit provides this capability. For MIMD machines, the coordination mechanism is determined by the machine organization. A PE-to-PE MIMD machine (e.g., the Cosmic Cube) passes messages among the PEs to achieve coordination. A shared-memory MIMD machine (e.g., the NYU Ultracomputer) uses read/modify/write primitives on shared memory locations to synchronize the PEs. A number of such primitives for shared-memory MIMD machines have been proposed; several are reviewed below.

Two well known primitives are test-and-set and compare-and-swap; they are used, for example, on the IBM 370/168 (Arnold et al. [74], MacKinnon [74]). Another synchronization mechanism is the full/empty bit as used in the Denelcor HEP (Smith [85]); here a bit is associated with each memory word and a load from an empty location is blocked until a store into that location is performed.

The NYU Ultracomputer (Gottlieb et al. [83a]) uses the fetch-and-add primitive to coordinate PEs. Fetch-and-add (F&A) is essentially an indivisible add-to-memory operation that returns the old value of the referenced memory location. Interestingly, the (proposed) implementation of this primitive permits multiple fetch-and-adds on the same location to be satisfied in the time required for one memory reference,[6] i.e., concurrent fetch-and-adds are serialization-free.

Gottlieb and Kruskal [81] described a more general form of fetch-and-add, called fetch-and-ϕ (F&ϕ), where ϕ is an associative binary operation. A fetch-and-ϕ on a shared variable V replaces V with $\phi(V)$ and returns the old value of V. The IBM RP3 intends to support a number of F&ϕ operations, such as fetch-and-add, fetch-and-min, and fetch-and-store (Brantley et al. [85]). Another fetch-and-add extension, called add-and-λ, has been suggested by Harrison [86]. This operation permits more general functions, e.g., modular arithmetic, to be performed atomically on shared memory locations.

Zhu and Yew [85] have proposed a synchronization primitive for Cedar that associates a key and a data field with each memory location. A synchronization instruction consists of a memory location M, a key field, a relational operator, and a key and/or data operation (e.g., fetch, increment, increment-and-fetch[7]). The relational operation is applied to the key fields of M and the synchronization instruction; if the test is satisfied, the indicated operations are performed on M's key and/or data fields.

This concludes our general survey of parallel machine architecures, interconnection networks, and processor coordination primitives. We now detail our architectural model, including the fetch-and-add primitive, and describe how the model can be implemented.

1.4. Architectural Model

Our architectural model is an idealized shared-memory MIMD machine, dubbed a paracomputer by Schwartz [80], a WRAM by Borodin and Hopcroft [82], and a CRCW (concurrent read, concurrent write) PRAM[8] by Snir [82]. Memory loads and stores in a paracomputer are realized in one cycle. Furthermore, concurrent memory references to the same location are also satisfied in the time required for one memory reference. The effect of such concurrent operations is described by the *serialization principle* (Gottlieb et al. [83b]), which states that the effect of simultaneous actions by the PEs corresponds to some (unspecified) serial ordering of the actions.

---

[6] This primitive and its implementation are described in greater detail in the following section.

[7] This last operation is equivalent (semantically) to fetch-and-add.

[8] A PRAM is a parallel random access machine, introduced by Fortune and Wyllie [78], that is a multiple processor analogue of Aho et al.'s [74] RAM, in which concurrent reads are permitted but concurrent writes are not.

We enhance the paracomputer with the fetch-and-add primitive, which was described briefly in the previous section. The format of fetch-and-add is F&A(V, E), where V is an integer variable and E is an integer expression. Fetch-and-add increments V by E and returns the old value of V. The primitive is described by the following function, where brackets ([]) denote code that is executed indivisibly.

```
function F&A (V, E : integer);
var temp : integer;
   [ temp := V;        old value of V
     V    := V + E; ]
     return temp
end function F&A;
```

Fetch-and-add, like the other paracomputer memory operations, executes in a *single* cycle and satisfies the serialization principle. That is, the effect of simultaneous fetch-and-adds acting on the same variable is as if they were executed in some (unspecified) serial order. To illustrate this effect, we shall use an example from Gottlieb et al. [83b]: if PE_i executes

$$a_i := \text{F\&A}(V, E_i)$$

and PE_j simultaneously executes

$$a_j := \text{F\&A}(V, E_j)$$

and no other PE references V, then either

$$a_i = V \text{ and } a_j = V + E_i$$

or

$$a_i = V + E_j \text{ and } a_j = V.$$

In either case the final value of V is $V + E_i + E_j$.

The power of fetch-and-add as a coordination primitive stems from its ability to permit code sections to be executed in parallel that would otherwise have to be performed serially. For example, a group of PEs can simultaneously be assigned different indices of a shared array by having each PE execute

$$MyIndex := \text{F\&A}(I, 1),$$

where *MyIndex* is local to the PE and I is a shared index variable. In the absence of fetch-and-add, the index assignment would have to be performed serially.

A paracomputer represents an ideal but unrealizable parallel machine (e.g., due to fan-in limitations). However, two machines have been proposed, the NYU Ultracomputer and the IBM RP3, that can approximate the behavior of a paracomputer enhanced with the fetch-and-add primitive. The architecture of the RP3 includes the Ultracomputer. We describe the simpler machine, the Ultracomputer, below and refer the reader to Pfister et al. [85] and Brantley et al. [85], and the references therein, for details on the RP3.

1.5. The NYU Ultracomputer

The Ultracomputer is a shared-memory MIMD machine (see Gottlieb et al. [83a] for a detailed description of the Ultracomputer design). It consists of $N = 2^d$ autonomous PEs connected to a shared memory composed of N MMs by an interconnection network having the topology of an Omega network. The network is composed of $\log N$ stages of $(N/2)$ 2×2 crossbar switches; an Omega network of size $N = 8$ is shown in Figure 1-1. Hence, the single cycle memory access of a paracomputer is replaced with a $\log N$ cycle memory access in an Ultracomputer. The Ultracomputer's only coordination primitive is fetch-and-add.

1.5.1. The Interconnection Network

The novel hardware component of the Ultracomputer is its interconnection network. The network is pipelined and packet-switched. Since an Omega network is a blocking multi-stage network, messages can conflict at switches. This contention is resolved by associating a queue with each switch output port and inserting messages contending for the same output port onto the port's queue. These design characteristics permit a network bandwidth that is linear in N (as described in section 1.2).

Network bandwidth is not degraded by hot spot traffic to a single memory location since the network combines concurrent memory references to the same location; that is, through combining, concurrently issued loads and store to the same location do not serialize but are performed in time proportional to $\log N$, the number of stages in the network. We first describe how load and stores are combined and then describe how fetch-and-adds are implemented and combined.[9]

[9] We note that the network and memory organization of the Ultracomputer can be generalized to support fetch-and-ϕ operations, where ϕ is an associative binary operation (see Gottlieb and Kruskal [81] for details).

In an Omega network there is a unique path from each PE to each MM. When memory references (loads and stores) to the same location meet at a switch, the requests are combined and only one forwarded. Combined requests can be further combined with other requests as they progress towards an MM. Loads and stores referencing the same location are combined as follows:[10]

(1) load–load: one load is forwarded and the value obtained from memory is used to satisfy both loads.

(2) load–store: the store is forwarded and when the store acknowledge returns, the stored value is used to satisfy the load.

(3) store–store: one store is ignored and the other forwarded and the store acknowledge is sent to both.

1.5.2. Implementing Fetch-and-Add

To execute atomic fetch-and-adds, the MMs are augmented with adders. An MM upon receiving a F&A(V, E) replaces V with the sum $V+E$ and returns the old value of V to satisfy the fetch-and-add.

The network switches are enhanced with adders and local memories to support the combining of concurrent fetch-and-adds. When a F&A(V, E) and a F&A(V, F) meet at a switch, a F&A(V, $E+F$) is forwarded and E is stored in the switch's local memory. When the F&A(V, $E+F$) returns to the switch with a value of W, the switch returns F&A(V, E) with a value of W and the other F&A with $W+E$. As with loads and stores, a combined F&A can itself be combined with further requests.

The combining of fetch-and-adds with other requests for the same location is done as follows:

(1) F&A(V, E)–F&A(V, F): these are combined as described above;

(2) F&A(V, E)–load(V): the load is treated as F&A(V, 0);

(3) F&A(V, E)–store(V, F): forward store(V, $E+F$) and when the store acknowldege arrives, return F to satisfy the fetch-and-add.

[10] Notice that the resolution of each conflict corresponds to some serial ordering of the two requests.

We note that PE memory requests are satisfied in some (unspecified) order, which may differ from the order in which they were issued. The following two restrictions ensure that a PE's execution is serially reproducible. First, a PE can have at most one outstanding request to the *same* location in shared memory (see Gottlieb [87] for details). Second, a PE can have at most one outstanding request to a read-write (shared) variable, where variables are classified as private, read-only, and read-write (see Brantley et al. [85] and Collier [81] for details).

1.5.3. Memory Organization

Based on performance considerations the memory hierarchy of the Ultracomputer consists of registers and caches at each PE, shared main memory, and secondary storage.[11]

Memory references satisfied in the caches do not incur the large memory latency introduced by the network. Data marked as private or read-only may be cached. Shared read-write data, in general, cannot be cached. This restriction may be lifted when for a given period of time the data is known to be referenced by only a single PE or to be accessed read-only. Such temporarily cacheable data have been studied by McAuliffe [86].

1.5.4. Input and Output

Two input-output organizations have been proposed for the Ultracomputer. A front-end organization calls for several PEs to be connected to local peripheral devices and to perform system I/O; access to main memory is accomplished through the Omega network. While this scheme is inexpensive (with respect to needed additional hardware), block data transfers are inefficient as the network is packet-switched. Thus, a second strategy has been proposed.

A back-end organization uses dual-ported MMs to allow an additional DMA connection. MMs are grouped into clusters (with interleaving occurring within clusters). A circuit-switched network then connects I/O channels to cluster I/O ports.

1.6. A History of Multiprocessor Operating Systems

In the previous sections we have reviewed various parallel architectures, with an emphasis upon shared-memory machines, and in particular, the NYU Ultracomputer. Of course, a machine

[11] The implementation of virtual memory, specifically demand paging, is being studied by Teller [88].

is only partially described by its hardware. To complete its definition we need to examine its underlying software structures, in particular, its operating system. Therefore, we now proceed to survey operating systems for both shared-memory and message-passing MIMD computers. Our examples are drawn from both the commercial and academic sectors. In this review we concentrate upon each operating system's basic structure and note any of its features that we consider novel or innovative; we defer detailed discussions of software synchronization and coordination mechanisms until section 1.7.3. Also, when describing many of the following operating systems, we will often use the term *task* or *process* to refer to a program in execution (Saltzer [66], Dennis and Van Horn [66]).

One of the earliest shared-memory MIMD computers was the Burrough's D-825 (Anderson et al. [62]), which became available in 1960. The D-825 was a symmetrically designed machine: its resources were pooled and treated in an anonymous manner. The machine consisted of up to 4 PEs that shared both a common memory and I/O devices; that is, I/O devices were not dedicated to specific PEs, but availiable to PEs on an as needed basis. Two design goals that the machine was intended to meet were expandability and availability; both goals were met in part by the ability to configure the machine to have multiple PEs.

The operating system for the D-825 was called the Automatic Operating and Scheduling Program (AOSP). One copy of the AOSP was kept in shared memory. When a PE required an operating system service, the PE executed the correspond operating system code to perform the service; that is, PEs serviced their own requests. This organization was chosen so that if one PE failed, the remaining PEs could still continue to execute. To reduce shared data contention problems, multiple PEs were not permitted to execute AOSP concurrently; rather, AOSP ‘‘floated’’ from one PE to another PE as needed.

In the early 1970s, the DEC PDP-10 model KL-10 (DEC [71, 76a, 76b, 77]) was introduced. This shared-memory multiprocessor computer could be configured as either a symmetrical multiprocessor (SMP) or as a master/slave multiprocessor (M/S). Each configuration supported a maximum of 2 PEs. The SMP configuration was similiar in concept to the Burrough's D-825 (and therefore will not be commented on further). The M/S configuration designated one PE to be the ‘‘master’’ and the other PE to be the ‘‘slave’’. Master and slave shared memory, but all peripherals were attached to the master, from which all I/O was done.

The operating system for the DEC PDP-10 was called TOPS-10, and it supported both the SMP and M/S machine configurations. The SMP version of TOPS-10 consisted of a single, shared, reentrant copy of the operating system. Both PEs could concurrently execute the operating system if each one referenced a different data structure; when the same data structure was referenced, a PE had to gain exclusive access to the data structure by locking it.[12] This strategy has also been used in other operating systems such as Hydra and Mach; both are surveyed below.

The M/S version of TOPS-10 was not reentrant, but locks were embedded in the code to serialize the execution of parts of the operating system code (e.g., the scheduler) that were executed by both PEs. In general, the master PE serviced all operating system requests, both its own and those of the slave. However, the slave was permitted to schedule itself, i.e., to obtain work to do without the service of the master. An intention of the M/S configuration was to improve throughput by having the master run I/O-bound jobs and the slave run CPU-bound jobs. Observe that if the slave failed, the system could still function, but if the master malfunctioned, no I/O could be done.

The IBM System/370 model 158 MP (and 168 MP, as well) (Arnold et al. [74], MacKinnon [74]) was a shared-memory computer that could be configured as either a ‘‘true’’ shared-memory MIMD machine with two processors, referred to as multiprocessor mode, or as two separate uniprocessors, referred to as uniprocessor mode. The operating system, OS/VS2 (for Operating System Virtual System 2), supported both configurations. In uniprocessor mode, the PEs operated independently of one another. Each PE had its own main memory, I/O devices, and ran its own copy of the operating system. Once a job was assigned to execute on a given PE, it was executable by that PE only, i.e., the job never migrated to the other PE.

The multiprocessor mode of OS/VS2 was similar to the other symmetric organizations reviewed above. The PEs shared main memory, I/O devices, and there was one reentrant copy of the operating system. Furthermore, each PE could execute the operating system to service its own requests and jobs could migrate back and forth from one PE to the other PE. Instead of ‘‘floating’’ the operating system from PE to PE, as the Burrough’s AOSP did, multiple PEs could execute different operating system functions concurrently. This was accomplished by dividing the operating system into 13 components such that the components had little interaction with one

[12] Methods for locking are reviewed in section 1.7.3.

another. Related data structures (such as the dispatcher queues) were placed into a common component, and, in general, a lock was associated with each component (in some cases a lock was assigned to each member of a component, such as the logical channel queues). A PE executed a function associated with a given component only after it locked the component. This locking strategy was therefore similar to the one employed by DEC's TOPS-10 operating system, but with coarser granularity.

The division of the operating system into 13 components was viewed as a compromise between using one lock for the entire operating system (i.e., a ''floating'' master), which would serialize execution of all operating system functions, and using one lock per object, which might have had significant overheads when there were a large number of locks.[13] To prevent locking deadlocks, e.g., a situation where task *A* holds lock *X* and waits for lock *Y* while at the same time task *B* holds lock *Y* and waits for lock *X*, a lock hierarchy was defined in which locks were requested and released in a fixed order.

Two hardware instructions were also introduced to support the presence of multiple processors. Compare and swap is an indivisible instruction that compares the contents of one register with the contents of a memory location. If the two are equal, the memory location is replaced with the contents of a second register; otherwise, the contents of the memory location is stored in the first register. Compare and swap facilitated the testing and modifying of shared memory locations. The second hardware instruction is Signal Processor (SIGP), which permitted, among other things, one processor to start and stop the other. An application of this instruction was to aid in system recovery if a CPU should fail.

The first three operating systems that we have reviewed were developed in the commercial sector. We now shall review a number of academic projects.

C.mmp was a 16 PE shared-memory MIMD machine (Wulf and Bell [72]). Hydra (Wulf et al. [74, 75], Wulf and Harbison [78], Jones and Schwarz [80]) was the *kernel* or *nucleus* (Brinch Hansen [70]) for the machine's operating system: Hydra provided the basic mechanisms (e.g., low level scheduling, protection) upon which an operating system could be built.

[13] Acquiring a component's lock saved having to lock individually each data structure in the component.

Hydra was an object-oriented operating system (Jones [78a]) with a capability-based protection scheme (Dennis and Van Horn [66], Levy [84]). All physical and virtual resources (e.g., files, disks, procedures) were treated as ''objects''. A capability consists of an unforgeable token (e.g., a pointer) to an object plus a set of access rights that indicate what types of actions may be performed on that object. For a task to perform an operation on an object, it has to have a capability for that object that permits the operation to be performed. At any given time, the set of objects for which a task has capabilities defines its address space, called its local name space (LNS).

An important feature of Hydra was that it separated mechanism from policy (Brinch Hansen [70], Levin et al. [75]). Hydra implemented basic mechanisms, such as the creation and representation of new object types. These mechanisms were then used to implement policies, such as memory management, which were executed as user (i.e., non-privileged) tasks.

One reentrant copy of Hydra was kept in shared memory and multiple PEs could execute the same code concurrently when referencing different data structures. Locking was necessary only when multiple PEs needed to access the same data structure. Aside from a number of software locks (such as semaphores, which are reviewed in section 1.7.3), there was also a hardware kernel lock that permitted a task to remain on a PE while waiting to obtain a lock. The PE upon which the task waited received a hardware interrupt when the lock in question was acquired by the task. While the task waited, the PE did not consume any memory network bandwidth, but the PE itself was wasted. This mechanism was therefore suitable when the delay in obtaining a lock was expected to be small.

Another multiprocessor project at CMU was Cm^* (Swan et al. [77]). Cm^* consisted of 50 computer modules (Cms), where each Cm consisted of a PE connected to an MM (and possibly some I/O devices) via a bus. The Cms were partitioned into 5 equal sized clusters. A cluster's Cms were interconnected via a common bus (called the map bus). The clusters were connected to one another via intercluster buses. The memory of Cm^* was composed of the MMs of the individual Cms; there was no central memory per se. However, each PE could address the memory of any other Cm. The time to satisfy a memory request was on the order of 1, 3, or 9 time units depending upon whether a local, intracluster, or intercluster MM had to be accessed.

Two operating systems were developed for Cm^*: StarOS (Jones [78b], Jones et al. [79]) and Medusa (Ousterhaut et al. [80]). Like Hydra, both operating systems were object-oriented and supported capability-addressing. A key feature of both operating systems was the concept of a *task force* (TF). A TF is a collection of tasks (called activities in Medusa) that cooperate closely in solving a single problem. Due to this close interaction, an attempt was made to *coschedule* (Ousterhaut et al. [80]) TFs, that is, to assign each task of a TF to a different PE and to schedule these tasks to execute concurrently. StarOS and Medusa themselves were implemented as task forces; for example, the file manager and the task force manager were TFs. Communication between tasks was by message passing; synchronization was via special objects called semaphores (examined in section 1.7.3).

StarOS, unlike Medusa, hid some details of the hardware hierarchy of Cm^* at a possible performance loss. For example, in StarOS, each operating system function could execute within each cluster. In contrast, Medusa was divided into functionally disjoint TFs (called utilities), and due to the cost of remote memory references, a PE could execute a given function only if it could execute the code for that TF locally; otherwise, a (potentially intercluster) message was sent to a PE that could perform the function locally.

The Elxsi System 6400 (Sanguinetti [86]) is a commercially produced, bus-based multiprocessor. It can support up to 10 PEs, 6 MMs, and 4 I/O processors connected to a common bus. While this machine supports a shared-memory, its operating system, Embos (Olson [85]), supports a message-passing model of computation. In Embos, each resource is managed by one task. These tasks do not share any data; they communicate via message passing. One task can send messages to another task if it holds a *link* (i.e., capability) to that task; a link is thought of as a virtual circuit between two tasks. One or more links may point into the same *funnel*, and multiple funnels may be associated with a task. A task receives messages sent to a *funnel* in first-in, first-out order. As in the symmetric operating systems above, Embos permits a task to migrate from one to PE to another PE.

It is interesting to note that while Embos itself is based on the message passing paradigm, it supports both the shared-memory and message-passing models for user programs. Shared memory support was added late in the development cycle; the designers of Embos originally felt that the latter model was complicated and (mistakenly) believed that the users of their system would not be interested in this model of computation.

The NYU Ultracomputer design (Gottlieb et al. [83a]), unlike the previous machines mentioned in this section, is for a very large shared-memory parallel processor; one on the order of hundreds to thousands of PEs. The hardware for this machine has been described already in section 1.5.

The operating system for the Ultracomputer is called Symunix (Gottlieb et al. [87], Edler et al. [88a, 88b]), for symmetrical Unix (Ritchie and Thompson [74]). Symunix does not ''float'' from PE to PE (like Burrough's AOSP) nor is it divided into a number of large serially accessible components (like IBM's OS/VS2). Instead, Symunix advocates a self-service paradigm whereby each PE services its own operating system requests and shared data is held in concurrently accessible data structures. Hence, multiple PEs can concurrently perform the same operating system function using the same data structure without serializing. A number of concurrently accessible data structures (e.g., queues) and parallel coordination primitives (e.g., readers/writers) have been developed to support this approach and are reviewed in sections 1.9, 2, and 3.

Next, we consider operating systems for message-passing or PE-to-PE MIMD organizations. We concentrate upon distributed, as opposed to network, operating systems. A distributed operating system presents a unified view of the multiprocessor to a user; the interconnection network is transparent. For example, there is one global file system and the physical location of files and executing tasks is unknown to users. A network operating system, such as the Newcastle Connection (see Brownbridge et al. [82]), does not hide such information from the user (see Tanenbaum and Van Renesse [85] for details).

Locus (Walker et al. [83]) is a distributed Unix system that has been operational on a 17 processor machine at UCLA since 1981. Locus adhers closely to the description given above for distributed operating systems. We note three features. First, Locus permits files to be replicated to increase performance; the ''closer'' a copy is located to a task, the faster the task can access it. Locus is responsible for keeping the replicated copies of a file up to date. The second feature is that when a task requests an operating system service, it traps to the operating system. If the service is provided only on a remote PE, a message is sent to that PE to perform the service. The task is unaware of this transmission; it does not resume until the remote PE services the request.[14] Lastly, tasks can migrate from one PE to another PE. The site (PE) where a task is created, its *origin site*, is responsible for keeping track of the task's current location (i.e., PE). A task's origin site assists in forwarding messages to the task.

[14] This trapping mechanism is a primitive form of remote procedure call, which is discussed below.

Since 1982 the V distributed operating system (Cheriton and Zwaenepoel [83], Cheriton and Mann [84], Cheriton [84], Theimer et al. [85]) has been running on a system of 25 processors at Stanford University. In this system, each processor is dedicated to a specific function such as a workstation, file server, print server, etc. Interprocess communication is via message passing, e.g., send, receive, reply. Tasks are organized into groups called *teams*. A team of tasks share a common address space and therefore must be run on the same PE. An example of a team is a file server, where individual team members read, write, and open files. V permits tasks to migrate from one PE to another in response to system needs. When a task migrates, its relationship with its old PE is completely severed, e.g., local files are migrated with the task.

Eden (Almes et al. [85], Black [85]) is a distributed operating system that became operation in 1983 at the University of Washington in Seattle. It is an object-oriented, capability-addressed operating system. In Eden objects are called *ejects*. Ejects are active: associated with an eject are one or more tasks that perform the operations defined on the object. The tasks of an eject share the same address space. Ejects are *mobile*, i.e., they can migrate from PE to PE.

In Eden, interprocess communication is via remote procedure calls (RPCs) (see Spector [82] and Birrell and Nelson [84]). Basically, an RPC looks like a standard procedure call, but it is translated by a *stub* procedure into a message that is sent to a task that can perform the desired operation. The calling task is blocked until a reply is received, i.e., until a return is made from the invoked procedure.

The Amoeba project (Tanenbaum and Van Renesse [85], Tanenbaum et al. [86]) at Vrije Universiteit, Amsterdam, is another study in distributed operating systems. The current Amoeba machine consists of 24 PEs. The PEs are divided into two groups. One group of PEs is dedicated to specific functions (as in V). The other group is treated as a pool from which members are dynamically assigned to service requested functions and then returned to the group.

The operating system for Amoeba is distributed and object-oriented with capability addressing. Capabilities are protected cryptographically. Interprocess communication is via message passing.

In 1984, Carnegie-Mellon University began the implementation of an operating system called Mach (Accetta et al. [86], Rashid [86], Rashid et al. [87], Young et al. [87]), which is a successor to the Accent operating system (Rashid and Robertson [81]). Mach is intended to support both shared-memory and message-passing models of computation. It supports four basic abstractions: tasks, threads, ports, and messages. A task represents a collection of resources (e.g., a paged virtual address space, access to ports); it defines the execution environment for threads. A thread is the unit of computation; it is the entity executed by a PE. A thread consists basically of a program counter within a task. The threads of a task share the task's resources. A port is a communication channel (or queue) to which messages are sent and from which they are received. A message contains data and/or port capabilities. As in an object-oriented operating system, operations on an object (other than a message) are performed by sending messages to a port associated with the object. The virtual memory design of Mach supports both read/write and copy-on-write[15] memory sharing between tasks. Copy-on-write can also be used when large messages need to be sent. On a shared-memory multiprocessor, the threads of a task may be executed in parallel and the threads can use shared memory for communication and synchronization. Mach currently runs on a number of multiprocessors including the BBN Butterfly (BBN [85]), the Sequent Balance (Sequent [86]), the Encore MultiMax, multiprocessor VAXen (e.g., VAX 11/784), and the simulator for the IBM RP3.

1.7. Operating System Issues for Shared-Memory MIMD Machines

An operating system for a shared-memory MIMD machine needs to provide the same services as an operating system for a uniprocessor, e.g., processor scheduling, resource allocation and management, prevention of system deadlocks, memory and file protection, etc. In addition, the former must handle new problems introduced by the presence of multiple PEs, such as load balancing (i.e., ensuring that some PEs are not overloaded with work while others remain idle). As stated at the beginning of the chapter, it is important for the operating system to be efficient if the performance potential offered by the PEs is to be realized.

How should an operating system for a shared-memory MIMD machine be organized? In the previous section, three basic types are identified (see Enslow [77]): *master-slave, separate supervisors*, and *symmetric*. The first of these, master-slave, is the easiest to implement. The operating system resides in one PE, the master, and the other PEs, the slaves, direct their operating system requests to and have them serviced by the master (e.g., the M/S version of DEC's TOPS-10). There is no contention for system tables or code since they are accessed only by the master. This

[15] Copy-on-write (Bach [86]) permits two tasks to physically share the same data although logically each task has its own copy of the data. When a task updates the data, it is given a physically distinct copy.

organization is subject to catastrophic failure when the master PE malfunctions; also, the system becomes inefficient when the master cannot satisfy requests in a timely fashion: slaves idle waiting for the master to service their requests.

Associating a separate executive (i.e., operating system) with each processor (e.g., the uniprocessor mode of IBM's OS/VS2) treats the PEs of a multiprocessor as a set of independent uniprocessors. Each PE runs its own copy of the executive and has its own files and I/O devices. This organization is not subject to catastrophic failure but it does not support multiprocessing within a single job: tasks executing on different PEs cannot cooperate. In addition, since tasks cannot migrate from PE to PE, there can be processor load balancing problems.

A symmetrical or anonymous organization pools the resources (e.g., PEs, I/O devices, shared memory) of a machine and uses them on an as needed basis. The operating system is shared by the PEs and its execution is based on a self-service paradigm: each PE executes the operating system to service its own requests. Hence, this method (subject to how it is implemented) is free of master service bottlenecks (there being no master), but it may only succeed in shifting the bottleneck from the master to the data structures that are shared by the PEs: bottlenecks may arise when multiple PEs attempt to access concurrently the same data structure. In this organization, tasks can migrate freely among PEs, facilitating automatic load balancing, and tasks being executed by different PEs can cooperate on a single problem. In general, if a PE fails, the remaining PEs can still continue to execute. In short, this organization permits the hardware of the machine to be efficiently utilized.

We have seen a range of implementations for the symmetrical organization. Each implementations is based on the self-service paradigm; they differ primarily with respect to the level of parallelism supported, i.e., with respect to the maximum number of PEs that can service their own requests without serializing. Probably the simplest and easiest symmetrical organization to implement is one that permits the operating system, or master, to ‘‘float’’ from one PE to another PE on an as needed basis (e.g., Burrough's AOSP). However, treating the operating system as a single, serially accessible entity introduces a bottleneck when multiple PEs need to use it at the same time. To alleviate this bottleneck, the operating system can be partitioned into a small number of large serially accessible components (e.g., the multiprocessor mode of IBM's OS/VS2), or better yet, into a large number of small serially accessible components (e.g., CMU's Hydra). PEs can execute concurrently as long as they do not reference the same component (i.e., data structure); these two strategies, as noted above, shift the bottleneck for obtaining service from a master PE to obtaining exclusive access to a data structure. Finally, the latter type of bottleneck can be reduced (and in some cases eliminated) by using concurrently accessible data structures where possible (e.g., NYU's Symunix).

Each one of the above operating system organizations may experience performance degradation due to service and/or data access bottlenecks. We have already observed that the relative cost of a serial bottleneck rises linearly with respect to the number of PEs involved. While bottlenecks may be tolerable on small multiprocessors, they can become unacceptable on multiprocessors composed of hundreds or thousands of PEs, which are the sizes proposed for the next generation of multiprocessor supercomputers. Ideally then an operating system for a shared-memory MIMD machine should be free of bottlenecks. We note that the Ultracomputer's Symunix operating system comes closest to achieving this goal, but at a cost in both hardware and software. A major hardware expense is the combining interconnection network; however, without it, memory references to the same location would serialize, precluding bottleneck-free access to code and data strucutures. A software cost is that parallel algorithms are more complicated than their serial counterparts, and often the serial (uniprocessor) time to execute a parallel algorithm is greater than that of its serial counterpart.

We conclude that of the master-slave, separate supervisors, and symmetric (i.e., self-service) organizations, the last one, being free of the central server bottleneck (in an ideal implementation), is an attractive operating system paradigm for large shared-memory MIMD machines. A major problem that must be addressed for this scheme to be successful is preventing shared data access from becoming a bottleneck. Below, we review how this problem has been addressed with respect to scheduling, memory allocation, and PE coordination.

1.7.1. Scheduling

The scheduling problem deals with the assignment of work to PEs. In the context of a self-service organization, this means that each PE is responsible for its own scheduling, i.e., obtaining work for itself. We review various solutions to this problem based on the following terminology and scheduling model.

A *task* or *process* is a program in execution (Saltzer [66], Dennis and Van Horn [66]). A task is *running* when it is being executed by a PE; it is *ready* when it is capable of running but there is no available PE; and, it is *blocked* when its execution cannot proceed until it acquires a resource (such as a critical code section, which is reviewed below). Ready tasks are kept on a queue called the *Ready Queue*.[16] A blocked task is inserted onto a wait queue associated with the resource it has requested. When a task blocks or exits the system, it invokes a scheduling primitive that deletes a task from the Ready Queue, which is then run on the PE.

Suppose the Ready Queue is serially accessible. Then, it may become a bottleneck as the number of PEs grows. To avoid this situation, various proposals have been put forth by Sullivan et al. [77], Cezzar and Klappholz [83], Klappholz and Park [84], and Ni and Wu [85] that replace the Ready Queue with multiple queues $Q_1, Q_2, \ldots, Q_n$. In most of these proposals, when a task is created, it is inserted onto a randomly selected queue.[17] Sullivan et al. [77], Cezzar and Klappholz [83], and Ni and Wu [85] (in one version) associate each PE with one queue. Then, when a PE becomes free, it deletes a task from its assigned queue. Alternatively, Ni and Wu [85] (in a second version) and Cezzar and Klappholz [83] place the queues into a common pool. When a PE becomes free, it selects (e.g., at random) a queue from which to obtain a task. In general, the problem with multiple ready queues is that they may lead to load imbalancing; also, bottlenecks may still arise when multiple PEs concurrently access the same queue.

Gottlieb et al. [83b] describe a bottleneck-free scheduling scheme with good load balancing. The Ready Queue is represented by a concurrently accessible queue data structure. Hence, multiple PEs can access the Ready Queue without serializing; furthermore, since a central queue is used, no PE remains idle while a task is ready and thus the load is well balanced. Gottlieb et al. [83b] also describe two other concurrently accessible data structures, *queues with priority* and *queues with multiplicity*, that are used to support, respectively, task priorities and the constant-time spawning of multiple instances of a single task. A queue with priority is a queue consisting of items tagged with priorities; a priority insertion adds an item with priority to the queue, and a priority deletion deletes an item of highest priority from the queue. A queue with multiplicity or *multiqueue* is defined as a queue consisting of items tagged with multiplicities. That is, entries on the queue are of the form (I, M), representing M consecutive copies of an item I on a (hypothetical) much longer queue. The time required for a multiqueue insertion is constant (i.e., it is

[16] To simplify the description we assume all tasks are of equal priority.

[17] Ni and Wu [85] consider additional selection schemes that include picking the shortest queue or using a deterministic assignment policy.

independent of an item's multiplicity). *M* deletions can be concurrently performed on an item with multiplicity *M*, where each deletion makes a copy of the item, and the *M*th deletion to do so takes the item off of the multiqueue.[18]

A more specific scheduling problem concerns the execution of parallel loops within a program.[19] This problem is important since such loops account for the greatest percentage of program parallelism (Kuck et al. [84]). The basic problem is to have loop iterations be assigned to PEs in such a way as to minimize scheduling overheads and to achieve load balancing.

Solutions to this problem may be classified as either static or dynamic (Gajski and Peir [85], Polychronopoulos et al. [86]). *Static* loop scheduling determines the assignment of iterations to PEs at compile time. The solutions given assume that execution times of loop instances are approximately the same. Hence, the number of iterations executed by each PE can be evenly distributed and executed without run-time scheduling overhead (Polychronopoulos et al. [86], Cytron [87]). However, the assumption may not hold, as for example, when conditionals are present in a loop.

Dynamic loop scheduling assigns iterations to PEs at run-time as PEs become available. This approach supports load balancing but can incur run-time scheduling overheads, e.g., when a PE invokes the operating system scheduler (which may also cause the PE to be context-switched) in order to obtain a new iteration. Hence, a special case of dynamic loop scheduling, called *self-scheduling* (Smith [81]), that attempts to reduce the run-time scheduling overhead has been investigated (e.g., see Tang and Yew [86] and Fang et al. [87]). Basically, this method permits each PE that is executing a loop to determine for itself which iteration(s) to execute next. This is accomplished by having the PE access a set of shared variables, which include indices for the loop, from which it can determine the next iteration to be executed. The effectiveness of self-scheduling then depends upon the efficiency with which these shared variables can be accessed. For example, on an Ultracomputer with a combining network, multiple PEs can increment a shared index variable (via fetch-and-add) in a fully parallel manner. On more conventional machines, e.g., those with bus-based architectures, the PEs' increments to the shared variable

[18] Queues, queues with priority, and queues with multiplicity will be reviewed in greater detail below.

[19] Here we assume that parallel loops are explicitly identified either by a restructuring compiler (e.g., see Kuck [82]) or by a programmer.

must be done serially.

A variation of loop self-scheduling is to have PEs obtain multiple instances or ‘‘chunks’’ (Edler et al. [85] and Fang et al. [87]) of iterations at a time. This policy decreases scheduling overheads by reducing the number of accesses made to the shared loop index variables. Also, chunking permits parallel loops consisting of short instances to be executed efficiently by effectively combining the small instances into larger ones.

Kruskal and Weiss [85] have shown that chunking may reduce the time for all instances of a parallel loop to be executed; this is advantageous when the PEs executing a loop must synchronize (i.e., wait for the loop to be completed) before continuing with the program's execution.[20] Kruskal and Weiss suggest that an optimal loop scheduling scheme might be obtained by decreasing the number of iterations assigned to each PE as a loop's execution evolves. Indeed, Polychronopoulos and Kuck [87] have proposed such a policy, called *guided self-scheduling*, that varies the chunk size assigned to each PE to permit all of the PEs executing a loop to finish at approximately the same time.

1.7.2. Memory Allocation

Memory allocation, like scheduling, represents another potential system bottleneck. Recall that in the self-service operating system paradigm, PEs service their own requests. Therefore, access to the data structure that represents the pool of free memory space may cause contention and degrade system performance. To date, the subject of parallel memory allocation has not been as extensively studied as scheduling. This is reflected in our survey below.

A number of parallel memory methods have been based on the serial first fit method. The first fit method (Bailey et al. [64], Berztiss [65], Comfort [64], Ross [67], Knuth [68]) is a memory allocation scheme that permits memory to be dynamically partitioned into variable size blocks. Briefly, a request for a block B of size s searches the pool of available or unallocated blocks for the *first* block F whose size is greater than or equal to s. If no such F exists, the request fails. Otherwise, B is allocated from F and F's size is reduced by s; if this results in the complete allocation of F, F is removed from the pool. When a block is returned (deallocated), it

[20] Similarly, coscheduling (Ousterhaut et al. [80]) attempts to schedule the tasks associated with a given job to execute at the same time to increase performance by decreasing the amount of waiting due to synchronization.

is added to the available block pool. If the returned block is physically adjacent to a block in memory that is already in the pool, the two are coalesced to create a larger block. The block pool data structure is often represented by a single linked list (Berztiss [65], Comfort [64], Ross [67]).

The parallel first fit algorithms of Ellis and Olson [87] divide the linked list of free blocks into a set of sublists; blocks are ordered by starting address within a sublist (to facilitate block coalescing). They consider three different levels of list locking granularity: locking all sublists (which is equivalent to serial first fit), locking one sublist at a time, and locking individual blocks on a sublist. The authors suggest that the second method should be appropriate when the number of sublists and the number of tasks capable of making concurrent memory requests are equal; whereas, the third method should be selected when the number of concurrent requests is greater than the number of sublists. Algorithms based on only the third locking protocol are given.

Stone [82] and Bigler et al. [85] also outline parallel first fit algorithms based on the last locking protocol described above. An interesting feature of Stone's algorithm is that multiple tasks can perform allocations from the same available block in a fully parallel manner by decrementing the block's size via fetch-and-add (and by assuming the presence of a combining network).[21] Ellis and Olson use Stone's allocation method in their algorithms.

We are interested in serialization-free memory and scheduling algorithms since they eliminate performance bottleneck penalties. Such algorithms are, of course, a goal; often, a parallel algorithm will contain some serial code, such as when a task requires exclusive access to a shared data structure. In the next section we review a number of synchronization and coordination problems that are germane to our architectural model and algorithms.

1.7.3. Synchronization and Coordination

In 1965 Dijkstra [65] described the ‘‘critical section’’ (or mutual exclusion) problem. The problem involves coordinating a set of asynchronously executing tasks such that the tasks take turns using a resource. Each task accesses the resource by executing a ‘‘critical section’’ of code. No two tasks may be executing their critical sections at the same time. Furthermore, Dijkstra stated that a solution to the problem must be symmetric among the tasks, make no assumptions about task (PE) speeds, be unaffected if a task should stop outside of the critical section, and

[21] Stone's allocation technique is reviewed in chapter 4.

ensure that tasks do not deadlock, i.e., engage in an endless ‘‘after you, after you’’ scenario in attempting to enter the critical section. Syntactically, the critical section is bracketed by entry and exit protocols that coordinate the tasks:

entry protocol
    critical section
exit protocol

One of the earliest known solutions to the critical section problem for two processes is due to Dekker (see Dijkstra [68a]). It uses only indivisible reads and writes (i.e., no special synchronization primitives such as test-and-set were assumed). In Dekker’s solution, the entry protocol has a complicated loop structure, involving one outer loop and two inner loops. Doran and Thomas [80] provided a simpler solution to this problem that has only one loop in the entry protocol. Peterson [81] then followed with a solution that contained an even simpler loop structure than the one given by Doran and Thomas.

For a system of N tasks, Dijkstra [65] presented a solution to the critical section problem that uses, like the two process solutions above, only indivisible reads and writes. Knuth [66] noted that Dijkstra’s solution was subject to starvation, i.e., a task could wait indefinitely to enter the critical section. He gave a ‘‘fair’’ (i.e., starvation-free) solution that ensured that a task would wait at most $2^{N-1}-1$ turns before entering the critical section, where a turn is defined as a task’s execution of the critical code section. DeBruijn [67] then reduced the waiting time to $\frac{1}{2}N(N-1)$ turns. Eisenberg and McGuire [72] further shortened this time to $N-1$ turns. Recently, Lamport [87] has given a solution that requires a total of seven shared memory references in the absence of contention for the critical section; in contrast, the solutions above require $\theta(N)$ references to shared memory when there is no contention. Lamport notes that his solution admits starvation.

The space complexity of solutions to the critical section problem has also been examined. For example, Dijkstra’s solution uses one N-valued variable (i.e., a variable that can take on N different values) and N 3-valued variables (Burns [81]). Burns and Lynch [80] have shown that any solution to this problem using just indivisible reads and writes must have at least N shared variables. Burns [81] has proven that any symmetric solution must have at least one shared variable that takes on at least $N-1$ values.

The above solutions to the critical section problem are not immune to system problems, such as the failure of a PE. Hence, so-called ‘‘distributed system’’ solutions have been proposed that can continue to preserve mutual exclusion even in the presence of failed system components. Typically, these solutions assume that associated with each task are state variables that can by written to only by that task and read by all other tasks. When a task fails, it writes its state variables to indicate its malfunction. Of course, in reality, a faulty task may not behave so nicely. For example, it could just halt without indicating its failure, or worse, it could behave malevolently and try to disrupt the system. The latter type of behavior falls under the heading of the Byzantine Generals Problem (Pease et al. [80], Lamport et al. [82]), which poses the question of how nonfaulty tasks can continue to function in the presence of malevolent tasks. For solutions to this problem see Pease et al. [80], Lamport et al. [82], Dolev and Strong [83], and Dolev et al. [86, 87]. Below, we review solutions to the critical section problem for distributed systems that assume faulty tasks behave benevolently.

Lamport [74] gave a distributed solution to the critical section problem that was fair: waiting tasks enter the critical section based on a first-come, first-served (FCFS) discipline. Specifically, tasks are assigned ticket numbers, as in a bakery, when they execute their entry protocols, and a task is served (enters the critical section) when its number is called. Lamport's solution suffered two shortcomings: not all variables were bounded in value, and the system could become deadlocked by a task that repeatedly failed and restarted. A version of Lamport's solution that does not suffer from the latter problem was given by Hehner and Shyamasundra [81]. Rivest and Pratt [76] also improved upon Lamport's solution by eliminating both defects cited above; however, while their solution was fair, it did not serve waiting tasks in FCFS order. In addition, Rivest and Pratt's solution required $\theta(N)$ values to be associated with each state variable. This bound was reduced to $\theta(\log N)$ by Peterson and Fisher [77]. Katseff [78] presented a solution that served waiting tasks in FCFS order and that did not suffer from the shortcomings of Lamport's solution, but it did require $\theta(2^{2n})$ shared values per task. Peterson [83] has given a solution that needs only 4 shared values per task, (which is within one of the known lower bound (Peterson [79])); the solution is fair, but does not use a FCFS discipline to service waiting tasks.

Solutions to the critical section problem that are based on more powerful primitives than indivisible read and write have also been proposed. For example, Dennis and Van Horn [66] defined a lock bit that is manipulated by Lock and Unlock primitives. Access to a critical section is bracketed by Lock(W)/Unlock(W), where W is a 1-bit variable associated with the critical section. A task executing Lock(W) waits until W equals one and no other task is referencing W, and then sets W to zero; the testing and setting of W must be done indivisibly. Unlock(W) sets W to one.

The Lock/Unlock solution of Dennis and Van Horn, as well as all of the previous solutions cited above, require a task to busy-wait (cycle on a PE) when attempting to enter a critical section if another task is executing the critical section. Busy-waiting wastes PE cycles if there are other tasks ready to run.

In 1968 Dijkstra [68a] presented his now well-known semaphore solution to the critical section problem, which is free of busy-waiting. A semaphore is a nonnegative integer variable that can only be manipulated by the indivisible primitives P and V. A critical section *CS* is bracketed by P(S) and V(S), where S is the binary semaphore (i.e., it can only take on the value zero or one) associated with *CS*. P(S) decrements S by 1 if S is positive (after which the calling task enters *CS*); otherwise, the calling task is placed onto a wait list associated with S (and the PE executing the calling task obtains and executes a task from the Ready Queue, i.e., the PE is context-switched). V(S) selects a task from the wait list, if the list is not empty, and places the task onto the Ready Queue; otherwise, S is incremented by 1. Dijkstra did not indicate how a waiting task was to be selected, but some type of fair scheduling was assumed. In another description of semaphores, Dijkstra [68b] permitted the semaphore value to be negative when tasks were on the wait list in which case $|S|$ equals the number of waiting tasks. In this description he also stated that it was immaterial which of the waiting tasks was selected when a V was done. In yet a third description, Dijkstra [72] suggested that the wait list could be represented by a queue and waiting tasks processed in first-in, first-out (FIFO) order.

Busy-waiting implementations of P/V have also been proposed (e.g., see Shaw [74]). In such an implementation, a task is not placed onto a wait list (when it exeutes a P and the semaphore is nonpositive), but instead busy-waits.

The various ways in which a task can wait to enter a critical section has led Stark [82] to propose the following P/V protocol taxonomy. *Weak* protocols busy-wait, *block-set* protocols use unordered wait lists (i.e., resumed tasks are selected arbitrarily), and *block-queue* protocols maintain FIFO wait lists. The first two protocols are subject to starvation, the last is not.

In 1971, Courtois et al. [71] described the readers/writers problem, which is a generalization of the mutual exclusion problem. The readers/writers problem deals with two types of tasks, *readers* and *writers*, sharing a resource that may be concurrently accessed by either an arbitrary number of readers or by a single writer. Readers and writers *request*, *use*, and then *release* the resource. Let us refer to a task that has requested but not yet released the resource as an *operating* task and to an operating task whose resource request has been denied and not yet granted as a *waiting* task. When a reader requests the resource and there are no operating writers, the request is granted and the reader uses the resource; otherwise, the request is denied and the reader becomes a waiting reader (and either busy-waits or is inserted onto a read wait queue). Similarly, when a writer requests the resource and there are no operating tasks, the request is granted and the writer uses the resource; otherwise, the request is denied and the writer becomes a waiting writer (and either busy-waits or is inserted onto a write wait queue). Typically, readers and writers are treated equally or writers have priority to prevent them from being starved by readers.[22] In each case, when a reader releases the resource, it unblocks (grants the resource to) a waiting writer if one exists. On the other hand, when a writer releases the resource, it unblocks all waiting readers if readers and writers have equal priority but preferentially unblocks a waiting writer if writers have priority. Semaphore-based solutions to the readers/writers problem have been given by Courtois et al. [71], Brinch Hansen [73b], and Conradi [77].

Vantiborgh and van Lamsweerde [72] defined generalized P and V operations, P(S, N) and V(S, N), that modify a counting semaphore S (i.e., a semaphore that can assume values greater than one) by integer values $N \geq 1$. These operations allow simple solutions to be stated for a number of synchronization problems, such as the readers/writers problem.

Another basic synchronization mechanism is the event, which is commonly associated with the occurrence (or completion) of an operation such as I/O (Brinch Hansen [73a]). The primitive Await(E) delays a task (by having the task busy-wait or insert itself onto a wait list) until another task causes event E via Cause(E). As defined by Brinch Hansen, events are memoryless. Hence, if a task performs Await(E) after another task has performed Cause(E), the former task waits until the next Cause(E) is executed. On the other hand, the operating system for the Burroughs B5700/6700, Master Control Program (MCP), supports events that have memory (Organick [73]). Associated with an event is a *happened* bit that records whether the event has occurred. With this type of event, if a task performs Await(E) after another task executes Cause(E), the former task does not wait. An event can be reset (to *not happened*) via Reset(E).

[22] The alternative, writers starving readers, is less likely to occur since writers do not share the resource, i.e., it is not possible for one writer to begin using the resource before a previous writer has released it; of course, this behavior is possible for readers. We note that readers can be given priority, if need be.

Reed and Kanodia [79] proposed synchronization mechanisms called eventcounts and sequencers that are as powerful as semaphores but often yield simpler solutions to various coordination problems (e.g., producers and consumers). An eventcount records the number of times a particular event has occurred. Operations on an eventcount E include Advance(E) (an event has occurred), Read(E), and Await(E, V) (delay until E has occurred at least V times). Sequencers are used to determine which of several events occurred first.

To reduce programming errors due to missing or erroneously placed synchronization primitives (e.g., an unmatched P) and to make parallel programs more understandable, higher level synchronization constructs have been defined. Hoare [72] and Brinch Hansen [73a] define a *critical region* as a sequence S of statements that refer to a shared variable V:

region V **do** S **end**.

We will refer to a variable that is the subject of a region construct, such as V above, as a region variable. Region variable V can be accessed only within regions referencing V, and regions that reference the same region variable are mutually exclusive. Hoare [72] extended the critical region definition to include conditions associated with entry into a region. That is, a *conditional critical region* associates a boolean condition B with a region

region V **when cond** B **do** S **end**;

When a task executes the region and finds B true, it performs S; otherwise, the task temporarily exits the region and is inserted onto a wait queue associated with the region. When a task exits the region, it re-evaluates the conditions of waiting tasks and if some tasks' conditions are now true, one such task is resumed.

Regions for a given shared variable are placed at the points where the variable is accessed. Hoare [74] suggested that these regions (i.e., their operations on the shared variable) and the definition of the shared variable be (textually) gathered together into a single module called a *monitor* (also called a *secretary* by Dijkstra [72]). A task accesses a shared variable by invoking a monitor operation; the operations of a monitor are mutually exclusive.

This completes our survey of task scheduling, memory allocation, and task synchronization and coordination. Note that solutions to problems in these areas may involve parallel programs. That is, a self-service operating system for a shared-memory machine may itself be a parallel program. Hence, we are interested in techniques, other than testing, that permit the properties of such programs to be verified.

1.7.4. Proving Properties of Parallel Programs

A parallel program (even a short one solving a simple problem like critical sections or readers/writers) can have a large number of possible execution sequences, making it difficult to ensure merely by testing that the program is ‘‘correct,’’ i..e., satisfies certain program constraints, such as mutual exclusion. Hence, proof techniques for verifying the properties of parallel programs have been developed.

Assertional proof techniques consider the states of tasks executing a program;[23] conditions that are to be verified are stated as assertions over a program’s variables (Owicki [75] and Owicki and Gries [76]) and control locations (Keller [76], Lamport [77, 80]). Typically, an assertional technique proves that a program satisfies a verification condition when executed sequentially and then shows that this proof still holds under parallel execution. Many of these techniques are extensions of Hoare’s [69] axiomatic proof technique and Floyd’s [67] intermediate assertional method for sequential programs.

Owicki and Lamport [82] present an assertional proof technique that includes temporal assertions. These are assertions based on temporal logic (Pneulli [77], Manna and Pneulli [81]), which is an extension to logic that includes certain types of assertions about the future. For example, ‘‘$x > 0$ now and forever’’ is a temporal assertion. Such assertions make it convenient to state that something will eventually happen in a parallel program, e.g., a task will eventually enter its critical section.

In general, there are two types of properties proven for a parallel program (Lamport [77], Owicki and Lamport [82]). *Safety* properties state that something bad cannot happen. Examples include freedom from deadlock and mutual exclusion. On the other hand, *liveness* properties state that something good can eventually happen. For example, fairness and freedom from starvation are liveness properties. We note that not all proof techniques can be used to establish both types of properties. For example, while Owicki and Lamport’s [82] technique can be used to prove liveness and safety properties, the technique of Lamport [80] can prove only the latter.

[23] A non-assertional proof technique has also been given by Lamport [79]; program proofs are based on the execution sequence of program statements.

1.8. Thesis Outline

Our thesis addresses data structures and algorithms that can drastically reduce serialization in the memory, task, and resource management components of an operating system for a large shared-memory MIMD machine; our goal is to eliminate performance-inhibiting bottlenecks from the operating system. Our architectural model was defined in section 1.4 and we adopt the self-service paradigm described in section 1.6. More specifically, we follow the strategy espoused by Gottlieb [87] for the NYU Ultracomputer: each PE is responsible for servicing its own requests, and shared data are held in concurrently accessible structures. The thesis begins by reviewing and then extending several existing concurrently accessible data structures, e.g., queues and multiqueues. These modified data structures are then used to implement parallel algorithms for the operating system components.

A detailed description of our results, along with a layout of the thesis follows. The reader will note that we have extended Stark's [82] taxonomy of P/V implementations to other coordination functions: a *weak* implementation requires tasks to busy-wait (e.g., when a resource is not free); whereas, a *block-queue* implementation places a task onto a FIFO queue when the task must wait.

In chapter 2 we study parallel implementations of one type of structure that is integral to many operating system functions, the queue. We first review the sequentially allocated (i.e., contiguous array) queue implementation of Gottlieb et al. [83b] and their parallel algorithms for insertion (at the rear of the queue) and deletion (from the front of the queue). We then introduce a third type of operation, *Interior Access* (IA), that can reference an item at any position within a queue, not just at its front or rear. Next, we review the linked list implementation of queues outlined by Rudolph [82], and present (both fair and unfair) algorithms for insertion and deletion to such structures. For both the sequential and linked list implementations, the IA operation of *removal*–the deletion of an item from an interior queue position–is dealt with in detail; this operation plays a significant role in many operating system functions. For example, one may wish to remove a task from a queue as a prelude to destroying the task. An important feature of our queue implementations is that the inclusion of interior access operations does not reduce a queue's degree of concurrent accessibility.

In chapter 3 we consider a parallel queue variant: the multiqueue, a queue composed of items with multiplicity (section 1.7.1). We first review the multiqueue implementations of

Gottlieb et al. [80] and then present algorithms for insertion and deletion on sequentially allocated multiqueues that are more efficient than those originally proposed by Gottlieb et al. [80]. We then extend these new algorithms to support linked allocated multiqueues. To complement the insertion of *M* items onto a multiqueue, we define a *multi-deletion* of size *K* as the deletion of *K* items from a multiqueue; the multiqueue lends itself towards the efficient implementation of this operation since a single queue entry may contain multiple items. We also define operations that access interior multiqueue items, again, without reducing a multiqueue's maximum level of concurrency. Two specific interior access operations that are defined are *RaiseMult* and *LowerMult*. These operations raise and lower the multiplicity of a queued item, permitting applications to adjust item multiplicities to their needs. LowerMult may be used to remove an item from a multiqueue. We also observe that the insertion and deletion of arrays and queues of items may be efficiently supported on multiqueues by treating an array or queue of *M* items as a single item with multiplicity *M*.

In chapter 4 we address the problem of memory allocation. As noted previously (section 1.7.2), requests for memory on a shared-memory MIMD machine represent a potential system bottleneck. In this chapter two basic memory allocation techniques are studied. One dynamically divides memory into variable size blocks and satisfies requests of arbitrary size. The other one statically divides memory into fixed size blocks and satisfies requests for arbitrary numbers of blocks (which need not be contiguous). We present parallel algorithms for allocating variable size blocks based on the serial *boundary tag method* (Knuth [68]), which is a first fit allocation method, and the serial *binary buddy system* (Knowlton [65], Knuth[68]); both serial methods are reviewed in the chapter. Also, two parallel fixed size block algorithms are given.

In chapter 5 we consider the task and resource management components of the operating system. A representative set of task and resource primitives that could form the nucleus of the task and resource management components of an operating system for our architectural model is studied. The intention of this exercise is to demonstrate that serialization-free implementations of task and resource primitives are attainable. In addition, we present strategies that show how system activities (e.g., task creation, execution, (normal) termination, destruction, resource allocation) can be made parallel. In particular, we present highly parallel block-queue implementations of readers/writers and event primitives, as well as a weak (i.e., busy-waiting) implementation for events. The majority of the weak event primitives are serialization-free; notably, the important operation of testing an event's status is fully parallel. In our readers/writers implementations, reader activity is fully parallel when there are no writers. An important feature of both the block-queue readers/writers and event implementations is that tasks waiting (on a queue) for a resource can be resumed in parallel by multiple PEs. Also, for each block-queue implementation it is shown that blocked tasks may be destroyed without reducing an implementation's maximum level of parallelism.

In chapter 6 implementations of high-level parallel programming constructs (e.g., parallel loops) are considered. First, the parallel constructs are presented and translated into the task primitives of chapter 5. Next, shared variables are discussed in the context of block-structured languages. Then, performance issues are considered, and finally a method for dynamically optimizing the number of tasks assigned to the execution of a construct is given.

In chapter 7 we present a fair block-queue implementation of the readers/writers problem and prove that it satisfies the following criteria: readers and writers are mutually exclusive; writers are mutually exclusive; no deadlock can occur; and multiple readers can use the resource at the same time (when no writers are present).

Chapter 8 summarizes our results. We also outline how our queue and multiqueue implementations may be simplified when they are used only as repositories for unordered items, i.e., items that need not be processed in FIFO order.

Finally, we include two appendices that review the queue with priority and queue with multiplicity implementations of Gottlieb et al. [80].

Many of our algorithms use fetch-and-add based primitives developed by Gottlieb et al. [83b] and Rudolph [82]. We review these primitives below. The reader should observe that the implementations of the primitives permit multiple tasks to execute concurrently primitives that act on the same variable without serializing (when a combining network is present).

1.9. A Review of Fetch-and-Add Based Primitives

The implementations below and throughout the thesis are presented in a programming language notation that is very similar to Pascal (Wirth [71]); any major discrepancies between the two will be noted in the text.

1.9.1. The Test-Modify-Retest Paradigm

A very common pattern found in the implementations of Gottlieb et al. [83b] is the testing, modifying, and retesting of a shared variable. These patterns are described by the functions Test-Decrement-Retest (TDR) and Test-Increment-Retest (TIR). TDR(*V*, *Delta*) returns true if *V* can be decremented by *Delta* and remain nonnegative (as a side-effect, *V* is decremented by *Delta*); otherwise, it returns false. TIR(*V*, *Delta*, *Bound*) returns true if *V* can be incremented by *Delta* without exceeding *Bound* (as a side-effect, *V* is incremented by *Delta*); otherwise, the function returns false.

```
function TDR (V, Delta : integer);
    TDR := false;
    if V ≥ Delta then                       test V
        if F&A(V, –Delta) ≥ Delta then      modify and retest
            TDR := true
        else
            F&A(V, Delta)                   reset V
        end if
    end if
end function TDR;

function TIR (V, Delta, Bound : integer);
    TIR := false;
    if V + Delta ≤ Bound then                       test V
        if F&A(V, Delta) + Delta ≤ Bound then       modify and retest
            TIR := true
        else
            F&A(V, –Delta)                          reset V
        end if
    end if
end function TIR;
```

The first test of *V* in each function is necessary to prevent race conditions from occurring among tasks that are repeatedly performing these functions. This matter is discussed in detail in Gottlieb et al. [83b].

1.9.2. Weak P/V

Two weak P/V protocols are given. The first is fair, the second is not.

1.9.2.1. Version One

The fair protocol (Rudolph [82]) is based on Lamport's bakery service analogy. Upon entering the bakery (by executing a P), a task is assigned a ticket number. *Now_Serving* is the ticket number of the task currently being served (executing its critical section). When a task's service is complete, it advances *Now_Serving* by 1 (via a V) and the task whose ticket number matches *Now_Serving* is then served.

For this protocol, a semaphore is composed of an integer component *Ticket#* that is used to dispense ticket numbers to tasks, and an integer component *Now_Serving* that is the number of the task being served. Both components are initialized to zero. Code to bound the counters is included, but will not be explained, except to state that it is assumed that the constant *MaxR* bounds the maximum number of active tasks.[24]

First note that variables declared within procedures are local (i.e., non-shared) variables. Second note that the P/V implementation is for a binary semaphore, i.e., at most one task at a time can be in the critical code section associated with the semaphore.

```
Weak_Semaphore = record Ticket#, Now_Serving : integer end;

procedure Weak_P_V1 (S : Weak_Semaphore);
var my_number : integer;
    my_number := F&A(S.Ticket#, 1) mod MaxR;
    if my_number = MaxR−1 then
        F&A(S.Ticket#, −MaxR);
        F&A(S.Now_Serving, −MaxR)
    end if;
    repeat until my_number = S.Now_Serving mod MaxR
end procedure Weak_P_V1;

procedure Weak_V_V1 (S : Weak_Semaphore);
    F&A(S.Now_Serving, 1)
end procedure Weak_V_V1;
```

[24] A fully defined queue implementation is given is section 2.2.2.2 that bounds counters in a similar manner.

1.9.2.2. Version Two

An unfair implementation of P/V is easily expressed using TDR and fetch-and-add (Gottlieb et al. [83b]). A semaphore is represented by a nonnegative integer variable *S*. To enter a critical section, a task loops until it successfully performs a TDR of 1 on *S*. That is, the first task to perform a TDR on *S* when *S* is positive obtains the resource. A task can starve if it always happens to test *S* after other tasks have made *S* nonpositive.

```
Weak_Semaphore = integer;

procedure Weak_P_V2 (S : Weak_Semaphore);
   repeat until TDR(S, 1)
end procedure Weak_FCFS_P;

procedure Weak_V_V2 (S : Weak_Semaphore);
   F&A(S, 1)
end procedure Weak_V_V2;
```

1.9.3. Weak PChunk/VChunk

Gottlieb et al. [83b] refer to the generalized P/V operations of Vantiborgh and van Lamsweerde [72] as PChunk (PC) and VChunk (VC). Recall that PC/VC can modify a semaphore by an arbitrary positive integer value. The PC/VC procedures follow directly from the second version of P/V given above; they admit starvation.

```
Weak_Semaphore = integer;

procedure Weak_PC (S     : Weak_Semaphore;
                   Delta : integer);
   repeat until TDR(S, Delta)
end procedure Weak_PC;

procedure Weak_VC (S     : Weak_Semaphore;
                   Delta : integer);
   F&A(S, Delta)
end procedure Weak_VC;
```

1.9.4. Weak Readers/Writers Coordination Primitives

Two weak readers/writers implementations are given (Gottlieb et al. [83b]). The first one treats readers and writers equally and can thus easily starve writers.[25] The second one avoids this starvation by giving writers priority over readers.[26] For each implementation reader activity is fully parallel in the absence of writers.

Readers and writers request and release a resource by performing fetch-and-adds (via PCs and VCs) on a shared activity counter *RW_Act* associated with the resource. The implementations assume that the maximum number of readers is less than the constant *MaxR*, which is also the initial value of *RW_Act*.

1.9.4.1. Readers/Writers with Equal Priority

Each read request (or release) modifies the activity counter by one; each write request (or release) modifies it by *MaxR*.

```
procedure Read_Request (RW_Act : integer);
    Weak_PC(RW_Act, 1)
end procedure Read_Request;

procedure Read_Release (RW_Act : integer);
    Weak_VC(RW_Act, 1)
end procedure Read_Release;

procedure Write_Request (RW_Act : integer);
    Weak_PC(RW_Act, MaxR)
end procedure Write_Request;

procedure Write_Release (RW_Act : integer);
    Weak_VC(RW_Act, MaxR)
end procedure Write_Release;
```

[25] A fair weak readers/writers implementation appears in Rudolph [82].

[26] As previously indicated, reader priority can be implemented analogously but is less useful.

1.9.4.2. Readers/Writers with Writer Priority (WP)

In this implementation writers are given priority over readers. The variable $\#W$ (which is initially zero) equals the number of writers that have requested but not yet released the resource; a reader's request is delayed while $\#W$ is positive.

```
procedure WP_Read_Request (RW_Act, #W : integer);
    repeat until #W = 0;
    Weak_PC(RW_Act, 1)
end procedure WP_Read_Request;

procedure WP_Read_Release (RW_Act : integer);
  Weak_VC(RW_Act, 1)
end procedure WP_Read_Release;

procedure WP_Write_Request (RW_Act, #W : integer);
    F&A(#W, 1);
    Weak_PC(RW_Act, MaxR)
end procedure WP_Write_Request;

procedure WP_Write_Release (RW_Act, #W : integer);
    Weak_VC(RW_Act, MaxR);
    F&A(#W, –1)
end procedure Write_Release;
```

CHAPTER 2

Queues

The Ultracomputer operating system design utilizes the distributed processing of system-wide information kept in shared data structures. The success of this approach is influenced by the degree to which the contents of these structures can be concurrently accessed; software bottlenecks may develop if these structures are only serially accessible. In this chapter we study the parallelization of one type of structure that is integral to many operating system functions, the queue. The first half of this chapter examines sequentially allocated queues, i.e., concurrently accessible queue structures that are implemented using contiguous arrays. The second half deals with linked allocated queues, i.e., concurrently accessible queues implemented as sets of linked lists, with each list serially accessible.

Section 2.1 begins with a review of the sequentially allocated queue implementation of Gottlieb et al. [83b] and their algorithms for insertion (at the rear of the queue) and deletion (from the front of the queue). We then introduce a third type of operation, *Interior Access* (IA), that can reference an item at any position within the queue, not just at its front or rear. In section 2.2, after first reviewing the linked list implementation of queues introduced by Rudolph [82], we present algorithms for insertion and deletion to such structures. For both the sequential and linked list implementations, the IA operation of *removal*–the deletion of an item from an interior queue position–is dealt with in detail. (The importance of this operation was cited earlier, in chapter 1.) For both the sequential and linked implementations, the addition of interior access operations leaves invariant the number of concurrent operations that can be performed on the structure. That is, interior access does not reduce a structure's degree of concurrent accessibility.

The concurrently accessible queues of this chapter can also be used as a repository for unordered items, i.e., items that need not be processed in FIFO order. We refer to such a repository as a *pile*. In chapter 8 we consider how the queue structures and operations can be simplified when they need only support piles.

2.1. Sequentially Allocated Queues

In section 2.1.1 we review the basic queue structure for sequential allocation and generic procedures for insertion and deletion. Coordination problems due to concurrent queue operations are described in section 2.1.1.1; protocols to resolve them are given in section 2.1.1.2; fully defined insertion and deletion algorithms are given in section 2.1.1.3. A generic implementation for interior access operations is given in section 2.1.2, followed by protocols for coordinating insertions, deletions, and IA operations in section 2.1.2.1. Complete algorithms are given in section 2.1.2.2.

2.1.1. A Review of Insertion and Deletion

Gottlieb et al. [83b] implement a concurrently accessible queue of size *Qsize* as a circular array of cells *Cell*[0..*Qsize*−1]. The queue operations of insert (at the rear) and delete (from the front) are fully parallel, i.e., a maximum of *Qsize* operations, one per cell, can concurrently access the queue; no coordination is required among the *Qsize* operations. The queue is ‘‘logically distributed’’ (Jones and Schwarz [80]), that is, each cell is treated as a separately accessible resource with its own critical code section (or lock) as opposed to having a single critical code section associated with the entire queue. Thus, concurrent operations may be performed on different cells simultaneously. The First-In, First-Out (FIFO) property of the queue within this parallel environment is defined as follows: If the insertion of item *I* finishes before the insertion of item *J* starts, then a deletion of item *J* cannot be completed before a deletion of item *I* has started.

The front and rear of the queue are defined by the queue components *Qfront* and *Qrear*. Specifically, *Qfront* modulo *Qsize* and *Qrear* modulo *Qsize* are the (indices of the) front and rear queue cells. A deletion executes

$$d := \text{F\&A}(Qfront,\ 1)\ \mathbf{mod}\ Qsize$$

to determine its *deletion site* (*Cell*[*d*]). Similarly, an insertion executes

$$i := \text{F\&A}(Qrear,\ 1)\ \mathbf{mod}\ Qsize$$

to find its *insertion site* (*Cell*[*i*]). Through the fetch-and-add multiple deletion and insertion sites may be concurrently calculated without serializing references to *Qfront* and *Qrear*.

Two other queue components, *Qupper* and *Qlower*, record upper and lower bounds on the number of items in the queue. Both are initially zero. An insertion increments *Qupper* at its initiation and *Qlower* at its completion. A deletion does just the opposite, decrementing *Qlower* and *Qupper* at the start and end of its activity, respectively. The two counters differ by at most the number of active insertions and deletions. Thus, during periods of queue inactivity, *Qupper* = *Qlower* = the number of items in the queue.

Queue overflow and underflow occur when an insertion is attempted on a full queue (*Qupper* = *Qsize*) and a deletion on an empty one (*Qlower* = 0), respectively. In such cases, the operations return with failure.

An insertion first determines if space is available in the queue by performing a test-increment-retest (TIR) on *Qupper*. If the queue is found to be full, the insertion fails. Otherwise, an insertion site is calculated as described above. The insertion then waits until it can execute its critical code section for the site. After depositing its item into the cell, *Qlower* is incremented, signaling the presence of an item ready for deletion. Deletions are performed in a similar manner.

The basic queue data structure and operations are given below. We assume that the items acted upon by insertions and deletions are actually pointers to objects, where objects are application-defined structures, such as blocks of memory. A sequentially allocated queue composed of values, instead of pointers, can also be supported by the algorithms to be described. The definitions of the entry and exit protocols referenced within the insertion and deletion procedures are postponed until after the discussion of cell coordination problems in the next section.

```
Seq_Queue = record   Sequentially Allocated Queue
                Qlower, Qupper, Qsize, Qfront, Qrear : integer;
                Cell :          array[0..Qsize–1] of
                                record
                                   Value : Object_Pointer
                                end
            end;

Object = record
              Application_Defined_Attributes
         end

Object_Pointer = ↑Object;

Initially:
     Qlower, Qupper, Qfront, Qrear = 0;
```

```
function Seq_QInsert (Item : Object_Pointer;
                      Q    : Seq_Queue);
var i : integer;
    with Q do
    if TIR(Qupper, 1, Qsize) then
        i := F&A(Qrear, 1) mod Qsize;
        EntryProtocolForCell[i];
            Cell[i].Value := Item;
        ExitProtocolForCell[i];
        F&A(Qlower, 1);
        return true
    else
        return false
    end if
    end with
end function Seq_QInsert;

function Seq_QDelete (Item : Object_Pointer;
                      Q    : Seq_Queue);
var d : integer;
    with Q do
    if TDR(Qlower, 1) then
        d := F&A(Qfront, 1) mod Qsize;
        EntryProtocolForCell[d];
            Item := Cell[d].Value;
        ExitProtocolForCell[d];
        F&A(Qupper, −1);
        return true
    else
        return false
    end if
    end with
end function Seq_QDelete;
```

These insertion and deletion algorithms permit *Qfront* and *Qrear* to attain arbitrarily large values. Gottlieb et al. [83b] observe that these counters need only be maintained accurate modulo *Qsize*. A method for doing so, due to Rudolph [82] (an alternative method may be found in Gottlieb et al. [83b]), is reviewed below for *Qrear* (*Qfront* is similarly bounded).

When an insertion task detects that *Qrear* modulo *Qsize* has become zero, the task subtracts *Qsize* from *Qrear*. (The decrement does not disrupt the proper assignment of sites to other concurrent operations.) Between the time a task detects that the counter has to be adjusted and the decrement is performed, other insertions may enter the queue, each raising the counter's value by one. Each time the counter hits a multiple of *Qsize*, another task will pause to decrement it. A maximum of *Qsize* insertion tasks can be in the queue about to perform the decrement. Further insertions are blocked since the queue is full (*Qupper* = *Qsize*). Therefore, the maximum value *Qrear* can attain before being decremented is $Qsize^2$.

The code to calculate an insertion site and maintain the counter modulo the queue size is:

```
i := F&A(Qrear, 1) mod Qsize;
if  i = Qsize–1 then              Qrear was a multiple of Qsize
    F&A(Qrear, −Qsize)
end if;
```

All subsequent algorithms will contain code to bound queue counters.

Up to *Qsize* concurrent queue operations may be executed by assigning each to a different cell. Note however that since the processors do not execute in lock-step, no predictions can be made as to the order in which any two operations acting on distinct sites terminate. All that can be guaranteed is the FIFO queue property defined earlier. Moreover, the PEs, being asynchronous, do create coordination problems, which must be resolved by the cell protocols.

2.1.1.1. Cell Coordination Problems

No coordination is called for among concurrent queue operations acting on different cells. Due to the asynchrony of the PEs however, the possibility exists that two or more operations can be assigned to, and thus contend for, the same cell. In this section we review the causes for this contention and in the following section resolutions will be given.

A cell may be contended for (1) by an insertion and a deletion or (2) by concurrent insertions or concurrent deletions. We will refer to the first as *consecutive insertion/deletion* contention and to the second as *concurrent insertion/deletion* contention. The basic rule for resolving both types of contention is that a cell must be alternately accessed by an insertion and a deletion. Note that when a queue is only serially addressable, this alternation is implicitly guaranteed. Explicit mechanisms must be in place to achieve this same behavior for a concurrently accessible queue. The following scenarios illustrate how the alternation of insertions and deletions may be upset in the absence of such mechanisms.

Scenario 1: *Consecutive Insertion/Deletion Contention*

(1) On an initially empty queue, two insertions, I_0 and I_1, begin execution on *Cell*[0] and *Cell*[1], respectively.

(2) I_1 finishes first. At this point there is one item in the queue, in *Cell*[1], and *Qlower* = 1.

(3) Between the termination of I_0 and I_1, the first deletion D_0 may begin since there is an item in the queue (*Qlower*>0). D_0's deletion site, however, is *Cell*[0] not *Cell*[1]. Therefore, D_0 and I_0 will contend for the same cell. However, if D_0 does not wait for I_0 to complete, it will access an empty cell.

A similar problem, where an insertion arrives at a cell before a deletion has emptied it, occurs when the queue is nearly full.[1]

A full/empty flag associated with each cell would be sufficient to prevent an operation from prematurely accessing a cell if scenario one were the only source of contention. Unfortunately, a second source exists: multiple insertions or deletions may concurrently vie for the same cell. This problem, first told to us by Boris Lubachevsky, is highlighted below in more detail. We assume that cell alternation is enforced between insertions and deletions.

Scenario 2: *Concurrent Insertion/Deletion Contention*

(1) Insertions I_0 through I_{N-1} are begun on an initially empty queue of size N where *Cell*[j] is the insertion site for I_j, $0 \le j < N$. Further insertions are blocked at this time since the queue is full, i.e., *Qupper* = N.

(2) I_1 through I_{N-1} complete. At this point, $N-1$ items are in the queue ready to be deleted. I_0 has yet to insert its item into *Cell*[0].

(3) Two deletions, D_0 and D_1, begin execution. D_0 waits for I_0 to fill *Cell*[0]. D_1 empties *Cell*[1] and decrements *Qupper* (to $N-1$).

(4) The next insertion, I_N, can now begin since the queue is no longer full (*Qupper*<N); *Cell*[1] is, in fact, empty. However, I_N's insertion site is calculated to be *Cell*[0]. Thus, I_0 and I_N contend for *Cell*[0].

[1] As Boris Lubachevsky has stated, this is easily seen by viewing deletions as insertions of empty space.

2.1.1.2. Cell Protocols for Insertion and Deletion

Gottlieb et al. [83b] present four different weak cell protocols[2] that solve the coordination problems presented above. Two of these resolve cell contention using ‘‘global’’ information, i.e., data associated with a queue’s state, whereas the other two rely on ‘‘local’’ information, i.e., data associated with a cell’s state. The former are fair while the latter are not.

The global protocols define a serial ordering for insertions and deletions based on their fetch-and-adds of *Qrear* and *Qfront*. These counters are shared, or global, to all insertions and deletions. For example, the insertion site calculation may be performed in two steps as

$$t := \text{F\&A}(Qrear, 1);$$
$$i := t \ \mathbf{mod} \ Qsize;$$

to assign a ticket number (or timestamp) t and a site i to an insertion. Deletions may be assigned ticket numbers and sites in a similar manner. Using these assignments, one can then refer to the jth insertion (deletion) as the insertion (deletion) with ticket number j. One of the global protocols guarantees that an item inserted into the queue by the jth insertion must be removed by the jth deletion. The other global protocol divides insertions and deletions into groups of size *Qsize*, called *phases*, based on their ticket numbers. The jth phase consists of all insertions and deletions whose ticket numbers range from $j * Qsize$ to $(j+1) * Qsize - 1$. Access to the entire queue (and thereby the cells) is controlled on a phase basis, i.e., items inserted into the queue by phase j insertions may only be removed by phase j deletions. Furthermore, deletion phase j delays until deletion phase $j-1$ ends, and insertion phase j delays until insertion phase $j-2$ ends. Both protocols operate by delaying a given operation until its ticket or phase number is called (as in Lamport’s [74] bakery analogy). Starvation is avoided at the risk of reducing the ‘‘throughput’’ of the queue.[3] An example of the first protocol will be given for linked insertion and deletion (see section 2.2.2.2); the second protocol will be used in the implementation of logarithmic multiqueues (in chapter 3).

---

[2] Recall from chapter 1 that *weak* protocols use busy-waiting, as opposed to task-switching, to wait for a resource.

[3] That is, queue operations executed by fast PEs may have to wait for operations being executed by slower PEs to terminate.

The two local protocols resolve operator contention for a cell based on data that is associated with the cell. As a consequence, these protocols do not preserve the serial ordering of insertions and deletions as defined by the global protocols. Under the local protocols, access to a cell among a set of competing tasks is given to the first one that sets a lock that is local to the cell. As an example of how operators may access a cell in non-serial order, consider scenario two, where I_0 and I_N address the same cell. Using a local protocol, the latter can place its item into *Cell*[0] before the former, although I_0 preceded I_N into the queue (i.e., despite I_0 having a smaller ticket number than I_N).[4] Similarly, the *i*th item in the queue may not be removed by the *i*th deletion.

The local protocols described by Gottlieb et al. [83b] are not fair. However, as noted below, starvation-free versions may be easily derived. The advantage of local over global protocols is that the former can increase the potential throughput of the queue, especially if the protocols are not required to be fair, i.e., if starvation can be tolerated.

We now describe one of the local protocols (the other one is quite similar in style): Concurrent queue operations are coordinated through cell entry and exit protocols that are implemented using weak semaphore operations. An insertion and a deletion semaphore are associated with each cell. Initially, the insertion semaphore is *open* (i.e., it has a value of one) and the deletion semaphore is *closed* (i.e., it has a value of zero). Alternation of insertion and deletion (see scenario 1) is achieved through the semaphores: An insertion first closes its semaphore and, after depositing its item into the cell, opens the deletion semaphore. The deletion operates in a symmetrical manner. Multiple operations acting on the same cell (see scenario 2) busy-wait (by executing weak Ps) until they can proceed.

The insertion protocols for *Cell*[*i*] are:

Weak_P_V2(*Cell*[*i*].*Insert_Sem*);	*entry protocol*
deposit item into cell	
Weak_V_V2(*Cell*[*i*].*Delete_Sem*);	*exit protocol*

[4] Based on an extension of this example, Vladmir Lanin (Lanin [87]) has shown that there exist scenarios for which the local protocols will not preserve the FIFO queue property as defined in section 2.1.1. For example, in the above example, the deletion of the item inserted by insertion I_j $(1 < j < N)$ may not be started before the deletion of the item inserted by I_N is completed.

The deletion protocols for *Cell*[*d*] are:

```
Weak_P_V2(Cell[d].Delete_Sem);        entry protocol
    remove item from cell
Weak_V_V2(Cell[d].Insert_Sem);        exit protocol
```

The protocols are specified using the second version of P/V. Recall from chapter 1 that this semaphore implementation is not fair. Starvation may be prevented by substituting the first version of P/V, which is fair.

In the remainder of the thesis, when weak semaphore operations are cited in an implementation, we will generally base our descriptions of the data structures and algorithms on the second version of P and V. It should be understood that the first version of P and V can always be substituted.

2.1.1.3. Putting It All Together

In this section, fully defined insertion and deletion algorithms are given. All counters are bounded, and cells are accessed using the local protocol described in the previous section.

```
Seq_Queue = record   Sequentially Allocated Queue
                Qlower, Qupper, Qsize, Qfront, Qrear : integer;
                Cell :    array[0..Qsize−1] of
                          record
                               Value            : Object_Pointer;
                               Insert_Sem,
                               Delete_Sem       : Weak_Semaphore
                          end
            end;

Object = record
              Application_Defined_Attributes
         end

Object_Pointer = ↑Object;

Initially:
     Qlower, Qupper, Qfront, Qrear = 0;
     for 0≤j<Qsize,
       Cell[j].Delete_Sem              = 0;
       Cell[j].Insert_Sem              = 1;
```

```
function Seq_QInsert (Item : Object_Pointer;
                      Q    : Seq_Queue);
var i : integer;
    with Q do
    if TIR(Qupper, 1, Qsize) then
        i := F&A(Qrear, 1) mod Qsize;
        if i = Qsize−1 then F&A(Qrear, −Qsize) end if;
        Weak_P(Cell[i].Insert_Sem);
            Cell[i].Value := Item;
        Weak_V(Cell[i].Delete_Sem);
        F&A(Qlower, 1);
        return true
    else
        return false
    end if
    end with
end function Seq_QInsert;

function Seq_QDelete (Item : Object_Pointer;
                      Q    : Seq_Queue);
var d : integer;
    with Q do
    if TDR(Qlower, 1) then
        d := F&A(Qfront, 1) mod Qsize;
        if d = Qsize−1 then F&A(Qfront, −Qsize) end if;
        Weak_P(Cell[d].Delete_Sem);
            Item := Cell[d].Value;
        Weak_V(Cell[d].Insert_Sem);
        F&A(Qupper, −1);
        return true
    else
        return false
    end if
    end with
end function Seq_QDelete;
```

2.1.2. Interior Access

Queue operations are normally directed at the ends of the queue–insertions at the rear and deletions at the front. However (as mentioned earlier in chapter 1), at times it is necessary to violate this restriction and access an interior item.

The simplest way to implement an interior access operation is to grant it exclusive access to the entire queue. Unfortunately, this can drastically reduce the concurrent accessibility of the queue. The implementation strategy that has been adopted for interior access requires exclusive access only to individual cells and thus does not lower the level of concurrent queue activity.

The operations that may be performed on an interior item may be divided into two types. The first consists of operations that read and/or modify an item but leave it positioned on the queue as found. That is, the item cannot be taken out of its queue cell nor may data fields within the item that are referenced by the queue operations (e.g., see the definition of *Cell_Loc* below) be altered. Operations of the second type may have write access to the entire contents of an item and its queue.[5] For the sequentially allocated queues, the algorithms for type one and type two interior access are identical, aside from the specific actions an operation performs once exclusive access to a cell is obtained. Therefore, it suffices to provide only a single procedure to describe both types of interior access operations. This will not be true for the linked list version of queues (see section 2.2.3); a separate procedure for each type will be necessary.

Our primary example of a type two operation is the deletion of an interior item from a queue, which we refer to as a *removal*. In this operation, an item, regardless of its queue position, is replaced with a dummy value of nil. The queue counters–*Qfront*, *Qrear*, *Qlower*, *Qupper*–are not altered and a deletion that addresses a cell whose item has been removed simply returns nil. The calling routine may then decide whether or not to retry the deletion. Notice that as an alternative to substituting nil values for removed items, the queue could be compacted following an item's removal. While this would eliminate deletion checks for ''nil'' items, we do not see how to do this without suspending all other queue activity to adjust the queue's cells and counters.

An interior access operation is passed three parameters: an item I that is to be accessed, a sequentially allocated queue Q, and a function ϕ that is to be performed on I. If I is a member of Q,[6] the IA operation obtains exclusive access to the cell containing I,[7] performs $\phi(I)$, and returns with success (i.e., it returns true). Otherwise the operation fails (returns false). In general, an IA operation succeeds if an insertion of I into Q has completed and no deletion or removal of I has started. An IA operation fails if a deletion or removal of I from Q has completed and no insertion

[5] By this definition queue insertion (at the rear) and deletion (from the front) are also type two interior access operations. However, insertion and deletion will be treated as distinct from other interior access operations.

[6] Other criteria could also be used to decide if the item is to be accessed or not, such as whether it is a member of *any* queue or a queue of a given type (e.g., a semaphore wait queue). An example of an applications requiring a different test is the highly parallel implementation of the good fit (a hybrid of first fit and the buddy system) memory allocation algorithm, which is described in chapter 4.

[7] We assume that a queued item may be manipulated only by an insertion, deletion, or IA operation.

of *I* into *Q* has begun. An IA operation's success or failure is implementation dependent if an insertion, deletion, or removal of *I* into or from *Q* has begun but has not completed when the IA operation is performed.

The conditions causing an IA operation to fail deserve further explanation. As stated above, if *I* is not on *Q*, i.e., if it is on some queue other than *Q* or not queued at all, the attempt to access it fails. If an IA operation is performed on *I* when it is being inserted into *Q*, we could elect to have the operation fail, as *I* is not yet a member of *Q*, or succeed, by delaying the operation until the insertion completed. Later, in section 2.1.2.1, we will give implementations of interior access that support these different outcomes. Even when *I* has been inserted into the correct queue, there is no guarantee that the operation will succeed. Suppose *I* is in cell *c* of *Q* when an IA operation attempts to access it. Before the IA operation obtains exclusive access to *c*, *I* may be taken off of the queue by a removal or deletion (if *c* is at the front of *Q*). It may then remain unqueued or at some later time be inserted into another queue *Q* (*Q*≠Q) or into the queue it was last on, *Q*. In this last instance, *I* can be inserted back into cell *c* or some other cell *d*. Thus, when the IA operation accesses *c*, *I* may be

(1) unqueued,

(2) on a different queue,

(3) on the original queue but in a different cell (*d*) from the one being accessed, or

(4) on the original queue and in the original cell (*c*).

Clearly, the IA operation must fail in the first two cases. For (3), the IA operation could fail (since *I* has obviously been dequeued for some period of time) or be retried (since *I* is still on the specified queue) by releasing cell *c* and accessing cell *d*. In the fourth case, the operation succeeds since *I* is where it was expected to be.[8] At the end of this section, we will present two versions of the IA procedure, one fails on the third case above and the other succeeds.

The site of an interior operation is defined by its first parameter *I*. As indicated in Figure 2-

[8] In order to fail on (4), additional information indicating that *I* had moved off and then back onto the queue would have to be available to the operation.

1, I is assumed to contain a cell location field, *Cell_Loc*, that defines its IA site[9] (i.e., if I is in the ith cell of Q then $I\uparrow.Cell\_Loc$ points to $Q.Cell[i]$). The insertion adding I to the queue sets this field to the insertion site and when I is dequeued (by a deletion or removal), the field is set to nil. Thus, the IA operation examines *Cell_Loc* to determine whether I is queued.

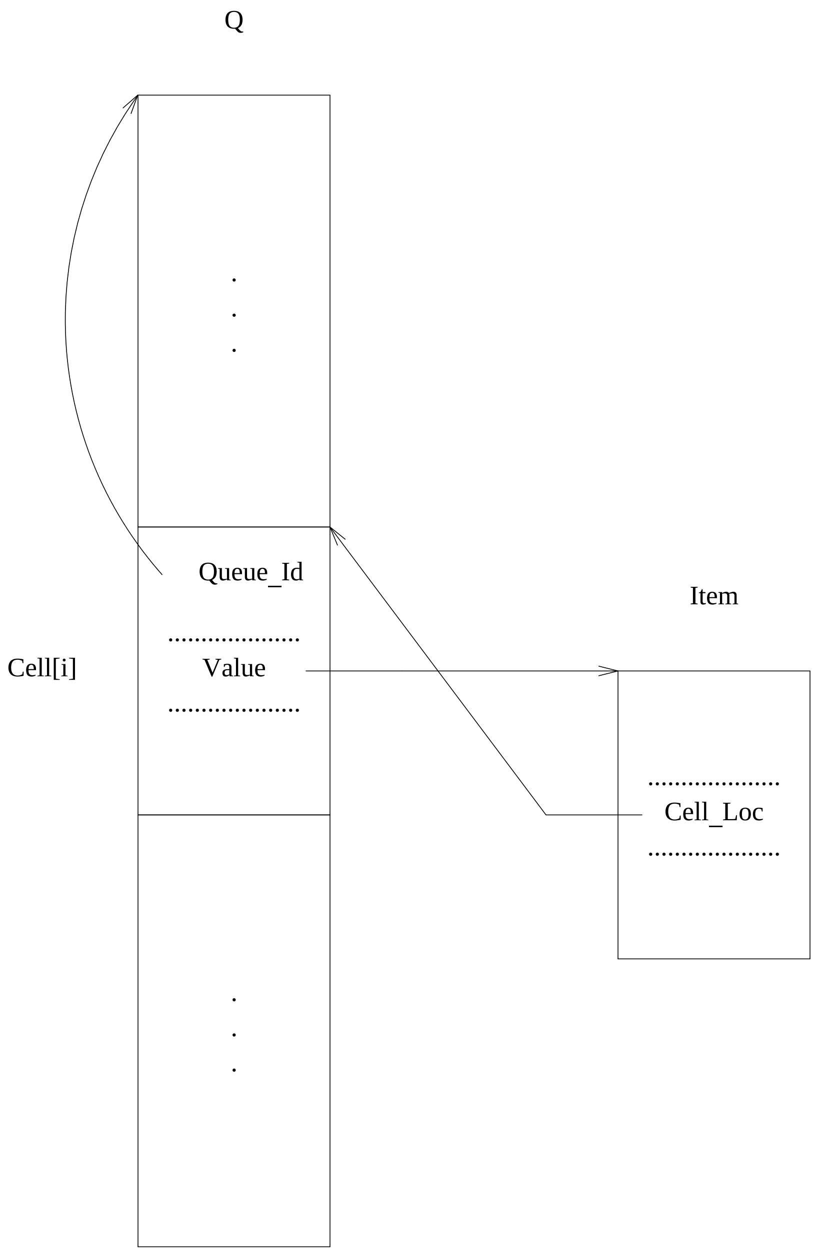
An item's cell location field only reveals that the item is queued by citing a specific cell within *some* queue; it does not identify to which queue the cell, and thereby the item, belongs. To remedy this, each queue cell has a queue identifier *Queue_Id* that names the cell's queue. The IA operation verifies that I belongs to Q by comparing the cell's queue identifier to Q, i.e. if I is in cell c ($I\uparrow.Cell\_Loc = c$), then I is a member of queue Q if c's queue identifier points to Q ($c\uparrow.Queue\_Id$ = **address**(Q)). The relationship between items, cells and queues is depicted in Figure 2-1.

I's cell is accessed through a critical code section to serialize potential deletions (since I may be at the front of the queue) as well as other IA operations acting on I. The cell protocols given for insertion and deletion (section 2.1.1.2) will be extended to coordinate all three operations.

As noted in case (3) above, a secondary problem related to referencing I's cell c is that between the time an IA operation checks c's queue identifier and executes c's cell entry protocol, another deletion or removal can access the cell and dequeue I from queue Q (modifying I's cell location along the way). Therefore, two versions of the IA procedure will be given, differing in the treatment of this case. The first one succeeds if, after acquiring exclusive access to cell c, I's location within Q has not changed (i.e., $I\uparrow.Cell\_Loc = c$). Otherwise, it fails. The second version succeeds if I is in Q, regardless of any cell migration it may have performed during the execution of the interior access operation, i.e., the operation cycles until I is accessed on Q or it is determined that I is no longer a member of Q. Both versions fail if I either is no longer queued ($I\uparrow.Cell\_Loc$ = nil) or resides on a different queue ($c\uparrow.Queue\_Id \neq$ **address**(Q)).

A modified queue structure supporting interior access and the two versions of the IA procedure are given below. The definitions of the cell entry/exit routines, CellEntryProtocol and CellExitProtocol, cited by the IA procedures will be presented in the next section.

[9] The alternative to having this field is to search the queue for the item.



Representation of an item inserted into Q.Cell[i].

Figure 2-1.

```
Seq_Queue = record   Sequentially Allocated Queue Supporting IA
                  Qlower, Qupper, Qsize, Qfront, Qrear : integer;
                  Cell : array[0..Qsize–1] of CellElement
              end;

CellElement = record
                   Queue_Id : Queue_Pointer;
                   Value      : Object_Pointer
               end;

Queue_Pointer = ↑Seq_Queue;

CellElement_Pointer = ↑CellElement;

Object = record
             Cell_Loc : CellElement_Pointer;
             Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
    Qlower, Qupper, Qfront, Qrear = 0;
    for 0≤j<Qsize,
      Cell[j].Queue_Id    = address of the queue;

function Seq_IA_V1 (Item : Object_Pointer;
                        Q     : Seq_Queue;
                        ϕ     : function);
var c : CellElement_Pointer;
    with Q do
    Seq_IA_V1 := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then        verify Item on Q
        CellEntryProtocol;
        if c = Item↑.Cell_Loc then                     verify Item still in cell c
            ϕ(Item);                                   access the item
            Seq_IA_V1 := true
        end if;
        CellExitProtocol
    end if
    end with
end function Seq_IA_V1;
```

```
function Seq_IA_V2 (Item : Object_Pointer;
                    Q    : Seq_Queue;
                    ϕ    : function);
var c    : CellElement_Pointer;
    done : boolean;

    with Q do
    Seq_IA_V2 := false;
    done := false;
    repeat
        c := Item↑.Cell_Loc;
        if c ≠ nil and c↑.Queue_Id = address(Q) then          verify Item on Q
            CellEntryProtocol;
            if c = Item↑.Cell_Loc then                          verify Item still in cell c
                ϕ(Item);                                        access the item;
                Seq_IA_V2 := true;
                done := true
            end if;
            CellExitProtocol
        else
            done := true
        end if
    until done
    end with
end function Seq_IA_V2;
```

2.1.2.1. Cell Protocols for Interior Access

Insertions and deletions execute the protocols described in section 2.1.1.2 to coordinate their cell accesses. These protocols do not exclude interior access operations. The basic protocol format for insertions and deletions to coordinate with one another and interior access operations is:

cell entry protocol for insertion or deletion (as defined in 2.1.1.2)
cell entry protocol for interior access
‘‘perform cell insertion or deletion’’
cell exit protocol for interior access
cell exit protocol for insertion or deletion

Notice that insertions and deletions act as interior access operations to gain final entry to a cell.

The inner and outer protocols may of course be both fair, both unfair, or combinations of the two (e.g., the outer protocol may be fair while the inner protocol is not). However, if insertions and deletions use unfair (i.e., version two) P/V protocols, as described in section 2.1.1.2, these queue operations can access a cell without having to execute the inner protocol cited above.

We give three different weak P/V protocol solutions for coordinating insertions, deletions, and IA operations. The first is a simple extension of the cell protocols already defined for insertion and deletion. The second optimizes the insertion and deletion protocols while raising the overhead for interior access. The third reduces the interior access overhead of the second solution. The solutions differ with respect to the behavior of *concurrent* insertion and interior access. Specifically, an IA operation performed on an item whose insertion has not completed fails when the third solution is used and succeeds when either the first or second solution is used.

2.1.2.1.1. Solution One

The first solution is an extension of the insertion and deletion protocols given in section 2.1.1.2. To permit interior access, a third semaphore *IA_Sem* is associated with each cell to serialize multiple IA operations (including insertions and deletions) on a given cell.

The critical code section and protocols for an insertion on a cell *i* are:

```
Weak_P_V2(Cell[i].Insert_Sem);
Weak_P_V2(Cell[i].IA_Sem);
    Cell[i].Value := I;
    I↑.Cell_Loc := address(Cell[i]);  mark I as queued
Weak_V_V2(Cell[i].IA_Sem);
Weak_V_V2(Cell[i].Delete_Sem);
```

The critical code section and protocols for a deletion on a cell *d* are:

```
Weak_P_V2(Cell[d].Delete_Sem);
Weak_P_V2(Cell[d].IA_Sem);
    I := Cell[d].Value;
    I↑.Cell_Loc := nil;             mark I as dequeued
Weak_V_V2(Cell[d].IA_Sem);
Weak_V_V2(Cell[d].Insert_Sem);
```

Finally, the interior access protocols[10] for a cell *c* consist of just P and V operations on *c*'s IA semaphore:

```
Weak_P_V2(c↑.IA_Sem);
    if c = I↑.Cell_Loc then  verify I still in cell c
        φ(I);                access I
        Seq_IA := true
    end if;
Weak_V_V2(c↑.IA_Sem);
```

[10] The protocols are given in the context of version one of the IA procedure.

Notice that if the entry protocols for insertion and deletion were reversed (with the interior access semaphore being closed before the insertion or deletion semaphore) then deadlock would be possible. For example, a deletion could close cell *d*'s *IA_Sem* before an insertion was performed on *d*. The deletion would then busy-wait forever on *Delete_Sem* and the insertion would do the same on *IA_Sem*.

2.1.2.1.2. Solution Two

We may simplify the insertion and deletion protocols just presented by making only the IA protocol responsible for guaranteeing mutual exclusion between IA operations and insertions and deletions. The protocols for insertion and deletion need only ensure their operations alternate. Thus, the original protocols defined in section 2.1.1.2 are sufficient for insertion and deletion. The second solution is based on the writer priority readers/writers algorithm, where IA operations are the writers and insertions and deletions are the readers.

An IA operation, after closing a cell's *IA_Sem*, decrements the cell's insertion and deletion semaphores by one through a naked fetch-and-add.[11] Any currently active insertion or deletion is unaffected, but new insertions or deletions are prevented from starting (since their semaphores are already closed). The IA operation then busy-waits until all insertion and deletion activity has stopped (at which time one semaphore will have a value of zero and the other a value of −1).

The cell protocols for insertion and deletion are included below for completeness.

The insertion protocols and critical code section for cell *i* are:

Weak_P_V2(*Cell*[*i*].*Insert_Sem*);
    Cell[*i*].*Value* := *I*;
    I↑.*Cell_Loc* := **address**(*Cell*[*i*]); *mark I as queued*
Weak_V_V2(*Cell*[*i*].*Delete_Sem*);

The critical code section and protocols for a deletion on cell *d* are:

[11] We note that in a strict sense this is a violation of the semaphore definition given in chapter 1, which requires that semaphores be accessed only by P and V. But, if the IA operation closed the insertion and deletion semaphores of a cell via P operations, it would always enter an infinite loop since one of these semaphores is always closed. A subsequent insertion or deletion would then be unable to access the cell.

```
Weak_P_V2(Cell[d].Delete_Sem);
    I := Cell[d].Value;
    I↑.Cell_Loc := nil;                mark I as dequeued
Weak_V_V2(Cell[d].Insert_Sem);
```

The simplification of the insertion and deletion protocols, as compared with solution one, comes at the price of complicating the IA entry and exit protocols. These are given below for an IA operation on cell *c*.

```
Weak_P_V2(c↑.IA_Sem);
F&A(c↑.Insert_Sem, −1);
F&A(c↑.Delete_Sem, −1);
repeat until (c↑.Insert_Sem = 0 and c↑.Delete_Sem = −1) or
             (c↑.Insert_Sem = −1 and c↑.Delete_Sem = 0);
    if c = I↑.Cell_Loc then
        ϕ(I);       access I
        Seq_IA := true
    end if;
F&A(c↑.Delete_Sem, 1);
F&A(c↑.Insert_Sem, 1);
Weak_V_V2(c↑.IA_Sem);
```

2.1.2.1.3. Solution Three

The first two solutions may appear overly conservative; they assume that an interior access operation and an insertion may attempt to access the same cell concurrently. To guard against this, the first solution has both operations close a semaphore (*IA_Sem*) associated with the cell, and the second has the IA operation decrement the insertion cell semaphore (*Insert_Sem*). We now present an IA protocol that eliminates explicit mutual exclusion operations between insertions and IA operations and does not require the latter to busy-wait on the insertion and deletion semaphores.

The protocol is based on the following observation: An IA operation can only succeed if the cell *c* to be accessed is full, i.e., *c*'s deletion semaphore must be open. When this condition is true, insertions are already blocked−*c*'s insertion semaphore is closed. Therefore, if an IA operation closes *c*'s deletion semaphore, the cell contains an item (it is full) and insertions do not have to be explicitly excluded. On the other hand, if the IA operation finds that the deletion semaphore is already closed, then the cell is empty (a deletion has already occurred) or being emptied (by an active deletion) or being filled (by an insertion).[12] Regardless of which case holds, the requested item has been taken off of the queue and the operation fails.

Therefore, the IA protocol decrements the deletion semaphore by −1 through a naked fetch-and-add. If this closes the semaphore, the IA operation has exclusive access to the cell. If the semaphore was already closed, the operation fails. When done (or if the semaphore was already closed), the IA operation increments the deletion semaphore by one. This is summarized below.

```
Weak_P_V2(c↑.IA_Sem);
if F&A(c↑.Delete_Sem, −1) = 1 then   cell is full
    if c = I↑.Cell_Loc then
        φ(I);                        access I
        Seq_IA := true
    end if
end if;
F&A(c↑.Delete_Sem, 1);
Weak_V_V2(c↑.IA_Sem)
```

It is possible for an IA operation to access an item *I* after an insertion has marked *I* as queued but before the insertion has opened the cell deletion semaphore, i.e., the insertion has not yet completed. When this occurs, the IA operation fails as the IA protocol reports that *I* is not queued (because the deletion semaphore is closed). Given the first or second protocol solution and this same scenario, an IA operation succeeds since it busy-waits until the insertion completes.

2.1.2.2. Putting It All Together

In this section, fully defined insertion, deletion, and IA operations are given. They use the second protocol solution of section 2.1.2.1.

```
Seq_Queue = record   Sequentially Allocated Queue Supporting IA
                Qlower, Qupper, Qsize, Qfront, Qrear : integer;
                Cell : array[0..Qsize–1] of CellElement
            end;
```

[12] No other IA operation may access the cell since IA operations are mutually exclusive (prior to decrementing a cell's *Delete_Sem*, an IA operation closes the cell's *IA_Sem*).

```
CellElement = record
                    Queue_Id     : Queue_Pointer;
                    Value        : Object_Pointer;
                    Insert_Sem,
                    Delete_Sem,
                    IA_Sem       : Weak_Semaphore
                end;

Queue_Pointer = ↑Seq_Queue;

CellElement_Pointer = ↑CellElement;

Object = record
              Cell_Loc : CellElement_Pointer;
              Application_Defined_Attributes
          end;

Object_Pointer = ↑Object;

Initially:
    Qlower, Qupper, Qfront, Qrear = 0;
    for 0≤j<Qsize,
      Cell[j].Queue_Id      = address of the queue;
      Cell[j].Insert_Sem    = 1;
      Cell[j].IA_Sem        = 1;
      Cell[j].Delete_Sem    = 0;

function Seq_QInsert (Item : Object_Pointer;
                      Q    : Seq_Queue);
var i : integer;
    with Q do
    if TIR(Qupper, 1, Qsize) then
        i := F&A(Qrear, 1) mod Qsize;
        if i = Qsize−1 then F&A(Qrear, −Qsize) end if;
        Weak_P(Cell[i].Insert_Sem);
            Cell[i].Value := Item;
            Item↑.Cell_Loc := address(Cell[i]);  mark Item as queued
        Weak_V(Cell[i].Delete_Sem);
        F&A(Qlower, 1);
        return true
    else
        return false
    end if
    end with
end function Seq_QInsert;
```

```
function Seq_QDelete (Item : Object_Pointer;
                      Q    : Seq_Queue);
var d : integer;
   with Q do
   if TDR(Qlower, 1) then
      d := F&A(Qfront, 1) mod Qsize;
      if d = Qsize−1 then F&A(Qfront, −Qsize) end if;
      Weak_P(Cell[d].Delete_Sem);
         Item := Cell[d].Value;
         Item↑.Cell_Loc := nil;   mark Item as dequeued
      Weak_V(Cell[d].Insert_Sem);
      F&A(Qupper, −1);
      return true
   else
      return false
   end if
   end with
end function Seq_QDelete;


function Seq_IA_V1 (Item : Object_Pointer;
                    Q    : Seq_Queue;
                    ϕ    : function);
var c : CellElement_Pointer;
   with Q do
   Seq_IA_V1 := false;
   c := Item↑.Cell_Loc;
   if c ≠ nil and c↑.Queue_Id = address(Q) then  verify Item on Q
      Weak_P(c↑.IA_Sem);
      F&A(c↑.Insert_Sem, −1);
      F&A(c↑.Delete_Sem, −1);
      repeat until (c↑.Insert_Sem = 0 and c↑.Delete_Sem = −1) or
                   (c↑.Insert_Sem = −1 and c↑.Delete_Sem = 0);
         if c = Item↑.Cell_Loc then                verify Item still in cell c
            ϕ(Item);                               access the item
            Seq_IA_V1 := true
         end if;
      F&A(c↑.Delete_Sem, 1);
      F&A(c↑.Insert_Sem, 1);
      Weak_V(c↑.IA_Sem)
   end if
   end with
end function Seq_IA_V1;
```

2.2. Linked Allocated Queues

Rudolph [82] observes that each cell of a sequentially allocated queue may be replaced with a linked list to yield a concurrently accessible queue of unlimited size–a ‘‘linked’’ queue.

Insertion and deletion on such a structure are similar to the corresponding operations on sequentially allocated queues just described. The major differences between the two sets of operations are the replacement of cell operations by linked list operations and the substitution of list protocols for cell protocols. Rudolph's description of linked insertion and deletion, reviewed in section 2.2.1, omitted the definitions of the linked list protocols. Therefore, we complete the implementations of these operations by giving local and global list protocols in sections 2.2.2.1 and 2.2.2.2. Interior access to linked queues is covered in section 2.2.3. Type one and two IA operations are dealt with separately, in sections 2.2.3.1 and 2.2.3.2., respectively.

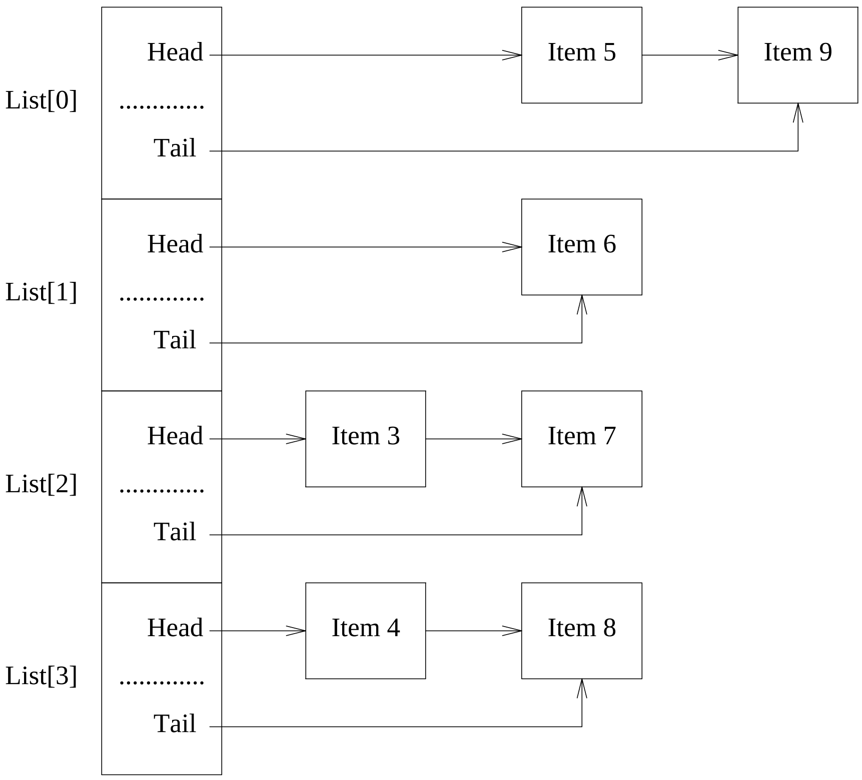
2.2.1. A Review of Linked Insertion and Deletion

A linked queue is implemented using *Qsize* conventional, i.e. serially accessible, linked lists. Each list has a distinguished node called a list header. This header contains two components, *Head* and *Tail*, pointing to the first and last items on the list, respectively. The list headers themselves are stored in a circular array *List*[0..*Qsize*–1], which is accessed like the sequentially allocated queues of section 2.1.

The queue itself has two shared counters, *Qfront* and *Qrear*, for the calculation of deletion and insertion sites. Since the queue can never become full (i.e., insertions always succeed), no upper bound count of the number of queued items (*Qupper*) is maintained. However, a lower bound count *Qlower* is still kept since underflow is possible (i.e., deletions can fail).

As many as *Qsize* insertions and deletions may be concurrently executed on a queue, one per list. These operations closely resemble their counterparts for sequentially allocated queues, i.e., they calculate their list sites (in the same way that cell sites are determined in section 2.1.1) and exclusively access the list at that site. Now, however, instead of performing cell assignments, an insertion at site *i* appends its item to the tail of *List*[*i*] and a deletion at site *d* removes the head item from *List*[*d*]. Items are placed onto the lists in a round robin fashion: The first item is inserted into *List*[0], the second into *List*[1], etc. After *Qsize* insertions, the next item is appended to *List*[0] and so on. This pattern of assignment keeps the lists balanced: during periods of inactivity the number of items on any two lists will differ by at most one. Also, note that an item and its immediate successor on a list are actually separated by $Qsize - 1$ logical queue positions.

The result of applying 9 insertions and 2 deletions to a queue of *Qsize* 4 is illustrated in Figure 2-2. The next insertion would be to list 1, the next deletion from list 2. Notice that many insertions can be performed on a given list before a deletion is ever done and similarly, consecutive deletions from a list can occur without any intervening insertions. Sites (i.e., lists) do not have to be alternately accessed by insertions and deletions.



Basic queue structure after 9 insertions and 2 deletions.

Figure 2-2.

2.2.2. List Protocols for Insertion and Deletion

From the cell contention scenarios described in section 2.1.1.1, it is clear that multiple operations may simultaneously contend for the same site. If a site is simply a cell, insertions and deletions have to alternate and multiple insertions and deletions are serialized. If a site is a list, alternation is not required but concurrent operations must still be serialized. Therefore, the list protocols are clearly not identical to the cell protocols. Since Rudolph [82] does not define list protocols, we present two implementations for insertion and deletion. These implementations use local and global list protocols similar to the cell protocols described in section 2.1.1.2.

2.2.2.1. A Local List Protocol

The local list protocol below, like its corresponding cell protocol, does not impose a serial ordering upon insertions and deletions. Thus, the *i*th deletion may not necessarily remove the item inserted by the *i*th insertion.[13]

Associated with each list is a binary semaphore *Access_Sem* that is closed by a queue operation before the list is updated. Furthermore, a deletion must first determine that the list is not empty. To do so the deletion performs a P operation on a counting semaphore *Item_Cnt* associated with the list. When there are no active insertions or deletions, the value of *Item_Cnt* equals the number of items in the list. By executing a P on *Item_Cnt*, a deletion effectively reserves an item for itself before acquiring exclusive access to the list. Thus, a list containing more than one item may have multiple deletions contending for it. An insertion increments a list's *Item_Cnt* (through a V) after appending an item to the list.

Qfront and *Qrear* are incremented by deletion and insertion, respectively, to calculate deletion and insertion sites. As there are only *Qsize* sites, *Qfront* and *Qrear* are maintained modulo *Qsize* (by using the procedure of section 2.1.1) and, therefore, prevented from overflowing. The maximum number of active queue operations equals *B*, the maximum number of active tasks.[14] (For example, since the queue cannot overflow, up to *B* insertion tasks can be active.) The maximum value that *Qfront* or *Qrear* can attain is $B * Qsize$: When a task is assigned to site $Qsize - 1$ (sites are numbered from 0 to $Qsize - 1$), it decrements the corresponding counter (*Qrear* or *Qfront*) by *Qsize*. Between the time the task detects that it has been assigned site $Qsize - 1$ and the decrement is performed, each of the remaining $B-1$ tasks can access the queue up to a maximum of *Qsize* times. (Eventually, each task would have to adjust the counter prior to accessing site $Qsize - 1$.) Hence, the maximum value either counter reaches is $B * Qsize$. In contrast, recall that the maximum number of active operations on a sequentially allocated queue equals *Qsize*, and the maximum value that *Qrear* (*Qfront*) can attain is $Qrear^2$ ($Qfront^2$).

[13] We may order deletions (or insertions) by the fetch-and-adds used to calculate their sites.

[14] One may choose $B = M * P$ where *M* bounds the multiprogramming level and *P* is the number of processors.

The linked queue data structures and algorithms are given below. Each queued item has a *Next* component to link the item to its successor on the list. Following Knuth [68], the tail of an empty list points to the list header, and a reference to the *Next* component of the list header is interpreted as a reference to the *Head* component.

```
Linked_Queue = record  Linked List Allocated Queue
                     Qlower, Qsize, Qfront, Qrear : integer;
                     List : array[0..Qsize–1] of ListElement
                 end;

ListElement = record
                   Head        : Object_Pointer;
                   Tail        : Pointer to ListElement or Object;15
                   Access_Sem,
                   Item_Cnt    : Weak_Semaphore
               end;

ListElement_Pointer = ↑ListElement;

Object = record
             Next : Object_Pointer;
             Application_Dependent_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear = 0;
   for all 0≤j<Qsize,
     List[j].Tail          = address(List[j]);
     List[j].Access_Sem = 1;
     List[j].Item_Cnt     = 0;
```

[15] The notation ‘‘*X* : Pointer to *A* or *B*’’ means that *X* can point to either a data structure of type *A* or *B*.

```
procedure Linked_QInsert (Item : Object_Pointer;
                          Q    : Linked_Queue);
var i : integer;
    with Q do
    i := F&A(Qrear, 1) mod Qsize;
    if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
    Item↑.Next := nil;                 mark end of list
    Weak_P(List[i].Access_Sem);
        List[i].Tail↑.Next := Item;    append item to end of list
        List[i].Tail := Item;
    Weak_V(List[i].Access_Sem);
    Weak_V(List[i].Item_Cnt);
    F&A(Qlower, 1)
    end with
end procedure Linked_QInsert;

function Linked_QDelete (Item : Object_Pointer;
                         Q    : Linked_Queue);
var d : integer;
    with Q do
    if TDR(Qlower, 1) then
        d := F&A(Qfront, 1) mod Qsize;
        if d = Qsize–1 then F&A(Qfront, –Qsize) end if;
        Weak_P(List[d].Item_Cnt);
        Weak_P(List[d].Access_Sem);
            Item := List[d].Head;                  take first item off list
            List[d].Head := Item↑.Next;            new Head of list
            if List[d].Head = nil then             list empty
                List[d].Tail := address(List[d])   reinitialize Tail
            end if;
        Weak_V(List[d].Access_Sem);
        return true
    else
        return false
    end if
    end with
end function Linked_QDelete;
```

As in the first cell protocol solution (section 2.1.2.1.1), if the order of the P operations in the deletion algorithm were reversed, deadlock would be possible.

2.2.2.2. A Global List Protocol

In the previous implementation, the item added to the queue by the *i*th insertion may not be removed by the *i*th deletion. We now present a fair list protocol for which the *i*th deletion is assured of receiving the item inserted by the *i*th insertion. This implementation is similar to the first global cell protocol described in section 2.1.1.2.

Insertions and deletions are assigned ticket numbers from two different sequences[16] as they arrive at the queue (i.e., when they calculate their sites). Associated with each list are two ticket numbers, *I#* and *D#*, defining which insertion and deletion, respectively, are next to access the list. Note that the sequencing of operations by ticket number guarantees that the item added to the queue by the jth insertion is removed by the jth deletion.

In this implementation, *Qrear* and *Qfront* are used to assign ticket numbers to insertions and deletions to order their site accesses. If we can place a limit, L, on the number of unique ticket numbers required, *Qrear* and *Qfront* can be bounded. Since a maximum of B insertions or deletions can be active at a single site, where B is the maximum number of active tasks, we require simply that $\lfloor L/Qsize \rfloor \geq B$, i.e., that each of the B operations must have a unique number. Therefore, we may let $L = B * Qsize$ and maintain the ticket numbers modulo L.[17] Then, *Qfront* and *Qrear* may be prevented from assuming arbitrarily large values.

Specifically, when a task receives a ticket number of $L-1$ (tickets are numbered from 0 to $L-1$), it decrements the counter by L. Between the time the task detects its ticket number is $L-1$ and the decrement is performed, each of the remaining $B-1$ tasks can access the queue up to a maximum of *Qsize* times. (Eventually, each task would have to adjust the counter prior to accessing *List*[*Qsize*–1].) Thus, the maximum value a counter attains is

$$L + (B-1) * Qsize = (2B-1) * Qsize.$$

To properly bound *Qrear* (and similarly *Qfront* for deletions), insertion ticket numbers can be calculated as shown below:

iticket# := F&A(*Qrear*, 1) **mod** *L*;
if *iticket#* = *L*–1 **then** F&A(*Qrear*, –*L*) **end if**;

To initialize the queue for this method, each list is designated as empty (*Item_Cnt* = 0) and accessible (*Access_Sem* = 1). For each list *j*, *I#* and *D#* are initialized to *j*.

---

[16] Two sequences are used so as not to force the alternation of insertions and deletions as in the global cell protocol implemented by Gottlieb et al. [83b].

[17] The list header ticket numbers for the next insertion and deletion, *I#* and *D#*, are also maintained modulo *L*.

```
Linked_Queue = record  Linked List Allocated Queue
                   Qlower, Qsize, Qfront, Qrear : integer;
                   List : array[0..Qsize–1] of ListElement
               end;

ListElement = record
                 Head        : Object_Pointer;
                 Tail        : Pointer to ListElement or Object;
                 I#,
                 D#          : integer;
                 Access_Sem,
                 Item_Cnt    : Weak_Semaphore
              end;

ListElement_Pointer = ↑ListElement;

Object = record
            Next : Object_Pointer;
            Application_Dependent_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear = 0;
   for all 0≤j<Qsize,
    List[j].Tail        = address(List[j]);
    List[j].I#          = j;
    List[j].D#          = j;
    List[j].Access_Sem = 1;
    List[j].Item_Cnt    = 0;
```

```
procedure Linked_QInsert (Item : Object_Pointer;
                          Q    : Linked_Queue);
var i, iticket# : integer;
    with Q do
    Item↑.Next := nil;
    iticket# := F&A(Qrear, 1) mod L;        get a number
    if iticket# = L−1 then F&A(Qrear, −L) end if;
    i := iticket# mod Qsize;                list site
    repeat until List[i].I# = iticket#;     wait till number called
    Weak_P(List[i].Access_Sem);
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
    Weak_V(List[i].Access_Sem);
    Weak_V(List[i].Item_Cnt);
    List[i].I# := (iticket# + Qsize) mod L; next ticket number
    F&A(Qlower, 1)
    end with
end procedure Linked_QInsert;
```

```
function Linked_QDelete (Item : Object_Pointer;
                         Q    : Linked_Queue);
var d, dticket# : integer;
    with Q do
    if TDR(Qlower, 1) then
        dticket# := F&A(Qfront, 1) mod L;
        if dticket# = L−1 then F&A(Qfront, −L) end if;
        d := dticket# mod Qsize;
        repeat until List[d].D# = dticket#;
        Weak_P(List[d].Item_Cnt);
        Weak_P(List[d].Access_Sem);
            Item := List[d].Head;
            List[d].Head := Item↑.Next;
            if List[d].Head = nil then
                List[d].Tail := address(List[d])
            end if;
        Weak_V(List[d].Access_Sem);
        List[d].D# = (dticket# + Qsize) mod L;
        return true
    else
        return false
  end if
  end with
end function Linked_QDelete;
```

We claim that the above implementations of insertion and deletion are starvation-free even if version two of the weak P/V operations is used to access the lists. (Recall that this version permits starvation.) The ticket numbering method ensures that at most one insertion and one deletion will concurrently vie for a list. A deletion may be prevented from acquiring a list for an extended period of time by a string of consecutive insertions but it will starve only if this string is infinitely long. We assume this is not possible due to physical limitations (e.g., memory is finite). The opposite situation, where a continuous stream of deletions monopolizes a list, cannot occur since each list contains a finite number of items. The remaining weak P operation is performed by deletions on a list's item count. Only one deletion at a time may execute this operation (due to ticket number restrictions) thereby eliminating any possibility of starvation at this point.

2.2.3. Interior Access

In this section we examine how to access an interior item of a linked queue.[18] We begin with the implementation of type one IA operations. Recall that these operations read and/or modify an item but leave the queue structure intact; type two operations have write access to the entire contents of an item and its queue (see section 2.1.2 for further details). We then implement a specific type two operation, the removal of an interior item, which has a significant impact upon both the queue structure and the insertion and deletion algorithms. Both types of IA operations are implemented without reducing the concurrent accessibility of the queue; it remains at *Qsize* operations, one per list.

The linked queue IA operations, like linked insertion and deletion, operate on lists. In the following, local list protocols are specified; as noted at the end of the next section, starvation-free algorithms may be implemented if these protocols are fair or if global protocols are used in their place.

2.2.3.1. Type One

Recall that an interior access operation has the form: IA(I, Q, ϕ). In the present case, Q is a linked queue. A type one IA operation

(1) verifies that item I is on the correct queue Q;

(2) requests exclusive access to L, the item's queue list;

[18] Our algorithms will be patterned after the first version of interior access given in section 2.1.2. The second version can also be easily supported.

(3) once permission is granted, verifies that *I* is still on *L*;

(4) performs $\phi(I)$;

(5) releases *L*.

This sequence is essentially identical to type one and two interior access operations on sequentially allocated queues (see section 2.1.2).

To accommodate linked queues, each item contains a list location field *List_Loc* that points to the item's list header when it is queued and is nil otherwise.[19] Assignments to this field are done within the critical code sections of linked insertion and deletion.

The IA procedure given below uses a local list protocol to access a list. For completeness, insertion and deletion algorithms based on this protocol are also listed. Note that the same IA procedure may be used with insertion and deletion algorithms that access lists through global protocols.

```
Linked_Queue = record  Linked Allocated Queue Supporting IA
                    Qlower, Qsize, Qfront, Qrear : integer;
                    List : array[0..Qsize–1] of ListElement
               end;

ListElement = record
                   Queue_Id      : Queue_Pointer;
                   Head          : Object_Pointer;
                   Tail          : Pointer to ListElement or Object;
                   Access_Sem,
                   Item_Cnt      : Weak_Semaphore
              end;

Queue_Pointer = ↑Linked_Queue;

ListElement_Pointer = ↑ListElement;

Object = record
              List_Loc     : ListElement_Pointer;
              Next         : Object_Pointer;
              Application_Dependent_Attributes
         end;
```

[19] This serves the same function as *Cell_Loc* in the sequential implementation.

```
Object_Pointer = ↑Object;

Initially:
    Qlower, Qfront, Qrear = 0;
    for all 0≤j<Qsize,
      List[j].Queue_Id     = address of the queue;
      List[j].Tail         = address(List[j]);
      List[j].Access_Sem = 1;
      List[j].Item_Cnt     = 0;

procedure Linked_QInsert (Item : Object_Pointer;
                          Q    : Linked_Queue);
var i : integer;
    with Q do
    i := F&A(Qrear, 1) mod Qsize;
    if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
    Item↑.Next := nil;
    Weak_P(List[i].Access_Sem);
        Item↑.List_Loc := address(List[i]);   mark as queued
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
    Weak_V(List[i].Access_Sem);
    Weak_V(List[i].Item_Cnt);
    F&A(Qlower, 1)
    end with
end procedure Linked_QInsert;
```

```
function Linked_QDelete (Item : Object_Pointer;
                         Q    : Linked_Queue);
var d : integer;
   with Q do
   if TDR(Qlower, 1) then
      d  := F&A(Qfront, 1) mod Qsize;
      if d = Qsize−1 then F&A(Qfront, −Qsize) end if;
      Weak_P(List[d].Item_Cnt);
      Weak_P(List[d].Access_Sem);
         Item := List[d].Head;
         List[d].Head := Item↑.Next;
         Item↑.List_Loc := nil;      mark as dequeued
         if List[d].Head = nil then
            List[d].Head↑.Tail := address(List[d])
         end if;
      Weak_V(List[d].Access_Sem);
      return true
   else
      return false
   end if
   end with
end function Linked_QDelete;
```

```
function  Linked_IA_TypeI (Item : Object_Pointer;
                           Q    : Linked_Queue;
                           ϕ    : function);
var h : ListElement_Pointer;
   with Q do
   Linked_IA_TypeI := false;
   h := Item↑.List_Loc;
   if h ≠ nil and  h↑.Queue_Id = address(Q) then
      Weak_P(h↑.Access_Sem);
         if h = Item↑.List_Loc then
            ϕ(Item);      access the item
            Linked_IA_TypeI := true
         end if;
      Weak_V(h↑.Access_Sem)
   end if
   end with
end function Linked_IA_TypeI;
```

If the second version of P/V is used in the above implementation, there exists the possibility that an insertion, deletion or IA operation may starve. For example, IA operations may repeatedly access a list, blocking out other operations. To prevent starvation, lists should be accessed using the first version of P/V, which is fair.

2.2.3.2. Type Two: Removal

Type two interior access operations can remove, add, or reposition items within a queue. Such manipulations may necessitate the modification of a queue's components (e.g., *Qfront* and *Qrear*) or the introduction of other mechanisms to maintain the correct functioning of the queue. We focus our attention on the implementation problems associated with one specific operation, the *removal* of an interior item, and their resolution.

The major additional problem faced in removing an interior item is that the FIFO property of a queue may be destroyed.[20] To illustrate this point, Figure 2-3 shows a queue after 9 insertions have been performed followed by the removal of items 3 and 5. Repeated calls to a naive deletion routine would deliver the remaining items in the following (incorrect) order: 1, 2, 6, 4, 8, 9, 7.

To remedy this situation we record the absence of removed items by associating a ''missing'' item count *#Missing* with each item (and list header). This count reflects the number of times an item's (or list header's) immediate list successor has been removed. In this manner, the holes which develop between items are recorded. Figure 2-4 displays the same queue as in Figure 2-3 but with the missing item counts included.

The removal of an item follows the five steps outlined in the previous section. The fourth step, the actual manipulation of the item, necessitates unchaining the item from its list, marking it as dequeued, and modifying the missing count of the item's predecessor. These three actions, stated in detail, are:

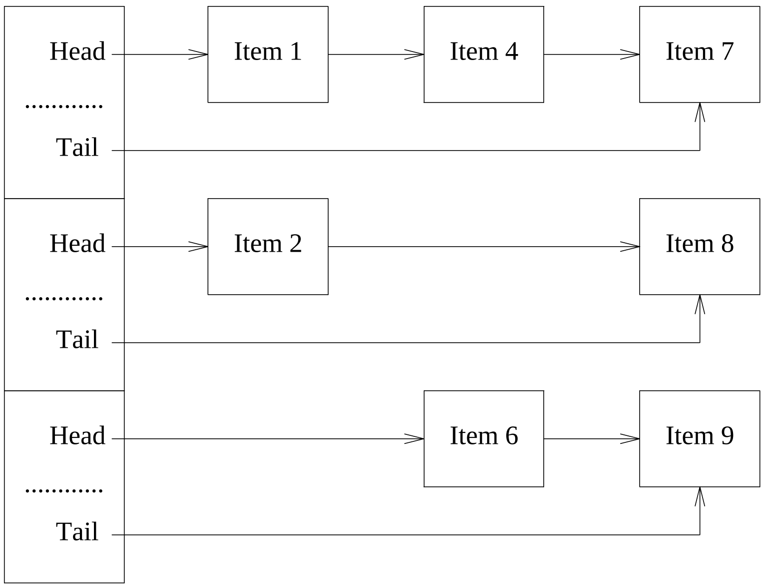
(1) Take the item off its list. This is facilitated by using doubly linked lists.

(2) Mark the item as dequeued (by setting its list location to nil).

(3) Add the item's missing count plus one (for the item itself) to its predecessor's missing count.

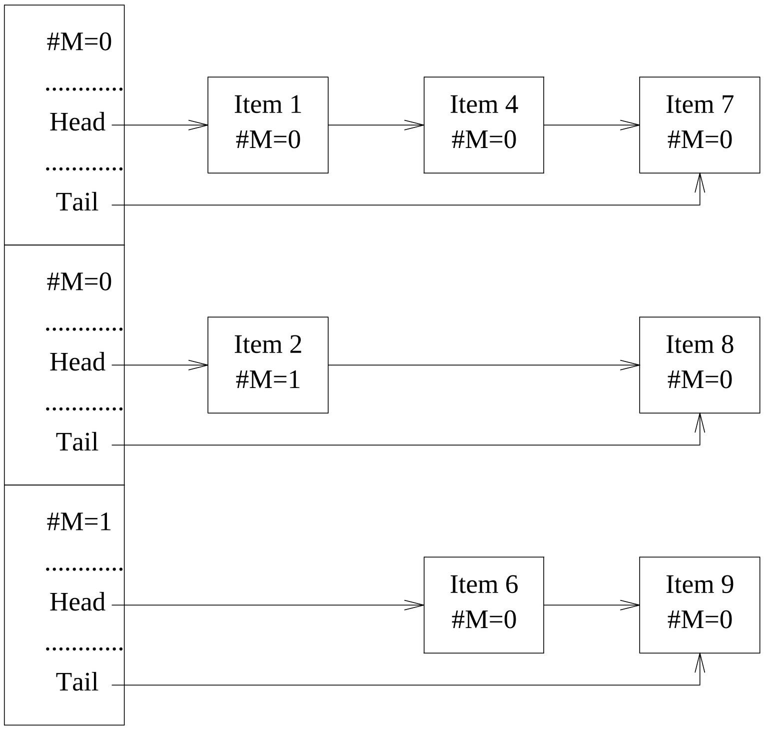
Linked deletion is modified in two ways to account for missing items. First, when an item is deleted, its missing item count is transferred to the item's list header. Absent items thus become

[20] This may be tolerable if the FIFO order is not required. However, the natural implementation of simply removing items would permit the constituent lists to have widely differing lengths. Thus, deletions may be (permanently) directed to empty lists and would then need to take some remedial action.



Basic queue structure after 9 insertions
and the removal of items 3 and 5.

Figure 2-3.



Basic queue structure after 9 insertions and the removal of items 3 and 5. #M denotes missing item counts.

Figure 2-4.

known to the delete routine once their immediate predecessor is taken from the queue. Second, a list header's missing count is examined to determine if its head item should be taken. If the count is zero, no missing elements precede the head item, which is then deleted. Otherwise, one or more removals have occurred and to preserve the FIFO ordering, the head item is not deleted. Instead, the list header's missing item count is decremented, reflecting that a ‘‘missing’’ item has been ‘‘deleted.’’ A nil item is returned to the caller of the delete routine. The appropriate response to receiving such a value is application dependent; one choice is to retry the deletion. The linked insertion algorithm must initialize an item's missing count to zero and its list header location to the list site address.

The algorithms for insertion, deletion and removal are given using local list protocols that are not fair. As noted in the previous section, starvation-free protocols may be also be implemented.

To simplify the queue operations, circular, doubly linked lists will be maintained. Each queued item has components *Next* and *Prev* to link the item to its successor and predecessor on the list. Following Knuth [68], the head and tail of an empty list point to the list header, and a reference to the *Next* or *Prev* component of the list header is interpreted as a reference to the *Head* or *Tail* component, respectively.

```
Linked_Queue = record  Linked Allocated Queue Supporting Removal
                   Qlower, Qsize, Qfront, Qrear : integer;
                   List : array[0..Qsize–1] of ListElement
               end;

ListElement = record
                   Queue_Id      : Queue_Pointer;
                   Head,
                   Tail          : Pointer to ListElement or Object;
                   #Missing      : integer;
                   Access_Sem,
                   Item_Cnt      : Weak_Semaphore
              end;

Queue_Pointer = ↑Linked_Queue;

ListElement_Pointer = ↑ListElement;

Object = record
              List_Loc : ListElement_Pointer;
              Next,
              Prev     : Pointer to ListElement or Object;
              #Missing : integer;
              Application_Dependent_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear = 0;
   for all 0≤j<Qsize,
     List[j].Queue_Id    = address of the queue;
     List[j].Head        = address(List[j]);
     List[j].Tail        = address(List[j]);
     List[j].#Missing    = 0;
     List[j].Access_Sem = 1;
     List[j].Item_Cnt    = 0;
```

```
procedure Linked_QInsert (Item : Object_Pointer;
                          Q    : Linked_Queue);
var i : integer;
    with Q do
    i := F&A(Qrear, 1) mod Qsize;
    if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
    Item↑.#Missing := 0;
    Item↑.Next := address(List[i]);           circular list
    Weak_P(List[i].Access_Sem);
        Item↑.List_Loc := address(List[i]);   mark as queued
        Item↑.Prev := List[i].Tail;
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
    Weak_V(List[i].Access_Sem);
    Weak_V(List[i].Item_Cnt);
    F&A(Qlower, 1)
    end with
end procedure Linked_QInsert;


function Linked_QDelete (Item : Object_Pointer;
                         Q    : Linked_Queue);
var d : integer;
    with Q do
    if TDR(Qlower, 1) then
        d := F&A(Qfront, 1) mod Qsize;
        if d = Qsize–1 then F&A(Qfront, –Qsize) end if;
        Weak_P(List[d].Item_Cnt);
        Weak_P(List[d].Access_Sem);
            if List[d].#Missing > 0 then  head list item removed
                List[d].#Missing := List[d].#Missing – 1;
                Item := nil
            else take first item off list
                Item := List[d].Head;
                List[d].#Missing := Item↑.#Missing;
                List[d].Head := Item↑.Next;
                List[d].Head↑.Prev := address(List[d]);
                Item↑.List_Loc := nil
            end if;
        Weak_V(List[d].Access_Sem);
        return true
    else
        return false
    end if
    end with
end function Linked_QDelete;
```

```
function Linked_QRemove (Item : Object_Pointer;
                         Q    : Linked_Queue);
var h : ListElement_Pointer;
    with Q do
    Linked_QRemove := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if Item↑.List_Loc = h then
                F&A(Item↑.Prev.#Missing, Item↑.#Missing + 1);
                Item↑.Prev↑.Next := Item↑.Next;
                Item↑.Next↑.Prev := Item↑.Prev;
                Item↑.List_Loc := nil;       mark as dequeued
                Linked_QRemove := true
            end if;
        Weak_V(h↑.Access_Sem)
    end if
    end with
end function Linked_QRemove;
```

In closing this section, we wish to point out that the interactions of the queue operations must be clearly understood when defining their critical code sections. The inclination to minimize the size of these sections through seemingly safe code movements must be carefully considered. For example, once an insertion list site is determined, the value to be assigned to an item's list location field is also fixed. But this assignment is postponed until the operation has exclusive access to the list. If the assignment were performed outside of the critical section, two operations could erroneously address the same item: Suppose the assignment is done after the critical section is exited. Between the time an insertion appends an item to a list but before it performs the list location assignment statement, the item can be deleted and have its list location set to nil by the deletion. The insertion would then overwrite this location with a valid list address, making the item appear to be queued. A removal performed on this item would then erroneously report that it had dequeued the item, while in actuality the item was not queued when the removal executed. Similarly, if the insertion performs the assignment before the critical section is entered, a removal executed on the item could again report success, even though it never actually dequeued the item.

2.3. Chapter Summary

Gottlieb et al. [83b] and Rudolph [82] gave concurrent insertion and deletion algorithms for sequentially and linked allocated queues. The key to achieving concurrency is that the fetch-and-add operation permits indices into the contiguous array of cells or linked list headers to be calculated in parallel by concurrently active queue operations. Each cell or list is then independently accessed. Solutions for resolving cell contention among concurrent operations include global and local cell protocols. We completed the linked list implementation of queues by giving two list protocols similar to the cell protocols cited above.

A major difference between the cell and list protocols of the sequential and linked list implementations is that the former must ensure the alternation of insertions and deletions whereas the latter does not. Both types of protocols must serialize access to their sites. Fair protocols offer starvation-free access but at the cost of reducing the throughput of the queue.

For both queue implementations a third type of operation for accessing an interior queue item has been introduced. Its implementation does not effect the concurrent accessibility of the queue–each site can be accessed in parallel by an insertion, deletion, or interior access operation. For all queue implementations, adding interior access requires that insertions and deletions perform certain extra bookkeeping tasks, notably marking an item as queued or dequeued. It is important to note that during periods of IA inactivity, the cost of performing an insertion or deletion is only raised by a (small) constant factor as compared to when interior access is not supported.

Removing an interior item from a sequentially allocated queue has proven to be a relatively simple task. Performing the same operation on a linked queue is complicated by the round-robin placement of items onto a queue's linked lists. To maintain the FIFO queue property, it was necessary to record how many times a given item's list successor had been removed. The alternative method of removing an item by compacting the queue after the item was taken is unattractive since it would suspend all other queue activity.

CHAPTER 3

Multiqueues

The maximum level of concurrent activity supported by the queue structures described in the preceding chapter is bounded by *Qsize*, the number of distinct cells or linked lists within a queue. Applications that enqueue and dequeue individual items are well served by these structures. They are less appropriate, from the standpoint of parallelism, for applications whose queue operations deal with arrays of items or copies of a given item as single entities. Examples of such applications include the allocation and deallocation of sets of memory pages (see chapter 4) and the initiation of multiple concurrent evaluations of a function, such as the iterates of a parallel loop (see chapter 6). These applications generate bursts of insertion or deletion activity to be performed, in part, by single tasks. Using a concurrently accessible queue, $\Omega(M)$ time is required for an application's task to insert M items and $\Omega(\lceil M/\min(N, Qsize)\rceil)$ time is required for M tasks to delete these items, where N is the number of PEs.[1] In this chapter we will examine an alternative queue structure that permits these same operations to be performed in constant time.

Gottlieb et al. [83b] define a queue with multiplicity or *multiqueue* as a queue consisting of items tagged with multiplicities. That is, entries on the queue are of the form (I, M), representing M consecutive copies of an item I on a (hypothetical) much longer queue. The time required for a multiqueue insertion is constant (i.e., it is independent of an item's multiplicity). The time for M deletions to be performed on an (I, M) entry is $\Omega(\lceil M/N\rceil)$, where N is the number of PEs. Each deletion makes a copy of I, and the Mth or *final* deletion to do so takes I off the multiqueue. The

[1] For the starvation-free implementations, the time complexities for the insertion and deletion operations described above are $\theta(M)$ and $\theta(\lceil M/\min(N, Qsize)\rceil)$, respectively, assuming that each PE executes its next instruction within a finite time and that the queue, if it is sequentially allocated, has at least M empty sites when the insertions are performed so that they succeed (if the queue is linked allocated, no site is ever full, i.e., each site can always accept an insertion). The insertion and deletion algorithms of these implementations busy-wait only to access sites, the rest of the code is done in constant time. However, since fair protocols are used to access sites, insertions and deletions are delayed by a bounded period of time.

ratio of insertions, to final deletions, to non-final deletions is $1{:}1{:}\overline{M}-1$, where $\overline{M}$ is the average item multiplicity. When $\overline{M}$ is large, the dominant multiqueue operation is the fully parallel non-final deletion.

Gottlieb et al. [80] gave two implementations for sequentially allocated multiqueues, designated as *linear* and *logarithmic*. The first permits only one deletion site to be accessible at a time, and thus if the multiplicity is bounded, concurrent deletions require time *linear* in the number of such operations. The second permits all *Qsize* sites of a multiqueue to be accessible to deletions, but at the cost of a log *Qsize* overhead for insertions and deletions. An analogue of the logarithmic multiqueue structure has also been used by Gottlieb et al. [80] to implement a parallel queue with priority (reviewed in Appendix A), which is logarithmic in the number *#P* of priorities (i.e., *Qsize*=*#P*).

The choice of which type of queue structure is best suited for a particular application should be guided by $\overline{M}$. When $\overline{M}$ is small, the standard queues of chapter 2 are most appropriate. When $\overline{M}$ is large, the linear multiqueue is appropriate, and when $\overline{M}$ is unknown, the logarithmic version should be selected.

In this chapter we present algorithms for insertion and deletion on sequentially allocated multiqueues that are more efficient than those originally proposed by Gottlieb et al. [80]. We then extend these algorithms to support linked list implementations of linear and logarithmic multiqueues. The dominant queue activity for these implementations, non-final deletion, is fully parallel when $\overline{M}$ is large.

To complement the insertion of *M* items onto a multiqueue, we define a *multi-deletion* of size *K* as the deletion of *K* items from a multiqueue. The multiqueue lends itself towards the efficient implementation of this operation since a single deletion site may contain multiple items.

We also define operations that access interior multiqueue items. These operations do not require exclusive access to an entire multiqueue, but only to specific sites. Thus, interior access adversely effects a queue's concurrency only when an item with large multiplicity located at an active deletion site is accessed. Deletions (and multi-deletions) are readers of sites and interior access operations (and insertions) are writers of sites. Readers/writers and writer priority readers/writers implementations of interior access that support insertion and deletion are presented. Algorithms for multi-deletion that permit interior access are not presented. However, we believe such algorithms can be derived from the basic multi-deletion algorithms that are given and from the deletion algorithms that do support interior access.

The readers/writers implementation is starvation-free, the writer priority readers/writers implementation is not. However, the latter, in the absence of interior access activity, executes non-final deletions as efficiently as when interior access is not supported; whereas, the former does not.

Two specific type two interior access operations are defined: *RaiseMult* and *LowerMult*. These operations raise and lower the multiplicity of a queued item, permitting applications to adjust item multiplicities to their needs. LowerMult may also be used to remove an item from a multiqueue.

As in the previous chapter, contention among operations for a multiqueue's cells or lists can occur, requiring P/V or readers/writers site protocols to be implemented. Unfortunately, in our multiqueue algorithms, even if the P/V or readers/writers protocols are fair, deletions may still starve. We will show that each multiqueue deletion algorithm can be made starvation-free by serializing part of the code. However, such a solution will not be advocated since it precludes non-final deletions from being fully parallel.

We also observe that the insertion and deletion of arrays and queues of items may be efficiently supported on multiqueues. An array of items is very similar in concept to an item with multiplicity. In fact, the latter may be viewed as a special case of the former: an item with multiplicity M is equivalent to an array of size M where each array element contains a copy of the same item. A set of objects (such as pointers to blocks of memory) held in an array can then be inserted onto a multiqueue in constant time. Similarly, a queue containing M items can be treated as an item with multiplicity M. That is, it can be attached to a multiqueue in constant time via a multiqueue insertion with multiplicity M. A multiqueue deletion on this entry then performs a standard queue deletion to obtain an item. Multiqueue applications using array and queue entries are given in sections 4.2 and 5.4, respectively.

In chapter 2 we refer to an unordered collection of items as a pile. Analogously, a pile with multiplicity or *multipile* is a pile that consists of items tagged with multiplicity. Multipiles may of course be implemented as multiqueues. However, since FIFO item processing is not required for multipiles, the multiqueue data structures and algorithms presented in this chapter can be modified to produce multipiles whose operations (e.g., insertion into and deletion from a multipile) are more efficient than their multiqueue counterparts. In chapter 8 we describe how this conversion can be done.

Linear multiqueues are addressed in the first section of this chapter, and logarithmic multiqueues are dealt with in the second section. For each type of queue, sequential and linked list structures are given along with algorithms for insertion, deletion, multi-deletion, and interior access.

3.1. Linear Multiqueues

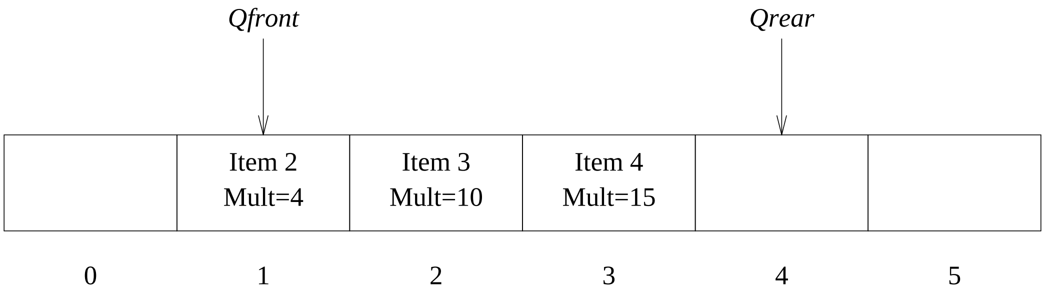
Sequentially allocated and linked list allocated linear multiqueues are covered in the next two sections. For each allocation technique, the basic multiqueue structure along with the operations of insertion and deletion are reviewed. Modified versions of insertion and deletion, which are more efficient than those originally proposed by Gottlieb et al. [80], are then given.

3.1.1. Sequential Allocation

3.1.1.1. A Review of Insertion and Deletion

A sequentially allocated linear multiqueue has the same basic structure as the sequentially allocated queue of chapter 2. A multiqueue is represented by an array of *Qsize* cells. Associated with each cell is a multiplicity component *Mult* that equals the current multiplicity of its item. The multiqueue components *Qfront* and *Qrear* taken modulo *Qsize* identify the front and rear of the queue. *Qupper* indicates how many cells are empty, bounding the number of insertions that can be performed. *Qlower* equals the number of items in the multiqueue, which is the sum of the multiplicities of the non-empty cells, and bounds the number of deletions. See Figure 3-1.

Multiqueue insertion sites are calculated using the procedure of chapter 2, i.e., by incrementing *Qrear* and taking the result modulo *Qsize* to obtain a site index. Deletion sites can not be determined in a similar manner since the number of items in a cell is no longer restricted to one, but to an item’s multiplicity. Sending each incoming deletion to a different cell would violate the FIFO queue property. For example, suppose insertions have been performed on three consecutive sites, *a*, *b*, and *c*, where *a* is the current deletion site, when a deletion *D* is performed. Before *D* accesses *a*, other deletions arrive and empty not only *a*, but *b* as well. If an insertion is then performed on *b before D* references it, the item at *b* will be deleted prematurely, i.e., ahead of the one at *c*. Since a cell with multiplicity *M* has to be accessed by *M* deletions before the next



Linear Multiqueue of *Qsize* 6 containing 29 items.

Figure 3-1.

cell is opened, deletions may access only one site at a time. If the front cell is of multiplicity M and $M+N$ deletions enter the queue, the first M must complete before the next N proceed to the following cell. Notice however that when converting multiqueues into multipiles, deletions can be sent to different sites since the latter does not support the FIFO queue property.

Qfront is accessed by incoming deletions to determine the current deletion site and by final deletions to advance the site to the next cell. That is, the former are readers and the latter are writers of *Qfront*. The implementation of Gottlieb et al. [80] (reviewed in Appendix B) has these two groups coordinate their access to *Qfront* through readers/writers protocols. In the modified implementation below, these protocols have been eliminated and replaced with readers/writers protocols between insertions and final deletions, reducing the execution time of non-final deletions, which is the dominant operation when the average item multiplicity is large.

3.1.1.2. Modified Insertion and Deletion

The modified implementation introduces a queue component *FrontMult* that is initialized to the multiplicity of an item I's cell when I moves to the front of the queue. A deletion cycles until *FrontMult* is positive and then decrements it by one to reserve a copy of the front item for itself.[2]

After decrementing *FrontMult*, a deletion reads (but does *not* modify) *Qfront* to determine the location of the front queue cell. A deletion site *d* is therefore calculated as follows:

$$
\begin{array}{l}
s := 0; \\
\textbf{repeat} \\
\quad \textbf{if } FrontMult > 0 \textbf{ then } s := \mathrm{F\&A}(FrontMult, -1) \textbf{ end if} \\
\textbf{until } s > 0; \\
d := Qfront;
\end{array}
\tag{1}
$$

Following this calculation, a deletion makes a copy of cell *d*'s item and decrements *d*'s multiplicity component *Mult* by one. The *final* deletion, the one decrementing *d*'s multiplicity to zero, takes the item off the queue, advances *Qfront* to the next cell *e*, and initializes *FrontMult* to *e*'s multiplicity. It is unnecessary for deletions to lock the front site as *FrontMult*'s initial value bounds the number of deletions. In fact, the queue component *FrontMult* and the front cell's multiplicity component *Mult* are lower and upper bounds on the number of copies of the front item that remain to be deleted.

FrontMult is updated not only by a final deletion but also by an insertion into an empty queue. These updates must be mutually exclusive to prevent erroneous initializations of *FrontMult*. The code fragments below show how this coordination is achieved, where we assume that a binary semaphore is associated with each cell.

A final deletion on *Cell*[*d*] executes

	n := (*d* + 1) **mod** *Qsize*;	*index of new front cell*
	Weak_P(*Cell*[*n*].*Access_Sem*);	*lock cell*
D1:	*Qfront* := *n*;	*new deletion site*
D2:	*FrontMult* := *Cell*[*n*].*Mult*;	*multiplicity of new site*
	Weak_V(*Cell*[*n*].*Access_Sem*);	*unlock cell*

to advance the deletion site to the next cell *n* and assign *n*'s multiplicity to *FrontMult*. The order of the statements in the critical code section is crucial. If they were reversed (D2 followed by D1), incoming deletions could reference the wrong site by decrementing *FrontMult* before the final deletion's update of *Qfront* was done.

An insertion places an item with multiplicity *M* into *Cell*[*i*] via

---

[2] We wait for *FrontMult* to be positive before a decrement is attempted to prevent looping deletions from generating an integer underflow.

```
   Weak_P(Cell[i].Access_Sem);
I1:      Cell[i].Value := Item;
I2:      Cell[i].Mult := M;
I3:      if Qfront = i then FrontMult := M end if;   front cell insertion
   Weak_V(Cell[i].Access_Sem);
```

The sequence of the above statements is again important. If I3 preceded I2, an incoming deletion could decrement *FrontMult* and then the front cell's multiplicity before the latter had been initialized by an insertion.

The insertion and final deletion cell references are critical code sections to ensure the proper initialization of a cell and *FrontMult*. Otherwise, a problem could develop between a final deletion on cell *i* and an insertion on the next cell *j* = *i*+1 modulo *Qsize*. If uncoordinated, a possible interleaving of the above statements is: I1, I2, D1, D2, I3. Unfortunately, this results in *FrontMult* being initialized twice. If any deletions decremented *FrontMult* between the two initializations, more than *M* deletions would access the front item.

With critical code sections, the initialization of *FrontMult* is always performed correctly: First, assume the deletion on cell *i* accesses cell *j* before the insertion. When the deletion exits the critical code section, *Qfront* points to *j* and *FrontMult* is zero since the cell is empty (*Cell*[*j*].*Mult*=0). The insertion then executes the critical code section, initializing both the multiplicity of the cell and *FrontMult* to its item's multiplicity. Now assume the insertion occurs before the deletion. In this case, *FrontMult* is not set by the insertion because *Qfront* still points to cell *i*. The deletion updates *Qfront* and *FrontMult* after the insertion exits its critical code section.

In chapter 2 (section 2.1.1.1), two cell contention problems for sequentially allocated queues are described: consecutive insertion/deletion contention and concurrent insertion/deletion contention. Their resolution represents a major component in the design effort of the queue's operations. These same cell contention problems are not present in multiqueues: Since only one multiqueue deletion site is active at a time, a deletion cannot be assigned a site before an insertion has taken place there nor can an insertion be directed to a site that has not yet been emptied by a final deletion. Therefore, consecutive insertion/deletion contention is precluded. Also, because just one deletion site is open, there is at most one final deletion active at a time; thus, a site is never contended for by multiple concurrent final deletions. In addition, limiting the number of active deletion sites to one prevents insertions from wrapping around the queue and contending with other insertions for a common site. Hence, concurrent insertion/deletion contention does not occur either. The only cell contention that is present is between an insertion on cell *i* and a final deletion on cell *d* when $i = d+1$ modulo *Qsize*. A method for resolving this contention was described above.

As a maximum of two operations, an insertion and a final deletion, can contend for a cell, fair cell protocols are unnecessary. However, it is possible for an incoming deletion *D* to starve while performing the site calculation (see code segment (1) above). Due to other deletion activity, *D* may never test *FrontMult* when it is positive. As a remedy, each deletion could execute this calculation as a critical code section; but, this introduces a potential bottleneck into the deletion algorithm, preventing non-final deletions from being fully parallel and therefore will not be pursued.

However, making the site calculation a critical code section is one method for implementing a global protocol that guarantees that the *i*th deletion takes the item inserted by the *i*th insertion, similar to the protocol given in section 2.2.2.2. The critical code section permits a serial ordering to be defined for deletions. (A serial ordering for insertion is defined by a fetch-and-add on the queue counter *Qrear*.) Without the critical code section, the relative order of deletions can not be determined based solely on their site calculations. Of course, one could also define such an ordering by having the deletions perform fetch-and-adds on a shared counter before or after the site calculation.

All of the linear multiqueue algorithms presented in this chapter resolve cell contention via local unfair protocols. A global (fair) protocol may be implemented by applying the technique cited above.

As in chapter 2, a multiqueue is assumed to contain pointers, referred to as items, to objects with multiplicity. We will often fail to differentiate between an item and its object. Our algorithms and data structures can be modified to support multiqueues of values instead of pointers.

```
Seq_LinMQueue =    Sequentially Allocated Linear MultiQueue
                   record
                      Qlower, Qupper, Qsize, Qfront, Qrear,
                      FrontMult : integer;
                      Cell : array[0..Qsize–1] of CellElement
                   end;

CellElement = record
                 Mult        : integer;
                 Value       : Object_Pointer;
                 Access_Sem  : Weak_Semaphore
              end;
```

```
Object = record
                Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qupper, Qfront, Qrear, FrontMult = 0;
   for 0≤j<Qsize,
     Cell[j].Mult                    = 0;
     Cell[j].Access_Sem              = 1;

function Seq_LinMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LinMQueue);
var i : integer;
   with Q do
   if TIR(Qupper, 1, Qsize)   then
      i := F&A(Qrear, 1) mod Qsize;
      if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
      Weak_P(Cell[i].Access_Sem);
         Cell[i].Value := Item;
         Cell[i].Mult  := M;
         if Qfront = i then FrontMult := M end if;
      Weak_V(Cell[i].Access_Sem);
      F&A(Qlower, M);
      return true
   else
      return false
   end if
   end with
end function Seq_LinMQInsert;
```

```
function Seq_LinMQDelete (Item : Object_Pointer;
                          Q    : Seq_LinMQueue);
var d, n, s : integer;
   with Q do
   if TDR(Qlower, 1) then
      Calc_Deletion_Site(d)_V1;
      Item := Cell[d].Value;
      if F&A(Cell[d].Mult, −1) = 1 then    final delete
         Open_Next_Cell_Site_PV;
         F&A(Qupper, −1);
      end if;
      return true
   else
      return false
   end if
   end with
end function Seq_LinMQDelete;
```

Deletion site calculation and final delete processing are defined by the macros Calc_Deletion_Site(d)_V1[3] and Open_Next_Cell_Site_PV given below.

```
macro Calc_Deletion_Site(d)_V1;
   s := 0;
   repeat
      if FrontMult > 0 then s := F&A(FrontMult, −1) end if
   until s > 0;
   d := Qfront;
end macro Calc_Deletion_Site(d)_V1;

macro Open_Next_Cell_Site_PV;
   n := (d + 1) mod Qsize;
   Weak_P(Cell[n].Access_Sem);
      Qfront := n;
      FrontMult := Cell[n].Mult;
   Weak_V(Cell[n].Access_Sem);
end macro Open_Next_Cell_Site_PV;
```

In the deletion algorithm, it is tempting to have a final deletion on cell *d* first decrement *Qupper* to enable an insertion to take place on the recently emptied cell and then have it open the next site. Unfortunately, an insertion could then occur on *d* before the front queue pointer was moved. The insertion would then erroneously initialize *FrontMult* (since *Qfront*=*d*).

[3] The suffix denotes version one.

After the final deletion of an item I with multiplicity M is performed, up to M tasks may hold pointers to I's object. We assume that mechanisms (not shown here) are in place to coordinate the eventual deallocation of this object.

The multiqueue structure and algorithms can be easily modified to enqueue and dequeue arrays of items. That is, each cell of the queue can contain (i.e., point to) an array of size M of items as opposed to an item with multiplicity M. The insertion algorithm remains virtually the same (only the type of its first parameter changes), and a deletion, instead of copying a cell's value, indexes into the cell's array (via a fetch-and-add) to copy one of its elements. A memory allocation scheme based on this concept is given in section 4.2 along with fully defined multiqueue data structures and operations.

3.1.1.3. Multi-Deletion

We now consider the generalized delete operation *multi-deletion* that removes M items at a time from a multiqueue, or more precisely, makes c_i copies of item i, where $\sum c_i = M$. The parameters of this operation are: M, the number of items to be deleted; *Table*, an array of size M to hold the items; and Q, a multiqueue. If Q does not contain at least M items, the operation fails. Otherwise, the multi-deletion copies as many items as possible (up to M) from the front queue cell and transfers them to *Table*. If the cell does not contain enough items to satisfy the request, the operation waits for the deletion site to be moved and then continues to acquire items. This process is repeated until M items are obtained. Consecutively queued items may not be obtained due to other deletion activity. If such behavior is unacceptable, then the multi-deletion would need to obtain exclusive access to the queue.

The deletion algorithm for multiqueues is transformed into a multi-deletion algorithm by decrementing queue and cell multiplicities (i.e., *FrontMult* and *Cell*[c].*Mult*) by values greater than one. For example, the multi-deletion attempts to obtain M items from the front cell via

$$s := \text{F\&A}(\mathit{FrontMult}, -M).$$

The actual number granted g equals the minimum of s and M. After copying g items from the front cell c, the multi-deletion decrements c's multiplicity by g:

$$t := \text{F\&A}(\mathit{Cell}[c].\mathit{Mult}, -g).$$

The *final* multi-deletion of cell c is the one that copies the last g items contained in the cell (i.e., the one that finds $t=g$).

In the best case, a multi-deletion obtains *M* items from one cell, having to calculate the deletion site only once. In the worst case, a multi-deletion references *M* sites, yielding the same asymptotic performance as the serial execution of *M* deletions.

```
function Seq_LinMulti-Delete (Table : array[1..M] of Object_Pointer;
                              M     : integer;
                              Q     : Seq_LinMQueue);
var d, g, m, n, s : integer;
   with Q do
   if TDR(Qlower, M) then
      m := M;
      repeat    keep taking items from the Qfront cell until satisfied
         s := 0;
         repeat
            if FrontMult > 0 then s := F&A(FrontMult, –m) end if
         until s > 0;
         d := Qfront;
         g := s min m;   number of items to copy from cell array
         Table[m–g+1..m] := Cell[d].Value;
         m := m – g;      remaining number to obtain
         if F&A(Cell[d].Mult, –g) = g then    final multi-delete
            Open_Next_Cell_Site_PV;   defined in sect. 3.1.1.2
            F&A(Qupper, –1)
         end if
      until m = 0;
      return true
   else
      return false
   end if
   end with
end function Seq_LinMulti-Delete;
```

Multi-deletion may also be performed on a multiqueue of item arrays, where the operation copies elements of queued arrays into a table. For example, if arrays of available memory blocks are kept on a multiqueue, then a set of blocks can be allocated by a single multi-deletion. Such an application, with the accompanying deletion algorithm, is given in section 4.2.

3.1.1.4. Interior Access

Interior access operations on a multiqueue are quite similar to the corresponding operations on the concurrently accessible queues described in chapter 2. Recall that for queues an interior access operation IA acquires exclusive access to a cell by closing the cell's semaphore. This technique can also used for multiqueues when the cell in question is *not* at the front of the queue.

Closing the semaphore of the front cell does not exclude deletions since their access is only controlled by the value of *FrontMult*, not by an entry cell protocol (such as a P semaphore operation). Rather than serializing deletions, which would eliminate much of a multiqueue's advantage over a simple queue, we coordinate them with IA operations by enforcing a readers/writers relationship: multiple deletions (the readers) or a single IA operation (the writer) may be performed on a front item. Readers/writers and writer priority readers/writers implementations of this relationship are given. The first requires deletions and IA operations to execute explicit readers/writers protocols to access the front cell. The second does not. Rather, incoming deletions execute the same deletion code as when interior access is not supported, and interior access operations acquire write access to the deletion site by manipulating queue components referenced in the deletion code. Basically, interior access operations make *FrontMult* negative to block incoming deletions from accessing the front item. Operating on *FrontMult* in this manner gives interior access operations priority over deletions.

For both implementations, a cell that is not at the front of the queue is only accessed by an insertion, a final deletion, or an IA operation. The readers/writers and writer priority readers/writers implementations serialize these operations via readers/writers and binary P/V cell protocols, respectively.[4]

The introduction of interior access operations raises the possibility of a given deletion or IA operation starving due to other IA activity. The readers/writers implementation is starvation-free when fair protocols are used, and, as noted in section 3.1.1.2, the deletion site calculation is a critical code section. The writer priority readers/writers implementation may starve deletions. However, we are willing to accept this possibility for improved deletion efficiency.

Type one and two interior access operations are covered in the next two sections. Recall that the first can only modify a queued item's application dependent information whereas the latter can modify both the item's application and queue dependent information. Generic type one procedures are given. They serve primarily to introduce the basic techniques that will be used in the implementations of the specific type two operations of RaiseMult and LowerMult, which

[4] Alternatively, the writer priority readers/writers implementation could use write request/release operations instead of P/V operations to access a nonfront cell. We do not elect to do this since deletions (readers) do not issue explicit read request/release operations.

increment and decrement, respectively, an item's multiplicity.

For each type of interior access operation, the readers/writers implementation is presented first, followed by the writer priority readers/writers implementation. When operations of both types are used in an application, we assume that their implementations will agree (i.e., only their readers/writers or writer priority readers/writers implementations will be chosen).

3.1.1.4.1. Type One

3.1.1.4.1.1. A Readers/Writers Implementation

In this implementation deletions, including final deletions, are readers of front cells, and insertions and interior access operations are writers of cells, regardless of whether the cells are at the front.

A deletion calculates the position *d* of the deletion site and then requests permission to read cell *d*. Once permission is granted, the deletion makes a copy of the front item and then releases *d*. Insertions and interior access operations execute write request and release operations before and after a cell access.

A final deletion is a writer of the front cell (when marking the cell's item as dequeued), but it does not issue a write request to gain exclusive access. By virtue of being the last reader of the cell, it already has such access. However, a final deletion on a cell *d* must still exclusively access (via a write request) the next cell *e* to open up a new deletion site. Notice that if an IA operation gains write access to a cell *c* that is not at the front of the queue, a final deletion on the preceding cell will be unable to reference *c* until the IA operation terminates. If *c* is at the front of the queue, all deletions will be blocked until the IA operation completes.

While a nonfront cell could be exclusively accessed via a binary P/V protocol, this would force two protocols, readers/writers and P/V, to be associated with each cell. Which protocol was desired would depend upon the cell's position relative to the front of the queue. We elect to implement the simpler scheme of always accessing a cell via a readers/writers protocol, regardless of its position. Notice that the time complexities of the readers/writers and P/V cell protocols are the same. The readers/writers and P/V implementations differ only in that the first increments a counter by ± 1 (for readers) and $\pm MaxR$ (for writers) and the second always uses increments of ± 1.

The multiqueue data structure (given in section 3.1.1.2) is extended to include a queue identifier *Queue_Id* with each cell. The identifier is referenced by interior access operations to verify whether an item belongs to the desired queue (see section 2.1.2 for details). Each cell's binary semaphore is replaced with a readers/writers activity counter *RW_Act*.[5] We assume that each item contains a cell location component *Cell_Loc* that defines its position within a queue.

The algorithms for insertion, deletion, and interior access follow. The last is patterned after the first sequentially allocated queue version of IA given in section 2.1.2. That is, for an IA operation to succeed, an interior item must be found in the first queue cell that the operation inspects; the operation fails if the item shifts from one cell to another within the multiqueue while the operation is in progress (see section 2.1.2 for details).[6] The insertion algorithm differs from the one given in section 3.1.1.2 only in that it marks an item as queued.

```
Seq_LinMQueue =    MultiQueue Supporting Type I IA
                   record
                      Qlower, Qupper, Qsize, Qfront, Qrear,
                      FrontMult : integer;
                      Cell : array[0..Qsize–1] of CellElement
                   end;

CellElement = record
                   Queue_Id : MQueue_Pointer;
                   Mult     : integer;
                   Value    : Object_Pointer;
                   RW_Act   : integer
              end;

MQueue_Pointer = ↑Seq_LinMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
              Cell_Loc : CellElement_Pointer;
              Application_Defined_Attributes
         end

Object_Pointer = ↑Object;
```

[5] The counter equals $MaxR(1-w)-r$, where r and w are, respectively, the number of active readers and writers, and $r+w<MaxR$. The counter has an initial value of *MaxR*.

[6] Throughout the remainder of this chapter, all interior access operations will be based on this technique. Analogues of other versions found in section 2.1.2 are also possible.

Initially:

```
Qlower, Qupper, Qfront, Qrear, FrontMult = 0;
for 0≤j<Qsize,
  Cell[j].Queue_Id      = queue address;
  Cell[j].Mult          = 0;
  Cell[j].RW_Act        = MaxR;
```

```
function Seq_LinMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LinMQueue);
var i : integer;
   with Q do
   if TIR(Qupper, 1, Qsize) then
      i := F&A(Qrear, 1) mod Qsize;
      if i = Qsize−1 then F&A(Qrear, −Qsize) end if;
      Write_Request(Cell[i].RW_Act);
         Item↑.Cell_Loc := address(Cell[i]);
         Cell[i].Value := Item;
         Cell[i].Mult := M;
         if Qfront = i then FrontMult := M end if;
      Write_Release(Cell[i].RW_Act);
      F&A(Qlower, M);
      return true
   else
      return false
   end if
   end with
end function Seq_LinMQInsert;
```

```
function Seq_LinMQDelete (Item : Object_Pointer;
                          Q    : Seq_LinMQueue);
var d, n, s : integer;
   with Q do
   if TDR(Qlower, 1) then
      Calc_Deletion_Site(d)_V1;   defined in sect. 3.1.1.2
      Read_Request(Cell[d].RW_Act);
         Item := Cell[d].Value;
         if F&A(Cell[d].Mult, −1) = 1 then    final delete
            Item↑.Cell_Loc := nil;
            Open_Next_Cell_Site_RW;
            F&A(Qupper, −1)
         end if;
      Read_Release(Cell[d].RW_Act);
      return true
   else
      return false
   end if
   end with
end function Seq_LinMQDelete;


macro Open_Next_Cell_Site_RW;
   n := (d + 1) mod Qsize;
   Write_Request(Cell[n].RW_Act);
      Qfront := n;
      FrontMult := Cell[n].Mult;
   Write_Release(Cell[n].RW_Act);
end macro Open_Next_Cell_Site_RW;


function Seq_LinMQIA_TypeI (Item : Object_Pointer;
                            Q    : Seq_LinMQueue;
                            ϕ    : function);
var c : CellElement_Pointer;
   with Q do
   Seq_LinMQIA_TypeI := false;
   c := Item↑.Cell_Loc;
   if c ≠ nil and c↑.Queue_Id = address(Q) then
      Write_Request(c↑.RW_Act);
         if c = Item↑.Cell_Loc then
            ϕ(Item);            access Item
            Seq_LinMQIA_TypeI := true
         end if;
      Write_Release(c↑.RW_Act)
   end if
   end with
end function Seq_LinMQIA_TypeI;
```

The deletion algorithm, as stated above, deadlocks when the multiqueue consists of only one cell, i.e., when *Qsize*=1. The final deletion of the cell tries to gain write access before doing a read release. We will disregard this degenerate case.

3.1.1.4.1.2. A Writer Priority Readers/Writers Implementation

The writer priority readers/writers implementation does not require deletions to explicitly issue read requests to access a deletion site. Recall that when interior access is not supported a deletion cycles until it can decrement the queue component *FrontMult* when it is positive to gain access to the front cell and then decrements the front cell's multiplicity component *Mult* after the access is completed. When *FrontMult* is positive, *Mult*–*FrontMult* equals the number of active deletions referencing the front item. When *FrontMult* is nonpositive, no new deletions can start and *Mult* equals the number of active deletions. The writer priority readers/writers implementation therefore uses a deletion's decrements on *FrontMult* and *Mult* in place of the read request and release operations of the preceding section to coordinate front cell accesses. All other cell accesses are coordinated via binary P/V cell protocols.

Insertion differs from the algorithm of the preceding section only in that P/V protocols replace readers/writers protocols.

Non-final deletion follows the algorithm of section 3.1.1.2, which does not support interior access. That is, a non-final deletion calculates the location *d* of the deletion site, copies the item *I* in cell *d*, and decrements *d*'s multiplicity. Thus, during periods of interior access inactivity, non-final deletion is as efficient as when interior access is not supported. The deletion that drives *d*'s multiplicity to zero is a final deletion. It obtains exclusive access to *d*, marks *I* as dequeued, releases *d*, and then opens the next deletion site. (The justification for why the front cell is locked is given at the end of this section.)

An interior access operation on the front cell issues a write request by making *FrontMult* very negative, preventing incoming deletions from accessing the cell, and then waits for the currently active deletions, i.e., deletions that have decremented *FrontMult* but not the front cell's multiplicity component *Mult*, to terminate.

The basic interior access procedure for accessing an item *I* in a cell *c* is:

```
L1:   lock cell c to exclude other IA operations
L2:   if I is still in c then
L3:      if I is a nonfront item then
L4:         access I
L5:      else I is the front item
L6:         if FrontMult > 0 then
L7:            f := F&A(FrontMult, –MaxMult);
L8:            if f > 0 then
L9:               repeat until c.Mult = f;
L10:              access I;
L11:              FrontMult := f  restore value
L12:           end if
L13:        end if
L14:     end if
L15:  end if;
L16:  unlock cell c;
```

MaxMult is assumed to be the maximum multiplicity of an item.

Let us step through the code for accessing a front item, lines L6 through L13. If an interior access operation on the front item finds that *FrontMult* is nonpositive (at L6 or L8), it fails since a final deletion is already in progress. The test of *FrontMult* (at L6) is done to prevent integer underflow: Suppose this test (at L6) were omitted, and a final deletion had decremented *FrontMult* to zero, but had not progressed further, in particular, it had not yet locked the front cell *d* to dequeue *d*'s item *I*. IA operations are then issued on *I*. Each one locks *d* (at L1), decrements *FrontMult* (at L7), and fails to access *I* since *FrontMult* (i.e., *f*) is nonpositive. If the number of IA operations is sufficient, *FrontMult* underflows.

If an IA operation decrements *FrontMult* (at L7) and finds (at L8) that *FrontMult*'s old value is nonpositive, the operation does not restore *FrontMult* to this value. Instead, the IA operation acts like a deletion that tests *FrontMult* when it is positive and then decrements *FrontMult* after it has become nonpositive due to other deletion activity. In this case, *FrontMult* remains negative until a final deletion or an insertion assigns it a positive value.

If an interior access operation decrements *FrontMult* (at L7) when it is positive, then *c.Mult*–*f* deletions are reading *c*. In this case, the interior access operation waits (at L9) for the deletions to finish (i.e., for *c.Mult* to equal *f*). It then accesses *c* after which it restores *FrontMult* to the multiplicity *f* of *c*. Deletions that access *FrontMult* after the IA operation makes *FrontMult* negative (at L7) busy-wait until *FrontMult* is assigned a positive value (at L11).[7] Notice that the IA operation cannot restore *FrontMult*'s old value (at L11) via a fetch-and-add of *MaxMult* since *FrontMult* may be less than $f - MaxMult$ due to deletion activity that occurred after the IA operation made *FrontMult* negative (at L7).

A final deletion is the last reader of a cell and also a writer of a cell when dequeueing an item. In the readers/writers implementation, a final deletion requests only read access to the front cell; it has write access by virtue of being the last reader. Now, in the writer priority readers/writers implementation, a final deletion must explicitly lock the front cell to dequeue an item. At first it may appear that this is unnecessary: *FrontMult* becomes nonpositive once a final deletion arrives, causing subsequent interior access operations to fail (at L6). However, if the front cell were not locked by a final deletion, an interior access operation and a final deletion could execute concurrently, resulting in the deadlocking of deletions and the IA operation. This is illustrated in the following scenario where we assume that final deletions do not lock the front cell to dequeue an item:

(1) An interior access operation on the front cell *c* is at L6.

(2) A final deletion makes *FrontMult* nonpositive and *c*'s multiplicity zero.

(3) The final deletion marks the item in *c* as dequeued and then opens the next deletion site *d*, which includes initializing *FrontMult* to *d*'s multiplicity *M*. Assume *M* is positive.

(4) The interior access operation executes L6 through L8 and then enters an infinite loop at L9 and incoming deletions perform the site calculation forever: No insertions can be done at *c* as the interior access operation has exclusive access to the cell (via L1). The interior access operation tested and decremented *FrontMult* when *FrontMult* had a positive value of f=M. It therefore waits (forever) for *c*'s multiplicity to equal f. No incoming deletions can proceed since the interior access operation has made *FrontMult* negative (at L7).

This deadlock scenario is precluded by having a final deletion exclusively access a front cell to dequeue an item. A final deletion can not perform the third step of the above scenario until the

[7] This busy-waiting is similar to that performed by incoming deletions that access *FrontMult* before a final deletion opens the next site.

IA operation releases the front cell, which is guaranteed to occur within a finite time (since *FrontMult* is nonpositive, the IA operation will unlock the cell after executing L6). Notice that a scenario similar to the one above can be constructed to demonstrate that deadlock could also occur if a final deletion on cell *d* first opened the next deletion site and then exclusively accessed *d*.

The data structures for this implementation differ from those of the preceding section only in the substitution of binary semaphores (which are initially open) for readers/writers activity counters.

```
Seq_LinMQueue =    MultiQueue Supporting Type I IA
                   record
                       Qlower, Qupper, Qsize, Qfront, Qrear,
                       FrontMult : integer;
                       Cell : array[0..Qsize–1] of CellElement
                   end;


CellElement = record
                   Queue_Id     : MQueue_Pointer;
                   Mult         : integer;
                   Value        : Object_Pointer;
                   Access_Sem : Weak_Semaphore
              end;

MQueue_Pointer = ↑Seq_LinMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
              Cell_Loc : CellElement_Pointer;
              Application_Defined_Attributes
         end

Object_Pointer = ↑Object;

Initially:
    Qlower, Qupper, Qfront, Qrear, FrontMult = 0;
    for 0≤j<Qsize,
      Cell[j].Queue_Id              = queue address;
      Cell[j].Mult                  = 0;
      Cell[j].Access_Sem            = 1;
```

```
function Seq_LinMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LinMQueue);
var i : integer;
    with Q do
    if TIR(Qupper, 1, Qsize) then
        i := F&A(Qrear, 1) mod Qsize;
        if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
        Weak_P(Cell[i].Access_Sem);
            Item↑.Cell_Loc := address(Cell[i]);
            Cell[i].Value := Item;
            Cell[i].Mult := M;
            if Qfront = i then FrontMult := M end if;
        Weak_V(Cell[i].Access_Sem);
        F&A(Qlower, M);
        return true
    else
        return false
    end if
    end with
end function Seq_LinMQInsert;


function Seq_LinMQDelete (Item : Object_Pointer;
                          Q    : Seq_LinMQueue);
var d, n, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Deletion_Site(d)_V1;   defined in sect. 3.1.1.2
            Item := Cell[d].Value;
            if F&A(Cell[d].Mult, –1) = 1 then    final delete
                Weak_P(Cell[d].Access_Sem);
                    Item↑.Cell_Loc := nil;
                Weak_V(Cell[d].Access_Sem);
                Open_Next_Cell_Site_PV;   defined in sect. 3.1.1.2
                F&A(Qupper, –1)
            end if;
        return true
    else
        return false
    end if
    end with
end function Seq_LinMQDelete;
```

```
function Seq_LinMQIA_TypeI (Item : Object_Pointer;
                            Q    : Seq_LinMQueue;
                            ϕ    : function);
var c  : CellElement_Pointer;
    f  : integer;
    with Q do
    Seq_LinMQIA_TypeI := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Weak_P(c↑.Access_Sem);
            if c = Item↑.Cell_Loc then
                if Item ≠ Cell[Qfront].Value then
                    ϕ(Item);                            access Item
                    Seq_LinMQIA_TypeI := true
                else  Item is at front of queue
                    if FrontMult > 0 then
                        f := F&A(FrontMult, −MaxMult);  block deletions
                        if f > 0 then                   f deletions to be done
                            repeat until c↑.Mult = f;   wait for deletions
                            ϕ(Item);                    access Item
                            FrontMult := f;             restore FrontMult
                            Seq_LinMQIA_TypeI := true
                        end if
                    end if
                end if
            end if;
        Weak_V(c↑.Access_Sem)
    end if
    end with
end function Seq_LinMQIA_TypeI;
```

An interior access operation can safely inspect *Qfront*, which can be updated by final deletions, even though it does not obtain exclusive access to this component explicitly. If cell *d* is the deletion site and the IA operation is accessing *d*, a final deletion cannot occur until the operation releases *d*. If *d* is not the front site, a final deletion on the preceding cell cannot make *d* the new site until the operation is completed.

3.1.1.4.2. Type Two

In this section, two primitives, RaiseMult and LowerMult, are presented that increment and decrement an interior item’s multiplicity. LowerMult can also be used to remove an item.

3.1.1.4.2.1. A Readers/Writers Implementation

Our readers/writers implementation of type two operations follows the basic strategy employed for implementing type one operations. Explicit readers/writers protocols are used to access a site, where interior access operations and insertions are writers and incoming deletions are readers. Final deletions issue write requests only when opening deletion sites.

RaiseMult and LowerMult increment and decrement, respectively, an item *I*'s multiplicity. As an aid to understanding how these updates are done recall that when *I* is a nonfront item in a cell *c*, *I*'s multiplicity is defined by *c*'s multiplicity component *Mult*. When *I* is the front item, its multiplicity is defined by the queue component *FrontMult* and the front cell's component *Mult*. These components represent lower and upper bounds on the number of copies of *I* that remain to be deleted.

RaiseMult is passed three parameters: an item *I*, a positive integer *K*, and a multiqueue *Q*. If *I* does not belong to *Q*, the primitive returns false. Otherwise, it raises *I*'s multiplicity by incrementing the following variables by *K*: the multiplicity component *Mult* of *I*'s cell, the lower bound number *Qlower* of items in the queue, and, if *I* is at the front of the queue, the front item multiplicity counter *FrontMult*. The primitive then returns true.

For the most part, RaiseMult increments a variable *V* by a value *K* via an assignment of $V+K$ when *V* is serially accessible and via a fetch-and-add of *K* when *V* is concurrently accessible. A serially accessible variable can always be incremented via a fetch-and-add, but, to help distinguish between the two variable types, fetch-and-add will be restricted to incrementing only concurrently accessible variables.

RaiseMult, like all interior access operations, first obtains exclusive access to an item *I*'s cell *c*. Hence, RaiseMult increments *c*'s *Mult* component via an assignment. It increments *Qlower* via a fetch-and-add since *Qlower* is accessible to other operations (e.g., deletions). Incrementing *FrontMult* (when *I* is the front item) is more involved than the execution of a single assignment or fetch-and-add due to possible deletion activity. This is explained in greater detail below.

Although RaiseMult excludes deletions from referencing the front cell by obtaining write access, deletions are not precluded from testing and decrementing *FrontMult* (when calculating deletion sites). Suppose RaiseMult locks the front cell and then tests *FrontMult* after deletions have made it nonpositive. Then RaiseMult can safely increment *FrontMult* via an assignment of *K*. On the other hand, if RaiseMult tests *FrontMult* when it has a positive value of *f*, an assignment of $f+K$ would not work since *FrontMult* may be decremented by deletions after RaiseMult tests it and before the assignment is performed. The subsequent assignment would result in

FrontMult being greater than the front item's multiplicity. However, incrementing *FrontMult* when it was positive via a fetch-and-add of K would also be incorrect: between RaiseMult's testing of *FrontMult* and the fetch-and-add of K, deletions can lower *FrontMult* to a negative value. The fetch-and-add would then result in *FrontMult* being less than the front item's multiplicity.

The above problem arises since deletions and RaiseMults both are readers and writers of *FrontMult*. We could of course coordinate these operations by having them request permission to read or write *FrontMult* or by serializing their access to the queue. However, these solutions increase the cost and/or decrease the concurrency of deletion. The solution we have elected to use does neither. It implements a writer priority readers/writers protocol that is similar in style to the writer priority readers/writers implementation of type one operations: RaiseMult increments the front item's multiplicity after first determining the current value of *FrontMult* and temporarily making it negative via

$$f := \text{F\&A}(FrontMult, -MaxMult).$$

Any deletions that access *FrontMult* after this is done will busy-wait until *FrontMult* becomes positive. RaiseMult then increments *FrontMult* by K via an assignment of $f+K$, if f is positive, and via an assignment of K, otherwise.

LowerMult has three input parameters: an item I, a positive integer K, a multiqueue Q; and one output parameter: a nonnegative integer M. If I is not a member of Q, the primitive returns false with the output parameter set to zero. Otherwise, M is set to I's multiplicity. If M exceeds K, LowerMult reduces I's multiplicity by K. If not, LowerMult removes I from the queue by reducing I's multiplicity by $M-1$ and placing a value of nil into I's queue cell. The primitive then returns true. Notice that the removal of an item can be requested by letting K=*MaxMult*.

The above method for removing an item parallels the convention used in chapter 2 (see section 2.1.2 for details). Like deletion in chapter 2 and as explained below, a multiqueue deletion performed on a cell whose item has been removed simply returns nil. Also, as noted in chapter 2, an alternative to replacing removed items with nil values is to compact the multiqueue, but we do not see how to do so without suspending all other queue activity.

To lower I's multiplicity by K or $M-1$, the multiplicity component *Mult* of I's cell, the queue component *Qlower*, and, if I is the front item, the queue component *FrontMult* must be decremented appropriately. Like the increments performed by RaiseMult, these decrements will for the most part be implemented via assignments when the variables are serially accessible and via fetch-and-adds when they are concurrently accessible.

When a cell's multiplicity is lowered by a value K, K items (or more precisely, K copies of the item in the cell) are effectively removed from the queue. Since *Qlower* equals the number of items in the queue, LowerMult performs a TDR of K, and not a fetch-and-add of $-K$, on this counter. Recall that TDR returns true if it performs a fetch-and-add of $-K$ on *Qlower* when *Qlower* is greater than or equal to K, and false otherwise. Thus, if the TDR of K on *Qlower* returns true, the items have been successfully removed from the queue. But it is possible for enough deletions to enter the queue and drive *Qlower* below K before the TDR is performed. Therefore, if TDR returns false, the K items that LowerMult tried to delete from *Qlower* are instead added to a nil item pool represented by the counter *NilPool*.[8] *Qlower* now equals the number of items in the queue plus the number of items in the nil pool. A deletion can return with an item from either one of these sources.

The actions of LowerMult may now be explained in greater detail. To simplify the descriptions, we will assume that TDRs performed on *Qlower* by LowerMult always succeed. This is with the understanding that in the implementation of LowerMult, a failed TDR of K on *Qlower* is followed by an increment by K of the *NilPool* queue component.

LowerMult first acquires exclusive access, via a write request, to an item I's cell c to determine whether I is at the front of the queue and to bar other interior access operations. If I is not at the front, M is set to the cell's multiplicity component *Mult*. If M exceeds K, *Mult* is decremented by K. Otherwise, I is removed, c is assigned a value of nil, and *Mult* is assigned a value of one. In either case, *Qlower* is decremented by the number of copies (K or $M-1$) taken from the cell, a write release is issued, and the primitive returns true.

When I is in the front queue cell c, LowerMult acts like a multi-deletion of size K by attempting to acquire K copies of the front item. Specifically, LowerMult decrements *FrontMult* via a fetch-and-add of $-K$. (Unlike RaiseMult, LowerMult and deletion may concurrently modify *FrontMult* via fetch-and-adds.) If *FrontMult*'s old value f is nonpositive, a final deletion is underway and LowerMult releases c and returns false. If f is greater than K, I's multiplicity is reduced: M is set to c's multiplicity component *Mult* which is then decremented, along with *Qlower*, by K.

[8] Without such a pool, LowerMult would have to request write access to the *entire* queue to prevent deletions from performing TDRs on *Qlower*.

The cell is then released and the primitive returns true. If *f* is positive but not greater than *K*, *I* is removed: At this point, *FrontMult* is nonpositive and, as a result, incoming deletions are blocked. The number of deletions that decremented *FrontMult* when it was positive and that are waiting to access *c* equals *Mult*–*f*. These deletions remain blocked until LowerMult does a write release of *c*. We elect to allow these deletions to reference the front item before proceeding with the removal. In order to do this, these deletions must be able to obtain read access to *c*. If LowerMult did a standard write release, another interior access operation could access the front item ahead of it. Hence, LowerMult becomes a reader of *c* by incrementing *c*'s readers/writers activity counter by *MaxR*–1, instead of by *MaxR*, which is the standard write release increment. This permits the deletions, but not other interior access operations, to access *c*. Once *Mult* equals *f*, the deletions are done. At this point, LowerMult is the last reader of *c*. It sets *M* to *Mult*, marks *I* as dequeued, places a nil value in *c*, assigns *Mult* and *FrontMult* a value of one for this nil value, decrements *Qlower* by *M*–1, does a read release of *c*, and returns true.

A deletion, as usual, first attempts to retrieve an item from the front of the queue. Failing this, it tries to extract one from the nil pool. This process is repeated until the deletion obtains an item from one of these sources. The majority of nil items are thus pushed to the end of the queue and only taken when the queue is empty or a final deletion does not open the next site before the current site is examined by an incoming deletion. The deletion algorithm checks for these nil items by using the version of the deletion site calculation macro given below.

The algorithm for insertion is the same as the one given to support the readers/writers implementation of type one interior access operations. See section 3.1.1.4.1.1 for details.

Seq_LinMQueue = *MultiQueue Supporting Type II IA*
record
    Qlower, Qupper, Qsize, Qfront, Qrear,
    FrontMult, NilPool : **integer**;
    Cell : **array**[0..Qsize–1] **of** CellElement
end

```
CellElement = record
                 Queue_Id : MQueue_Pointer;
                 Mult     : integer;
                 Value    : Object_Pointer;
                 RW_Act   : integer
              end;

MQueue_Pointer = ↑Seq_LinMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
            Cell_Loc : CellElement_Pointer;
            Application_Defined_Attributes
         end

Object_Pointer = ↑Object;

Initially:
   Qlower, Qupper, Qfront, Qrear, FrontMult, NilPool = 0;
   for 0≤j<Qsize,
     Cell[j].Queue_Id              = queue address;
     Cell[j].Mult                  = 0;
     Cell[j].RW_Act                = MaxR;

function Seq_LinMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LinMQueue);
var i : integer;
   with Q do
   if TIR(Qupper, 1, Qsize) then
      i := F&A(Qrear, 1) mod Qsize;
      if i = Qsize−1 then F&A(Qrear, −Qsize) end if;
      Write_Request(Cell[i].RW_Act);
         Item↑.Cell_Loc := address(Cell[i]);
         Cell[i].Value := Item;
         Cell[i].Mult := M;
         if Qfront = i then FrontMult := M end if;
      Write_Release(Cell[i].RW_Act);
      F&A(Qlower, M);
      return true
   else
      return false
   end if
   end with
end function Seq_LinMQInsert;
```

```
function Seq_LinMQDelete (Item : Object_Pointer;
                          Q    : Seq_LinMQueue);
var d, n, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Deletion_Site(d)_V2;
        Read_Request(Cell[d].RW_Act);
            Item := Cell[d].Value;
            if F&A(Cell[d].Mult, −1) = 1 then    final delete
                if Item ≠ nil then Item↑.Cell_Loc := nil end if;
                Open_Next_Cell_Site_RW;   defined in sect. 3.1.1.4.1
                F&A(Qupper, −1)
            end if;
        Read_Release(Cell[d].RW_Act);
        return true
    else
        return false
    end if
    end with
end function Seq_LinMQDelete;


macro Calc_Deletion_Site(d)_V2;
    s := 0;
    repeat
        if FrontMult > 0 then
            s := F&A(FrontMult, −1)
        elseif TDR(NilPool, 1) then   take item from ‘‘nil’’ pool
            Item := nil;
            return true
        end if
    until s > 0;
    d := Qfront;
end macro Calc_Deletion_Site(d)_V2;
```

```
function Seq_LinMQRaiseMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LinMQueue);
var c  : CellElement_Pointer;
    f  : integer;
    with Q do
    Seq_LinMQRaiseMult := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Write_Request(c↑.RW_Act);
            if c = Item↑.Cell_Loc then
                if Item ≠ Cell[Qfront].Value then
                    c↑.Mult := c↑.Mult + K;
                else Item is at front of queue
                    f := F&A(FrontMult, −MaxMult);  stop deletions
                    c↑.Mult := c↑.Mult + K;
                    if f > 0 then f copies left
                        FrontMult := f + K
                    else  all copies taken
                        FrontMult := K
                    end if
                end if;
                F&A(Qlower, K);
                Seq_LinMQRaiseMult := true
            end if;
        Write_Release(c↑.RW_Act)
    end if
    end with
end function Seq_LinMQRaiseMult;
```

```
function Seq_LinMQLowerMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LinMQueue;
                             M    : integer);
var c  : CellElement_Pointer;
    f  : integer;

    with Q do
    Seq_LinMQLowerMult := false;  M := 0;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Write_Request(c↑.RW_Act);
            if c = Item↑.Cell_Loc then
                if Item ≠ Cell[Qfront].Value then
                    M := c↑.Mult;
                    if c↑.Mult > K then lower multiplicity
                        c↑.Mult := c↑.Mult – K;
                        if not TDR(Qlower, K) then
                            F&A(NilPool, K)
                        end if
                    else remove Item
                        Item↑.Cell_Loc := nil;
                        c↑.Mult := 1;
                        c↑.Value := nil;
                        if not TDR(Qlower, M – 1) then
                            F&A(NilPool, M – 1)
                        end if
                    end if;
                    Seq_LinMQLowerMult := true
                else Item is at front of queue
                    if FrontMult > 0 then
                        f := F&A(FrontMult, –K);
                        if f > K then lower multiplicity
                            M := f;
                            c↑.Mult := c↑.Mult – K;
                            if not TDR(Qlower, K) then
                                F&A(NilPool, K)
                            end if;
                            Seq_LinMQLowerMult := true
                        elseif f > 0 then remove item
                            F&A(c↑.RW_Act, MaxR–1); become a reader
                            repeat until c↑.Mult = f;       wait for deletions
                            Item↑.Cell_Loc := nil;  mark as dequeued
                            M := f;   number of copies removed
                            c↑.Mult := 1;
                            c↑.Value := nil;
                            if not TDR(Qlower, M – 1) then
                                F&A(NilPool, M – 1)
                            end if;
                            FrontMult := 1;
                            Read_Release(c↑.RW_Act);
                            Seq_LinMQLowerMult := true;
                            return
```

```
                    end if  f > K
                end if  FrontMult > 0
            end if  Item ≠ Cell[Qfront].Value
        end if;  c = Item↑.Cell_Loc
        Write_Release(c↑.RW_Act)
    end if
    end with
end function Seq_LinMQLowerMult;
```

Notice that LowerMult assigns *FrontMult* a value of one *after* the front item is marked dequeued. If the order of these operations were reversed, a deletion and LowerMult could both obtain and mark the same item as dequeued, which would be incorrect. To illustrate how this could occur, suppose LowerMult sets *FrontMult* to one and then marks the front item *I* dequeued. A deletion arrives, decrements *FrontMult*, copies *I*, decrements the front cell's multiplicity component *Mult*, and marks *I* dequeued as well (as it is a final deletion). Both operations return *I* to their callers, resulting in one too many copies of *I* being in circulation. The problem is that LowerMult is a reader of *I* at the time it assigns a positive value to *FrontMult*. Once this assignment is done, deletions, being readers, can access the front item. All versions of LowerMult presented in this chapter will therefore dequeue an item before permitting deletion activity to resume.

A LowerMult operation that attempts to decrement *Qlower* via a TDR may temporarily make *Qlower* negative if the TDR returns false. Recall that a TDR of *K* on *Qlower* first compares *Qlower* and *K*. If *Qlower* is less that *K*, TDR returns false. Otherwise, it decrements *Qlower* by *K*. If *Qlower*'s old value is greater than or equal to *K*, TDR returns true. If not, TDR increments *Qlower* by *K* and returns false. Between the test and decrement of *Qlower* by a LowerMult *L*'s TDR, a sufficient number of other LowerMults and/or deletions can execute TDRs on *Qlower* to bring the value down to zero. In this case, *L*'s TDR will return false, and *Qlower* will be negative between the TDR's decrement and increment. During this interval, incoming deletions will fail although there are items in the queue. We claim that this is acceptable since the same situation can occur when only deletions and insertions are executed on a concurrently accessible queue or multiqueue. For example, suppose a queue contains one item, i.e., *Qlower*=1. Let $j>1$ deletions perform TDRs concurrently on *Qlower*, temporarily making it equal $1-j$. One TDR succeeds; each of the remaining $j-1$ TDRs, having failed, increments *Qlower* by 1. However, before all $j-1$ increments are done, an insertion and then a deletion are executed. The deletion fails to obtain the newly inserted item since all of the increments to *Qlower* have not completed. To the deletion, the queue appears empty.

The possibility of a LowerMult's TDR on *Qlower* succeeding may be raised by using *m*, the minimum of *Qlower* and *K*, as the decrement. If TDR(*Qlower*, *m*) succeeds, then only $K{-}m$ nil items are added to the pool. If it fails, then *K* nil items are placed in the pool.

3.1.1.4.2.2. A Writer Priority Readers/Writers Implementation

The writer priority readers/writers implementation of RaiseMult and LowerMult does not require incoming deletions to execute explicit readers/writers protocols. Instead, the queue component *FrontMult* and a cell's multiplicity component *Mult* are manipulated by these primitives to acquire write access to the front cell. The technique for doing this is patterned after the one given in the writer priority readers/writers implementation of type one operations (see section 3.1.1.4.1.2). Also, like the latter implementation, a final deletion, after decrementing a cell's multiplicity, obtains exclusive access to a cell before dequeueing an item. The data structures differ from those of the preceding section only in that a binary semaphore (which is initially open), instead of a readers/writers activity counter, is associated with each cell.

RaiseMult, in order to increment an item *I*'s multiplicity by a value *K*, first closes the binary semaphore associated with *I's* cell *c*. Then, if *c* is not the front cell, RaiseMult has exclusive access to *I* and therefore proceeds as in the above readers/writers implementation. However, if *c* is the front cell, RaiseMult makes *FrontMult* negative (via a fetch-and-add of –*MaxMult*) to block incoming deletions. If *FrontMult*'s old value *f* is found to be nonpositive, the final deletion of *c* has already arrived, and within a finite time will be busy-waiting on *c*'s semaphore. Therefore, RaiseMult opens the semaphore and returns false. If *f* is positive, the final deletion has not yet arrived and $Mult{-}f$ non-final deletions are active, where *Mult* is *c*'s multiplicity component. As each deletion completes, it will decrement *Mult* by one via a fetch-and-add. Hence, RaiseMult increments *Mult* by *K* via a fetch-and-add, and not by an assignment of $Mult{+}K$ as is done in the readers/writers implementation.[9] *Qlower* and *FrontMult* are incremented, respectively, via a fetch-and-add and an assignment as in the readers/writers implementation.

[9] To correctly perform the assignment, RaiseMult would have to busy-wait until the deletions were done (i.e., when *Mult* equaled *f*). Doing the increment with a fetch-and-add avoids this wait.

LowerMult obtains exclusive access to items in nonfront cells by closing the associated cell semaphores and proceeding as in the previous section. It differs from its readers/writers counterpart primarily in how it deals with an item in a front cell. We will therefore restrict ourselves to describing this part of the algorithm.

Like RaiseMult, LowerMult, when accessing a front cell *c*, blocks incoming deletions by making *FrontMult* negative via a fetch-and-add of −*MaxMult*. If *FrontMult*'s old value *f* is found to be nonpositive, the primitive returns false as a final deletion has arrived. If *f* is greater than *K*, the value by which *c*'s multiplicity is to be decremented, LowerMult decrements *FrontMult* and *c*'s multiplicity via an assignment and a fetch-and-add (just as RaiseMult uses an assignment and a fetch-and-add to increment these two components above). If *f* is positive and not greater than *K*, LowerMult busy-waits until non-final deletion activity has ceased (i.e., when *c*'s multiplicity equals *f*) and then removes *c*'s item as in the readers/writers implementation.

```
Seq_LinMQueue =    MultiQueue Supporting Type II IA
                   record
                      Qlower, Qupper, Qsize, Qfront, Qrear,
                      FrontMult, NilPool : integer;
                      Cell : array[0..Qsize–1] of CellElement
                   end


CellElement = record
                   Queue_Id     : MQueue_Pointer;
                   Mult         : integer;
                   Value        : Object_Pointer;
                   Access_Sem   : Weak_Semaphore
              end;

MQueue_Pointer = ↑Seq_LinMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
              Cell_Loc : CellElement_Pointer;
              Application_Defined_Attributes
         end

Object_Pointer = ↑Object;
```

```
Initially:
    Qlower, Qupper, Qfront, Qrear, FrontMult, NilPool = 0;
    for 0≤j<Qsize,
      Cell[j].Queue_Id                    = queue address;
      Cell[j].Mult                        = 0;
      Cell[j].Access_Sem                  = 1;

function Seq_LinMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LinMQueue);
var i : integer;
    with Q do
    if TIR(Qupper, 1, Qsize) then
        i := F&A(Qrear, 1) mod Qsize;
        if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
        Weak_P(Cell[i].Access_Sem);
            Item↑.Cell_Loc := address(Cell[i]);
            Cell[i].Value := Item;
            Cell[i].Mult := M;
            if Qfront = i then FrontMult := M end if;
        Weak_V(Cell[i].Access_Sem);
        F&A(Qlower, M);
        return true
    else
        return false
    end if
    end with
end function Seq_LinMQInsert;
```

```
function Seq_LinMQDelete (Item : Object_Pointer;
                          Q    : Seq_LinMQueue);
var d, n, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Deletion_Site(d)_V2;   defined in sect. 3.1.1.4.2.1
        Item := Cell[d].Value;
        if F&A(Cell[d].Mult, −1) = 1 then    final delete
            if Item ≠ nil then
                Weak_P(Cell[d].Access_Sem);
                    Item↑.Cell_Loc := nil;
                Weak_V(Cell[d].Access_Sem)
            end if;
            Open_Next_Cell_Site_PV;   defined in sect. 3.1.1.2
            F&A(Qupper, −1)
        end if;
        return true
    else
        return false
    end if
    end with
end function Seq_LinMQDelete;
```

```
function Seq_LinMQRaiseMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LinMQueue);
var c  : CellElement_Pointer;
    f  : integer;

    with Q do
    Seq_LinMQRaiseMult := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Weak_P(c↑.Access_Sem);
            if c = Item↑.Cell_Loc then
                if Item ≠ Cell[Qfront].Value then
                    c↑.Mult := c↑.Mult + K;
                    F&A(Qlower, K);
                    Seq_LinMQRaiseMult := true
                else Item is at front of queue
                    if FrontMult > 0 then
                        f := F&A(FrontMult, −MaxMult);  stop deletions
                        if f > 0 then f copies left
                            F&A(c↑.Mult, K);
                            FrontMult := f + K;
                            F&A(Qlower, K);
                            Seq_LinMQRaiseMult := true
                        end if
                    end if
                end if
            end if;
        Weak_V(c↑.Access_Sem)
    end if
    end with
end function Seq_LinMQRaiseMult;
```

```
function Seq_LinMQLowerMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LinMQueue;
                             M    : integer);
var c  : CellElement_Pointer;
    f  : integer;

    with Q do
    Seq_LinMQLowerMult := false;  M := 0;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Weak_P(c↑.Access_Sem);
            if c = Item↑.Cell_Loc then
                if Item ≠ Cell[Qfront].Value then
                    M := c↑.Mult;
                    if c↑.Mult > K then lower multiplicity
                        c↑.Mult := c↑.Mult – K;
                        if not TDR(Qlower, K) then
                            F&A(NilPool, K)
                        end if
                    else remove Item
                        Item↑.Cell_Loc := nil;
                        c↑.Mult := 1;
                        c↑.Value := nil;
                        if not TDR(Qlower, M – 1) then
                            F&A(NilPool, M – 1)
                        end if
                    end if;
                    Seq_LinMQLowerMult := true
                else Item is at front of queue
                    if FrontMult > 0 then
                        f := F&A(FrontMult, –MaxMult);  stop deletions
                        if f > K then lower multiplicity
                            M := f;
                            F&A(c↑.Mult, –K);
                            if not TDR(Qlower, K) then
                                F&A(NilPool, K)
                            end if;
                            FrontMult := f – K;
                            Seq_LinMQLowerMult := true
                        else
                            if f > 0 then remove item
                                repeat until c↑.Mult = f;
                                Item↑.Cell_Loc := nil;
                                M := f;   number of copies removed
                                c↑.Mult := 1;
                                c↑.Value := nil;
                                if not TDR(Qlower, M – 1) then
                                    F&A(NilPool, M – 1)
                                end if;
                                FrontMult := 1;
                                Seq_LinMQLowerMult := true
                            end if  f > 0
```

```
                    end if  f > K
                 end if  FrontMult > 0
              end if  Item ≠ Cell[Qfront].Value
           end if;  c = Item↑.Cell_Loc
        Weak_V(c↑.Access_Sem)
     end if
     end with
  end function Seq_LinMQLowerMult;
```

Notice that LowerMult decrements *FrontMult* by *MaxMult* in this implementation and by *K* in the readers/writers implementation. The *MaxMult* decrement is necessary to avoid a race condition between LowerMult and deletion that could permanently block the latter. To illustrate how this could occur, suppose LowerMult does use a decrement of *K*, instead of *MaxMult*, to lower the front multiplicity component *FrontMult*. Let *FrontMult* have a value of $f>K$ when LowerMult decrements it by *K*. Then, before LowerMult decrements the front cell's multiplicity component *Mult* by *K*, $f-K$ deletions arrive, decrement *FrontMult* (making it nonpositive), copy the front item, and decrement *Mult*. No additional incoming deletions can access the front cell until *FrontMult* is assigned a positive value. LowerMult then decrements *Mult* by *K*. Now there can be no final or subsequent deletions: *FrontMult* is nonpositive and there are no active deletions.

Decrementing *FrontMult* by *MaxMult* prevents deletions from racing ahead of LowerMult and prematurely decrementing the front cell's multiplicity component. Alternatively, LowerMult could decrement *FrontMult* by *K* if it opened the next deletion site upon finding that the cell's multiplicity was zero. This race can not occur in the readers/writers implementation as LowerMult first obtains write permission to the front cell, which blocks deletions from decrementing the front item's multiplicity component, before decrementing *FrontMult*.

3.1.2. Linked List Allocation

A linear multiqueue of unbounded size is obtained by replacing the array of *Qsize* cells with an array of *Qsize* serially accessible linked lists. This structure supports the same levels of concurrent activity as the sequentially allocated linear multiqueue. That is, a maximum of $Qsize-1$ insertions (one at each nonfront list) and *M* deletions (where *M* is the multiplicity of the front item) may be performed in parallel.

Insertion and deletion algorithms are given first, followed by both types of IA operations. All implementations use local unfair list protocols. If starvation-free algorithms are desired, fair versions of these protocols may be substituted. Also, as noted in section 3.1.1.2, an incoming deletion may starve if the site calculation is not a critical code section.

3.1.2.1. Insertion and Deletion

The linked allocated linear multiqueue structure consists of an array of lists *List*[0..*Qsize*–1]. Associated with each list is a counter *HeadMult* that defines the multiplicity of the list's head item. Similarly, each item includes a component *Mult* that defines its multiplicity while it occupies a nonhead list position. The *HeadMult* component is not essential. However, as noted at the end of this section, its inclusion improves the performance of deletion.

Insertions and deletions follow the corresponding algorithms for sequentially allocated multiqueues given in section 3.1.1.2 with linked list operations being substituted for cell operations. An item with multiplicity *M* is inserted at site *i* by appending it to the *i*th list. If the list is empty, its head multiplicity component *HeadMult* is initialized to *M* and, if the list is also the current deletion site (*i*=*Qfront*), *FrontMult* is initialized to *M*, enabling deletions to commence.

A major difference between the sequential and linked list versions is that the first limits the number of active insertions per site to one whereas the second places no bounds on the number of concurrent insertions that may be assigned to a site. The reason, of course, is that a cell can only accommodate a single item at a time whereas a linked list has an essentially infinite capacity. Since multiple insertions on a linked allocated multiqueue may contend for the same site, each insertion requires exclusive access to its site to not only block out final deletions but other insertions as well.

A final deletion obtains exclusive access to a list *d* to dequeue *d*'s head item. It also obtains exclusive access to the following list *e* to open the next deletion site.[10] Exclusive access to both lists may be obtained before either operation is done, or list *e* and then list *d* may be exclusively accessed, i.e., the deletion exclusively accesses only one list at a time. The remaining alternative for accessing the lists is to operate on *d* and then *e*. This sequence would be incorrect as it could allow more items to be deleted than had been inserted. To illustrate this point, assume this last sequence is followed, and a final deletion is executed on a list *d* that contains one item *I* with multiplicity one.

[10] In the deletion algorithm below, we isolate this section as the macro Open_Next_List_Site. In the degenerate case when the number of lists is one (i.e., *Qsize*=1), the macro should not be executed.

(1) The final deletion exclusively accesses *d* and dequeues *I*. Before the deletion opens up the next site, by updating *Qfront* and *FrontMult*:

(1.1) An insertion of an item *J* with multiplicity M_J exclusively accesses *d*, adds *J* to *d*, and initializes *FrontMult* to M_J since the queue front pointer *Qfront* is set to *d* and *d* is empty.

(1.2) $N \le M_J$ deletions decrement *FrontMult*.

(2) The final deletion exclusively accesses the next list *e* and opens up the next deletion site by setting *Qfront* to *e* and assigning *FrontMult* the multiplicity M_K of *e*'s head item *K*.

(3) The *N* deletions of (1.2) access *Qfront* and copy *K*.

(4) M_K deletions decrement *FrontMult*, access *Qfront*, and copy *K*.

Hence, *N* too many copies of *K* are made.

We elect to have final deletions open sites and then dequeue items using the second list acquisition method cited above, namely, to exclusively access only one list at a time, since this does not reduce queue concurrency.[11] This method has the added benefit of allowing deletions that arrive while a final deletion is in progress to proceed to the next list (i.e., site) without having to wait for the head item on the current list to be dequeued. There does exist the possibility that deletions will then wraparound the queue and return to a list where a final deletion is still in progress, but this is easily detected and resolved: Between the time a final deletion *D* on list *d* advances the deletion site to the next list *e* and accesses list *d*, final deletions on lists *e* through *c* may be performed, where *c* precedes *d*. Let *C* be the final deletion on list *c*. Both *C* and *D* will contend for list *d*. If the list is first acquired by *D*, then there is no deletion site wraparound. However, if *C* references list *d* ahead of *D*, then the deletion site will be moved to list *d* and *FrontMult* initialized to the head multiplicity of *d*. *FrontMult*, however, will be set to zero by *C* since *D* has not yet had the opportunity to dequeue its item and update the list's head multiplicity. Thus, no deletions may be performed on this site until *D* completes. After *D* acquires list *d*, it initializes *FrontMult* for a second time if queue wraparound has occurred (*Qfront*=*d*).

[11] The first method will be selected to support interior access. See section 3.1.2.3.1.

The linked queue data structures are similar to those of chapter 2. Each queued item *I* has a *Next* component to link the item to its successor on the list. Following Knuth [68], the tail of an empty list points to the list header, and a reference to the *Next* (or *Mult*) component of the list header is interpreted as a reference to the *Head* (or *HeadMult*) component.

```
Linked_LinMQueue =      record    Linked Allocated Linear MultiQueue
                            Qlower, Qsize, Qfront, Qrear,
                            FrontMult : integer;
                            List : array[0..Qsize–1] of ListElement
                        end;

ListElement = record
                   HeadMult    : integer;
                   Head        : Object_Pointer;
                   Tail        : Pointer to ListElement or Object;
                   Access_Sem  : Weak_Semaphore
              end;

List_Pointer = ↑ListElement;

Object = record
              Mult        : integer;
              Next        : Object_Pointer;
              Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
    Qlower, Qfront, Qrear, FrontMult = 0;
    for 0≤j<Qsize,
      List[j].HeadMult   = 0;
      List[j].Tail       = address(List[j]);
      List[j].Access_Sem = 1;
```

```
procedure Linked_LinMQInsert (Item : Object_Pointer;
                              M    : integer;   Item's Multiplicity
                              Q    : Linked_LinMQueue);
var i : integer;
   with Q do
   i := F&A(Qrear, 1) mod Qsize;
   if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
   Item↑.Next := nil;
   Item↑.Mult := M;
   Weak_P(List[i].Access_Sem);
      List[i].Tail↑.Next := Item;
      List[i].Tail := Item;
      if List[i].Head = Item then           list was empty
         List[i].HeadMult  := M;            promote item's multiplicity
         if Qfront = i then FrontMult := M end if
      end if;
   Weak_V(List[i].Access_Sem);
   F&A(Qlower, M)
   end with
end procedure Linked_LinMQInsert;


function Linked_LinMQDelete (Item : Object_Pointer;
                             Q    : Linked_LinMQueue);
var d, n, s : integer;
   with Q do
   if TDR(Qlower, 1)   then
      Calc_Deletion_Site(d)_V1;   defined in sect. 3.1.1.2
      Item := List[d].Head;
      if F&A(List[d].HeadMult, –1) = 1 then   final delete
         Open_Next_List_Site_PV;
         Weak_P(List[d].Access_Sem);
            List[d].Head := Item↑.Next;
            List[d].HeadMult := Item↑.Next↑.Mult;  new multiplicity
            if List[d].Head = nil then
               List[d].Tail := address(List[d])
            elseif Qfront = d  then   queue wraparound
               FrontMult := Item↑.Next↑.Mult
            end if;
         Weak_V(List[d].Access_Sem)
      end if;
      return true
   else
      return false
   end if
   end with
end function Linked_LinMQDelete;
```

```
macro Open_Next_List_Site_PV;
    n := (d + 1) mod Qsize;
    Weak_P(List[n].Access_Sem);
        Qfront := n;
        FrontMult := List[n].HeadMult;
    Weak_V(List[n].Access_Sem);
end macro Open_Next_List_Site_PV;
```

When an item is at the front of a list, the list component *HeadMult*, and not the item component *Mult*, defines the item's multiplicity. This improves deletion at the expense of insertion, which is an acceptable tradeoff as deletions are the dominant activity (except when item multiplicities are always one, in which case neither operation dominates). Deletions (including final deletions when opening up deletion sites) reference only a list's *HeadMult* component to determine the multiplicity of the list's front item.[12] If this component were omitted, deletions would have to reference a list's header component *Head* to locate the front item *I* and then *I*'s *Mult* component to determine the multiplicity. Insertion is more expensive since it must initialize a list's *HeadMult* component when the list is empty.

3.1.2.2. Multi-Deletion

Multi-deletion from a linked allocated multiqueue is analogous to the same operation performed on a sequentially allocated multiqueue (see section 3.1.1.3). The operation keeps visiting deletion sites until it obtains *M* items.

[12] The presence of this component therefore simplifies deletion when LowerMult is supported: The multiplicity of a nil (i.e., removed) item that is at the head of a list is recorded in the list's *HeadMult* component.

```
function Linked_LinMulti-Delete (Table : array[1..M] of Object_Pointer;
                                 M     : integer;
                                 Q     : Linked_LinMQueue);
var d, g, m, n, s : integer;
    item          : Object_Pointer;
   with Q do
   if TDR(Qlower, M)   then
      m := M;
      repeat    keep taking items from the Qfront list item until satisfied
         s := 0;
         repeat
            if FrontMult > 0 then s := F&A(FrontMult, –m) end if
         until s > 0;
         d := Qfront;
         g := s min m;   number of copies to make of front item
         Table[m–g+1..m] := List[d].Head
         m := m – g;     remaining number to obtain
         if F&A(List[d].HeadMult, –g) = g then    final multi-delete
            Open_Next_List_Site_PV;   defined in sect. 3.1.2.1
            item := List[d].Head;
            Weak_P(List[d].Access_Sem);
               List[d].Head := item↑.Next;
               List[d].HeadMult := item↑.Next↑.Mult;  new multiplicity
               if List[d].Head = nil then
                  List[d].Tail := address(List[d])
               elseif Qfront = d then   queue wraparound
                  FrontMult := item↑.Next↑.Mult
               end if;
            Weak_V(List[d].Access_Sem)
         end if
      until m = 0;
      return true
   else
      return false
   end if
   end with
end function Linked_LinMulti-Delete;
```

3.1.2.3. Interior Access

This section presents type one and type two interior access operations. For each one, readers/writers and writer priority readers/writers implementations are given.

3.1.2.3.1. Type One

3.1.2.3.1.1. A Readers/Writers Implementation

Accessing an interior item of a linked allocated multiqueue can be implemented by following the same readers/writers strategy as employed for sequentially allocated multiqueues (see section 3.1.1.4). For each list, IA operations, insertions and final deletions are writers, and incoming deletions are readers. The IA procedure is identical to the one given in section 3.1.1.4.1 with lists substituted for cells.

The linked allocated multiqueue insertion and deletion algorithms must mark an item as queued or dequeued when it is put on or taken off a list to support interior access. Furthermore, readers/writers code must bracket a deletion's reference to the front list.

It is perhaps surprising that a final deletion does not have to *explicitly* request write access to a list in order to dequeue an item. However, every deletion obtains read access to the front list and holds it until done. When the final deletion is performed, it is the final reader of the front item and no insertions or interior access operations may access the list since they are writers.

In this implementation, a final deletion opens the next deletion site while it is a reader of the current site. That is, it obtains exclusive access to both sites' lists before proceeding. Therefore, the queue front pointer *Qfront* cannot wraparound the queue to where a final deletion is in progress (cf. section 3.1.2.1). If we wished to avoid a final deletion having exclusive access to two sites concurrently, the final deletion would have to release the current site *s*, open up the next site, and then re-acquire *s*.

```
Linked_LinMQueue = MultiQueue Supporting Type I IA
                   record
                      Qlower, Qsize, Qfront, Qrear, FrontMult : integer;
                      List : array[0..Qsize] of ListElement
                   end;

ListElement = record
                 Queue_Id : MQueue_Pointer;
                 HeadMult : integer;
                 Head     : Object_Pointer;
                 Tail     : Pointer to ListElement or Object;
                 RW_Act   : integer
              end;

MQueue_Pointer = ↑Linked_LinMQueue;
```

```
ListElement_Pointer = ↑ListElement;

Object = record
            List_Loc : ListElement_Pointer;
            Mult     : integer;
            Next     : Object_Pointer;
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear, FrontMult = 0;
   for 0≤j<Qsize,
     List[j].Queue_Id  = address of queue;
     List[j].HeadMult = 0;
     List[j].Tail      = address(List[j]);
     List[j].RW_Act   = MaxR;

    procedure Linked_LinMQInsert (Item : Object_Pointer;
                                  M    : integer;
                                  Q    : Linked_LinMQueue);
    var i : integer;
       with Q do
       i := F&A(Qrear, 1) mod Qsize;
       if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
       Item↑.Next := nil;
       Item↑.Mult := M;
       Write_Request(List[i].RW_Act);
          Item↑.List_Loc := address(List[i]);
          List[i].Tail↑.Next := Item;
          List[i].Tail := Item;
          if List[i].Head = Item then
             List[i].HeadMult  := M;   promote item's multiplicity
             if Qfront = i then FrontMult := M end if
          end if;
       Write_Release(List[i].RW_Act);
       F&A(Qlower, M)
       end with
    end procedure Linked_LinMQInsert;
```

```
function Linked_LinMQDelete (Item : Object_Pointer;
                             Q    : Linked_LinMQueue);
var d, m, n, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Deletion_Site(d)_V1;   defined in sect. 3.1.1.2
        Read_Request(List[d].RW_Act);
            Item := List[d].Head;
            if F&A(List[d].HeadMult, −1) = 1 then    final delete
                Open_Next_List_Site_RW;
                List[d].Head := Item↑.Next;
                List[d].HeadMult := Item↑.Next↑.Mult;
                Item↑.List_Loc := nil;
                if List[d].Head = nil then
                    List[d].Tail := address(List[d])
                end if
            end if;
        Read_Release(List[d].RW_Act);
        return true
    else
        return false
    end if
    end with
end function Linked_LinMQDelete;


macro Open_Next_List_Site_RW;
    n := (d + 1) mod Qsize;
    Write_Request(List[n].RW_Act);
        Qfront := n;
        FrontMult := List[n].HeadMult;
    Write_Release(List[n].RW_Act);
end macro Open_Next_List_Site_RW;
```

```
function Linked_LinMQIA_TypeI (Item : Object_Pointer;
                               Q    : Linked_LinMQueue;
                               ϕ    : function);
var h : ListElement_Pointer;
    with Q do
    Linked_LinMQIA_TypeI := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Write_Request(h↑.RW_Act);
            if h = Item↑.List_Loc then
                ϕ(Item);        access Item
                Linked_LinMQIA_TypeI := true
            end if;
        Write_Release(h↑.RW_Act);
    end if
    end with
end function Linked_LinMQIA_TypeI;
```

3.1.2.3.1.2. A Writer Priority Readers/Writers Implementation

The writer priority readers/writers implementation of type one operations is patterned after its sequential counterpart with lists substituted for cells. Briefly, deletions and interior access operations are coordinated through operations on the queue component *FrontMult* and the front list's multiplicity component *HeadMult*, not through explicit readers/writers protocols.

In this implementation, a final deletion first obtains exclusive access to both the front list *L* and the next list *N*. It then opens up the next deletion site, releases *N*, dequeues the head item of *L*, and releases *L*. Thus, while the final deletion is in progress, the deletion site may not wrap back around to *L*.

Recall that when interior access is not supported, a final deletion on a list *L* first accesses the next list *N* to open the next site and then accesses *L* to dequeue the item. Now, in the presence of interior access, this same sequence could cause an IA operation and deletions to deadlock as illustrated in the following scenario (which is similar to the deadlock scenario given in section 3.1.1.4.1.2):

(1) An interior access operation locks the front list *L*.

(2) A final deletion of *L*'s head item *I* is begun. The deletion decrements *L*'s head multiplicity component *HeadMult* down to zero and opens up the next site, initializing *FrontMult* to the new site's multiplicity *M*, which we assume is greater than 1. The deletion then waits to lock *L* to dequeue *I*.

(3) The interior access operation makes *FrontMult* negative to block further deletion activity. It then waits for *L*'s *HeadMult* component to equal *FrontMult*'s old value *M*, indicating that there are no deletions reading *I*. The interior access operation and deletion deadlock.

The data structures for this implementation differ from those of the preceding section only in that binary semaphores (which are initially open) and P/V protocols replace readers/writers activity counters and protocols.

```
Linked_LinMQueue = MultiQueue Supporting Type I IA
                   record
                      Qlower, Qsize, Qfront, Qrear, FrontMult : integer;
                      List : array[0..Qsize] of ListElement
                   end;

ListElement = record
                   Queue_Id    : MQueue_Pointer;
                   HeadMult    : integer;
                   Head        : Object_Pointer;
                   Tail        : Pointer to ListElement or Object;
                   Access_Sem : Weak_Semaphore
              end;

MQueue_Pointer = ↑Linked_LinMQueue;

ListElement_Pointer = ↑ListElement;

Object = record
              List_Loc : ListElement_Pointer;
              Mult     : integer;
              Next     : Object_Pointer;
              Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear, FrontMult = 0;
   for 0≤j<Qsize,
     List[j].Queue_Id     = address of queue;
     List[j].HeadMult     = 0;
     List[j].Tail         = address(List[j]);
     List[j].Access_Sem = 1;
```

```
procedure Linked_LinMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LinMQueue);
var i : integer;
   with Q do
   i := F&A(Qrear, 1) mod Qsize;
   if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
   Item↑.Next := nil;
   Item↑.Mult := M;
   Weak_P(List[i].Access_Sem);
      Item↑.List_Loc := address(List[i]);
      List[i].Tail↑.Next := Item;
      List[i].Tail := Item;
      if List[i].Head = Item then
         List[i].HeadMult  := M;   promote item's multiplicity
         if Qfront = i then FrontMult := M end if
      end if;
   Weak_V(List[i].Access_Sem);
   F&A(Qlower, M)
   end with
end procedure Linked_LinMQInsert;


function Linked_LinMQDelete (Item : Object_Pointer;
                             Q    : Linked_LinMQueue);
var d, m, n, s : integer;
   with Q do
   if TDR(Qlower, 1) then
      Calc_Deletion_Site(d)_V1;   defined in sect. 3.1.1.2
      Item := List[d].Head;
      if F&A(List[d].HeadMult, –1) = 1 then    final delete
         Weak_P(List[d].Access_Sem);
            Open_Next_List_Site_PV;   defined in sect. 3.1.2.1
            List[d].Head := Item↑.Next;
            List[d].HeadMult := Item↑.Next↑.Mult;
            Item↑.List_Loc := nil;  dequeue Item
            if List[d].Head = nil then
               List[d].Tail := address(List[d])
            end if
         Weak_V(List[d].Access_Sem)
      end if;
      return true
   else
      return false
   end if
   end with
end function Linked_LinMQDelete;
```

```
function Linked_LinMQIA_TypeI (Item : Object_Pointer;
                               Q    : Seq_LinMQueue;
                               ϕ    : function);
var h  : ListElement_Pointer;
    f  : integer;
    with Q do
    Linked_LinMQIA_TypeI := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if h = Item↑.List_Loc then
                if Item ≠ List[Qfront].Head then
                    ϕ(Item);                              access Item
                    Linked_LinMQIA_TypeI := true
                else  Item is at front of queue
                    if FrontMult > 0 then
                        f := F&A(FrontMult, −MaxMult);    block deletions
                        if f > 0 then                     f deletions to be done
                            repeat until h↑.Mult = f;     wait for deletions
                            ϕ(Item);                      access Item
                            FrontMult := f;               restore FrontMult
                            Linked_LinMQIA_TypeI := true
                        end if
                    end if
                end if
            end if;
        Weak_V(h↑.Access_Sem)
    end if
    end with
end function Linked_LinMQIA_TypeI;
```

3.1.2.3.2. Type Two

3.1.2.3.2.1. A Readers/Writers Implementation

The linked allocated readers/writers versions of RaiseMult and LowerMult closely parallel their sequential counterparts with lists substituted for cells. Now, a primitive's actions are based on whether an item is positioned at the front of the queue, the head of a nonfront list, or the interior of a list.

RaiseMult and LowerMult increment and decrement, respectively, an item *I*'s multiplicity. Again, to aid our understanding of how these updates are done recall that when *I* is an interior item of a list *L*, *I*'s multiplicity is defined by its multiplicity component *Mult*. When *I* is at the head of *L*, its multiplicity is defined by the *HeadMult* component of *L*. Finally, when *I* is at the front of the queue, its multiplicity is defined by the *FrontMult* component of the multiqueue and the *HeadMult* component of the front list, which represent lower and upper bounds on the number of copies of *I* that remain to be deleted.

RaiseMult increments an item *I*'s multiplicity by a value *K* as follows: If *I* is an interior list item, *I*'s multiplicity component *Mult* and *Qlower* are incremented by *K*; if *I* is the head item of a nonfront list, the list's head multiplicity component *HeadMult* and *Qlower* are incremented by *K*; finally, if the item is the front item of the queue, the front list's head multiplicity component *HeadMult*, *Qlower*, and *FrontMult* are all incremented by *K*. The remaining details of the RaiseMult algorithm, including how the above increments are performed, follow directly from the sequential implementation.

LowerMult decrements an item's multiplicity by a value *K* in a symmetrical manner to RaiseMult. The details on performing these operations follow directly from the corresponding sequential implementation of LowerMult. If the item's multiplicity is not greater than *K*, the item is removed.

Removing an item *I* with multiplicity *M* from the interior of a linked allocated multiqueue requires that missing item counts (see section 2.2.3.2) be kept to ensure the FIFO queue property: When an item is removed, its premature deletion is recorded by incrementing its predecessor's missing item count by one. This accounts for only one of the *M* copies of *I*. *Qlower* is decremented by M–1 to account for the others. As in the sequential version, if it turns out that *Qlower* cannot be decremented (via a TDR) due to deletion activity, the M–1 copies are added to a pool of nil items. See section 3.1.1.4.2.1 for a justification of this protocol.

The specific actions associated with the removal of an item *I* with multiplicity *M* depend upon *I*'s location within a list *L*. If *I* is an interior item, i.e., *I* is not the head item of *L* or *L*'s missing item count is positive, *I*'s predecessor's missing item count is incremented by *I*'s missing item count plus one (to account for one copy of *I* itself). If *I* is the head item of a nonfront list, i.e., *I* is the head of a nonfront list *L* and *L*'s missing item count is zero, *L*'s head multiplicity component *HeadMult* is changed from *M* to 1 to limit the number of future deletions performed on the site to one and *L*'s missing item count is increased by *I*'s plus one. If *I* is the front item of the queue, i.e., *I* is the head of a front list *L* and *L*'s missing item count is zero, then *L*'s *HeadMult* component and missing item count are modified as described in the preceding case. In addition, *FrontMult* is assigned a value of one to be consistent with this substitution. In each case, after the above operations have been performed, the remaining M–1 copies of *I* are deducted from

Qlower (or added to a nil pool if *Qlower* is less that $M-1$), and *I* is dequeued.

An insertion of an item *I* with multiplicity *M* into a list *L* initializes *L*'s head multiplicity component to *M* if *L* is empty. The introduction of removal however alters the test for empty lists: *L* is empty when there are no items linked to its header *and* its missing item count is zero. If *L*'s missing item count has a positive value *P*, then *P* distinct items with multiplicity have been removed from *L* and have migrated to the head of the list. *P* deletions must visit *L* before a copy of the first item linked to *L*'s header can be made.

A final deletion's actions are dependent upon whether the head item *I* of a list *L* has been removed, i.e., is represented by a nil item. If *L*'s missing item count is zero, this has not occurred. In this case, *I* is dequeued and its successor *J* on *L* is moved to the front of the list. If *I*'s missing item count is zero, *J* is the new head item and *L*'s head multiplicity is set to *J*'s. On the other hand, if *I*'s missing item count has a positive value *P*, *P* nil (or ‘‘removed’’) items are at the front of *L* ahead of *J*. The head multiplicity of *L* is then set to 1 for the first of these nil items.

If a final deletion finds that list *L*'s missing item count is positive, then the front item has been removed. In response, the final deletion decrements the missing item count by one for the deletion of a nil item. If the count remains positive, the head multiplicity of *L* is set to one for the next nil item. Otherwise, the head multiplicity is set to the multiplicity of the first item on the list.

The linked allocated multiqueue data structure includes a nil pool *NilPool* for the queue and a readers/writers activity counter *RW_Act* for each doubly linked list. In addition, a missing item count *#Missing* is associated with each item and list header. Each queued item *I* has components *Next* and *Prev* to link the item to its successor and predecessor on the list. Following Knuth [68], the head and tail of an empty list point to the list header, and a reference to the *Next* or *Prev* (or *Mult*) component of the list header is interpreted as a reference to the *Head* or *Tail* (or *HeadMult*) component.

Note in the implementation below that after LowerMult becomes the last reader of a front list *F*,[13] it decrements *F*'s readers/writers activity counter by 1–*MaxR* to then become a writer.

[13] Recall from section 3.1.1.4.2.1 that LowerMult is originally a writer of an item's site. If an item is at the front of the queue and is to be removed, LowerMult becomes a reader to allow deletions that are currently in progress to complete. LowerMult prevents new deletions (readers) from accessing the site by making the queue component *FrontMult* very negative.

This simplifies the code, allowing LowerMult to always perform write releases of lists.

```
Linked_LinMQueue = MultiQueue Supporting Type II IA
                   record
                      Qlower, Qsize, Qfront, Qrear,
                      FrontMult, NilPool : integer;
                      List : array[0..Qsize–1] of ListElement
                   end;

ListElement = record
                 Queue_Id  : MQueue_Pointer;
                 HeadMult  : integer;
                 Head,
                 Tail      : Pointer to ListElement or Object;
                 #Missing  : integer;
                 RW_Act    : integer
              end;

MQueue_Pointer = ↑Linked_LinMQueue;

ListElement_Pointer = ↑ListElement;

Object = record
            List_Loc : ListElement_Pointer;
            Mult     : integer;
            Next,
            Prev     : Pointer to ListElement or Object;
            #Missing : integer;
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear, FrontMult, NilPool = 0;
   for 0≤j<Qsize,
     List[j].Queue_Id      = address of queue;
     List[j].HeadMult      = 0;
     List[j].Head          = address(List[j]);
     List[j].Tail          = address(List[j]);
     List[j].#Missing      = 0;
     List[j].RW_Act        = MaxR;
```

```
procedure Linked_LinMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LinMQueue);
var i : integer;
    with Q do
    i := F&A(Qrear, 1) mod Qsize;
    if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
    Item↑.Next := address(List[i]);
    Item↑.Mult := M;
    Item↑.#Missing := 0;
    Write_Request(List[i].RW_Act);
        Item↑.List_Loc := address(List[i]);
        Item↑.Prev := List[i].Tail;
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
        if List[i].Head = Item and List[i].#Missing = 0 then
            List[i].HeadMult := M;   promote item's multiplicity
            if Qfront = i then FrontMult := M end if
        end if;
    Write_Release(List[i].RW_Act);
    F&A(Qlower, M)
    end with
end procedure Linked_LinMQInsert;
```

```
function Linked_LinMQDelete (Item : Object_Pointer;
                             Q    : Linked_LinMQueue);
var d, m, n, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Deletion_Site(d)_V2;   defined in sect. 3.1.1.4.2.1
        Read_Request(List[d].RW_Act);
            Item := List[d].Head;
            if F&A(List[d].HeadMult, −1) = 1 then    final delete
                Open_Next_List_Site_RW;   defined in sect. 3.1.2.3.1.1
                if List[d].#Missing = 0 then     dequeue item
                    List[d].Head := Item↑.Next;
                    List[d].Head↑.Prev := address(List[d]);
                    List[d].#Missing := Item↑.#Missing;
                    m := Item↑.Next↑.Mult;
                    Item↑.List_Loc := nil
                else     head of list is a ‘‘missing item’’
                    List[d].#Missing := List[d].#Missing − 1;
                    m := Item↑.Mult;
                    Item := nil
                end if;
                if List[d].#Missing > 0 then
                    List[d].HeadMult := 1   mult. of ‘‘missing item’’
                else
                    List[d].HeadMult := m   mult. of head list item
                end if
            end if;
        Read_Release(List[d].RW_Act);
        return true
    else
        return false
    end if
    end with
end function Linked_LinMQDelete;
```

```
function Linked_LinMQRaiseMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LinMQueue);
var h : ListElement_Pointer;
    f : integer;

    with Q do
    Linked_LinMQRaiseMult := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Write_Request(h↑.RW_Act);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior list item
                    Item↑.Mult := Item↑.Mult + K
                else head of nonfront list?
                    if Item ≠ List[Qfront].Head then
                        h↑.HeadMult := h↑.HeadMult + K
                    else Item is at front of queue
                        f := F&A(FrontMult, −MaxMult);  stop deletions
                        h↑.HeadMult := h↑.HeadMult + K;
                        if f > 0 then f copies left
                            FrontMult := f + K
                        else  all copies taken
                            FrontMult := K
                        end if
                    end if
                end if;
                F&A(Qlower, K);
                Linked_LinMQRaiseMult := true
            end if;
        Write_Release(h↑.RW_Act)
    end if
    end with
end function Linked_LinMQRaiseMult;
```

```
function Linked_LinMQLowerMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LinMQueue;
                                M    : integer);
var dequeueItem   : boolean;
    h             : ListElement_Pointer;
    f             : integer;

    with Q do
    Linked_LinMQLowerMult := dequeueItem := false;
    f := M := 0;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Write_Request(h↑.RW_Act);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior list item
                    M := Item↑.Mult;
                    if Item↑.Mult > K then lower multiplicity
                        if not TDR(Qlower, K) then
                            F&A(NilPool, K)
                        end if;
                        Item↑.Mult := Item↑.Mult – K
                    else remove Item
                        dequeueItem := true
                    end if;
                    Linked_LinMQLowerMult := true
                else head of nonfront list?
                    if Item ≠ List[Qfront].Head then
                        M := h↑.HeadMult;
                        if h↑.HeadMult > K then lower multiplicity
                            if not TDR(Qlower, K) then
                                F&A(NilPool, K)
                            end if;
                            h↑.HeadMult := h↑.HeadMult – K
                        else remove Item
                            h↑.HeadMult := 1;
                            dequeueItem := true
                        end if
                        Linked_LinMQLowerMult := true
                    else Item is at front of queue
                        if FrontMult > 0 then
                            f := F&A(FrontMult, –K);
                            if f > K then lower multiplicity
                                M := f;
                                h↑.HeadMult := h↑.HeadMult – K;
                                if not TDR(Qlower, K) then
                                    F&A(NilPool, K)
                                end if;
                                Linked_LinMQLowerMult := true
                            else
```

```
                        if f > 0 then remove item
                            F&A(h↑.RW_Act, MaxR–1);
                            repeat until h↑.HeadMult = f;
                            F&A(h↑.RW_Act, 1–MaxR);
                            M := f;   number of copies removed
                            h↑.HeadMult := 1;
                            dequeueItem := true;
                            Linked_LinMQLowerMult := true
                        end if  f > 0
                    end if  f > K
                end if  FrontMult > 0
            end if  Item ≠ List[Qfront].Head
        end if;  h↑.Head ≠ Item of h↑.#Missing > 0
        if dequeueItem then
            F&A(Item↑.Prev↑.#Missing, Item↑.#Missing + 1);
            Item↑.Prev↑.Next := Item↑.Next;
            Item↑.Next↑.Prev := Item↑.Prev;
            Item↑.List_Loc := nil;
            if not TDR(Qlower, M – 1) then
                F&A(NilPool, M – 1)
            end if;
            if f > 0 then FrontMult := 1 end if
        end if
    end if;  h = Item↑.List_Loc
    Write_Release(h↑.RW_Act)
  end if
  end with
end function Linked_LinMQLowerMult;
```

3.1.2.3.2.2. A Writer Priority Readers/Writers Implementation

The writer priority readers/writers implementation of RaiseMult and LowerMult differs from the preceding implementation in that P/V protocols are substituted for readers/writers protocols and the explicit readers/writers protocols used to coordinate deletions and interior access operations are eliminated. Instead, deletions and interior access operations are coordinated via the queue component *FrontMult* and the front list's head multiplicity component *HeadMult* (and final deletions are now required to lock lists before dequeueing items) as described in section 3.1.2.3.1.2. Additional comments on the implementation are unnecessary.

```
Linked_LinMQueue = MultiQueue Supporting Type II IA
                   record
                      Qlower, Qsize, Qfront, Qrear,
                      FrontMult, NilPool : integer;
                      List : array[0..Qsize–1] of ListElement
                   end;

ListElement = record
                 Queue_Id    : MQueue_Pointer;
                 HeadMult    : integer;
                 Head,
                 Tail        : Pointer to ListElement or Object;
                 #Missing    : integer;
                 Access_Sem : Weak_Semaphore
              end;

MQueue_Pointer = ↑Linked_LinMQueue;

ListElement_Pointer = ↑ListElement;

Object = record
            List_Loc : ListElement_Pointer;
            Mult     : integer;
            Next,
            Prev     : Pointer to ListElement or Object;
            #Missing : integer;
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qfront, Qrear, FrontMult, NilPool = 0;
   for 0≤j<Qsize,
     List[j].Queue_Id       = address of queue;
     List[j].HeadMult       = 0;
     List[j].Head           = address(List[j]);
     List[j].Tail           = address(List[j]);
     List[j].#Missing       = 0;
     List[j].Access_Sem     = 1;
```

```
procedure Linked_LinMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LinMQueue);
var i : integer;
   with Q do
   i := F&A(Qrear, 1) mod Qsize;
   if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
   Item↑.Next := address(List[i]);
   Item↑.Mult := M;
   Item↑.#Missing := 0;
   Weak_P(List[i].Access_Sem);
      Item↑.List_Loc := address(List[i]);
      Item↑.Prev := List[i].Tail;
      List[i].Tail↑.Next := Item;
      List[i].Tail := Item;
      if List[i].Head = Item and List[i].#Missing = 0 then
         List[i].HeadMult := M;   promote item's multiplicity
         if Qfront = i then FrontMult := M end if
      end if;
   Weak_V(List[i].Access_Sem);
   F&A(Qlower, M)
   end with
end procedure Linked_LinMQInsert;
```

```
function Linked_LinMQDelete (Item : Object_Pointer;
                             Q    : Linked_LinMQueue);
var d, m, n, s : integer;
  with Q do
  if TDR(Qlower, 1) then
    Calc_Deletion_Site(d)_V2;   defined in sect. 3.1.1.4.2.1
    Item := List[d].Head;
    if F&A(List[d].HeadMult, −1) = 1 then    final delete
      Weak_P(List[d].Access_Sem);
        Open_Next_List_Site_PV;   defined in sect. 3.1.2.1
        if List[d].#Missing = 0 then     dequeue item
          List[d].Head := Item↑.Next;
          List[d].Head↑.Prev := address(List[d]);
          List[d].#Missing := Item↑.#Missing;
          m := Item↑.Next↑.Mult;
          Item↑.List_Loc := nil
        else    head of list is a ‘‘missing item’’
          List[d].#Missing := List[d].#Missing – 1;
          m := Item↑.Mult;
          Item := nil
        end if;
        if List[d].#Missing > 0 then
          List[d].HeadMult := 1   mult. of ‘‘missing item’’
        else
          List[d].HeadMult := m   mult. of head list item
        end if;
      Weak_V(List[d].Access_Sem);
    end if;
    return true
  else
    return false
  end if
  end with
end function Linked_LinMQDelete;
```

```
function Linked_LinMQRaiseMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LinMQueue);
var h : ListElement_Pointer;
    f : integer;

    with Q do
    Linked_LinMQRaiseMult := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior list item
                    Item↑.Mult := Item↑.Mult + K;
                    F&A(Qlower, K);
                    Linked_LinMQRaiseMult := true
                else head of nonfront list?
                    if Item ≠ List[Qfront].Head then
                        h↑.HeadMult := h↑.HeadMult + K;
                        F&A(Qlower, K);
                        Linked_LinMQRaiseMult := true
                    else Item is at front of queue
                        if FrontMult > 0 then
                            f := F&A(FrontMult, –MaxMult);  stop deletions
                            if f > 0 then
                                F&A(h↑.HeadMult, K);
                                FrontMult := f + K;
                                F&A(Qlower, K);
                                Linked_LinMQRaiseMult := true
                            end if
                        end if
                    end if
                end if
            end if;
        Weak_V(h↑.Access_Sem)
    end if
    end with
end function Linked_LinMQRaiseMult;
```

```
function Linked_LinMQLowerMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LinMQueue;
                                M    : integer);
var dequeueItem     : boolean;
    h               : ListElement_Pointer;
    f               : integer;

    with Q do
    Linked_LinMQLowerMult := dequeueItem := false;
    f := M := 0;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior list item
                    M := Item↑.Mult;
                    if Item↑.Mult > K then lower multiplicity
                        if not TDR(Qlower, K) then
                            F&A(NilPool, K)
                        end if;
                        Item↑.Mult := Item↑.Mult – K
                    else remove Item
                        dequeueItem := true
                    end if;
                    Linked_LinMQLowerMult := true
                else head of nonfront list?
                    if Item ≠ List[Qfront].Head then
                        M := h↑.HeadMult;
                        if h↑.HeadMult > K then lower multiplicity
                            if not TDR(Qlower, K) then
                                F&A(NilPool, K)
                            end if;
                            h↑.HeadMult := h↑.HeadMult – K
                        else remove Item
                            h↑.HeadMult := 1;
                            dequeueItem := true
                        end if
                        Linked_LinMQLowerMult := true
                    else Item is at front of queue
                        if FrontMult > 0 then
                            f := F&A(FrontMult, –MaxMult);  stop deletions
                            if f > K then lower multiplicity
                                M := f;
                                F&A(h↑.HeadMult, –K);
                                if not TDR(Qlower, K) then
                                    F&A(NilPool, K)
                                end if;
                                FrontMult := f – K;
                                Linked_LinMQLowerMult := true
                            else
```

```
                        if f > 0 then remove item
                            repeat until h↑.HeadMult = f;
                            M := f;   number of copies removed
                            h↑.HeadMult := 1;
                            dequeueItem := true;
                            Linked_LinMQLowerMult := true
                        end if  f > 0
                    end if  f > K
                end if  FrontMult > 0
            end if  Item ≠ List[Qfront].Head
        end if;  h↑.Head ≠ Item of h↑.#Missing > 0
        if dequeueItem then
            F&A(Item↑.Prev↑.#Missing, Item↑.#Missing + 1);
            Item↑.Prev↑.Next := Item↑.Next;
            Item↑.Next↑.Prev := Item↑.Prev;
            Item↑.List_Loc := nil;
            if not TDR(Qlower, M − 1) then
                F&A(NilPool, M − 1)
            end if;
            if f > 0 then FrontMult := 1 end if
        end if
    end if;  h = Item↑.List_Loc
  Weak_V(h↑.Access_Sem)
 end if
 end with
end function Linked_LinMQLowerMult;
```

3.2. Logarithmic Multiqueues

The linear multiqueue implementation supports only one active deletion site at a time, limiting the number of concurrent deletions to the multiplicity M of the queue's front item. If more than M deletions are concurrently executed, the remainder are blocked until the first M finish. Thus, the deletion site can become a bottleneck if the number of active deletions is much larger than $\overline{M}$, the average item multiplicity.

An obvious solution for avoiding deletion bottlenecks is to provide multiple deletion sites, as is done for the queues of chapter 2. This is complicated by the variability of an item's multiplicity which prevents deletions from determining their sites simply based on the number of deletions already performed. One strategy for opening up additional sites is to have a deletion, upon finding that the current site is exhausted, search the remainder of the queue for a non-empty site (without waiting for the final deletion of the current site to terminate). However, this does not eliminate the deletion bottleneck (because of the searching that is performed), and, as previously noted in section 3.1.1.1, the possibility of violating the FIFO queue property[14] then exists.

Gottlieb et al. [80] present an alternative multiqueue implementation (which is reviewed in Appendix B.2) that supports multiple deletion sites while maintaining the FIFO queue property. The queue consists of a sequentially allocated array augmented with a complete binary tree that guides deletions to sites. The possibility of a deletion site becoming a bottleneck is eliminated at the expense of adding overheads that are logarithmic in the size of the array to both insertion and deletion.

Sequential and linked list implementations of the logarithmic multiqueue are given in the next two sections. For each, insertion, deletion, multi-deletion, and interior access algorithms are presented. The original deletion algorithm of Gottlieb et al. [80] has been modified below to increase the efficiency of non-final deletions.

3.2.1. Sequential Allocation

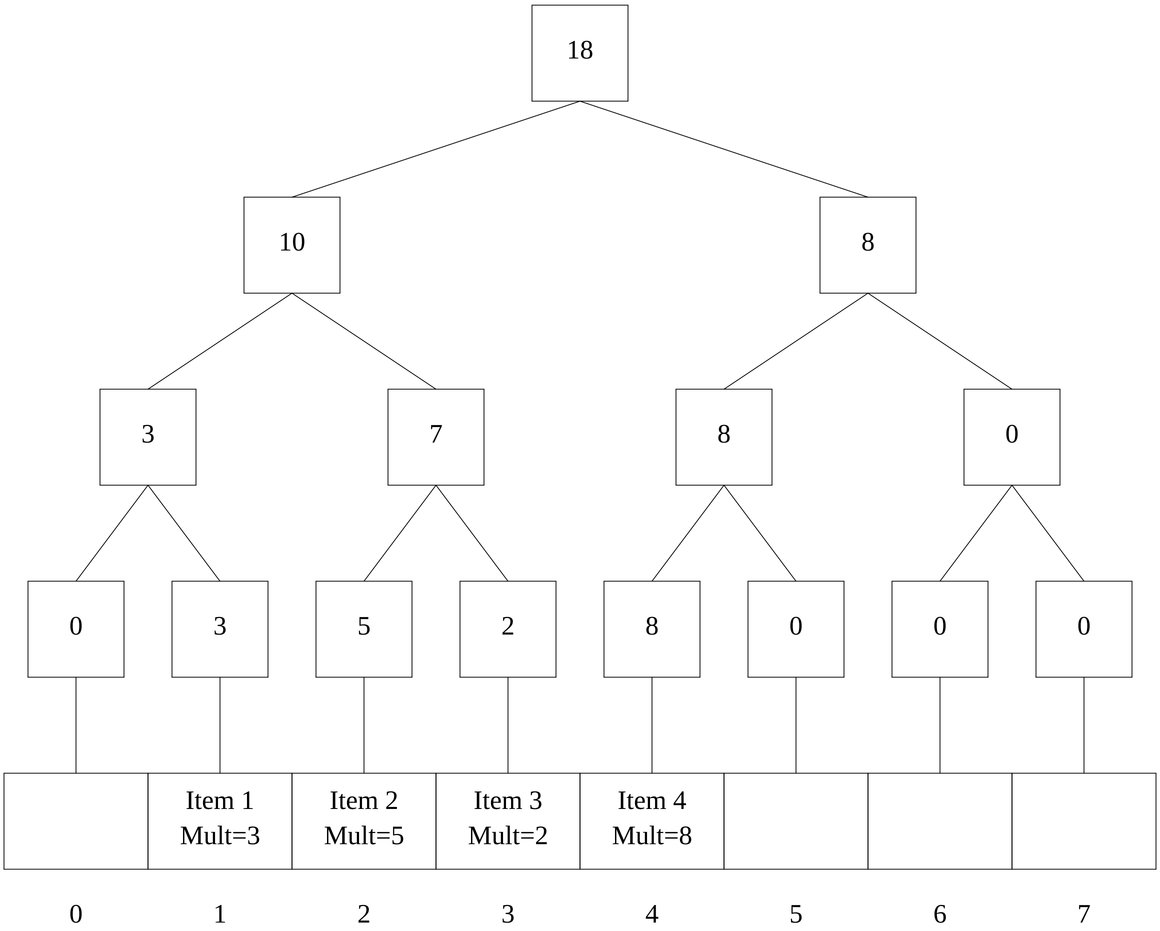
We first review the basic method presented by Gottlieb et al. [80] for performing multi-queue insertion and deletion and then proceed to the modified algorithms.

3.2.1.1. A Review of Insertion and Deletion

A multiqueue is represented by an array $Cell[0..Qsize-1]$, where $Qsize=2^h$. This structure is augmented with a complete binary tree of height h where leaf i of the tree maps to $Cell[i]$ of the queue. Each node contains a counter recording the number of items associated with the leaves of its subtree. A leaf's counter is set to its corresponding cell's multiplicity. The counter of an interior node equals the sum of its left and right sons' counts. The counters are adjusted by the queue operations as items are added to and removed from the queue. See Figure 3-2.

As before, an insertion calculates a site based on the number of insertions already performed modulo the queue size. After placing an item with multiplicity M into $Cell[i]$, the insertion ascends the tree from leaf i to the root, incrementing each visited node's count by M. A deletion starts at the root and, aided by the counters, descends the tree to the leftmost leaf with a non-zero count; the cell associated with this leaf is the deletion site. As the deletion travels down the tree, it decrements each node's count by one, reserving one item within that node's subtree for itself.

[14] Recall (from chapter 2) that this property states that if the insertion of item I finishes before the insertion of item J starts, then a deletion of item J cannot be completed before a deletion of item I has started.



Logarithmic multiqueue of *Qsize* 8 containing 18 items.

Figure 3-2.

(Note that concurrent deletions can descend the same path to a cell.) As in the linear implementation, the deletion copies the item from the cell and decrements the cell's multiplicity by one. When the multiplicity reaches zero, the item is dequeued.

This describes the overall flow of the operations, but it is incomplete as it stands since deletions may fail to remove the queued items in FIFO order. This is most easily demonstrated with a queue containing *Qsize* items, each with multiplicity one: Suppose the queue is serially accessed by a deletion D_0, an insertion I_0, and then by another deletion D_1. D_0 descends the queue's tree to the leftmost leaf and dequeues the item in *Cell*[0]. I_0 then fills *Cell*[0] and ascends the tree, incrementing the counters that D_0 decremented. D_1 then descends the tree, following the same path as D_0, and references *Cell*[0] instead of *Cell*[1]. The items in the cells are taken out of order. Hence, deletions will traverse the wrong path if insertions wraparound the queue and prematurely update a tree's counters. We note that the above behavior, while being incorrect for multiqueues is acceptable for multipiles. Since multipile items are unordered, they need not be deleted in FIFO order. Chapter 8 pursues this point in greater detail.

The violation of the FIFO queue property above can also be explained in terms of insertion and deletion phases,[15] where insertions j through j+*Qsize*–1 are referred to as the phase j insertions and deletions that copy items added to the queue by phase j insertions are referred to as the phase j deletions. Originally, the queue above contains *Qsize* items, where each one is inserted by a phase j insertion. D_0 is a phase j deletion that deletes an item inserted by one of these insertions. I_0 belongs to the next phase, j+1, but after it ascends the tree, the fact that the items in the first two cells belong to phases j+1 and j, respectively, cannot be determined from the tree node counters: these counters only indicate which leaves are non-empty. Thus, while D_1 is a phase j deletion, it is guided by the counters to a phase j+1 item, instead of to a phase j item.

One way to resolve this problem is to block all phase j+1 insertions until all of the phase j deletions have terminated, but this is overly restrictive. A less severe method (due to Gottlieb et al. [80]) is to allow one deletion phase and two consecutive insertion phases to be active at the same time. That is, deletions in phase j are blocked until the deletions in phase j–1 are finished, and insertions in phase j are blocked until the insertions in phase j–2 are finished. This is done by associating a tree and a cell array with each phase. Deletions and insertions only reference the trees and cells for their specific phases. Hence, deletions of one phase are unaffected by insertions of other phases, i.e., they are immune to the effects of insertion wraparound. For example, phase j insertions can be filling in the last k cells of one array, phase j deletions can be emptying any cell of this array where a phase j insertion has completed, and phase j+1 insertions can be filling in the first *Qsize*–k cells of the other array. The phase j deletions do not access items added to the queue by phase j+1 insertions since the trees and cells for these phases are distinctly separate.

[15] The phase concept was introduced by Gottlieb et al. [80] and is reviewed in section 2.1.1.2.

The implementation of Gottlieb et al. [80] of this insertion-deletion phase strategy requires the last deletion of a phase to update an index that identifies which tree the deletions of the next phase are to access. References to this index by incoming phase $j+1$ deletions and the last phase j deletion are coordinated via a readers/writers protocol. In the next section, a modified version of this implementation is given that requires only final deletions and insertions to coordinate via explicit readers/writers protocols. Hence, the cost of a non-final deletion is lower in the modified implementation that in the original implementation of Gottlieb et al. [80]. When the average item multiplicity $\overline{M}$ is large, non-final deletions are the dominant activity and thus the modified implementation gives better overall performance.

Before presenting the modified insertion and deletion algorithms, we wish to point out that while both the linear and logarithmic multiqueues support the FIFO queue property, the latter is more sensitive than the former to the order in which item's are removed when insertions and deletions are *concurrently* executed. For example, suppose two phase p insertions, I_1 and I_2, are performed on the first two sites of an empty multiqueue, and I_2 completes first. Let T be the time interval between the terminations of the two insertions. Under the logarithmic implementation, deletions descending the tree during T are guided to the item at the second site while those arriving after this period will be directed to the first site. Once the item at the first site is dequeued, deletions return to the second site. This deletion sequence could never occur under the linear implementation: Regardless of the relative order in which insertions terminate, the item at the first site must be dequeued before the next site becomes available for deletion.

3.2.1.2. Modified Insertion and Deletion

The initial implementation to be described limits the number of active insertion and deletion phases to two and one, respectively. After presenting the basic algorithms, we describe how the multiqueue structure and algorithms may be extended to support $\#p \geq 2$ insertion phases and $\#p-1$ deletion phases.

A logarithmic multiqueue requires a cell array and a tree of counters for each active phase. As the initial implementation supports two insertion phases, a multiqueue is represented by a two-dimensional array $Cell[0..Qsize-1, 0..1]$, where $Qsize = 2^h$, and a complete binary tree of height h, where each node of the tree is composed of two counters.[16] Phase information is communicated through two queue components: the current deletion phase index *DPhase*, which is used to index the node counters and cells, and the deletion phase multiplicity *PhaseMult*, which equals the number of items in the current deletion phase (and is analogous to the linear multiqueue component *FrontMult*).

The ith insertion's phase number equals $\lfloor i/Qsize \rfloor$, where a fetch-and-add on the queue counter *Qrear* determines i. In the current implementation phase numbers are kept modulo two. A phase p insertion ascends the tree from the leaf associated with its site to the tree's root incrementing the pth counter in each node visited. When the insertion reaches the root, its item's multiplicity is added to the current deletion phase multiplicity *PhaseMult* if the insertion's phase matches the current deletion phase (p = *DPhase*).

A deletion belongs to the current deletion phase *DPhase* if it decrements *PhaseMult* when the latter is positive. The deletion then descends the tree to a leaf that contains a *DPhase* item. Each node on the path from the root to the leaf has its *DPhase* counter decremented by 1. These operations are summarized by the following code sequence.[17] As explained after the presentation of the algorithms, a deletion may have to repeatedly test both counters of a node's subtrees before being able to descend further down the tree.

[16] Two-dimensional arrays and trees containing two counters, as opposed to two distinct one-dimensional arrays and two trees, are used here to be consistent with the linked allocated implementation of multiqueues for which such data structures are more space efficient. See section 3.2.2 and Appendix B.2 for details.

[17] They are also encapsulated in the Calc_Leaf_Site macro that appears together with the deletion algorithm below.

```
s := 0;
repeat
    if PhaseMult > 0 then s := F&A(PhaseMult, −1) end if
until s > 0;
phase := DPhase;         deletion can join current phase
F&A(Cnt[Root, phase], −1);
n := Root;
while n is not a leaf do    find leaf whose cell is not empty
    if TDR(Cnt[leftson(n), phase], 1) then
        n := leftson(n)
    elseif TDR(Cnt[rightson(n), phase], 1) then
        n := rightson(n)
    end if
end while;
```

A *final* phase *j* deletion, aside from its other duties (e.g., informing insertions that a site is now empty by decrementing *Qupper*), decrements a final deletion counter *#FD* (which is initialized to *Qsize*) to determine if it is the *last* deletion of the current phase. If so, it reinitializes *#FD* to *Qsize* and opens the next deletion phase by setting *DPhase* to *j*+1 and *PhaseMult* to the root's phase $j+1$ multiplicity count.[18] *PhaseMult* is not set to zero because insertions from phase *j*+1 may have completed already and incremented the root's count for this phase.

The last deletion of phase *j* and all insertions of phase *j+1* reference *DPhase*, *PhaseMult*, and the root's count for phase *j*+1. These references must be performed using readers/writers coordination between the *Qsize* insertions of phase *j*+1 (the readers) and the last deletion of phase *j* (the writer) to guarantee the proper initialization of *PhaseMult*.[19]

To increment the root's count, and possibly the deletion phase multiplicity, by a multiplicity of *M*, a phase *p* insertion executes

```
Read_Request(Phase_Act);
I1:   F&A(Cnt[Root, p], M);
I2:   if p = DPhase then F&A(PhaseMult, M) end if;
Read_Release(Phase_Act);
```

and, to open the next phase, the last deletion of the current phase executes

---

[18] This sequence is defined by the Open_Next_Phase macro also presented below. It is analogous to the Open_Next_Cell_Site macro defined earlier in section 3.1.1.2.

[19] This problem is analogous to the updating of *FrontMult* for linear multiqueues, which is discussed in section 3.1.1.2.

```
Write_Request(Phase_Act);
    D1:   DPhase := (DPhase+1) mod 2;
    D2:   PhaseMult := Cnt[Root, DPhase];
    Write_Release(Phase_Act);
```

Without readers/writers coordination, a possible interleaving of the above statements is: D1, I1, I2, D2, which results in an erroneous update of *PhaseMult* by an insertion.

Finally, before presenting the algorithms, we review the method for bounding the counter *Qrear* (which is similar to the method given for the sequentially allocated queues of chapter 2). Each insertion increments *Qrear* by 1 to generate a raw index *i* from which cell (*i* modulo *Qsize*) and phase ($\lfloor i/Qsize \rfloor$) indices can be derived. When the last insertion of an odd phase is performed and finds that *Qrear* equals twice the queue size, it decrements this counter by 2**Qsize*. The maximum value that *Qrear* can attain is 3**Qsize*–1: Between the time the last insertion of an odd phase *p* detects that *Qrear* equals 2**Qsize* and decrements it by 2**Qsize*, *Qsize*–1 phase *p* deletions and *Qsize*–1 phase *p*+1 insertions can occur, where each of the insertions increments *Qrear* by one, driving it to 3**Qsize*–1. Further insertion activity is then suspended until a deletion is performed. This occurs when the last phase *p* insertion completes, enabling the last phase *p* deletion to execute and open the next deletion phase.

```
Seq_LogMQueue = Sequentially Allocated Logarithmic MultiQueue
                    record
                        Qlower, Qupper, Qsize, Qrear,
                        PhaseMult, DPhase, #FD : integer;
                        Phase_Act          : integer;
                        Cnt                : array[1..2*Qsize–1, 0..1] of integer;
                        Cell               : array[0..Qsize–1, 0..1] of CellElement
                    end;

CellElement = record
                  Mult  : integer;
                  Value : Object_Pointer
              end;

Object = record
             Application_Defined_Attributes
         end

Object_Pointer = ↑Object;
```

Initially:

```
Qlower, Qupper, Qrear, DPhase, PhaseMult, Cnt[] = 0;
#FD = Qsize;
Phase_Act = MaxR;
for 0≤j<Qsize, 0≤k≤1,
  Cell[j, k].Mult = 0;

function Seq_LogMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LogMQueue);
var i, phase, n : integer;
  with Q do
  if TIR(Qupper, 1, Qsize) then
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Cell[i, phase].Value := Item;
    Cell[i, phase].Mult  := M;

    n := leaf corresponding to Cell[i, phase];
    while  n ≠ Root do   ascend tree
      F&A(Cnt[n, phase], M);
      n := parent(n)
    end while;

    Read_Request(Phase_Act);
      F&A(Cnt[Root, phase], M);
      if phase = DPhase then F&A(PhaseMult, M) end if;
    Read_Release(Phase_Act);
    F&A(Qlower, M);
    return true
  else
    return false
  end if
  end with
end function Seq_LogMQInsert;
```

```
function Seq_LogMQDelete (Item : Object_Pointer;
                          Q    : Seq_LogMQueue);
var d, phase, n, s : integer;
  with Q do
  if TDR(Qlower, 1) then
    Calc_Leaf_Site(n)_V1;
    d := cell corresponding to leaf n;
    Item := Cell[d, phase].Value;
    if F&A(Cell[d, phase].Mult, −1) = 1  then   final delete
      F&A(Qupper, −1);
      if F&A(#FD, −1) = 1 then  last delete
        #FD := Qsize;
        Open_Next_Phase
      end if
    end if;
    return true
  else
    return false
  end if
  end with
end function Seq_LogMQDelete;

macro Calc_Leaf_Site(n)_V1;
  s := 0;
  repeat
    if PhaseMult > 0 then s := F&A(PhaseMult, −1) end if
  until s > 0;
  phase := DPhase;        deletion can join current phase
  F&A(Cnt[Root, phase], −1);
  n := Root;
  while n is not a leaf do  find leaf whose cell is not empty
    if TDR(Cnt[leftson(n), phase], 1) then
      n := leftson(n)
    elseif TDR(Cnt[rightson(n), phase], 1) then
      n := rightson(n)
    end if
  end while;
end macro Calc_Leaf_Site(n)_V1;

macro Open_Next_Phase;
  Write_Request(Phase_Act);
    DPhase := (phase+1) mod 2;
    PhaseMult := Cnt[Root, DPhase];
  Write_Release(Phase_Act);
end macro Open_Next_Phase;
```

At first glance it may not be evident why a deletion *D* in its tree descent tests the counts of a node *N*'s right son if *N*'s left son's count is nonpositive (the reader should refer to the Calc_Leaf_Site macro above for details). One might assume that if *D* finds *N*'s left subtree is empty, then the right subtree must not be. Recall that D is guaranteed of receiving an item since it has already decremented the queue's deletion phase multiplicity (*PhaseMult*), and furthermore, this item must be in the tree rooted at N as D has been guided to N by the tree's node counters. However, in a complicated scenario given by Rudolph [82], this assumption is shown to be false. The basic problem lies in the fact that TDR is non-atomic: A node's counter may be tested by one deletion's TDR between the counter's being decremented and incremented by another deletion's TDR. The necessity of testing the right subtree counter is illustrated below, where we assume that N's left and right subtrees contain one item each.

(1) Two phase p deletions, D_1 and D_2, try to descend N's left subtree L by performing TDRs on L's phase p counter c_L. Both TDRs test and decrement c_L, driving it to -1. D_1's TDR succeeds, and D_1 descends L. Before D_2's TDR completes, by incrementing c_L by one:

(1.1) A phase p insertion is performed on a cell that is attached to a leaf of L. The insertion ascends the tree, incrementing c_L to zero along the way.

(1.2) Another phase p deletion D_3 descends the tree to N where it performs a TDR on c_L which fails since the counter is zero.

(2) D_2's TDR on c_L completes, incrementing c_L to 1.

(3) Both D_2 and D_3 perform TDRs on c_R, the phase p counter for N's right subtree R. Recall that c_R equals 1. The deletion whose TDR succeeds then descends R, and the other deletion tries once more to descend L.

If D_2 and D_3 assumed that R contained an item for each one of them and did not test c_R, both would have descended R, where only a single item resided.

While a phase j deletion D may have to test a node's subtree counters numerous times, it will not do so for an unbounded period of time (assuming that PEs execute with only a bounded delay between instructions). In the worst case, D waits for the remaining $Qsize-1$ phase j deletions to complete descending the tree before acquiring the last item of the phase.

Although insertions and deletions exclude one another from accessing the current deletion phase index (*DPhase*) through readers/writers synchronization, the implementation of this protocol does not have to be fair to avoid starvation. Insertions cannot starve one another since they are readers. A last deletion can never be indefinitely blocked by a sequence of insertions since the queue is full (*Qupper* = *Qsize*) after *Qsize* insertions have been performed. Conversely, insertions cannot be starved by a sequence of last deletions since all of the insertions of the current deletion phase must complete before a last deletion can commence.

It is possible for an incoming deletion to starve while trying to determine whether it is part of the current deletion phase. The operation could always be unlucky and check *PhaseMult* after another deletion had driven it to zero. If desired, the deletion phase calculation, like the deletion site calculation for linear multiqueue deletions, can be transformed into a critical code section to prevent starvation. But, unlike linear multiqueues, the presence of the critical section is not now sufficient to guarantee that the *i*th deletion takes the item inserted by the *i*th insertion[20] as a deletion's descent through the tree can be influenced by the actions of other operations.[21]

The current multiqueue implementation only permits one deletion and two insertion phases to be active at any time. The multiqueue data structure can be extended to support $\#p-1$ ($\#p \geq 2$) deletion phases and $\#p$ insertion phases by associating a counter array of size $\#p$ with each node, making the cell array of size *Qsize* by $\#p$, and replacing the variables *PhaseMult* and *#FD* with arrays of size $\#p-1$. A phase p insertion uses p modulo $\#p$ to index each array. If the insertion belongs to one of the $\#p-1$ active deletion phases, it increments *PhaseMult*[p modulo $\#p$] with its item's multiplicity. A deletion joins phase d if it can claim an item for itself from *PhaseMult*[d]. The deletion then uses d to index the node and cell arrays. *DPhase* equals the oldest active deletion phase, i.e., deletion phases *DPhase* through $DPhase + \#p - 2$ can be active.

3.2.1.3. Multi-Deletion

A multi-deletion of M items can be more efficient than M consecutive deletions. In the best case, the multi-deletion only accesses one deletion site to obtain its M items, decrementing the counters of h nodes, where h is the height of the multiqueue's tree. In the worst case, the multi-deletion yields the same performance as M deletions, i.e., M sites are accessed, requiring $\theta(Mh)$

[20] A serial ordering for logarithmic multiqueue operations may be defined by using a method similar to the one given in section 3.1.1.2 for defining a serial ordering for linear multiqueue operations.

[21] An example of this has already been given in the previous section. Specifically, in contrasting the results of *concurrent* operations executed on linear and logarithmic multiqueues, we saw that for the latter, a deletion could access site j before site i, where $j>i$, if the insertion at j finished before the insertion at i. Thus, the ith deletion may remove the item not inserted by the ith insertion.

node counters to be decremented.

A multi-deletion of M items fails if the queue does not contain at least M items. Otherwise, the operation requests as many items as possible (up to M) from the current deletion phase p. If p does not contain a sufficient number of items, the remainder will be acquired from the next phase when it is opened. Suppose the multi-deletion is granted $G \le M$ items from phase p. The operation collects these items by starting at the root of the queue's tree, and, aided by the node counters, descends the tree towards the leftmost leaves with non-zero counts. For each node N visited, let g_N equal the initial number of items the multi-deletion is granted from the tree rooted at N (e.g., $g_{root}=G$). The multi-deletion, like a standard multiqueue deletion, repeatedly requests up to g_N items, first from N's left subtree and then from the right subtree until it is granted $g_S \le g_N$ items from a subtree S of N. The multi-deletion then decrements g_N by g_S, giving the number of items (g_N) to be obtained from N's other subtree. The multi-deletion then continues its descent by requesting g_S items from S's subtrees. For a leaf L, the multi-deletion makes g_L copies of the item in the cell associated with L. The operation then ascends the tree to an ancestor A of L such that g_A is positive, i.e., it seeks a node where it still has items on reserve. The operation then descends a subtree of A to collect its remaining g_A items.

The multi-deletion algorithm uses the auxiliary function *Request* (N, R, G) to request R items from node N. The number of items granted is $G \le R$. *Request* fails if no items are granted. Otherwise, it succeeds.[22]

[22] Request differs from Test-Decrement-Retest function (TDR) (defined in chapter 1) in that the value by which N is decremented is the maximum of R and N instead of just R. The initial ''redundant'' test is needed here just as in TDR.

```
function Request (Count, R, G : integer);
var OldCount : integer;
    if Count > 0 then
        OldCount := F&A(Count, –R);
        if OldCount ≥ R then            grant request in full
            G := R;
            return true
        elseif OldCount > 0 then        grant request in part
            G := OldCount;
            F&A(Count, R–G);            correct Count by excess
            return true
        else
            F&A(Count, R)               deny request
        end if
    end if;
    return false
end function Request;
```

```
function Seq_LogMulti-Delete (Table : array[1..M] of Object_Pointer;
                              M     : integer;
                              Q     : Seq_LogMQueue);
var d, i, m, n, phase, s, son, total_granted : integer;
   g : array[1..2*MaxQsize–1] of integer;

   with Q do
   if TDR(Qlower, M) then
      m := M;
      total_granted := i := 0;  i is index for Table
      repeat  items also taken from next phase if PhaseMult<m
         g := s := 0;
         repeat
            if PhaseMult > 0 then s := F&A(PhaseMult, –m) end if
         until s > 0;
         phase := DPhase;
         g[Root] := s min m;                    number of items to obtain from this phase
         total_granted := total_granted + g[Root];
         m := m – g[Root];                      number to be obtained in next phase
         F&A(Cnt[Root, phase], –g[Root]);
         n := Root;
         repeat   make descents until get g[Root] items from current phase
            while n is not a leaf do
               if Request(Cnt[leftson(n), phase], g[n], g[leftson(n)]) then
                  g[n] := g[n] – g[leftson(n)];  n := leftson(n)
               elseif Request(Cnt[rightson(n), phase], g[n], g[rightson(n)]) then
                  g[n] := g[n] – g[rightson(n)];  n := rightson(n)
               end if
            end while;
            d := cell corresponding to leaf n;
            Table[i+1..i+g[n]] := Cell[d, phase].Value;
            i := i + g[n];
            if F&A(Cell[d, phase].Mult, –g[n]) = g[n] then
               F&A(Qupper, –1);                 final delete
               if F&A(#FD, –1) = 1 then         last delete
                  #FD := Qsize;
                  Open_Next_Phase   defined in sect. 3.2.1.2
               end if
            end if;
            if i < total_granted then
               repeat   ascend tree until find non-empty subtree
                  n := parent(n)
               until g[n] > 0
            end if
         until i = total_granted
      until m = 0;
      return true
   else
      return false
   end if
   end with
end function Seq_LogMulti-Delete;
```

3.2.1.4. Interior Access

We have already introduced many of the concepts and techniques needed to implement type one and type two interior access operations on logarithmic multiqueues.[23] Readers/writers and writer priority readers/writers implementations of type one and type two operations follow. These implementations are patterned, in part, after those presented in the first half of this chapter. Specifically, the cells of logarithmic multiqueues can be treated and accessed like the cells of linear multiqueues. The major new complications in implementing interior access are maintaining the auxiliary trees of logarithmic multiqueues and preserving the one-deletion-phase-at-a-time constraint. Our descriptions of the implementations will concentrate upon these two issues. We remark that the readers/writers implementations can be converted into starvation-free implementations by using fair readers/writers protocols and having incoming deletions test and decrement the queue component *PhaseMult* within a critical code section (as described in section 3.2.1.2).

3.2.1.4.1. Type One

3.2.1.4.1.1. A Readers/Writers Implementation

In this implementation (like the preceding interior access readers/writers implementations of sections 3.1.1.4.1.1 and 3.1.2.3.1.1), deletions read cells and insertions and interior access operations write cells. However, insertions do not issue explicit write requests (unlike interior access operations), rather they do so implicitly, resulting in a more efficient implementation. We explain how this is done below.

The tree structure of a multiqueue is ignored in this implementation since type one interior access operations do not alter cell multiplicities (the tree structure is not ignored in the writer priority readers/writers implementation that follows). Thus, the IA procedure is identical to the one given for sequentially allocated linear multiqueues (in section 3.1.1.4.1). An interior access operation explicitly requests write permission to a cell and then releases the cell when done. Similarly, after a deletion descends a multiqueue's tree to a leaf *i*, its actions parallel the linear multiqueue deletion algorithm (which is also given in section 3.1.1.4.1), i.e., the deletion explicitly requests permission to read cell *i* and then releases *i* when done. The final deletion of a cell

[23] Recall that type one IA operations can update only a queued item's application dependent information, whereas type two IA operations can update an item's queue dependent information as well.

can dequeue an item without requesting write access since it is the last reader.

Unlike deletions, insertions are not explicitly coordinated with interior access operations via a readers/writers protocol. An insertion does not mark an item as queued (through an assignment to the item's cell location component) until the item has been placed into a cell. This effectively prevents an IA operation from prematurely accessing the item. However, immediately after an item is marked, an IA operation may be performed. In particular, the item's insertion may still be ascending the queue's tree when its item is accessed by an IA operation. Again, this is permissible since type one operations do not alter cell multiplicities.

Associated with each cell of the multiqueue is a readers/writers activity counter and queue identifier. Each item contains a cell location component that indicates the item's queue status.

```
Seq_LogMQueue = MultiQueue Supporting Type I IA
                record
                    Qlower, Qupper, Qsize, Qrear,
                    PhaseMult, DPhase, #FD  : integer;
                    Phase_Act : integer;
                    Cnt         : array[1..2*Qsize–1, 0..1] of integer;
                    Cell        : array[0..Qsize–1, 0..1] of CellElement
                end;

CellElement = record
                  Queue_Id : MQueue_Pointer;
                  Mult      : integer;
                  Value     : Object_Pointer;
                  RW_Act    : integer
              end;

MQueue_Pointer = ↑Seq_LogMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
             Cell_Loc : CellElement_Pointer;
             Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;
```

Initially:

```
Qlower, Qupper, Qrear, DPhase, PhaseMult, Cnt[] = 0;
#FD = Qsize;
Phase_Act = MaxR;
for 0≤j<Qsize, 0≤k≤1,
  Cell[j, k].Queue_Id          = address of queue;
  Cell[j, k].Mult              = 0;
  Cell[j, k].RW_Act            = MaxR;
```

```
function Seq_LogMQInsert (Item : Object_Pointer;
                          M    : integer;     item's multiplicity
                          Q    : Seq_LogMQueue);
var i, phase, n : integer;
  with Q do
  if TIR(Qupper, 1, Qsize) then
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize−1 then F&A(Qrear, −2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Cell[i, phase].Value := Item;
    Cell[i, phase].Mult  := M;
    Item↑.Cell_Loc := address(Cell[i, phase]);
    n := leaf corresponding to Cell[i, phase];

    while  n ≠ Root do   ascend tree
      F&A(Cnt[n, phase], M);
      n := parent(n)
    end while;

    Read_Request(Phase_Act);
      F&A(Cnt[Root, phase], M);
      if phase = DPhase then F&A(PhaseMult, M) end if;
    Read_Release(Phase_Act);
    F&A(Qlower, M);
    return true
  else
    return false
  end if
  end with
end function Seq_LogMQInsert;
```

```
function Seq_LogMQDelete (Item : Object_Pointer;
                          Q    : Seq_LogMQueue);
var d, phase, n, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Leaf_Site(n)_V1;   defined in sect. 3.2.1.2
        d := cell index corresponding to leaf n;
        Read_Request(Cell[d, phase].RW_Act);
            Item := Cell[d, phase].Value;
            if F&A(Cell[d, phase].Mult, –1) = 1 then  final delete
                Item↑.Cell_Loc := nil;
                F&A(Qupper, –1);
                if F&A(#FD, –1) = 1 then
                    #FD := Qsize;
                    Open_Next_Phase   defined in sect. 3.2.1.2
                end if
            end if;
        Read_Release(Cell[d, phase].RW_Act)
        return true
    else
        return false
    end if
    end with
end function Seq_LogMQDelete;


function Seq_LogMQIA_TypeI (Item : Object_Pointer;
                            Q    : Seq_LogMQueue;
                            ϕ    : function);
var c : CellElement_Pointer;
    with Q do
    Seq_LogMQIA_TypeI := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Write_Request(c↑.RW_Act);
            if c = Item↑.Cell_Loc then
                ϕ(Item);        access Item
                Seq_LogMQIA_TypeI := true
            end if;
        Write_Release(c↑.RW_Act);
    end if
    end with
end function Seq_LogMQIA_TypeI;
```

3.2.1.4.1.2. A Writer Priority Readers/Writers Implementation

The writer priority readers/writers implementation eliminates the need for deletions to execute readers/writers protocols by adapting the linear multiqueue writer priority technique of section 3.1.1.4.1.2 to logarithmic multiqueues. Recall that for linear multiqueues, an interior access operation requests write access to the front cell by making the front multiplicity queue component *FrontMult* very negative. If *FrontMult*'s old value *f* is nonpositive, the operation fails as a final deletion has already begun. Otherwise, the operation has blocked *f* deletions from accessing the front site (as they can only proceed if they can decrement *FrontMult* when it is positive). It therefore waits for the front cell's multiplicity to equal *f*, which indicates that all active deletions have completed, and then accesses the front cell. We extend this same technique to the cells of a logarithmic multiqueue by treating each cell like an active deletion site of a linear multiqueue, where the cell's corresponding leaf counter serves as the site's *FrontMult* component.

To implement this strategy we modify insertion and deletion so that they perform the same operations on leaf counters that their linear multiqueue counterparts perform on queue component *FrontMult*. Specifically, insertions and deletions, respectively, increment and decrement leaf counters via assignments and test-decrement-retest-no-increment (TDRNI) functions instead of with fetch-and-adds and TDRs. TDRNI differs from TDR only in that after decrementing a variable *V* by a value *K* and finding *V*'s old value to be nonpositive, no increment of *V* by *K* is done.[24] That is, TDRNI does not undo the decrement when it returns false. Also, as can be seen from an inspection of the algorithms, these modifications raise neither the cost of insertion nor deletion.

When interior access is not supported these modifications are clearly safe. An insertion can increment a leaf's count via an assignment since concurrent increments are not possible: only one insertion can be active at a site, and deletions never increment leaf counts since they test leaves via TDRNIs. Once a deletion makes a leaf count nonpositive, it remains so until an insertion assigns it a positive value. These modifications to insertion and deletion allow the techniques already developed for linear multiqueue writer priority interior access to be directly applied to logarithmic multiqueues.

Interior access operations access cells exclusively to lock out other interior access operations. In addition, the absence of explicit readers/writers protocols between interior access and deletion requires interior access operations and either final deletions or insertions to be mutually

[24] See the second version of the Calc_Leaf_Site macro below. The TDRNI function has been expanded in line. Note that the loop body of the linear multiqueue deletion site calculation given in section 3.1.1.2 consists of a TDRNI operation.

exclusive at the cell level. Otherwise, an interior access operation could access the wrong item: Suppose neither exclusion were supported. Between an interior access operation's test that an item *I* is in cell *c* and its making the leaf count associated with *c* negative, a final deletion of *I* can occur, followed (eventually) by an insertion of an item *J* into *c*, making the leaf count positive. The interior access operation, after making the leaf count negative, would erroneously access *J*. Clearly, if final deletions access their cells exclusively, this problem will not occur. The same is true if insertions increment leaf counts and place items into cells atomically.[25] In this case, if a final deletion decrements a leaf count before an interior access operation can make it negative, the interior access operation fails: The count can not be made positive again until the interior access operation releases the cell and an insertion is performed.

We elect to make insertions and interior access operations mutually exclusive to prevent the following scenario. Suppose insertions and interior access operations were permitted to concurrently access the same cells. An insertion could mark an item *I* as queued, and then an interior access operation could attempt to access *I* before the insertion had initialized the corresponding leaf count to *I*'s multiplicity. The interior access would fail (as the leaf count would be nonpositive, implying that the cell was empty). Although this would not be an error (the insertion was not done), the scenario is ruled out by making insertion and interior access mutually exclusive at the cell level.

```
Seq_LogMQueue = MultiQueue Supporting Type I IA
                record
                    Qlower, Qupper, Qsize, Qrear,
                    PhaseMult, DPhase, #FD  : integer;
                    Phase_Act : integer;
                    Cnt       : array[1..2*Qsize–1, 0..1] of integer;
                    Cell      : array[0..Qsize–1, 0..1] of CellElement
                end;
```

[25] Linear multiqueue insertion also performs these same operations atomically, where the leaf count is replaced by a queue's *FrontMult* component.

```
CellElement = record
                    Queue_Id     : MQueue_Pointer;
                    Mult         : integer;
                    Value        : Object_Pointer;
                    Access_Sem : Weak_Semaphore
                end;

MQueue_Pointer = ↑Seq_LogMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
                Cell_Loc : CellElement_Pointer;
                Application_Defined_Attributes
            end;

Object_Pointer = ↑Object;
```

Initially:

```
    Qlower, Qupper, Qrear, DPhase, PhaseMult, Cnt[] = 0;
    #FD = Qsize;
    Phase_Act = MaxR;
    for 0≤j<Qsize, 0≤k≤1,
      Cell[j, k].Queue_Id              = address of queue;
      Cell[j, k].Mult                  = 0;
      Cell[j, k].Access_Sem            = 1;
```

```
function Seq_LogMQInsert (Item : Object_Pointer;
                          M    : integer;      item's multiplicity
                          Q    : Seq_LogMQueue);
var i, phase, n : integer;
  with Q do
  if TIR(Qupper, 1, Qsize) then
     i := F&A(Qrear, 1) mod 2*Qsize;
     if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
     phase := i div Qsize;
     i := i mod Qsize;
     Weak_P(Cell[i, phase].Access_Sem);
        Cell[i, phase].Value := Item;
        Cell[i, phase].Mult  := M;
        Item↑.Cell_Loc := address(Cell[i, phase]);
        n := leaf corresponding to Cell[i, phase];
        Cnt[n, phase] := M;
     Weak_V(Cell[i, phase].Access_Sem);

     n := parent(n);
     while  n ≠ Root do   ascend tree
        F&A(Cnt[n, phase], M);
        n := parent(n)
     end while;

     Read_Request(Phase_Act);
        F&A(Cnt[Root, phase], M);
        if phase = DPhase then F&A(PhaseMult, M) end if;
     Read_Release(Phase_Act);
     F&A(Qlower, M);
     return true
  else
     return false
  end if
  end with
end function Seq_LogMQInsert;
```

```
function Seq_LogMQDelete (Item : Object_Pointer;
                          Q    : Seq_LogMQueue);
var d, phase, n, s : integer;

   with Q do
   if TDR(Qlower, 1) then
      Calc_Leaf_Site(n)_V2;
      d := cell index corresponding to leaf n;
      Item := Cell[d, phase].Value;
      if F&A(Cell[d, phase].Mult, −1) = 1 then  final delete
         Item↑.Cell_Loc := nil;
         F&A(Qupper, −1);
         if F&A(#FD, −1) = 1 then
            #FD := Qsize;
            Open_Next_Phase   defined in sect. 3.2.1.2
         end if
      end if;
      return true
   else
      return false
   end if
   end with
end function Seq_LogMQDelete;
```

```
macro Calc_Leaf_Site(n)_V2;
    supporting type II and writer priority type I IA
    s := 0;
    repeat
        if PhaseMult > 0 then s := F&A(PhaseMult, −1) end if
    until s > 0;
    phase := DPhase;        deletion can join current phase
    F&A(Cnt[Root, phase], −1);
    n := Root;
    while n's left son is not a leaf do
        if TDR(Cnt[leftson(n), phase], 1) then
            n := leftson(n)
        elseif TDR(Cnt[rightson(n), phase], 1) then
            n := rightson(n)
        end if
    end while;
    loop until find non-empty leaf of n using TDRNI
    while true do
        if Cnt[leftson(n), phase] > 0 then
            if F&A(Cnt[leftson(n), phase], −1) > 0 then
                n := leftson(n);
                exit while
            end if
        end if;
        if Cnt[rightson(n), phase] > 0 then
            if F&A(Cnt[rightson(n), phase], −1) > 0 then
                n := rightson(n);
                exit while
            end if
        end if
    end while;
end macro Calc_Leaf_Site(n)_V2;
```

```
function Seq_LogMQIA_TypeI (Item : Object_Pointer;
                            Q    : Seq_LogMQueue;
                            ϕ    : function);
var c : CellElement_Pointer;
    f, n, p : integer;
    with Q do
    Seq_LogMQIA_TypeI := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Weak_P(c↑.Access_Sem);
            if c = Item↑.Cell_Loc then
                n := leaf corresponding to cell c;
                p := phase of cell c;
                if Cnt[n, p] > 0 then
                    f := F&A(Cnt[n, p], –MaxMult);   block deletions
                    if f > 0 then                     final deletion not done
                        repeat until f = c↑.Mult;     wait for deletions
                        ϕ(Item);                      access Item
                        Cnt[n, p] := f;               restore leaf count
                        Seq_LogMQIA_TypeI := true
                    end if
                end if
            end if;
        Weak_V(c↑.Access_Sem);
    end if
    end with
end function Seq_LogMQIA_TypeI;
```

The above implementation could also be implemented without modifying insertion and deletion as described at the start of this section. Leaf counts could be updated via F&As and TDRs. However, we believe that a F&A-TDR based approach would be no more efficient, especially with regard to non-final deletions, than the existing implementation. Furthermore, a F&A-TDR approach would not allow us simply to apply the linear multiqueue techniques already developed. They would have to be modified to be compatible with this new approach. Finally, as we shall see throughout the thesis, it is very simple to introduce subtle errors or timing problems into applications based on TDR. These are generally attributable to the function's non-atomicity. One such example has already been given in section 3.2.1.2 and another one will be presented in the next section.

3.2.1.4.2. Type Two

In this section readers/writers and writer priority readers/writers implementations of RaiseMult and LowerMult are given. Recall that these operations increment and decrement an item's multiplicity, respectively, and that LowerMult removes an item if the decrement would otherwise make the item's multiplicity nonpositive.

3.2.1.4.2.1. A Readers/Writers Implementation

Explicit readers/writers protocols are used by deletions (the readers) and insertions, RaiseMults, and LowerMults (the writers) to access cells. Each cell is treated as an active deletion site by the last two operations. RaiseMult and LowerMult update cell multiplicities and leaf counts in the same manner as their corresponding linear multiqueue versions update front cell multiplicities and the queue component *FrontMult*. Therefore, insertions and deletions operate on cell multiplicities and leaf counts using the assignment-TDRNI approach of the preceding section.

RaiseMult acts like an insertion when incrementing an item's multiplicity by a value K. The multiplicity of the item's cell and all nodes on the path from the cell's leaf to the root are incremented by K.

Insertion and RaiseMult are mutually exclusive (when accessing cells) to preclude the permanent blocking of deletion. Suppose they were not. Let an insertion place an item I with multiplicity M into cell c, set c's multiplicity to M, and mark I as queued. A RaiseMult with an increment of K then accesses I, sets c's multiplicity and leaf count to M+K, and then ascends the tree incrementing each interior node by K. The insertion then sets c's leaf count to M, effectively lowering the count by K, and then ascends the tree incrementing each interior node by M. Now, when $M + K$ deletions descend the tree, K of them wait forever for c's leaf count to become positive.

When LowerMult decrements an item's cell multiplicity by a value K, it removes K items from the tree. To maintain the node counts, LowerMult could ascend the tree and decrement each node's count on the path from the cell's leaf to the root by K, but if it met a deletion, one of them would have to retreat. As this method seems fraught with problems, we elect to implement the following, simpler alternative. For each phase p, we associate with each leaf a counter *LeafNilPool*[p]. Recall that each leaf for each phase p also has a counter *Cnt*[p] that equals the multiplicity of its corresponding phase p cell (when there is no queue activity). LowerMult, upon decrementing a phase p cell's multiplicity by K, increments and decrements by K the corresponding leaf components $LeafNilPool[p]$ and $Cnt[p]$, respectively. Thus, each copy of an item that is removed from a cell is replaced with a nil item in the corresponding leaf nil pool. A leaf's parent's phase p count equals the sum of its son's $LeafNilPool[p]$ and $Cnt[p]$ components.

After a deletion descends the tree to a node *N* whose children are leaves, it first tries to obtain an item from either one of the cells attached to a child of *N*. If both cells are empty, it then tries to acquire a nil item from either child's nil pool. It cycles until it obtains an item from one of these two sources.

To preserve the one-deletion-phase-at-a-time constraint, and thereby the FIFO queue property, all leaf nil pools of the current phase must be emptied (via deletion) of their items before the next phase is opened. Suppose this were not done, and a deletion that obtained an item from such a pool simply returned to its caller. Let the last two active deletions of a phase p reach a leaf l at the same time, where l's $LeafNilPool[p]$ and $Cnt[p]$ components both equal one. That is, there is one phase p item in l's phase p cell c and one phase p nil item in the pool. One deletion D_1 pauses at this point. The other deletion decrements l's $Cnt[p]$ component, empties c, and opens the next phase. In time, a phase $p+2$ insertion fills c and updates l's $Cnt[p+2]$ component before all of the phase $p+1$ deletions have begun. Since only two insertion phase counts are maintained in a tree, a leaf's $Cnt[p+2]$ component actually maps onto its $Cnt[p]$ component. Therefore when D_1, the last phase p deletion, finally decrements l's $Cnt[p]$ component (which has been incremented by a phase $p+2$ insertion), it obtains a phase $p+2$ item while phase $p+1$ items have yet to be deleted.

To guarantee that all of the deletions of a phase p have completed, we associate with a phase p a final deletion phase counter $\#FD[p]$ that equals the number of phase p final and nil pool deletions that remain to be done. Initially, for a phase p, $\#FD[p]$ equals *Qsize*. When LowerMult reduces a phase p item's multiplicity by a value K, it increments $\#FD[p]$ and then the $LeafNilPool[p]$ component of this item's leaf by K.[26] Deletions from nil pools are treated just like final deletions from cells: After obtaining a nil phase p item from a leaf, a deletion decrements $\#FD[p]$ by 1 to see whether it should open up the next deletion phase.[27] A final deletion counter is associated with each phase since an item belonging to either the current or next deletion phase can have its multiplicity lowered.

---

[26] If these increments were reversed, a deletion from a nil pool could open up the next site prematurely.

[27] See the third version of the Calc_Leaf_Site macro below.

```
Seq_LogMQueue =     MultiQueue Supporting Type Two IA
                    record
                       Qlower, Qupper, Qsize, Qrear,
                       PhaseMult, DPhase : integer;
                       Phase_Act         : integer;
                       #FD               : array[0..1] of integer;
                       Cnt               : array[1..2*Qsize–1, 0..1] of integer;
                       LeafNilPool       : array[1..Qsize, 0..1] of integer;
                       Cell              : array[0..Qsize–1, 0..1] of CellElement
                    end;

CellElement = record
                 Queue_Id : MQueue_Pointer;
                 Mult     : integer;
                 Value    : Object_Pointer;
                 RW_Act   : integer
              end;

MQueue_Pointer = ↑Seq_LogMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
            Cell_Loc : CellElement_Pointer;
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qupper, Qrear, DPhase, PhaseMult = 0;
   Cnt[], LeafNilPool[] = 0;  #FD[] := Qsize;
   Phase_Act = MaxR;
   for 0≤j<Qsize, 0≤k≤1,
     Cell[j, k].Queue_Id      = address of queue;
     Cell[j, k].Mult          = 0;
     Cell[j, k].RW_Act        = MaxR;
```

```
function Seq_LogMQInsert (Item : Object_Pointer;
                          M    : integer;      item's multiplicity
                          Q    : Seq_LogMQueue);
var i, phase, n : integer;

  with Q do
  if TIR(Qupper, 1, Qsize) then
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Write_Request(Cell[i, phase].RW_Act);
      Cell[i, phase].Value := Item;
      Cell[i, phase].Mult  := M;
      Item↑.Cell_Loc := address(Cell[i, phase]);
      n := leaf corresponding to Cell[i, phase];
      Cnt[n, phase] := M;
    Write_Release(Cell[i, phase].RW_Act);

    n := parent(n);
    while  n ≠ Root do   ascend tree
      F&A(Cnt[n, phase], M);
      n := parent(n)
    end while;

    Read_Request(Phase_Act);
      F&A(Cnt[Root, phase], M);
      if phase = DPhase then F&A(PhaseMult, M) end if;
    Read_Release(Phase_Act);
    F&A(Qlower, M);
    return true
  else
    return false
  end if
  end with
end function Seq_LogMQInsert;
```

```
function Seq_LogMQDelete (Item : Object_Pointer;
                          Q    : Seq_LogMQueue);
var d, phase, n, s : integer;

  with Q do
  if TDR(Qlower, 1) then
    Calc_Leaf_Site(n)_V3;
    d := cell index corresponding to leaf n;
    Read_Request(Cell[d, phase].RW_Act);
      Item := Cell[d, phase].Value;
      if F&A(Cell[d, phase].Mult, −1) = 1 then  final delete
        if Item ≠ nil then Item↑.Cell_Loc := nil end if;
        F&A(Qupper, −1);
        if F&A(#FD[phase], −1) = 1 then
          #FD[phase] := Qsize;
          Open_Next_Phase   defined in sect. 3.2.1.2
        end if
      end if;
    Read_Release(Cell[d, phase].RW_Act)
    return true
  else
    return false
  end if
  end with
end function Seq_LogMQDelete;
```

```
macro Calc_Leaf_Site(n)_V3;
    supporting type II IA on sequentially allocated logarithmic multiqueues
    s := 0;
    repeat
        if PhaseMult > 0 then s := F&A(PhaseMult, −1) end if
    until s > 0;
    phase := DPhase;         deletion can join current phase
    F&A(Cnt[Root, phase], −1);
    n := Root;
    while n's left son is not a leaf do
        if TDR(Cnt[leftson(n), phase], 1) then
            n := leftson(n)
        elseif TDR(Cnt[rightson(n), phase], 1) then
            n := rightson(n)
        end if
    end while;

    while true do  loop until find item in cell or nil pool
        if Cnt[leftson(n), phase] > 0 then
            if F&A(Cnt[leftson(n), phase], −1) > 0 then
                n := leftson(n);
                exit while
            end if
        end if;
        if Cnt[rightson(n), phase] > 0 then
            if F&A(Cnt[rightson(n), phase], −1) > 0 then
                n := rightson(n);
                exit while
            end if
        end if;
        if    TDR(LeafNilPool[leftson(n), phase], 1) or
              TDR(LeafNilPool[rightson(n), phase], 1) then
            if F&A(#FD[phase], −1) = 1 then
                #FD[phase] := Qsize;
                Open_Next_Phase   defined in sect. 3.2.1.2
            end if;
            Item := nil;
            return true
        end if
    end while;
end macro Calc_Leaf_Site(n)_V3;
```

```
function Seq_LogMQRaiseMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LogMQueue);
var c : CellElement_Pointer;
    f, n, p : integer;

    with Q do
    Seq_LogMQRaiseMult := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Write_Request(c↑.RW_Act);
            if c = Item↑.Cell_Loc then
                n := leaf corresponding to c;
                p := phase of c;
                f := F&A(Cnt[n, p], –MaxMult);  suspend deletions
                c↑.Mult := c↑.Mult + K;
                if f > 0 then  f copies left
                    Cnt[n, p] := f + K
                else  all copies taken
                    Cnt[n, p] := K
                end if;
                Write_Release(c↑.RW_Act);

                n := parent(n);
                while  n ≠ Root do   ascend tree
                    F&A(Cnt[n, p], K);
                    n := parent(n)
                end while;

                Read_Request(Phase_Act);
                    F&A(Cnt[Root, p], K);
                    if p = DPhase then F&A(PhaseMult, K) end if;
                Read_Release(Phase_Act);
                F&A(Qlower, K);
                Seq_LogMQRaiseMult := true;
                return
            end if;
        Write_Release(c↑.RW_Act)
    end if
    end with
end function Seq_LogMQRaiseMult;
```

```
function Seq_LogMQLowerMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LogMQueue;
                             M    : integer);
var c : CellElement_Pointer;
    f, n, p : integer;
    with Q do
    Seq_LogMQLowerMult := false;  M := 0;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Write_Request(c↑.RW_Act);
            if c = Item↑.Cell_Loc then
                n := leaf corresponding to c;
                p := phase of c;
                if Cnt[n, p] > 0 then
                    f := F&A(Cnt[n, p], –K);
                    if f > K then lower multiplicity
                        M := f;
                        c↑.Mult := c↑.Mult – K;
                        F&A(#FD[p], K);
                        F&A(LeafNilPool[n, p], K);
                        Seq_LogMQLowerMult := true
                    elseif f > 0 then  remove item
                        F&A(c↑.RW_Act, MaxR–1); become a reader
                        repeat until c↑.Mult = f;        wait for deletions
                        Item↑.Cell_Loc := nil;
                        M := f;   number of copies removed
                        c↑.Mult := 1;
                        c↑.Value := nil;
                        F&A(#FD[p], M – 1);
                        F&A(LeafNilPool[n, p], M – 1);
                        Cnt[n, p] := 1;
                        Read_Release(c↑.RW_Act);
                        Seq_LogMQLowerMult := true;
                        return
                    end if
                end if
            end if;
        Write_Release(c↑.RW_Act)
    end if
    end with
end function Seq_LogMQLowerMult;
```

In the previous section it was claimed that type one interior access could be implemented with insertion and deletion incrementing and decrementing leaf counts with F&As and TDRs, respectively. A similar claim for the implementation of type two interior access will not be made. Instead, we will present what appears to be a straightforward F&A-TDR based implementation of type two interior access that, as it turns out, contains at least one subtle timing problem attributable to the non-atomicity of TDR. The point of this exercise is to show that such an implementation is more complex than one might at first assume.

The operations differ from those above in that insertion, RaiseMult, and LowerMult modify leaf counts via F&As, and deletion modifies these counts via TDR. To raise an item I's multiplicity by a value R, RaiseMult locks (via a write request) I's cell c, increments c's multiplicity and the corresponding leaf's count by R via F&As (without first decrementing the latter by $-$*MaxMult*), and then unlocks c (via a write release). To lower I's multiplicity by a value of L, LowerMult locks c and decrements the leaf count by L via a F&A. If the leaf count's old value V is nonpositive, LowerMult increments the leaf count by L via a F&A, unlocks c, and returns false as a final deletion has begun. If V is greater than L, LowerMult decrements c's multiplicity by L, unlocks c, and returns true. If V is positive but not greater than L, LowerMult removes I by placing a nil value in c, decrementing c's multiplicity by $V-1$, and incrementing the leaf count by $L-(V-1)$ via a F&A. It then unlocks c and returns true. While a cell c is locked by either RaiseMult or LowerMult, deletions may perform TDRs on c's leaf count, but they may not access c (their read requests will be denied).

This implementation is correct as long as a LowerMult operation on an item I in a cell c is not performed while deletions are performing TDRs on c's leaf count. Otherwise, the LowerMult operation may fail to access I, even though I is queued, or allow too may copies of an item to be made by deletions, or deadlock itself and deletion.

For example, suppose an item I with multiplicity one is in a cell c.

(1) A RaiseMult locks c.

(2) N deletions start to perform TDRs on c's leaf count. Each TDR decrements the count by 1 via a F&A. One TDR returns true. Before each of the remaining TDRs increments the count by 1 via a F&A and returns false:

  (2.1) RaiseMult increments the leaf count and the cell's multiplicity by R via F&As and unlocks c. The leaf count's value V equals $R-(N-1)$, and the cell's multiplicity *Mult* equals $R+1$.

  (2.2) A LowerMult locks c and decrements the leaf count by L via a F&A.

(3) The remaining $N-1$ TDRs complete. The leaf count's new value V' equals $R-L$.

(4) The LowerMult operation proceeds.

Two errors are now possible. First, if V (the leaf count's value at the time the LowerMult is performed) is nonpositive, LowerMult increments the leaf count by L and returns false as it appears that a final deletion has begun. But, in fact, this is not the case: R deletions will access I after LowerMult returns.

The second error can occur if V is positive but not greater than L ($L \geq V > 0$). In this case, LowerMult removes I. Before doing so, LowerMult waits for deletions that are reading c to complete, which occurs when c's multiplicity *Mult* equals V. In the present scenario, this means that LowerMult expects $Mult - V$ or N deletions to access c. If $R > L$, then the actual number of deletions that will access c equals $1 + V'$: the deletion whose TDR succeeded in step (2) and the $R - L$ deletions that perform TDRs on the leaf count after step (3) is completed. If $N = 1 + V'$, then the expected number of deletions will indeed access c. If $N > 1 + V'$, an insufficient number of deletions will be performed. In this case, *Mult* will never equal V, and, consequently, LowerMult will wait forever. If $N < 1 + V'$, then more than the expected number N of deletions will be performed. Too many copies of I may be made by deletions, and, if LowerMult fails to detect the completion of the Nth deletion (i.e., when $Mult = V$), it may then enter an infinite loop. On the other hand, if $L \geq R$, then LowerMult has made the leaf count nonpositive ($V' \leq 0$). Only one deletion, the one whose TDR succeeded in step 2, will access c. Therefore, if $N = 1$, LowerMult will be waiting for the correct number of deletions to complete. Otherwise, it will wait forever. When LowerMult enters an infinite loop, no final deletion will be performed at c, and, thus, the next deletion phase will never be opened. As a result, deletion activity ceases once the other sites of the current phase are emptied.

3.2.1.4.2.2. A Writer Priority Readers/Writers Implementation

The following writer priority readers/writers implementation is derived from the preceding implementation, the logarithmic multiqueue writer priority readers/writers type one interior access implementation of section 3.2.1.4.1.2, and the linear multiqueue writer priority readers/writers type two interior access implementation of section 3.1.1.4.2.2. It is given without further comment.

```
Seq_LogMQueue =      MultiQueue Supporting Type Two IA
                     record
                        Qlower, Qupper, Qsize, Qrear,
                        PhaseMult, DPhase : integer;
                        Phase_Act         : integer;
                        #FD               : array[0..1] of integer;
                        Cnt               : array[1..2*Qsize–1, 0..1] of integer;
                        LeafNilPool       : array[1..Qsize, 0..1] of integer;
                        Cell              : array[0..Qsize–1, 0..1] of CellElement
                     end;


CellElement = record
                 Queue_Id    : MQueue_Pointer;
                 Mult        : integer;
                 Value       : Object_Pointer;
                 Access_Sem  : Weak_Semaphore
              end;

MQueue_Pointer = ↑Seq_LogMQueue;

CellElement_Pointer = ↑CellElement;

Object = record
            Cell_Loc : CellElement_Pointer;
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
   Qlower, Qupper, Qrear, DPhase, PhaseMult = 0;
   Cnt[], LeafNilPool[] = 0;  #FD[] := Qsize;
   Phase_Act = MaxR;
   for 0≤j<Qsize, 0≤k≤1,
      Cell[j, k].Queue_Id       = address of queue;
      Cell[j, k].Mult           = 0;
      Cell[j, k].Access_Sem     = 1;
```

```
function Seq_LogMQInsert (Item : Object_Pointer;
                          M    : integer;     item's multiplicity
                          Q    : Seq_LogMQueue);
var i, phase, n : integer;

   with Q do
   if TIR(Qupper, 1, Qsize) then
      i := F&A(Qrear, 1) mod 2*Qsize;
      if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
      phase := i div Qsize;
      i := i mod Qsize;
      Weak_P(Cell[i, phase].Access_Sem);
         Cell[i, phase].Value := Item;
         Cell[i, phase].Mult  := M;
         Item↑.Cell_Loc := address(Cell[i, phase]);
         n := leaf corresponding to Cell[i, phase];
         Cnt[n, phase] := M;
      Weak_V(Cell[i, phase].Access_Sem);

      n := parent(n);
      while  n ≠ Root do   ascend tree
         F&A(Cnt[n, phase], M);
         n := parent(n)
      end while;

      Read_Request(Phase_Act);
         F&A(Cnt[Root, phase], M);
         if phase = DPhase then F&A(PhaseMult, M) end if;
      Read_Release(Phase_Act);
      F&A(Qlower, M);
      return true
   else
      return false
   end if
   end with
end function Seq_LogMQInsert;
```

```
function Seq_LogMQDelete (Item : Object_Pointer;
                          Q     : Seq_LogMQueue);
var d, phase, n, s : integer;
   with Q do
   if TDR(Qlower, 1) then
      Calc_Leaf_Site(n)_V3;   defined in sect. 3.2.1.4.2.1
      d := cell index corresponding to leaf n;
      Item := Cell[d, phase].Value;
      if F&A(Cell[d, phase].Mult, −1) = 1 then  final delete
         if Item ≠ nil then Item↑.Cell_Loc := nil end if;
         F&A(Qupper, −1);
         if F&A(#FD[phase], −1) = 1 then
            #FD[phase] := Qsize;
            Open_Next_Phase   defined in sect. 3.2.1.2
         end if
      end if;
      return true
   else
      return false
   end if
   end with
end function Seq_LogMQDelete;
```

```
function Seq_LogMQRaiseMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LogMQueue);
var c : CellElement_Pointer;
    n, f, p : integer;

    with Q do
    Seq_LogMQRaiseMult := false;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Weak_P(c↑.Access_Sem);
            if c = Item↑.Cell_Loc then
                n := leaf corresponding to c;
                p := phase of c;
                if Cnt[n, p] > 0 then
                    f := F&A(Cnt[n, p], –MaxMult);
                    if f > 0 then  f copies left
                        F&A(c↑.Mult, K);
                        Cnt[n, p] := f + K;
                        Weak_V(c↑.Access_Sem);

                        n := parent(n);
                        while  n ≠ Root do   ascend tree
                            F&A(Cnt[n, p], K);
                            n := parent(n)
                        end while;

                        Read_Request(Phase_Act);
                            F&A(Cnt[Root, p], K);
                            if p = DPhase then F&A(PhaseMult, K) end if;
                        Read_Release(Phase_Act);
                        F&A(Qlower, K);
                        Seq_LogMQRaiseMult := true;
                        return
                    end if
                end if
            end if;
        Weak_V(c↑.Access_Sem)
    end if
    end with
end function Seq_LogMQRaiseMult;
```

```
function Seq_LogMQLowerMult (Item : Object_Pointer;
                             K    : integer;
                             Q    : Seq_LogMQueue;
                             M    : integer);
var c : CellElement_Pointer;
    f, n, p : integer;

    with Q do
    Seq_LogMQLowerMult := false;  M := 0;
    c := Item↑.Cell_Loc;
    if c ≠ nil and c↑.Queue_Id = address(Q) then
        Weak_P(c↑.Access_Sem);
            if c = Item↑.Cell_Loc then
                n := leaf corresponding to c;
                p := phase of c;
                if Cnt[n, p] > 0 then
                    f := F&A(Cnt[n, p], –MaxMult);
                    if f > K then lower multiplicity
                        M := f;
                        F&A(c↑.Mult, –K);
                        F&A(#FD[p], K);
                        F&A(LeafNilPool[n, p], K);
                        Cnt[n, p] := f – K;
                        Seq_LogMQLowerMult := true
                    elseif f > 0 then  remove item
                         repeat until c↑.Mult = f;       wait for deletions
                         Item↑.Cell_Loc := nil;
                         M := f;   number of copies removed
                         c↑.Mult := 1;
                         c↑.Value := nil;
                         F&A(#FD[p], M – 1);
                         F&A(LeafNilPool[n, p], M – 1);
                         Cnt[n, p] := 1;
                         Seq_LogMQLowerMult := true
                    end if
                end if
            end if;
        Weak_V(c↑.Access_Sem)
    end if
    end with
end function Seq_LogMQLowerMult;
```

3.2.2. Linked List Allocation

A linked allocated logarithmic multiqueue has the same structure as a sequentially allocated logarithmic multiqueue except the cells of the latter are replaced with serially accessible linked lists. Each leaf of the count tree is associated with the head item of a single linked list–two lists are *not* needed since a standard list can contain items from insertions belonging to different phases without introducing the incorrect behavior observed for sequentially allocated multiqueues (as described in section 3.2.1.1). The linked list implementation does require *#p* counters per tree node to support the concurrent execution of *#p* insertion and *#p*–1 deletion phases. Note that a given linked list may hold items belonging to more than *#p* distinct phases.

The insertion and deletion algorithms below restrict the number of insertion and deletion phases to two and one respectively (i.e., *#p*=2) and do not support interior access. Following the presentation of these algorithms, interior access operations on linked allocated multiqueues are given.

3.2.2.1. Insertion and Deletion

An insertion of an item with multiplicity M calculates a site i (using the sequentially allocated logarithmic multiqueue technique of section 3.2.1.2) and then appends an item to list i. If the list was empty, the insertion increments the tree's counters by M, and, in addition, if the insertion's phase matches the current deletion phase, the insertion adds M to the current deletion phase multiplicity *PhaseMult*. An insertion does not ascend the tree if its item is added to a non-empty list. Instead, the item's multiplicity will be promoted up the tree by a deletion when the item moves to the head of its list.

A deletion descends the tree, guided by the counters for its phase, and copies the item from the head of the indicated list. A final deletion dequeues its item and moves the item's successor S to the head of the list. If a final deletion is also the last deletion of its phase, the next phase is opened (by updating *DPhase* and *PhaseMult*). Regardless, a final deletion then returns if the deleted item has no successor. Otherwise, it acts as a phase p+1 insertion, incrementing the node counts on the path from the list's leaf to the root by S's multiplicity M. In addition, the deletion adds M to *PhaseMult* if phase $p+1$ has been opened. (Before the deletion reaches the tree's root, the *last* deletion of phase p can occur and advance the deletion phase to p+1.) When the last deletion of phase p ascends the tree carrying the multiplicity of an item from phase $p+1$, the multiplicity is always added to *PhaseMult*.

Insertions and last deletions are, respectively, readers and writers of the root, the current deletion phase index *DPhase*, and the deletion phase multiplicity *PhaseMult*. A last deletion cannot fail (i.e., starve) to open the next phase. In the worst case, the last deletion waits for *Qsize* insertions (and/or final deletions acting as insertions), one per list, to ascend the tree and access the root before it performs its write activities. However, as noted throughout this chapter, incoming deletions can starve in attempting to join the current deletion phase if *PhaseMult* is not accessed via a critical code section (within the Calc_Leaf_Site macro).

Insertions calculate their phase numbers through fetch-and-adds on the queue counter *Qrear*. It is still sufficient to maintain this counter modulo twice the queue size as is done in the sequential version by having the last insertion of an odd phase decrement *Qrear* by 2**Qsize* when *Qrear* reaches twice the queue size. However, the maximum value that the counter can attain is greater than the value $(3 * Qsize - 1)$ that it can reach in the sequential version. In the latter, since only two insertion phases are active at a time, only one insertion decrements *Qrear* at any given time. This is not true in the linked list version as multiple insertions, each being the last insertion of a different odd phase, may be executing concurrently. Let us assume that the maximum number of active tasks equals *B* and that each task performs $2 * Qsize$ insertions. Each task's last insertion increments *Qrear* to an even multiple of the queue size (making the insertion the last insertion of an odd phase) and then decrements it by twice the queue size. After all of the increments are performed, but before any decrement is done, *Qrear* attains the maximum value of $2 * B * Qsize$.

```
Linked_LogMQueue =    Linked Allocated Logarithmic MultiQueue
                      record
                         Qlower, Qsize, Qrear, #FD,
                         PhaseMult, DPhase : integer;
                         Phase_Act : integer;
                         Cnt     : array[1..2*Qsize–1,0..1] of integer;
                         List    : array[0..Qsize–1] of ListElement
                      end;

ListElement = record
                 HeadMult    : integer;
                 Head        : Object_Pointer;
                 Tail        : Pointer to ListElement or Object;
                 Access_Sem  : Weak_Semaphore
              end;

ListElement_Pointer = ↑ListElement;

Object = record
            Mult          : integer;
            Next          : Object_Pointer;
            Application_Defined_Attributes
         end;
```

```
Object_Pointer = ↑Object;

Initially:
    Qlower, Qrear, DPhase, PhaseMult, Cnt[] = 0;
    #FD = Qsize;
    Phase_Act = MaxR;
    for 0≤j<Qsize,
      List[j].HeadMult        = 0;
      List[j].Tail            = address(List[j]);
      List[j].Access_Sem      = 1;

procedure Linked_LogMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LogMQueue);
var i, phase, n : integer;
    with Q do
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize−1 then F&A(Qrear, −2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Item↑.Next := nil;
    Item↑.Mult := M;

    Weak_P(List[i].Access_Sem);
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
        if List[i].Head := Item  then  promote multiplicity
            List[i].HeadMult := M;
            Weak_V(List[i].Access_Sem);

            n := leaf corresponding to list i;
            while  n ≠ Root do   ascend tree
                F&A(Cnt[n, phase], M);
                n := parent(n)
            end while;

            Read_Request(Phase_Act);
                F&A(Cnt[Root, phase], M);
                if phase = DPhase then F&A(PhaseMult, M) end if;
            Read_Release(Phase_Act)
        else
            Weak_V(List[i].Access_Sem)
        end if;

    F&A(Qlower, M)
    end with
end procedure Linked_LogMQInsert;
```

```
function Linked_LogMQDelete (Item : Object_Pointer;
                             Q    : Linked_LogMQueue):
var d, m, n, phase, s : integer;
  with Q do
  if TDR(Qlower, 1) then
    Calc_Leaf_Site(n)_V1;   defined in sect. 3.2.1.2
    d := list corresponding to leaf n;
    Item := List[d].Head;
    if F&A(List[d].HeadMult, −1) = 1 then   final delete
      Weak_P(List[d].Access_Sem);     dequeue Item
        m := Item↑.Next↑.Mult;
        List[d].HeadMult := m;
        List[d].Head := Item↑.Next;
        if List[d].Head = nil then
          List[d].Tail := address(List[d])
        end if;
      Weak_V(List[d].Access_Sem);
      if F&A(#FD, −1) = 1 then
        #FD := Qsize;
        Open_Next_Phase   defined in sect. 3.2.1.2
      end if;
      if m > 0 then                          list is not empty
        phase := (phase+1) mod 2;
        while  node ≠ Root do                ascend tree like an insertion
          F&A(Cnt[node, phase], m);
          node := parent(node)
        end while;
        Read_Request(Phase_Act);             update root like an insertion
          F&A(Cnt[Root, phase], m);
          if phase = DPhase then F&A(PhaseMult, m) end if;
        Read_Release(Phase_Act)
      end if
    end if;
    return true
  else
    return false
  end if
  end with
end function Linked_LogMQDelete;
```

3.2.2.2. Multi-Deletion

Multi-deletion from a linked list allocated logarithmic multiqueue is a combination of two of the deletion algorithms already presented. First, the multi-deletion descends a queue's tree like a multi-deletion on a sequentially allocated logarithmic multiqueue (section 3.2.1.3) and collects its items. Second, if it is the final deletion of an item *I*, it follows the deletion algorithm of the preceding section, ascending the tree if *I* has a list successor.

```
function Linked_LogMulti-Delete (Table : array[1..M] of Object_Pointer;
                                 M     : integer;
                                 Q     : Linked_LogMQueue);
var d, i, m, n, phase, r, s, son, total_granted : integer;
   item : Object_Pointer;
   g : array[1..2*MaxQsize–1] of integer;

   with Q do
   if TDR(Qlower, M) then
      r := M;  initial request
      total_granted := i := 0;
      repeat  items also taken from next phase if PhaseMult<r
         g := s := 0;
         repeat
            if PhaseMult > 0 then s := F&A(PhaseMult, –r) end if
         until s > 0;
         phase := DPhase;
         g[Root] := s min r;   number of items to obtain from this phase
         total_granted := total_granted + g[Root];
         r := r – g[Root];   number to be obtained in next phase
         F&A(Cnt[Root, phase], –g[Root]);
         n := Root;

         repeat   make descents until get g[Root] items from current phase
            while n is not a leaf do
               if Request(Cnt[leftson(n), phase], g[n], g[leftson(n)]) then
                  g[n] := g[n] – g[leftson(n)];
                  n := leftson(n)
               elseif Request(Cnt[rightson(n), phase], g[n], g[rightson(n)]) then
                  g[n] := g[n] – g[rightson(n)];
                  n := rightson(n)
               end if
            end while;
            d := list corresponding to leaf n;
            Table[i+1..i+g[n]] := List[d].Head;
            i := i + g[n];

            if F&A(List[d].HeadMult, –g[n]) = g[n] then  final delete
               item := List[d].Head;
               Weak_P(List[d].Access_Sem);     dequeue item
                  m := item↑.Next↑.Mult;
                  List[d].HeadMult := m;
                  List[d].Head := item↑.Next;
                  if List[d].Head = nil then
                     List[d].Tail := address(List[d])
                  end if;
               Weak_V(List[d].Access_Sem);
               if F&A(#FD, –1) = 1 then
                  #FD := Qsize;
                  Open_Next_Phase   defined in sect. 3.2.1.2
               end if;
```

```
                    final delete processing continued
                     if m > 0 then                    list is not empty
                          phase := (phase+1) mod 2;
                          while  n ≠ Root do          ascend tree like an insertion
                               F&A(Cnt[n, phase], m);
                               n := parent(n)
                          end while;
                          Read_Request(Phase_Act);  update root like an insertion
                               F&A(Cnt[Root, phase], m);
                               if phase = DPhase then F&A(PhaseMult, m) end if;
                          Read_Release(Phase_Act)
                     end if
                end if;   F&A(List[d].HeadMult ...
                if i < total_granted then
                     repeat   ascend tree until find non-empty subtree
                          n := parent(n)
                     until g[n] > 0
                end if
           until i = total_granted
      until r = 0;
      return true
 else
      return false
 end if
 end with
end function Linked_LogMulti-Delete;
```

3.2.2.3. Interior Access

Readers/writers and writer priority readers/writers implementations for type one and two interior access are presented in the next two sections. Their descriptions are brief as very few new concepts or techniques are introduced. Once again we remark that the readers/writers implementations can be converted into startvation-free implementations by using fair readers/writers protocols and having incoming deletions test and decrement the queue component *PhaseMult* within a critical code section (as described in section 3.2.1.2).

3.2.2.3.1. Type One

3.2.2.3.1.1. A Readers/Writers Implementation

In this implementation, type one operations do not reference a multiqueue's tree of counters, but only the linked lists. Thus, the interior access algorithm differs from the one given for the readers/writers implementation of sequentially allocated linear multiqueues (of section 3.1.1.4.1) only by the substitution of lists for cells. IA operations and insertions are writers of lists and deletions are readers of lists. Since a final deletion is the last reader of a head list item, it dequeues this item without requesting write access. Insertion differs from the corresponding sequentially allocated logarithmic multiqueue algorithm in that sites (i.e., lists) now must be locked by the operations. Multiple insertions can be performed on a site, and each queued item can be the target of an interior access operation.

```
Linked_LogMQueue =      MultiQueue Supporting Type I IA
                        record
                           Qlower, Qsize, Qrear, #FD,
                           PhaseMult, DPhase : integer;
                           Phase_Act      : integer;
                           Cnt            : array[1..2*Qsize–1, 0..1] of integer;
                           List           : array[0..Qsize–1] of ListElement
                        end;

ListElement = record
                 Queue_Id : MQueue_Pointer;
                 HeadMult : integer;
                 Head     : Object_Pointer;
                 Tail     : Pointer to ListElement or Object;
                 RW_Act   : integer
              end;

MQueue_Pointer = ↑Linked_LogMQueue;

ListElement_Pointer = ↑ListElement;

Object = record
            List_Loc : ListElement_Pointer;
            Mult     : integer;
            Next     : Object_Pointer;
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;
```

```
Initially:
    Qlower, Qrear, DPhase, PhaseMult, Cnt[] = 0;
    #FD = Qsize;
    Phase_Act = MaxR;
    for 0≤j<Qsize,
      List[j].Queue_Id        = address of queue;
      List[j].HeadMult        = 0;
      List[j].Tail            = address(List[j]);
      List[j].RW_Act          = MaxR;

procedure Linked_LogMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LogMQueue);
var i, phase, n : integer;
    with Q do
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Item↑.Next := nil;
    Item↑.Mult := M;

    Write_Request(List[i].RW_Act);
        Item↑.List_Loc := address(List[i]);
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
        if List[i].Head = Item then
            List[i].HeadMult := M;    promote multiplicity
            Write_Release(List[i].RW_Act);
            n := leaf corresponding to list i;

            while  n ≠ Root do   ascend tree
                F&A(Cnt[n, phase], M);
                n := parent(n)
            end while;

            Read_Request(Phase_Act);
                F&A(Cnt[Root, phase], M);
                if phase = DPhase then F&A(PhaseMult, M) end if;
            Read_Release(Phase_Act)
        else
            Write_Release(List[i].RW_Act)
        end if;
    F&A(Qlower, M)
    end with
end procedure Linked_LogMQInsert;
```

```
function Linked_LogMQDelete (Item : Object_Pointer;
                             Q    : Linked_LogMQueue):
var d, m, n, phase, s : integer;

  with Q do
  if TDR(Qlower, 1) then
    Calc_Leaf_Site(n)_V1;   defined in sect. 3.2.1.2
    d := list corresponding to leaf n;
    Read_Request(List[d].RW_Act);
      Item := List[d].Head;

      if F&A(List[d].HeadMult, –1) = 1 then   final delete
        dequeue Item - list is already locked through Read Request
        m := Item↑.Next↑.Mult;
        List[d].Head := Item↑.Next;
        List[d].HeadMult := m    head item's mult.
        Item↑.List_Loc := nil;
        if List[d].Head = nil then
          List[d].Tail := address(List[d])
        end if;
        Read_Release(List[d].RW_Act);

        if F&A(#FD, –1) = 1 then
          #FD := Qsize;
          Open_Next_Phase   defined in sect. 3.2.1.2
        end if;

        if m > 0 then                  list is not empty
          phase := (phase+1) mod 2;
          while  n ≠ Root do           ascend tree like an insertion
            F&A(Cnt[n, phase], m);
            n := parent(n)
          end while;

          Read_Request(Phase_Act);  update root like an insertion
            F&A(Cnt[Root, phase], m);
            if phase = DPhase then F&A(PhaseMult, m) end if;
          Read_Release(Phase_Act)
        end if
      else  non-final deletion
        Read_Release(List[d].RW_Act)
      end if;
    return true
  else
    return false
  end if
  end with
end function Linked_LogMQDelete;
```

```
function Linked_LogMQIA_TypeI (Item : Object_Pointer;
                               Q    : Linked_LogMQueue;
                               ϕ    : function);
var h : ListElement_Pointer;
   with Q do
   Linked_LogMQIA_TypeI := false;
   h := Item↑.List_Loc;
   if h ≠ nil and h↑.Queue_Id = address(Q) then
      Write_Request(h↑.RW_Act);
         if h = Item↑.List_Loc then
            ϕ(Item);        access Item
            Linked_LogMQIA_TypeI := true
         end if;
      Write_Release(h↑.RW_Act);
   end if
   end with
end function Linked_LogMQIA_TypeI;
```

3.2.2.3.1.2. A Writer Priority Readers/Writers Implementation

This writer priority readers/writers implementation, like those that precede it, eliminates the requirement that incoming deletions execute explicit readers/writers protocols in order to access a head list item. Interior access operations on such items bar deletions by making the corresponding leaf counts very negative. To access an interior list item, it is sufficient for an IA operation to lock the corresponding list as insertions, final deletions, and IA operations exclusively access lists to perform their operations.

```
Linked_LogMQueue =    MultiQueue Supporting Type I IA
                      record
                         Qlower, Qsize, Qrear, #FD,
                         PhaseMult, DPhase : integer;
                         Phase_Act    : integer;
                         Cnt          : array[1..2*Qsize–1, 0..1] of integer;
                         List         : array[0..Qsize–1] of ListElement
                      end;

ListElement = record
                 Queue_Id   : MQueue_Pointer;
                 HeadMult   : integer;
                 Head       : Object_Pointer;
                 Tail       : Pointer to ListElement or Object;
                 Access_Sem : Weak_Semaphore
              end;
```

```
MQueue_Pointer = ↑Linked_LogMQueue;

ListElement_Pointer = ↑ListElement;

Object = record
              List_Loc : ListElement_Pointer;
              Mult     : integer;
              Next     : Object_Pointer;
              Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;
```

Initially:

```
   Qlower, Qrear, DPhase, PhaseMult, Cnt[] = 0;
   #FD = Qsize;
   Phase_Act = MaxR;
   for 0≤j<Qsize,
     List[j].Queue_Id       = address of queue;
     List[j].HeadMult       = 0;
     List[j].Tail           = address(List[j]);
     List[j].Access_Sem     = 1;
```

```
procedure Linked_LogMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LogMQueue);
var i, phase, n : integer;
    with Q do
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Item↑.Next := nil;
    Item↑.Mult := M;

    Weak_P(List[i].Access_Sem);
        Item↑.List_Loc := address(List[i]);
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
        if List[i].Head = Item then
            List[i].HeadMult := M;    promote multiplicity
            n := leaf corresponding to list i;
            Cnt[n, phase] := M;
            Weak_V(List[i].Access_Sem);

            n := parent(n);
            while  n ≠ Root do   ascend tree
                F&A(Cnt[n, phase], M);
                n := parent(n)
            end while;

            Read_Request(Phase_Act);
                F&A(Cnt[Root, phase], M);
                if phase = DPhase then F&A(PhaseMult, M) end if;
            Read_Release(Phase_Act)
        else
            Weak_V(List[i].Access_Sem)
        end if;
    F&A(Qlower, M)
    end with
end procedure Linked_LogMQInsert;
```

```
function Linked_LogMQDelete (Item : Object_Pointer;
                             Q    : Linked_LogMQueue):
var d, m, n, phase, s : integer;

   with Q do
   if TDR(Qlower, 1) then
      Calc_Leaf_Site(n)_V2;   defined in sect. 3.2.1.4.1.2
      d := List index corresponding to leaf n;
      Item := List[d].Head;
      if F&A(List[d].HeadMult, −1) = 1 then   final delete
         Weak_P(List[d].Access_Sem);
            m := Item↑.Next↑.Mult;
            List[d].Head := Item↑.Next;
            List[d].HeadMult := m    head item's mult.
            Item↑.List_Loc := nil;
            if List[d].Head = nil then
               List[d].Tail := address(List[d])
            end if;
            Cnt[n, (phase+1) mod 2] := m;
         Weak_V(List[d].Access_Sem);

         if F&A(#FD, −1) = 1 then
            #FD := Qsize;
            Open_Next_Phase   defined in sect. 3.2.1.2
         end if;

         if m > 0 then                  list is not empty
            phase := (phase+1) mod 2;
            n := parent(n);
            while  n ≠ Root do          ascend tree like an insertion
               F&A(Cnt[n, phase], m);
               n := parent(n)
            end while;

            Read_Request(Phase_Act);
               F&A(Cnt[Root, phase], m);
               if phase = DPhase then F&A(PhaseMult, m) end if;
            Read_Release(Phase_Act)
         end if
      end if;
      return true
   else
      return false
   end if
   end with
end function Linked_LogMQDelete;
```

```
function Linked_LogMQIA_TypeI (Item : Object_Pointer;
                               Q    : Linked_LogMQueue;
                               ϕ    : function);
var h : ListElement_Pointer;
    f, n, p : integer;
    with Q do
    Linked_LogMQIA_TypeI := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item then
                    Item);
                    Linked_LogMQIA_TypeI := true
                else  Item is at head of list
                    n := leaf corresponding to list h;
                    p := phase of head item of h;
                    if Cnt[n, p] > 0 then
                        f := F&A(Cnt[n, p], −MaxMult);      block deletions
                        if f > 0 then                       final delete not done
                            repeat until f = h↑.Mult;       wait for deletions
                            ϕ(Item);                        access Item
                            Cnt[n, p] := f;    restore leaf count
                            Linked_LogMQIA_TypeI := true
                        end if
                    end if
                end if
            end if;
        Weak_V(h↑.Access_Sem);
    end if
    end with
end function Linked_LogMQIA_TypeI;
```

3.2.2.3.2. Type Two

3.2.2.3.2.1. A Readers/Writers Implementation

The actions of RaiseMult and LowerMult are very similar to their linked list linear multi-queue counterparts (of section 3.1.2.3.2.1) with the exception that if an item is at the head of its list, the item is assumed to be at an active deletion site and is processed accordingly.

When RaiseMult is invoked with an increment of K, the item's *Mult* component and the queue's *Qlower* component are both increased by K. When the item is at the head of a list, the counters in the nodes on the path from the list's leaf to the root are also incremented by K.

For LowerMult the situation is more complicated. When LowerMult decreases an interior list item's multiplicity by a value K, the item's *Mult* component and the queue's *Qlower* component are each decremented by K, providing *Mult* exceeds K. Otherwise, the item is removed, and *Qlower* is decremented by $Mult-1$.[28] In both cases, since the item is at an interior list position, its multiplicity has not yet been promoted up the tree (it belongs to a future deletion phase). Therefore, the node counters can be ignored. However, due to deletion activity, enough deletions may have already entered the queue to drive *Qlower* below LowerMult's decrement value of V (i.e., K or $Mult - 1$). In this event, *Qlower* can not be decremented by V. Instead, V nil items are added to a queue nil pool *QNilPool*. As described below, deletions unable to join the current deletion phase will be able to acquire (nil) items from this pool.

When LowerMult decrements the multiplicity of a phase p item I that is at the head of a list L by a value K (to lower the multiplicity or as a prelude to removing the item), the final deletion counter $\#FD[p]$ and the phase p leaf nil pool counter $LeafNilPool[p]$ associated with this item are incremented by K, and L's head multiplicity component *HeadMult* is decremented by K. *Qlower* is not modified as I's multiplicity has already been promoted up the tree.

A deletion attempts to join the current deletion phase by decrementing *PhaseMult*; it succeeds if the old value was positive. If instead it was nonpositive, the deletion attempts to obtain a nil (or removed) item from the queue's *QNilPool*, in which case it returns immediately to its caller. The deletion cycles until allowed to descend the tree or acquire an item from the pool.[29] Lastly, note that a final deletion of an item I ascends the tree with a multiplicity of one when I's successor has been removed (i.e., when I's ''missing item'' count is positive).

```
Linked_LogMQueue = Multiqueue Supporting Type II IA
        record
            Qlower, Qsize, Qrear, QNilPool,
            PhaseMult, DPhase : integer;
            Phase_Act       : integer;
            #FD             : array[0..1] of integer;
            Cnt             : array[1..2*Qsize–1, 0..1] of integer;
            LeafNilPool     : array[1..Qsize, 0..1] of integer;
            List            : array[0..Qsize–1] of ListElement
        end;
```

[28] The removal of an item also requires the inclusion of ''missing item'' counts to maintain the FIFO queue property (section 2.2.3.2).

[29] See the fourth version of the Calc_Leaf_Site macro below.

```
ListElement = record
                 Queue_Id  : MQueue_Pointer;
                 HeadMult : integer;
                 Head,
                 Tail        : Pointer to ListElement or Object;
                 #Missing  : integer;
                 RW_Act    : integer
              end;

MQueue_Pointer = ↑Linked_LogMQueue;

ListElement_Pointer = ↑ListElement;

Object = record
            List_Loc : ListElement_Pointer;
            Mult       : integer;
            Next,
            Prev       : Pointer to ListElement or Object
            #Missing : integer
            Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
    Qlower, Qrear, DPhase, PhaseMult, QNilPool = 0;
    Cnt[], LeafNilPool[] = 0;  #FD[] = Qsize;
    Phase_Act = MaxR;
    for 0≤j<Qsize,
      List[j].Queue_Id            = address of queue;
      List[j].HeadMult            = 0;
      List[j].Head                = address(List[j]);
      List[j].Tail                = address(List[j]);
      List[j].#Missing            = 0;
      List[j].RW_Act              = MaxR;
```

```
procedure Linked_LogMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LogMQueue);
var i, phase, n : integer;

    with Q do
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Item↑.Next := address(List[i]);
    Item↑.Mult := M;
    Item↑.#Missing := 0;

    Write_Request(List[i].RW_Act);
        Item↑.List_Loc := address(List[i]);
        Item↑.Prev := List[i].Tail;
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
        if List[i].Head := Item and List[i].#Missing = 0 then
            List[i].HeadMult := M;    promote multiplicity
            n := leaf corresponding to list i;
            Cnt[i, phase] := M;
            Write_Release(List[i].RW_Act);

            n := parent(n);
            while  n ≠ Root do   ascend tree
                F&A(Cnt[n, phase], M);
                n := parent(n)
            end while;

            Read_Request(Phase_Act);
                F&A(Cnt[Root, phase], M);
                if phase = DPhase then F&A(PhaseMult, M) end if;
            Read_Release(Phase_Act)
        else
            Write_Release(List[i].RW_Act)
        end if;

    F&A(Qlower, M)
    end with
end procedure Linked_LogMQInsert;
```

```
function Linked_LogMQDelete (Item : Object_Pointer;
                             Q    : Linked_LogMQueue):
var d, m, n, phase, s : integer;
    with Q do
    if TDR(Qlower, 1) then
        Calc_Leaf_Site(n)_V4;
        d := list corresponding to leaf n;
        Read_Request(List[d].RW_Act);
            Item := List[d].Head;
            if F&A(List[d].HeadMult, –1) = 1 then   final delete
                if List[d].#Missing = 0 then  dequeue Item
                    m := Item↑.Next↑.Mult;
                    List[d].#Missing := Item↑.#Missing;
                    List[d].Head := Item↑.Next;
                    List[d].Head↑.Prev := address(List[d]);
                    Item↑.List_Loc := nil
                else  ‘‘missing’’ item
                    List[d].#Missing := List[d].#Missing–1;
                    m := Item↑.Mult;  Item := nil
                end if;
                if List[d].#Missing > 0 then
                    List[d].HeadMult := 1;    for a ‘‘nil’’ item
                    m := 1
                else
                    List[d].HeadMult := m    head item’s mult.
                end if;
                Cnt[n, (phase+1) mod 2] := m;
                Read_Release(List[d].RW_Act);
                if F&A(#FD[phase], –1) = 1 then
                    #FD[phase] := Qsize;
                    Open_Next_Phase   defined in sect. 3.2.1.2
                end if;
                if m > 0 then                          list is not empty
                    phase := (phase+1) mod 2;
                    n := parent(n);
                    while  n ≠ Root do                 ascend tree like an insertion
                        F&A(Cnt[n, phase], m);
                        n := parent(n)
                    end while;
                    Read_Request(Phase_Act);
                        F&A(Cnt[Root, phase], m);
                        if phase = DPhase then F&A(PhaseMult, m) end if;
                    Read_Release(Phase_Act)
                end if
            else  non-final deletion
                Read_Release(List[d].RW_Act)
            end if;
        return true
    else
        return false
    end if
    end with
end function Linked_LogMQDelete;
```

```
macro Calc_Leaf_Site(n)_V4;
    supporting type II IA on linked allocated logarithmic multiqueues
    s := 0;
    repeat
        if PhaseMult > 0 then
            s := F&A(PhaseMult, –1)
        elseif TDR(QNilPool, 1) then
            Item := nil;
            return
        end if
    until s > 0;
    phase := DPhase;          deletion can join current phase
    F&A(Cnt[Root, phase], –1);
    n := Root;
    while n's left son is not a leaf do
        if TDR(Cnt[leftson(n), phase], 1) then
            n := leftson(n)
        elseif TDR(Cnt[rightson(n), phase], 1) then
            n := rightson(n)
        end if
    end while;

    while true do  loop until find item in list or nil pool
        if Cnt[leftson(n), phase] > 0 then
            if F&A(Cnt[leftson(n), phase], –1) > 0 then
                n := leftson(n);
                exit while
            end if
        end if;
        if Cnt[rightson(n), phase] > 0 then
            if F&A(Cnt[rightson(n), phase], –1) > 0 then
                n := rightson(n);
                exit while
            end if
        end if;
        if    TDR(LeafNilPool[leftson(n), phase], 1) or
              TDR(LeafNilPool[rightson(n), phase], 1) then
            if F&A(#FD[phase], –1) = 1 then
                #FD[phase] := Qsize;
                Open_Next_Phase   defined in sect. 3.2.1.2
            end if;
            Item := nil;
            return true
        end if
    end while;
end macro Calc_Leaf_Site(n)_V4;
```

```
function Linked_LogMQRaiseMult (Item : Object_Pointer;
                                 K    : integer;
                                 Q    : Linked_LogMQueue);
var h : ListElement_Pointer;
    f, n, p : integer;

    with Q do
    Linked_LogMQRaiseMult := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Write_Request(h↑.RW_Act);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior list item
                    Item↑.Mult := Item↑.Mult + K
                    F&A(Qlower, K);
                    Linked_LogMQRaiseMult := true
                else Item is at head of list
                    n := leaf corresponding to list h;
                    p := phase of Item;
                    f := F&A(Cnt[n, p], –MaxMult);  suspend deletions
                    h↑.HeadMult := h↑.HeadMult + K;
                    if f > 0 then f copies left
                        Cnt[n, p] := f + K
                    else  all copies taken
                        Cnt[n, p] := K
                    end if;
                    Write_Release(h↑.RW_Act);

                    n := parent(n)
                    while  n ≠ Root do  ascend tree like an insertion
                        F&A(Cnt[n, p], K);
                        n := parent(n)
                    end while;

                    Read_Request(Phase_Act);
                        F&A(Cnt[Root, p], K);
                        if p = DPhase then F&A(PhaseMult, K) end if;
                    Read_Release(Phase_Act)
                    F&A(Qlower, K);
                    Linked_LogMQRaiseMult := true;
                    return
                end if;
            end if;
        Write_Release(h↑.RW_Act)
    end if
    end with
end function Linked_LogMQRaiseMult;
```

```
function Linked_LogMQLowerMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LogMQueue;
                                M    : integer);
var dequeueItem   : boolean;
    h             : ListElement_Pointer;
    f, n, p       : integer;

    with Q do
    Linked_LogMQLowerMult := dequeueItem := false;
    f := M := 0;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Write_Request(h↑.RW_Act);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior item
                    M := Item↑.Mult;
                    if Item↑.Mult > K then lower multiplicity
                        if not TDR(Qlower, K) then
                            F&A(QNilPool, K)
                        end if;
                        Item↑.Mult := Item↑.Mult – K
                    else remove Item
                        if not TDR(Qlower, M – 1) then
                            F&A(QNilPool, M – 1)
                        end if;
                        dequeueItem := true
                    end if;
                    Linked_LogMQLowerMult := true
                else Item is at head of list
                    n := leaf corresponding to list h;
                    p := phase of Item;
                    if Cnt[n, p] > 0 then
                        f := F&A(Cnt[n, p], –K);
                        if f > K then lower multiplicity
                            M := f;
                            h↑.HeadMult := h↑.HeadMult – K;
                            F&A(#FD, K);
                            F&A(LeafNilPool[n, p], K);
                            Linked_LogMQLowerMult := true
                        elseif f > 0 then  remove item
                            F&A(h↑.RW_Act, MaxR–1);
                            repeat until h↑.HeadMult = f;
                            F&A(h↑.RW_Act, 1–MaxR);
                            M := f;   number of copies removed
                            h↑.HeadMult := 1;
                            F&A(#FD, M – 1);
                            F&A(LeafNilPool[n, p], M – 1);
                            dequeueItem := true;
                            Linked_LogMQLowerMult := true
                        end if  f > k
                    end if  Cnt[n, p] > 0
                end if;  h↑.Head ≠ Item of h↑.#Missing > 0
```

```
                if dequeueItem then
                    F&A(Item↑.Prev↑.#Missing, Item↑.#Missing + 1);
                    Item↑.Prev↑.Next := Item↑.Next;
                    Item↑.Next↑.Prev := Item↑.Prev;
                    Item↑.List_Loc := nil;
                    if f > 0 then Cnt[n, p] := 1 end if
                end if
            end if;  h = Item↑.List_Loc
        Write_Release(h↑.RW_Act)
    end if
    end with
end function Linked_LogMQLowerMult;
```

3.2.2.3.2.2. A Writer Priority Readers/Writers Implementation

This last implementation applies the writer priority techniques of this chapter to the preceding implementation. The algorithms follow.

```
Linked_LogMQueue = Multiqueue Supporting Type II IA
        record
            Qlower, Qsize, Qrear, QNilPool,
            PhaseMult, DPhase : integer;
            Phase_Act       : integer;
            #FD             : array[0..1] of integer;
            Cnt             : array[1..2*Qsize–1, 0..1] of integer;
            LeafNilPool     : array[1..Qsize, 0..1] of integer;
            List            : array[0..Qsize–1] of ListElement
        end;

ListElement = record
            Queue_Id    : MQueue_Pointer;
            HeadMult    : integer;
            Head,
            Tail        : Pointer to ListElement or Object;
            #Missing    : integer;
            Access_Sem : Weak_Semaphore
        end;

MQueue_Pointer = ↑Linked_LogMQueue;

ListElement_Pointer = ↑ListElement;
```

```
Object = record
            List_Loc : ListElement_Pointer;
            Mult     : integer;
            Next,
            Prev     : Pointer to ListElement or Object
            #Missing : integer
            Application_Defined_Attributes
        end;

Object_Pointer = ↑Object;

Initially:
    Qlower, Qrear, DPhase, PhaseMult, QNilPool = 0;
    Cnt[], LeafNilPool[] = 0;  #FD[] = Qsize;
    Phase_Act = MaxR;
    for 0≤j<Qsize,
      List[j].Queue_Id          = address of queue;
      List[j].HeadMult          = 0;
      List[j].Head              = address(List[j]);
      List[j].Tail              = address(List[j]);
      List[j].#Missing          = 0;
      List[j].Access_Sem        = 1;
```

```
procedure Linked_LogMQInsert (Item : Object_Pointer;
                              M    : integer;
                              Q    : Linked_LogMQueue);
var i, phase, n : integer;

    with Q do
    i := F&A(Qrear, 1) mod 2*Qsize;
    if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
    phase := i div Qsize;
    i := i mod Qsize;
    Item↑.Next := address(List[i]);
    Item↑.Mult := M;
    Item↑.#Missing := 0;

    Weak_P(List[i].Access_Sem);
        Item↑.List_Loc := address(List[i]);
        Item↑.Prev := List[i].Tail;
        List[i].Tail↑.Next := Item;
        List[i].Tail := Item;
        if List[i].Head := Item and List[i].#Missing = 0 then
            List[i].HeadMult := M;    promote multiplicity
            n := leaf corresponding to list i;
            Cnt[i, phase] := M;
            Weak_V(List[i].Access_Sem);

            n := parent(n);
            while  n ≠ Root do   ascend tree
                F&A(Cnt[n, phase], M);
                n := parent(n)
            end while;

            Read_Request(Phase_Act);
                F&A(Cnt[Root, phase], M);
                if phase = DPhase then F&A(PhaseMult, M) end if;
            Read_Release(Phase_Act)
        else
            Weak_V(List[i].Access_Sem)
        end if;

    F&A(Qlower, M)
    end with
end procedure Linked_LogMQInsert;
```

```
function Linked_LogMQDelete (Item : Object_Pointer;
                             Q    : Linked_LogMQueue):
var d, m, n, phase, s : integer;
  with Q do
  if TDR(Qlower, 1) then
    Calc_Leaf_Site(n)_V4;   defined in sect. 3.2.2.3.2.1
    d := List index corresponding to leaf n;
    Item := List[d].Head;
    if F&A(List[d].HeadMult, –1) = 1 then   final delete
      Weak_P(List[d].Access_Sem);
        if List[d].#Missing = 0 then
          m := Item↑.Next↑.Mult;
          List[d].#Missing := Item↑.#Missing;
          List[d].Head := Item↑.Next;
          List[d].Head↑.Prev := address(List[d]);
          Item↑.List_Loc := nil
        else  ‘‘missing’’ item
          List[d].#Missing := List[d].#Missing–1;
          m := Item↑.Mult;
          Item := nil
        end if;
        if List[d].#Missing > 0 then
          List[d].HeadMult := 1;    for a ‘‘nil’’ item
          m := 1
        else
          List[d].HeadMult := m    head item’s mult.
        end if;
        Cnt[n, (phase+1) mod 2] := m;
      Weak_V(List[d].Access_Sem);
      if F&A(#FD[phase], –1) = 1 then
        #FD[phase] := Qsize;
        Open_Next_Phase   defined in sect. 3.2.1.2
      end if;
      if m > 0 then                      list is not empty
        phase := (phase+1) mod 2;
        n := parent(n);
        while  n ≠ Root do               ascend tree like an insertion
          F&A(Cnt[n, phase], m);
          n := parent(n)
        end while;
        Read_Request(Phase_Act);         update root like an insertion
          F&A(Cnt[Root, phase], m);
          if phase = DPhase then F&A(PhaseMult, m) end if;
        Read_Release(Phase_Act)
      end if
    end if;
    return true
  else
    return false
  end if
  end with
end function Linked_LogMQDelete;
```

```
function Linked_LogMQRaiseMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LogMQueue);
var h : ListElement_Pointer;
    f, n, p : integer;

    with Q do
    Linked_LogMQRaiseMult := false;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior list item
                    Item↑.Mult := Item↑.Mult + K;
                    F&A(Qlower, K);
                    Linked_LogMQRaiseMult := true
                else Item is at head of list
                    n := leaf corresponding to list h;
                    p := phase of Item;
                    if Cnt[n, p] > 0 then
                        f := F&A(Cnt[n, p], –MaxMult);  suspend deletions
                        if f > 0 then final deletion not here
                            F&A(h↑.HeadMult, K);
                            Cnt[n, p] := f + K;
                            Weak_V(List[d].Access_Sem);

                            n := parent(n);
                            while  n ≠ Root do  ascend tree
                                F&A(Cnt[n, p], K);
                                n := parent(n)
                            end while;

                            Read_Request(Phase_Act);
                                F&A(Cnt[Root, p], K);
                                if p = DPhase then F&A(PhaseMult, K) end if;
                            Read_Release(Phase_Act)
                            F&A(Qlower, K);
                            Linked_LogMQRaiseMult := true;
                            return
                        end if
                    end if
                end if
            end if;
        Weak_V(h↑.Access_Sem)
    end if
    end with
end function Linked_LogMQRaiseMult;
```

```
function Linked_LogMQLowerMult (Item : Object_Pointer;
                                K    : integer;
                                Q    : Linked_LogMQueue;
                                M    : integer);
var dequeueItem    : boolean;
    h              : ListElement_Pointer;
    f, n, p        : integer;

    with Q do
    Linked_LogMQLowerMult := dequeueItem := false;
    f := M := 0;
    h := Item↑.List_Loc;
    if h ≠ nil and h↑.Queue_Id = address(Q) then
        Weak_P(h↑.Access_Sem);
            if h = Item↑.List_Loc then
                if h↑.Head ≠ Item or h↑.#Missing > 0 then  interior item
                    M := Item↑.Mult;
                    if Item↑.Mult > K then lower multiplicity
                        if not TDR(Qlower, K) then
                            F&A(QNilPool, K)
                        end if;
                        Item↑.Mult := Item↑.Mult – K
                    else remove Item
                        if not TDR(Qlower, M – 1) then
                            F&A(QNilPool, M – 1)
                        end if;
                        dequeueItem := true
                    end if;
                    Linked_LogMQLowerMult := true
                else Item is at head of list
                    n := leaf corresponding to list h;
                    p := phase of Item;
                    if Cnt[n, p] > 0 then
                        f := F&A(Cnt[n, p], –MaxMult);
                        if f > K then lower multiplicity
                            M := f;
                            F&A(h↑.HeadMult, –K);
                            F&A(#FD[p], K);
                            F&A(LeafNilPool[n, p], K);
                            Cnt[n, p] := f – K;
                            Linked_LogMQLowerMult := true
                        elseif f > 0 then  remove item
                            repeat until h↑.HeadMult = f;
                            M := f;   number of copies removed
                            h↑.HeadMult := 1;
                            F&A(#FD[p], M – 1);
                            F&A(LeafNilPool[n, p], M – 1);
                            dequeueItem := true;
                            Linked_LogMQLowerMult := true
                        end if  f > K
                    end if  Cnt[n, p] > 0
                end if;  h↑.Head ≠ Item of h↑.#Missing > 0
```

```
                if dequeueItem then
                    F&A(Item↑.Prev↑.#Missing, Item↑.#Missing + 1);
                    Item↑.Prev↑.Next := Item↑.Next;
                    Item↑.Next↑.Prev := Item↑.Prev;
                    Item↑.List_Loc := nil;
                    if f > 0 then Cnt[n, p] := 1 end if
                end if
            end if;  h = Item↑.List_Loc
        Weak_V(h↑.Access_Sem)
    end if
    end with
end function Linked_LogMQLowerMult;
```

3.3. Chapter Summary

Multiqueues permit multiple copies of the same item and arrays of distinct items to be inserted and deleted with far greater efficiency than can be achieved with the concurrently accessible queues described in chapter 2. For each type of multiqueue, linear and logarithmic, a sequential and a linked list implementation has been given. The performance of the sequentially allocated multiqueue insertion and deletion algorithms of Gottlieb et al. [80] has been improved by eliminating the need for non-final deletions to execute readers/writers protocols.

Four new multiqueue operations have been defined: multi-deletion, generic type one interior access, and the specific type two interior access primitives of RaiseMult and LowerMult. The first takes advantage of the multiqueue structure to efficiently implement the deletion of multiple items. The second allows an interior item's application dependent information to be accessed. The third increments an item's multiplicity, and the fourth decrements an item's multiplicity or, if this is not possible, removes the item.

In general, supporting interior access raises the cost of insertion and deletion, but, during periods of interior access inactivity, the concurrency permitted for queue operations is not reduced. Readers/writers and writer priority readers/writers implementations of type one and type two interior access operations have been given. The former are fair (i.e., the implementations given in this chapter can be modified to prevent starvation), but increase the cost of non-final deletion when compared to implementations that do not support interior access. The writer priority variants do not prevent starvation, but, in the absence of interior access activity, non-final deletions are as efficient as when interior access is not supported.

CHAPTER 4

Memory Management

Requests for memory on a shared-memory MIMD machine, such as the Ultracomputer, represent a potential system bottleneck. To achieve high memory utilization it is desirable to treat memory as a single shared resource from which all storage requests are serviced. However, such a centralized policy runs the risk of partially or fully serializing service if multiple requests cannot be processed in parallel and/or the data structure used to maintain the free memory pool is not concurrently accessible. On the Ultracomputer, these obstacles towards parallelizing memory allocation are overcome by having each PE service its own requests using serialization-free algorithms and by maintaining the memory pool with concurrently accessible data structures (e.g., queues and multiqueues). The parallel algorithms to be presented could be incorporated into a segmented and/or paged memory management system for the Ultracomputer.[1]

In this chapter two basic memory allocation techniques are studied. One dynamically divides memory into variable size blocks and satisfies requests of arbitrary size. The other one statically divides memory into fixed size blocks and satisfies requests for arbitrary numbers of blocks (which need not be contiguous).

Our parallel variable size block algorithms are based on two serial methods: the *boundary tag method* (Knuth [68]), which is a first fit allocation method (Bailey et al. [64], Berztiss [65], Comfort [64], Ross [67]), and the *buddy system* (Knowlton [65], Knuth[68]). Both methods are reviewed below. The parallel fixed size block algorithms are simple fetch-and-add and queue/multiqueue applications.

[1] Complete (hardware/software) descriptions of such systems are beyond the scope of this thesis. The feasibility of one virtual memory system, demand paging, has been studied by Teller [88].

We will observe that due to concurrency a parallel method's behavior may not always mirror that of its serial counterpart. In particular, our parallel variants of the boundary tag method and the buddy system may fail to merge adjacent blocks of memory when their corresponding serial versions would do so.

4.1. Variable Size Allocation

This section begins with a review of several serial first fit techniques, focusing primarily on Knuth's boundary tag method. A parallel implementation of the latter is then given. Next, Knuth's serial buddy system is reviewed, and two different parallel implementations based on this system are presented, followed by a parallel scheme called *good fit* that combines features of both first fit and the buddy system.

4.1.1. First Fit

4.1.1.1. Serial Methods

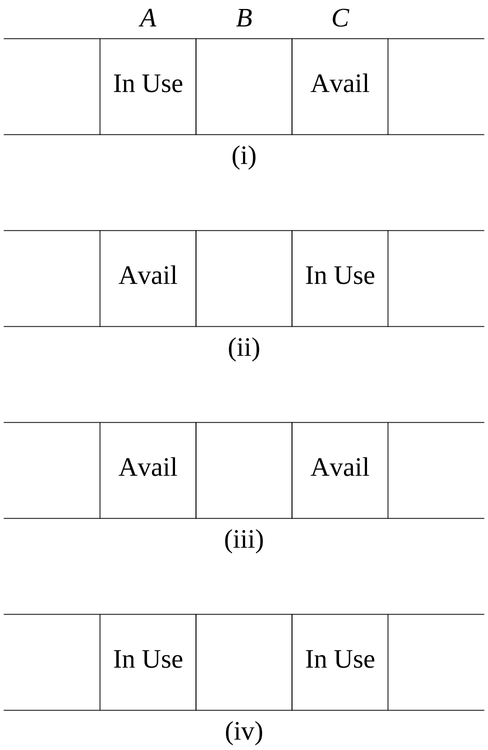
The *first fit* method is a memory allocation scheme that permits memory to be dynamically partitioned into variable size blocks. Two primitives are supported: GetBlock and ReturnBlock. The first requests that a block B of size s_B be allocated from the pool of available or unallocated blocks. A search is conducted through this pool for the *first* block F whose size s_F is greater than or equal to s_B. If no such F exists, the primitive fails. Otherwise, B is allocated from the front (or rear) of F and s_F is reduced by s_B. If this results in the complete allocation of F (i.e., s_F=0), F is removed from the pool.

ReturnBlock deallocates a block B by returning it to the available block pool. If B is physically adjacent to a block in memory that is already in the pool, the two are coalesced to create a larger block. The four possible block configurations that may exist when B is returned are depicted in Figure 4-1, where blocks A, B, and C are assumed to be contiguous. The first three of these result in block mergers. In the first case, C is available and A is in use (allocated). C is appended to B by incrementing B's size by C's and replacing C with B in the block pool. In the next case, A is available and C is in use. B is appended to A by simply incrementing the latter's size by the former's. Similarly, in the third case, where both A and C are available, B and C are appended to A by adding their sizes to A's and removing C from the pool. In the last case, B is added to the block pool as both of its neighbors are in use; no merging can take place. In the following, we will often refer to blocks *A* and *C* as *B*'s left and right neighbors, and to the first and last words of a block as its left and right boundaries.

The block pool data structure is often represented by an array (Bailey et al. [64]) or a linked list (Berztiss [65], Comfort [64], Ross [67]). The array requires storage that is external to the allocatable memory region of the order of the maximum number of available blocks that can be in the pool. In comparison, if the space within unallocated blocks is used to manage the list, the linked list's external space requirement is $O(1)$, consisting essentially of a pointer to the head of the list. However, in many cases the block pool operations of search, merge, insert, and remove can be more efficiently executed on an array than on a linked list.

A major problem faced by a first fit implementation based on either one of these data structures is how to determine whether a returned block's neighbors are already in the pool so mergers

---



Possible memory configurations when block *B* is deallocated.

Figure 4-1.

can occur. The simplest solution is to serially search through the pool for the block's neighbors. Of course the entire pool will be searched if at least one of the neighbors is allocated. A second solution is to order the blocks in the pool by ascending address (Bailey et al. [64], Comfort [64], Ross [67]). Then, when a block B is returned, the pool is searched for B's insertion point. If an available block is adjacent to B in memory, it will appear as B's predecessor or successor in the pool. This sorted block address solution then requires, on average, only half of the blocks in the pool to be examined to locate a block's neighbors. A third solution, the boundary tag method (Knuth [68]), involves no searching for a block's neighbors at the cost of reserving a small amount of space within *each* block (be it allocated or available) for the exclusive use of the method.

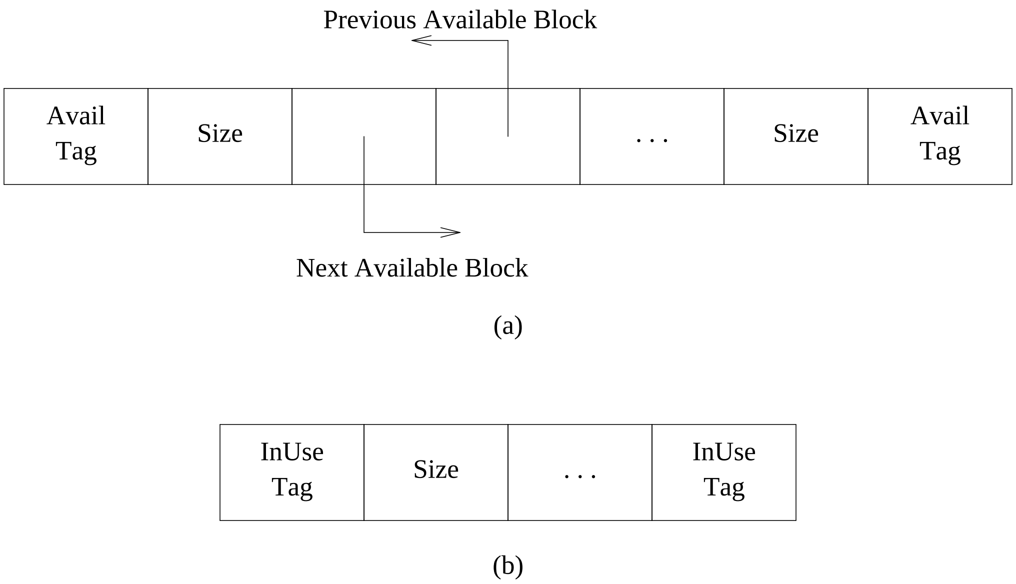
The boundary tag method derives its name from the fact that it *tags* free and allocated blocks at both of their boundaries as *available* and *in use*, respectively. The tags, which can be represented by single bits within the first and last words of a block, are the space ''overhead'' of the method alluded to above. In addition to these tags, the size of an available block is recorded at both boundaries and that of an in use block at its left boundary.[2] Note that the size of a block includes the space ε occupied by its tag and size values. Thus, the usable space of an allocated block of size s is actually $s-\varepsilon$.

The pool of available blocks is organized as a doubly linked list, where the free space within the available blocks is used to store the links for the list. Figure 4-2 depicts the structures of the two block types. Below, we will often refer to the tag (size) associated with a block B's left or right boundary as its left or right tag (size), respectively. Also, we will frequently identify a block's starting address with its name.

By explicitly tagging blocks as in use or available, the coalescing of a block with its neighbors can be done without searching the linked list. The status of a block B's neighbor is determined by examining the appropriate tag, i.e., the right tag of the left neighbor or the left tag of the right neighbor. Both tags are at a known fixed offset from B's boundaries. If B is merged with its available left neighbor L, L's right boundary is extended to the end of B. This new boundary of L is tagged available and L's new size (its previous size plus B's size) is recorded at L's left and

---

[2] If tags are represented by single bits, then the first word of an in use block can typically be used to store the block's tag and size. Alternatively, the size of an in use block could be placed in an external location to the block. The boundary tag method requires that the size of an in use block be known only at the time the block is deallocated.



Schematics for (a) Available and (b) In Use Blocks.

Figure 4-2.

(new) right boundaries. Notice that given only the address of *B*'s left boundary, the address of *L*'s left boundary can be found since *L*'s size is recorded at both of its boundaries. When *B* is merged with its available right neighbor *R*, *B*'s right boundary is shifted to the end of *R*. *B*'s size is incremented by *R*'s and recorded at *B*'s left and (new) right boundaries. Also, *B*'s left tag is changed to available (its right tag is not updated since *R*'s right boundary is already tagged available), and the links that point to and from *R* are moved to point to and from *B*; *R* is effectively replaced with *B* on the list.

While the boundary tag method eliminates searching the available list for a block's neighbors, the list must still be scanned to locate the first block (if one exists) that is large enough to satisfy a given request. If each search starts from the front of the list, there will be a tendency for small blocks to accumulate at this end, increasing the number of blocks that must be examined. By rotating the starting position of the search (e.g., beginning a new search from where the last one terminated), these small blocks may be distributed more evenly throughout the list, lessening their adverse affect on searches (Knuth [68]). This optimization is applicable to any first fit allocation scheme, not just the boundary tag method, and is referred to as ‘‘circular first fit’’ by Finkel [86].

As Finkel notes, a disadvantage of circular first fit arises when the available space is much larger than the average request size, but there are occasionally requests for large blocks. In the steady state, circular first fit distributes moderately sized blocks throughout the list, resulting in blocks incapable of satisfying large requests. Noncircular first fit is less likely to exhibit this behavior since, by allocating space from one end of the list, large blocks will tend to accumulate at the other end (Shore [75]). The tradeoff, of course, is that noncircular first fit has a higher search overhead than circular first fit.

Knuth [68] points out that the free block remaining after a partial allocation is performed may be smaller than any subsequent request. Such available but unallocatable blocks increase the length of the linked list and searches conducted over it. To prevent the generation of such blocks, Knuth recommends that a minimum allocatable block size *MinSize* be defined. Then, when a block B of size s_B is allocated from a block F of size s_F, F will remain on the list if $s_F - s_B \geq MinSize$. Otherwise, if $s_F - s_B < MinSize$, F will be removed from the list and the size of B will be raised to s_F.

The advantage of supporting a minimum block size is that it reduces the length of the list. A disadvantage of this technique is that it can cause requests to fail that would otherwise be satisfied. For example, suppose blocks A and B are contiguous, and A is free and B is in use, where s_A and s_B are the sizes of A and B. Let a block C of size s_C be requested where $s_C + \varepsilon = s_A$ and $\varepsilon < MinSize$; a full allocation of A is done to satisfy this request, even though s_C is less than s_A, since the minimum block size constraint rules out a partial allocation of A. B is then returned, followed by a request for a block D of size s_D, where $s_B < s_D \leq s_B + \varepsilon$. This request fails if there is no available block of size s_D or greater. On the other hand, if there were no minimum block size and we started from the same scenario as above, then the request would succeed since at least one block of sufficient size would exist, namely, the one coalesced from the last ε words of A and B, after C was allocated and B was returned.

4.1.1.2. A Parallel Implementation

We now present a parallel memory allocation implementation that satisfies concurrent requests based on a first fit strategy. Specifically, we elect to implement a parallel version of

Knuth's boundary tag method since this approach, unlike the others mentioned in the previous section, does not require the pool of available blocks to be searched to locate a block's neighbors. Like Knuth's serial algorithm, our parallel version requires space to be reserved within in use blocks for tag and size values.

The pool of available blocks is accessed by each GetBlock and ReturnBlock operation. To reduce the likelihood of this pool becoming a bottleneck and to increase its maximum level of concurrency, the pool is represented, not by one linked list as in Knuth's serial version, but by a fixed number *#Sublists* of independently accessible sublists. This organization is closely related to that of the concurrently accessible linked allocated queues of chapter 2. We choose not to use such a queue to represent this pool as the operations performed on it, which are described below, are better characterized as list rather than queue operations.[3]

GetBlock and ReturnBlock execute the following operations on sublists and/or their blocks: *seaching* a sublist for a block, *allocating* space from a block, *inserting* a new block into a sublist, *merging* a block with one that is being returned, *removing* a block from a sublist. The first two operations are performed by GetBlock and the next two by ReturnBlock. The last is common to both. GetBlock removes a block *C* if *C* is fully allocated. If blocks *A*, *B*, and *C* are contiguous, and *A* and *C* are available, then when *B* is returned, ReturnBlock removes *C* from its sublist and appends *C* and *B* to *A*.

Ideally, to prevent each sublist from becoming a serialization point, all of the operations cited above should be able to execute in parallel. As a first step in this direction, our implementation defines search and allocate operations to be readers of sublists and insertions, mergers, and removals to be writers of sublists. Thus, the most common type of operation, searching, is serialization-free (in the absence of writer activity). Writers are given priority over readers to prevent the former from being starved by the latter. This does not, however, make it possible for writers to starve readers: First, removals executed by GetBlocks cannot lock out searches (which are also the precursors to allocations) for an indefinite period of time as the former can only occur as a consequence of the latter being successful. Second, since the number of insertions (which add blocks to the available list) and mergers (which create larger blocks) that can be performed is

[3] We recognize that one could argue that these operations are merely very powerful type one and type two interior access queue operations as described in chapter 2.

limited by how many blocks of memory are allocated, these two operations cannot starve searches. In the worst case, searches will have to wait until all allocated blocks are returned and merged into one large available block before gaining access to a sublist. Third, removals executed by ReturnBlocks cannot starve readers since they are subject to the same limitations as mergers: A ReturnBlock only invokes a remove operation as part of a merge operation.[4]

A GetBlock request for a block B of size s_B searches each sublist until a block F of size s_F is found such that $s_F \geq s_B$ (if no such block exists, GetBlock fails). In an attempt to balance sublist activity, the ith GetBlock request begins by searching sublist i modulo *#Sublists*. A serial ordering of GetBlocks is defined by having the operations perform fetch-and-adds on a shared counter *AvailFront*.

In addition to being able to concurrently search the same sublist, GetBlocks can concurrently test and modify the size of the same available block via the TDR function. Specifically, each GetBlock request for a block B of size s_B tests (and possibly decrements) an available block F's size s_F with a TDR(s_F, s_B).[5]

Parallel testing of block sizes can in turn lead to concurrent allocations being performed on the same block (when the TDRs of GetBlocks on a block all return true). Such allocations are done by associating with each available block F a pointer end_F that points to the first location beyond the right boundary of F. A GetBlock of a block B of size s_B then performs its test and (possible) allocation from a block F in parallel with other GetBlocks by using the following test and allocate code fragment, which is due to Stone [82]:

$$\textbf{if } \mathrm{TDR}(s_F, s_B) \textbf{ then } B := \mathrm{F\&A}(end_F, -s_B) - s_B \textbf{ end if}$$

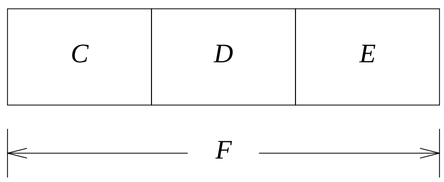
Notice that in this application the fetch-and-add is used not only on the counter s_F (within the TDR) but on the pointer variable end_F as well.

[4] The three write operations listed above do not need exclusive access to an entire sublist, but to, at most, a block and its successor and predecessor. Thus, we could consider implementing write (and read) operations by locking small neighborhoods of a sublist, e.g., by associating a read/write lock with each available block. The advantage of this scheme is that when a sublist is long, neighborhood locking could support a higher degree of parallelism than sublist locking, i.e., granting read/write access to an entire sublist. The concurrency achieved through this scheme would have to be weighed against the cost of manipulating the locks.

[5] Recall that TDR returns true if $s_F \geq s_B$ (and, as a side-effect, decrements s_F by s_B) and false otherwise.

If, at the time a GetBlock executes the test-allocate code fragment, $s_F - s_B \geq MinSize$, where *MinSize* is the minimum allocatable block size, *B* will be allocated from the right boundary of *F*, and *B*'s left and right tags will be set to in use. As a result of this allocation, *F*'s right boundary will be redefined (i.e., shifted towards to its left boundary) and retagged as available. On the other hand, if $s_F - s_B < MinSize$ when GetBlock performs the test and allocate code fragment, *F* will be fully allocated. In this case, *F* (i.e., *B*) will be marked as in use, its sublist *j* will be released (recall that the search and allocate operations are readers of lists), and then write access to *j* will be sought in order to remove *F*. Since *F* is marked as in use before *j* is released, *F* can not be merged with a neighboring block while GetBlock waits to obtain *j*.

After a partial allocation is peformed on an available block *F*, *F*'s new right boundary is tagged available, and the allocated block is tagged in use (at both boundaries). While GetBlocks can subdivide *F* concurrently, the tagging of *F* and blocks allocated from it must be done serially, based on the order in which the blocks are allocated from the right end of *F*. If this were not done, an in use block could be erroneously tagged available: To illustrate the problem, suppose the tagging of blocks is not serialized and two concurrent allocations are performed on *F*, resulting in its being subdivided into three (not necessarily equal) sections, as shown below, where *C* is the segment of *F* that remains after the allocations are done.



If the allocations are executed serially, then the first allocation, the one obtaining *E*, tags *E* as in use and the right boundary of *D* as available. At this point, the contiguous region represented by *C* and *D* equals the remaining segment of *F*. The second allocation, the one obtaining *D*, then tags *D* as in use and *C*'s right boundary as available. However, if the allocations are concurrent, the allocation of *E* could terminate after the allocation of *D*, resulting in *D* being tagged as available (by the allocator of *E*) when it was not.

We serialize the tagging of allocated blocks from the same block *F* via the positions of their right boundaries with respect to *F*'s. Let $oldend_F$ initially point to the same location as end_F (i.e., to the left boundary of *F*'s right neighbor). An allocation of a block *B* of size s_B, after decrementing s_F and end_F by s_B, waits until B's right boundary is adjacent to $oldend_F$. It then sets the right tag of B's left neighbor, the tag of F's new right boundary, to available and decrements $oldend_F$ by s_B. This process is incorporated into the test and allocate code fragment as follows:

```
if TDR(s_F, s_B) then                        allocate B from F
    B := F&A(end_F, -s_B) - s_B;
    repeat until B + s_B = oldend_F;
    righttagloc_F := B - 1;                  new location of F's right tag
    righttagloc_F↑ := available;             mark right boundary as available
    lefttag_B := righttag_B := inuse;
    oldend_F := oldend_F - s_B
end if
```

Notice that the new location l of F's right tag is calculated as an offset from B's, and not F's, left boundary. It cannot be assumed that l equals $F+s_F-1$ since other GetBlocks can test and (at least temporarily) alter s_F. Also, B must be marked as in use before $oldend_F$ is decremented. Otherwise, the next block allocated from F, B's left neighbor, could be returned before B's tags are initialized to in use. In the process, B's left tag could be tested while it was still undefined and indicate that B was available, leading to an erroneous merger of the two blocks.

ReturnBlock merges a block that is being returned with any of its neighbors that are available. In order for ReturnBlock to locate directly the sublist to which a block's available neighbor N belongs, as opposed to searching the sublists for N, the tags of N are represented as pointers to N's sublist. The tags of an in use block are represented as nil pointers (which could be implemented possibly as single bits).

When a block B is returned and both of its neighbors are in use, B is added to a sublist. Specifically, B's insertion increments a shared counter *AvailRear* to generate a ticket number p, obtains write access to sublist s, where s equals p modulo *#Sublists*, sets B's tags to point to s, and then inserts B into s. This evenly divides the number of write operations (due to insertions) among the sublists and also helps to balance the sublists' lengths.

On the other hand, when a block B is returned and at least one of its neighbors N is available, write access to *N*'s sublist *j* (which is identified by *N*'s tags) is obtained.[6] If *B*'s other neighbor *M* is also on sublist *j*, all three blocks will be merged with only one sublist access. If *M* is on a different sublist *k* from *N*, *N* will be removed from *j*, *j* will be released, and *B* and *N* will be merged. Write access to sublist *k* will then be obtained, and *B* and *M* will be coalesced. If *M* is in use, only *B* and *N* will be merged.

A ReturnBlock of a block *B* cannot assume that after (1) determining that a neighbor *N* of *B* is available and then (2) locking *N*'s sublist *j*, *N* will still belong to *j*. Between these two steps, a GetBlock can search *j* and allocate and remove *N*, or a ReturnBlock of a block *A*, where *A* is *N*'s other neighbor, can remove *N* to merge it with *A* and *A*'s other neighbor. Therefore, after *N*'s sublist is locked, the ReturnBlock of block *B* must reexamine *N*'s tag to verify that *N* still resides on *j*. If *N* has moved to a different sublist, *j* will be released, and write access to the new sublist will be obtained, where once again *N*'s tag will be checked. This process will be repeated until the sublist containing *N* is locked or *N* becomes unavailable, i.e., is allocated.[7]

The available block list consists of *#Sublists* doubly linked circular lists. The headers of these sublists are kept in a circular array *Sublist*[0..*#Sublists*–1]. Associated with each sublist is a writer priority readers/writers activity counter. *AvailFront* and *AvailRear* are counters, initialized to zero, that are used to distribute block insertions and searches among the sublists. The allocatable memory region *R* is assumed to initially consist of one available block that is found on *Sublist*[0]. All other sublists are empty. *MinSize* defines the minimum allocatable block size. It functions as described in the previous section.

The structures of available and in use blocks are as depicted in Figure 4-2, with available blocks having two additional components, *End* and *OldEnd*. These are not shown in this figure, but have been defined above. For simplicity, it is assumed that a block's tags occupy one word each. Also, while not explicitly stated in the block data structure definition below, it should be understood that both tags of an available block point to a sublist header, i.e., to an element of the array *Sublist*, and that those of an in use block are both nil pointers.

---

[6] As in the serial version of the boundary tag method, the availability of the left and right neighbors of a block is determined by inspecting their right and left tags, respectively.

[7] It is possible for *N* to be removed from and then reinserted into *j* before the ReturnBlock locks *j*. In this case, the ReturnBlock proceeds as if *N* had never been removed. The retesting of a block's tag is of course analogous to the retesting of a queued item's list location when an interior access operation is performed. In addition, the merger of a block with an available neighbor and the removal of a block from the available list may be viewed as type one and two interior access queue operations (see section 2.2.3).

Two notational conventions need to be cited. First, each available block has components *Next* and *Prev* to link the block to its successor and predecessor on a sublist. Following Knuth [68], the head and tail of an empty sublist point to the sublist header, and a reference to the *Next* or *Prev* component of the sublist header is interpreted as a reference to the *Head* or *Tail* component, respectively. Second, given a pointer to a block *B*, we use the expressions on the left hand side as a shorthand for those on the right hand side:

$$B\uparrow.RightTag \equiv (B + B\uparrow.LeftSize - 1)\uparrow$$
$$B\uparrow.RightSize \equiv (B + B\uparrow.LeftSize - 2)\uparrow$$

```
AvailList = record
        AvailRear, AvailFront, #Sublists, MinSize : integer;
        Sublist :                 array[0..#Sublists–1] of SublistElement
    end;

SublistElement = record
            #W,
            RW_Act : integer;
            Head,
            Tail     : Pointer to SublistElement or Block
        end;

SublistElement_Pointer = ↑SublistElement;

BlockType = (Avail, InUse);

Block = record
        LeftTag   : BlockType;  SublistElement_Pointer
        LeftSize  : integer;
        case BlockType of
        Avail     : (Next,
                     Prev        : Pointer to Block or SublistElement;
                     OldEnd,
                     End         : Block_Pointer;
                     Space       : array[7..LeftSize–2] of Word;
                     RightSize   : integer);
        InUse     : (UsableSpace : array[3..LeftSize–1] of Word)
        end
        RightTag : BlockType   SublistElement_Pointer
    end;

Block_Pointer = ↑Block;
```

Initially:

```
AvailRear := AvailFront := 0;
Sublist[0] contains one block representing memory region R;
Sublist[1..#Sublists−1] are empty;
for 0≤j<#Sublists,
   Sublist[j].#W          := 0;      no writers are active
   Sublist[j].RW_Act      := MaxR;
```

```
function GetBlock_FF (B     : Block_Pointer;
                      Size  : integer;
                      AL    : AvailList);
var a : integer;
    f : Pointer to Block or SublistElement;
    h : SublistElement_Pointer;
    rightsize_of_f        : ↑integer;
    rightboundary_of_B,
    rightboundary_of_f    : ↑SublistElement_Pointer;  pointer to a block's right tag

    with AL do
    a := F&A(AvailFront, 1) mod #Sublists;
    if a = #Sublists–1 then F&A(AvailFront, –#Sublists) end if;
    for 1 to #Sublists do                          search sublists for a suitable block
        h := address(Sublist[a]);                  remember where to stop sublist search
        WP_Read_Request(Sublist[a].RW_Act, Sublist[a].#W);
            f := Sublist[a].Head;                  candidate block
            while f ≠ h do                         not at end of sublist
                if TDR(f↑.LeftSize, Size) then     allocate B from f
                    B := F&A(f↑.End, –Size) – Size;
                    if B – f < MinSize then
                        Size := Size + B – f;
                        B := f
                    end if;
                    repeat until B + Size = f↑.OldEnd;
                    rightboundary_of_f := B – 1;
                    rightboundary_of_f↑ := h;      set f's new right tag to avail
                    rightsize_of_f := B – 2;
                    rightsize_of_f↑ := f – B;      set f's right size
                    rightboundary_of_B := B + Size – 1;
                    rightboundary_of_B↑ := nil;    mark B as InUse
                    B↑.LeftTag := nil;
                    B↑.LeftSize := Size;
                    f↑.OldEnd := f↑.OldEnd – Size;
                    WP_Read_Release(Sublist[a].RW_Act);
                    if B = f then                  remove f
                        WP_Write_Request(Sublist[a].RW_Act, Sublist[a].#W);
                            f↑.Prev↑.Next := f↑.Next;
                            f↑.Next↑.Prev := f↑.Prev;
                        WP_Write_Release(Sublist[a].RW_Act, Sublist[a].#W)
                    end if
                    return true
                end if
                f := f↑.Next                       try next block on sublist
            end while;
        WP_Read_Release(Sublist[a].RW_Act);
        a := a + 1 mod #Sublists                   try next sublist
    end for;
    return false
    end with
end function GetBlock_FF;
```

```
procedure ReturnBlock_FF (B    : Block_Pointer;
                          AL : AvailList);
var h                    : SublistElement_Pointer;
    ln, rn               : Block_Pointer;
    rightboundary_of_ln  : ↑SublistElement_Pointer;    pointer to block's right tag
    size_of_ln,
    size_of_rn           : ↑integer;
    size                 : integer;
    with AL do
    attempt to merge B with its right neighbor rn
    rn := B + B↑.LeftSize;                          rn's address
    rightboundary_of_ln := B – 1;                   right boundary of B's left neighbor ln
R:  h := rn↑.LeftTag;                               rn's sublist location
    if h ≠ nil then                                 rn is available and on sublist h
        WP_Write_Request(h↑.RW_Act, h↑.#W);         lock rn's sublist
            if rn↑.LeftTag = h then                 rn still on sublist h
                if rightboundary_of_ln↑ = h then
                    ln on same sublist; merge ln, B and rn
                    size_of_ln := B – 2;            address of ln's size
                    ln := B – size_of_ln↑;
                    ln↑.LeftSize := ln↑.LeftSize + B↑.LeftSize + rn↑.LeftSize;
                    ln↑.RightSize := ln↑.LeftSize;
                    ln↑.OldEnd := ln↑.End := rn↑.OldEnd;
                    rn↑.Prev↑.Next := rn↑.Next;  remove rn
                    rn↑.Next↑.Prev := rn↑.Prev
                    WP_Write_Release(h↑.RW_Act, h↑.#W);
                    return
                elseif rightboundary_of_ln↑ ≠ nil then   ln is on a different sublist
                    B↑.LeftSize := B↑.LeftSize + rn↑.LeftSize;   merge B and rn
                    rn↑.RightTag := nil;     mark B-rn block as in use
                    rn↑.Prev↑.Next := rn↑.Next;   remove rn
                    rn↑.Next↑.Prev := rn↑.Prev;
                    WP_Write_Release(h↑.RW_Act, h↑.#W);
                else  merge B and rn, ln is in use
                    B↑.Prev := rn↑.Prev;     link B to sublist
                    B↑.Next := rn↑.Next;
                    B↑.Prev↑.Next := B;
                    B↑.Next↑.Prev := B;
                    B↑.LeftTag := h;           mark B as available
                    B↑.End := B↑.OldEnd := rn↑.OldEnd;
                    B↑.LeftSize := B↑.LeftSize + rn↑.LeftSize;
                    B↑.RightSize := B↑.LeftSize;
                    WP_Write_Release(h↑.RW_Act, h↑.#W);
                    return
                end if
            else
                WP_Write_Release(h↑.RW_Act, h↑.#W);  rn is not on sublist h, try again
                go to R
            end if
    end if;
```

```
     attempt to merge B with its left neighbor ln
L:   h := rightboundary_of_ln↑;                    ln's sublist location
     if h ≠ nil then                               ln is on sublist h
         WP_Write_Request(h↑.RW_Act, h↑.#W);
             if rightboundary_of_ln↑ = h then
                 ln is still on list h, append B to ln
                 size_of_ln := B – 2;              ln's size is in next to last word
                 ln := B – size_of_ln↑;
                 ln↑.LeftSize := ln↑.LeftSize + B↑.LeftSize;
                 ln↑.RightSize := ln↑.LeftSize;
                 ln↑.End := ln↑.OldEnd := ln + ln↑.LeftSize;
                 B↑.RightTag := h;         mark ln's right tag as available
                 WP_Write_Release(h↑.RW_Act, h↑.#W);
                 return
             else  ln is no longer on sublist h, see if it's on another sublist
                 WP_Write_Release(h↑.RW_Act, h↑.#W);
                 go to L
             end if
     end if;
     InsertBlock(B, AL)   add B to the available list AL
     end with
end procedure ReturnBlock_FF;


procedure InsertBlock (B    : Block_Pointer;
                       AL : AvailList);
var i : integer

     with AL do
     i := F&A(AvailRear, 1) mod #Sublists;                        generate sublist index
     if i = #Sublists–1 then F&A(AvailRear, –#Sublists) end if;
     B↑.RightSize := B↑.LeftSize;                                 record B's size at both ends
     B↑.End := B↑.OldEnd := B + B↑.LeftSize;
     WP_Write_Request(Sublist[i].RW_Act, Sublist[i].#W);
         B↑.LeftTag := B↑.RightTag := address(Sublist[i]);        tag B as available
         B↑.Next := Sublist[i].Head;                              insert B at front of sublist
         B↑.Prev := address(Sublist[i]);
         B↑.Next↑.Prev := B;
         Sublist[i].Head := B;
     WP_Write_Release(Sublist[i].RW_Act, Sublist[i].#W)
     end with
end procedure InsertBlock;
```

Unlike serial implementations of first fit, our parallel implementation can miss mergers between adjacent available blocks due to concurrency. For example, when adjacent blocks *A* and *B* are returned, the ReturnBlock of *A* can test *B*'s tag at the same time the ReturnBlock of *B* tests *A*'s tag. Since both blocks are in use at this time, no merger occurs and the two blocks are added to the available list. A similar scenario can arise when two blocks having a common neighbor are returned. Suppose blocks *A*, *B*, *C*, and *D* are contiguous, where *A* and *C* are available, but on different sublists, and *B* and *D* are in use. Let the ReturnBlock of *B* remove *C* from its sublist, temporarily marking it as in use, as a prelude to merging it with *A* and *B*. While *C* is so marked, the ReturnBlock of *D* executes. Once again, two adjacent blocks, *D* and *C* (or more precisely, the amalgam of *A*, *B*, and *C*), are being returned at the same time, and their merger can be missed.

A special process could be invoked (perhaps periodically or when a GetBlock fails) to search the available list and merge any adjacent blocks found. In addition, one could reexamine a returned block's neighbors' tags after the block was inserted into a sublist for additional mergers.

4.1.2. The Buddy System

4.1.2.1. The Serial Method

Let us summarize briefly Knuth's description of the serial (binary) buddy system. Assume that we are to manage a contiguous memory region R of size 2^r and are to support two primitives GetBlock and ReturnBlock, where the first primitive requests a block of size 2^j units (a j-block) and the second returns a block obtained previously.

For each j ($0 \le j \le r$), a doubly linked list Q_j of available j-blocks is maintained. Initially, the entire region R is available so Q_r has a single entry and all the other lists are empty. Associated with each block's left boundary are a tag, which indicates if the block is *available* or *in use*, and a size, which equals the length of the block and includes the space within the block for the tag and size values.[8]

If a j-block is desired but none is available, we obtain a larger k-block, $k>j$ (if this is not possible, the request fails). This block is split into equal halves, called buddies. One of these buddies is added to Q_{k-1} and the other is split into buddies, etc., until a j-block appears. When a j-block is returned, we check to see if its buddy is available. If it is not, the block is inserted into Q_j. Otherwise, the buddy is removed from Q_j, the two blocks are coalesced, and the resulting (j+1)-block is returned.

[8] As in the boundary tag method, an in use block's size may be recorded externally to the block. In addition, for the buddy system, the tag may also reside externally. We will assume that both are kept within the block. An alternative is to require the user of these algorithms to pass the size of the block to ReturnBlock explicitly.

The buddy of the j-block at address A has address $A \oplus 2^j$ (where $\oplus$ is exclusive-or) and is thus easily determined given A. Note however that when a j-block is returned, its buddy is available only if the block at its buddy's address is so tagged *and* is of size j. For example, suppose the j-block at address A is split into two $(j-1)$-blocks and then the $(j-1)$-block at address A is split into two $(j-2)$-blocks. The buddy of the $(j-1)$-block at address $A \oplus 2^{j-1}$ and the buddy of the $(j-2)$-block at address $A \oplus 2^{j-2}$ have the same address, A. But, since the block at address A is of size $j-2$, it is the buddy of the latter.

The buddy system, by partitioning available blocks by size, satisfies a j-block request by searching for a non-empty Q_i ($j \le i \le r$) as compared to the first fit approach of seaching the available list for a block of size 2^j or greater. In returning a j-block, the availability of its buddy is determined directly from the data structures and the address relationship between buddies. Thus, like the boundary tag method, no searching is done to locate and remove a block's available buddy.

The buddy system supports requests for block sizes that are powers of 2. If an application needs a block of size s, where $2^j < s < 2^{j+1}$, then a block of size 2^{j+1} is allocated. Thus, the buddy system can suffer from *internal fragmentation* (Randell [69]), i.e., space within in use blocks may be wasted. First fit experiences this type of fragmentation only when a minimum block size is supported. Of course, both allocation methods are subject to *external fragmentation* (Randell [69]) where available space between in use blocks is wasted.

4.1.2.2. Parallel Implementations

Two parallel variants of the buddy system are presented. The first maps available memory blocks onto the nodes of a binary tree whereas the second resembles the serial method described above.

4.1.2.2.1. A Tree Implementation

Our first implementation maps available blocks onto the nodes of a complete binary tree. Obtaining an available j-block, corresponding to deleting a block from Q_j, involves descending the tree while decrementing counters associated with each node visited. Returning a block, corresponding to inserting a block into a queue, involves ascending the tree while incrementing the counters. Coalescing a j-block with its buddy is essentially a deletion of both blocks from the tree followed by an insertion of their corresponding $(j+1)$-block into the tree. This tree traversal technique is the same as the one used for the trees of logarithmic multiqueues (section 3.2) and queues with priority (Appendix A).

The buddy system allows the memory region *R* to be subdivided into blocks of size 2^{r-1}, 2^{r-2}, ... , 1. This division permits us to represent these blocks by a complete binary tree *T* of height *r*. The root of *T* represents *R* and the two children of a node *N* represent the two halves of the block represented by *N*. Thus, nodes at height *j* represent *j*-blocks and in particular, leaves (at height 0) represent 0-blocks. In what follows we often identify a node with the block it represents.

Associated with each node *N* are *r*+1 counters, $C[N, j]$ j=0, ... , *r*, specifying the number of available *j*-blocks in the subtree rooted at *N*.[9] Since these counters are central to understanding the algorithm, let us comment informally before presenting their precise definition. If the subtree at *N* is completely available, the counts at *N* are all zero except for $C[N, \text{height}(N)] = 1$. If the entire subtree is not available, the counts at *N* are the sums of the counts at *N*'s children. The counts do not reflect the possibility of splitting blocks, e.g., if $C[N, \text{height}(N)] = 1$, representing the availability of the entire subtree, then $C[N, j] = 0$ for all $j<\text{height}(N)$. This condition prevents one task getting a block while a concurrent task gets one of its subblocks. We do not require the counts of a node's children to be all zero when the entire subtree is available since this would force the counts to be changed when a block is in fact split or coalesced. A tree for a region of size 4 is shown in Figure 4-3. Notice that there are available 0-blocks at addresses 1 and 2.

To precisely define the counts, assume that no GetBlock or ReturnBlock primitives are active. Then,

(1) for any node,

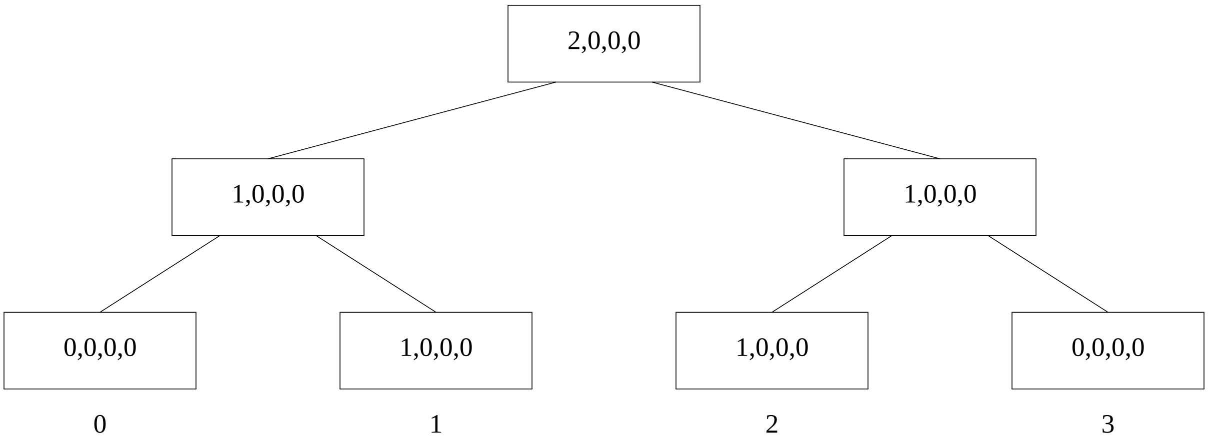
(i) $C[N, j] \geq 0$,

(ii) $C[N, j] = 0$ if $j > \text{height}(N)$,

(iii) $C[N, j] \leq 2^{\text{height}(N)-j}$ if $j \leq \text{height}(N)$;

(2) for a leaf, $C[N, 0] = 1$ iff the 0-block is available;

[9]Actually, a node at height *h* only needs *h* counters.



Counts shown for a region of size 4 where 0-blocks have been allocated at locations 0 and 3.

Figure 4-3.

(3) for a node at height $h > 0$,

(i) $C[N, h] = 1$ iff $C[\text{left}(N), h-1] = C[\text{right}(N), h-1] = 1$,

(ii) $C[N, h-1] = 0$ unless $C[N', h-1] = 1$ for precisely one child N' of N, in which case $C[N, h-1] = 1$;

(4) for a node at height $h \geq p \geq 1$,

$$C[N, h-p] = C[\text{left}(N), h-p] + C[\text{right}(N), h-p].$$

Clearly the counts are changed by GetBlock and ReturnBlock. Since concurrent activity is possible, it is important to correctly interpret the meaning of $C[N, j]$ while the primitives are in progress. The implementation guarantees two properties.

(1) The counts are conservative, i.e. they are lower bounds on the number of available j-blocks.

(2) Counts in an ancestor are at least as conservative as counts in a descendant.

These two properties hold trivially when no primitives are in progress since the counts are then accurate.[10]

The properties continue to hold during periods of activity since GetBlock *lowers* counts from the root *down* to *N before* claiming *N* and ReturnBlock *raises* counts from *N up* to the root *after* relinquishing *N*.

To obtain a *j*-block we proceed as follows. First a TDR of $C[root, j]$ is attempted. If this succeeds a *j*-block will be obtained; if it fails we attempt to claim a (j+1)-block with a TDR of $C[root, j+1]$; etc. Assume that we succeed in claiming a (j+p)-block (if we are unable to claim any such block the GetBlock request fails). Then GetBlock descends the tree, using the counts to guide its descent, until an available (j+p)-block is found (the counts are decremented during the descent).[11] A *j*-block is then removed from this (j+p)-block and blocks of size 2^j, 2^{j+1}, ... , 2^{j+p-1} are returned. Returning a *j*-block involves incrementing the counts from *N* (*j*'s node) up to the root. Although it appears that memory becomes increasingly more fragmented as execution continues, this is not the case since the following simple technique permits a block to be coalesced with its buddy. ReturnBlock, after returning *N*, descends the tree and attempts to obtain *N* and its buddy simultaneously. If this fails, no coalescing is possible; if it succeeds, the common parent of *N* and its buddy is returned (which may recursively lead to more coalescing).

As indicated above, we store the size of a block in the block itself, even if the latter is in use.

Height = 0..r; *memory region is of size* 2^r

Nodes = 1..$2^{(r+1)}$–1; *complete binary tree*

CntTree = **record**
    r : **integer**;
    C : **array**[Nodes, Height] **of integer**
end;

[10] When a node *N* is claimed by GetBlock, the counts in *N*'s descendants are not updated and are thus inaccurate and not conservative. However, no subsequent GetBlock can access these nodes until *N* is returned and, by that time, the counts are accurate.

[11] This process is similar to performing a deletion on a logarithmic multiqueue or queue with priority. ReturnBlock is analogous to performing an insertion on either type of queue.

```
Block  = record              in use block
           Size              : integer;  length of block is 2^Size
           UsableSpace : array[2..2^Size] of Word
         end;

Block_Pointer = ↑Block;

Initially:
   ∀n∈Nodes,
      C[n, height(n)]        = 1;
      C[n, 0..height(n)–1] = 0;

function GetBlock_BuddyTree (B     : Block_Pointer;
                             j     : integer;
                             Tree : CntTree);
var h, k, n, t : integer;
   with Tree do
   for k := j to r do  search for free block of size j or greater
      if TDR(C[root, k], 1) then go to L end if
   end for;
   return false;

   Descend tree until reach node at height k whose entire
   subtree is free; decrement node counts along the way
L: n := root of Tree;
   while height(n) > k do
      if TDR(C[leftson(n), k], 1) then
         n :=  leftson(n)
      elseif TDR(C[rightson(n), k], 1) then
         n := rightson(n)
      end if
   end while;

   Descend subtree to a height of j.  This is equivalent
   to the splitting of larger blocks to obtain a j-block.
   for h := k–1 downto j do
      n := leftson(n);
      C[n, h] := 0          mark n's subtree unavailable
   end for;

   B := block corresponding to node n;
   for h := j+1 to r do    add buddies of split blocks to node counts
      n := parent(n);
      for t := j to (k min h)–1 do
         F&A(C[n, t], 1)
      end for
   end for
   return true
   end with
end function GetBlock_BuddyTree;
```

```
procedure ReturnBlock_BuddyTree (B     : Block_Pointer;
                                 Tree : CntTree);
var i, k, n : integer;
    with Tree do
    k := B↑.Size;
    i := index corresponding to k-block B;
    F&A(C[i, k], 1);    mark k-block B as available
    if i = root then return end if;
    n := i;
    while n ≠ root do
        n := parent(n);
        F&A(C[n, k], 1)            add k-block to node
    end while;

    Try to descend tree to B's node i, acquiring two k-blocks at each level.
    while TDR(C[n, k], 2) do
        n := child of n on path to block B
    end while;
    if n = i then  have seized B and its buddy
        B := parent(B);
        B↑.Size := B↑.Size + 1;
        ReturnBlock_BuddyTree(B, Tree)
    else coalesce failed; ascend tree and return k-blocks
        while n ≠ root do
            n := parent(n);
            F&A(C[n, k], 2)
        end while
    end if
    end with
end procedure ReturnBlock_BuddyTree;
```

Due to concurrency, available buddies may fail to be coalesced. For example, suppose there is one available *j*-block ($j<r$) when three *j*-block requests are executed. Through their TDRs the three requests simultaneously test and decrement the root's *j*-block count, driving it to -1. The two that fail will, in completing their TDRs, each increment the root's *j*-block count by 1 and then request (j+1)-blocks. Before these increments are performed, let two in use *j*-blocks, which are buddies, be returned. Each of their ReturnBlocks increments the root's *j*-block count by 1, raising the count from -1 to 1. Since the count indicates that only one *j*-block is available, the two returned *j*-blocks will not be coalesced (their ReturnBlocks' TDRs of 2 on the count will fail). As noted for the parallel implementation of first fit, a special process could be invoked (possibly when a memory request failed) to locate and coalesce such available buddies.

4.1.2.2.2. A Queue Implementation

Our second implementation is a straightforward parallelization of the serial buddy system. For each block size *j*, a concurrently accessible linked allocated queue Q_j (see section 2.2) holds the available *j*-blocks. Recall that this data structure consists of a concurrently accessible circular array of pointers to serially accessible linked lists. The queue insertion and deletion algorithms are sufficient to obtain and return blocks if we ignore the coalescing of buddies. To implement this last point interior access to a queue is needed to obtain a block's buddy. Interior removal (section 2.2.3.2) provides this capability: It deletes a given block from any position within a specified queue.

All blocks are tagged as either available or in use at their left boundaries. To support interior queue removal, an available block contains a counter *#Missing* that is accessed by queue deletions and interior removals (see section 2.2.3.2). While an in use block's size is explicitly recorded at its left boundary, an available block's size is implicitly determined from the queue it is on: the length of a block on Q_j is 2^j. Also, as in the parallel version of the boundary tag method, the tag of an available *j*-block points to the list of Q_j of which it is a member, and the tag of an in use block is nil. Insertion, deletion, and removal set a block's tag as part of their standard operation.

An available *j*-block *B* at the front of its queue can be the target of both a GetBlock that is requesting a *j*-block and a ReturnBlock that is attempting to merge *B* with its buddy. As described in chapter 2, contention for a block is resolved at the queue primitive level through references to a block's tag and list, ensuring that no two memory requests obtain the same block.

When a block is removed from its queue for the purpose of merging it with its buddy, a nil block is left in its place. GetBlock, upon receiving such a block, discards it and seeks a new block.

In the serial buddy system, a *j*-block *B* is coalesced with the block *C* at its buddy's address if *C* is available and is of size *j*. In the parallel version of this method, *C* is available and is of size *j* if it is a member of Q_j. Thus, when *B* is returned, a queue removal of *C* from Q_j is attempted.[12] If the removal succeeds, B's buddy has been obtained and the two are coalesced. This process is repeated until the removal fails.

It will once again be assumed that each block's tag occupies one word.

BuddyQ = **record**
        r : **integer**; *memory region is of size* 2^r
        Q : **array**[0..r] **of** Linked_Queue
    end

BlockType = (Avail, InUse);

Block = **record**
        Tag : BlockType; *list or nil pointer*
        case BlockType **of**
        Avail : (Next,
            Prev : Block_Pointer;
            #Missing : **integer**; *removed block count*
            Space : **array**[5..2^j]) **of Word**) *where Block*$\in$ $Q[j]$
        InUse : (Size : **integer**; *block's length is* 2^{Size}
            UsableSpace : **array**[3..2^{Size}] **of Word**)
    end;

Block_Pointer = $\uparrow$Block;

Initially:
    Q[r] *contains one r-block*;
    Q[0..r–1] *are empty*.

[12] Recall that this primitive succeeds if its item C is on the correct queue Q_j, and fails otherwise.

```
function GetBlock_BuddyQ (B    : Block_Pointer;
                          j    : integer;
                          BQ   : BuddyQ);
var k, length     : integer;
    c             : Block_Pointer;

    with BQ do
    for k := j to r do
        repeat     locate a free block
            GetBlock_BuddyQ := Linked_QDelete(B, Q[k])
        until not GetBlock_BuddyQ or B ≠ nil;
        if GetBlock_BuddyQ then exit for end if
    end for;

    if GetBlock_BuddyQ then                split k-block to obtain a j-block
        for length := k–1 downto j do
            c := B ⊕ 2^length;             B's buddy
            Linked_QInsert(c, Q[length])   add c to available queue
        end for;
        B↑.Size := j                       record size of InUse block
    end if
    end with
end function GetBlock_BuddyQ;


procedure ReturnBlock_BuddyQ (B :  Block_Pointer;
                              BQ : BuddyQ);
var c         : Block_Pointer;
    success   : boolean;

    with BQ do
    success := true;
    while B↑.Size < r and success do              try to coalesce B with its buddy c
        c := B ⊕ 2^(B↑.Size);
        if Linked_QRemove(c, Q[B↑.Size]) then      merge B and c
            B := parent(B);                        B min c
            B↑.Size:= B↑.Size + 1
        else
            success := false
        end if
    end while;
    Linked_QInsert(B, Q[B↑.Size])                  place B on available queue
    end with
end procedure ReturnBlock_BuddyQ;
```

Like the tree-based buddy system, this version may also fail to coalesce buddies. For example, suppose buddies *A* and *B* are returned at the same time. If the ReturnBlock of *A* inspects *B*'s tag at the same time the ReturnBlock of *B* inspects *A*'s tag, then neither operation will attempt a merger.[13]

4.1.2.2.3. Comparison

The tree implementation of the buddy system is serialization-free since all memory requests are processed solely through fetch-and-adds. The queue implementation delays a request when operations contend for access to the same list of a queue. Expanding the size of the queue, by adding more lists, alleviates this problem.

The queue implementation takes advantage of the free space within available blocks to store data needed for the management of the queues. External space is only required for each queue's list headers and counters. The tree implementation does not rely upon queues or lists to keep track of free blocks; instead, the counts associated with each node suffice. But, unlike the queue implementation, a node's counters cannot be kept within its corresponding block because the physical relationship between blocks and nodes is not one-to-one. Consider the block at address zero in Figure 4-3. It can correspond to any one of three nodes–the root, the root's left son, or the leftmost leaf–depending upon its size. After a 0-block is allocated at this address, its ancestors' counts must still be preserved to permit the further allocation and deallocation of memory, precluding the placement of counters within the block.

Ignoring list contention, the queue implementation of the buddy system allocates and deallocates blocks in constant time. The major cost of these operations is in accessing the queue. On the other hand, the tree implementation's operations are logarithmic in the size of the memory region *R*. Thus, when *R* is large, the first implementation should be preferred over the second.

4.1.3. Good Fit

Good fit is a parallel memory allocation scheme that incorporates features from the parallel versions of the boundary tag method for first fit and the queue-based buddy system. Like first fit, good fit allocates variable size blocks of memory (whereas the buddy system restricts the sizes to powers of 2); and, as in the buddy system, it partitions available blocks based on their sizes into different queues (whereas first fit maintains one list).

[13] Eric Freudenthal has recently provided a solution to the ‘‘missed’’ coalesce problem that relies on a readers/readers protocol (Freudenthal [87]). Basically, each memory block is placed (based on its starting address) into one of two groups; buddies are never placed into the same group. When blocks are returned, only one group at a time may attempt coalesce operations; hence, mergers will not be missed.

In good fit, the available pool of blocks is repesented by k concurrently accessible queues. Associated with each queue Q_j ($0 \le j < k$) is a range $R_j = [m_j, m_{j+1})$, $0 < m_j < m_{j+1}$. An available block of size s is kept on Q_j, where $s \in R_j$.

Blocks have the same internal structure as those in the parallel boundary tag method except a missing block counter *#Missing* is added to each available block (to support interior queue removal), and the components *End* and *OldEnd* are eliminated. As in the parallel version of the boundary tag method, both tags of an available block point to the list of which the block is a member. Our implementation of good fit requires a block's tags to be set by the queue primitives of insertion, deletion, and removal. That is, an insertion sets *both* tags of a block to available (i.e., to point to the block's queue list), and a deletion or a removal sets them to in use (i.e., to nil).

A GetBlock request for a block B of size s_B tests queues i through k, where $s_B \in R_i$, until a non-empty Q_j is found (if all of these queues are empty, the request fails). The first block F on this queue is then deleted. The blocks within Q_j do not have to be tested as in first fit since any block in this queue (in particular, the first one) can satisfy the request–a *good fit* is assured. As in first fit, GetBlock returns F if *F's* size s_F less s_B is smaller than the minimum allocatable block size. Otherwise, B is allocated from the right boundary of F, F's size is decremented by B's, and F is inserted into Q_j, where $s_F \in R_j$.

A ReturnBlock of a block B merges B with its available neighbors by removing them from their queues and forming a larger block B', which is then returned. If both neighbors of B are in use, B is inserted into the appropriate queue. ReturnBlock uses the queue primitives RemoveLeft and RemoveRight to remove B's left and right neighbors, respectively, from their queues. These primitives fail if the blocks they are trying to remove are in use. RemoveLeft and RemoveRight are based on the standard interior queue removal primitive of chapter 2.

Due to certain types of mergers and allocations that can occur while a ReturnBlock is in progress, the standard remove operation can not support RemoveLeft, although it can support RemoveRight: Let us suppose that just the standard remove primitive is invoked to obtain a block's neighbors. Recall that this primitive is passed a block N's starting address and the queue Q that N is to be removed from. To simplify the following discussion, we will assume that the available list consists of a single queue (k=1). First, consider how a block's right neighbor is removed. A ReturnBlock of a block B computes the starting address A_R of B's right neighbor R

by first adding the size of *B* to the starting address of *B*, and then issuing a remove of *R*. Between these two steps, another ReturnBlock can remove *R*, merge *R* with its right neighbor, and then insert (the larger sized) *R* into the queue, or a GetBlock can delete *R*, allocate a block from *R*'s right boundary, and then insert (the smaller sized) *R* back into the queue. However, neither one of these scenarios alters A_R. Thus, the standard remove primitive invoked by the ReturnBlock of *B* will not be adversely affected by what either one of these other operations can do to *R*. This is not true for similar scenarios involving the removal of a block's left neighbor.

For a ReturnBlock of a block *B* to remove *B*'s left neighbor *L* (using the standard remove primitive), it first computes *L*'s starting address A_L, by subtracting the size of *L*[14] from the starting address of *B*, and then issues its remove of *L*. Between these two steps, (i) another ReturnBlock can remove *L*, merge *L* with its left neighbor *K*, and then insert (the larger sized) *K* into the queue, or (ii) a GetBlock can delete *L*, allocate a block *M* from *L*'s right boundary, and then insert (the smaller sized) *L* back into the queue. After (i) occurs, the address A_L computed by the ReturnBlock of *B* points into the interior of available block *K*. In this case, the ReturnBlock of *B* should not proceed to execute its removal of *L* as this block no longer exists. If the removal were performed, the outcome would be undefined as the remove primitive assumes that it is always passed a valid starting block address. Following the completion of (ii), the removal of *L* by the ReturnBlock of *B* can succeed, but by this time *M* has replaced *L* as *B*'s left neighbor. Hence, *B* and *L* should not be merged.

Removing *B*'s left neighbor *L* is complicated by the volatility of *L*'s starting address, as illustrated above. However, since the boundaries of *B* itself are constant while it is in use, the *starting* address of its right neighbor and the *ending* address of its left neighbor are also constant during this time. Therefore, the primitives RemoveRight and RemoveLeft are passed the starting address of *B* from which they can compute the starting and ending addresses of *B*'s right and left neighbors, respectively. RemoveRight then inspects the right neighbor's left tag and RemoveLeft inspects the left neighbor's right tag to determine if a neighbor is available (i.e., queued), and if so, to which list it is a member.[15] Note that unlike the standard remove primitive, these primitives are not passed the name of a specific queue from which to remove a block. As long as the

[14] Recall that *L*'s size is recorded at its right boundary and is at a fixed offset relative to *B*'s starting address.

[15] This is similar to what is done in the parallel implementation of the boundary tag method.

block is queued (on any one of the k queues comprising the pool of available blocks), it is taken. Which queue the block belongs to is unimportant for the purpose of merging the block with its neighbor. The identity of the queue merely provides additional information relating to the size of the block. Therefore, RemoveRight and RemoveLeft do not check a block's queue as is done in the standard remove primitive. Aside from this difference, RemoveRight is identical to this primitive. RemoveLeft is similar to RemoveRight, but it references a block's right tag in place of the left one.[16]

We will once again assume that a block's tags occupy one word each and are implemented as pointers to headers of lists, as in the parallel buddy queue implementation. Both GetBlock and ReturnBlock use the ancillary function QIndex(s) that returns i, where $s \in R_i$. For the convenience of the ReturnBlock implementation, it will also be assumed that the ''blocks'' adjacent to the allocatable memory region R are unavailable, i.e., the memory locations bordering R are tagged as in use.

```
AvailList = record
            MinSize, k : integer;
            Q : array[1..k] of Linked_Queue;
            R : array[1..k] of  range sizes for queues
                record
                    LB, UB : integer  lower and upper bounds
                end
        end

BlockType = (Avail, InUse);
```

[16] Recall that we are assuming for the good fit implementation that both tags of a block are altered whenever a queue primitive enqueues or dequeues the block. This assumption applies to RemoveLeft and RemoveRight as well.

```
Block = record
            LeftTag   : BlockType;   list or nil pointer
            LeftSize  : integer;
            case BlockType of
            Avail     : (Next,
                          Prev          : Block_Pointer;
                          #Missing      : integer;   for removed blocks
                          Space         : array[6..LeftSize–2] of Word;
                          RightSize     : integer);
            InUse     : (UsableSpace : array[3..LeftSize–1] of Word)
            end
            RightTag : BlockType   list or nil pointer
        end;

Block_Pointer = ↑Block;
```

Initially:
  Q[k] *contains one block corresponding to region R*;
  Q[1..k–1] *are empty*.

```
function GetBlock_GF (B     : Block_Pointer;
                      Size : integer;
                      AL   : AvailList);
var f     : Block_Pointer;
    j, s  : integer;

    with AL do
    for j := QIndex(Size) to k do
        repeat
            GetBlock_GF := Linked_QDelete(f, Q[j])
        until not GetBlock_GF or f ≠ nil;
        if GetBlock_GF then exit for end if
    end for;

    if GetBlock_GF then          have acquired a block
        s := f↑.LeftSize – Size;
        if s ≥ MinSize then       split block
            f↑.LeftSize := s;
            f↑.RightSize := s;    f↑.RightSize is at f+s–1
            B := f + s;
            B↑.LeftSize := Size;
            B↑.LeftTag := nil;   F's (B's) right tag is set to nil by QDelete
            Linked_QInsert(f, Q[QIndex(s)])
        else
            B := f
        end if
    end if
    end with
end function GetBlock_GF;
```

```
procedure ReturnBlock_GF (B   : Block_Pointer;
                          AL : AvailList;
var ln, rn          : Block_Pointer;
    size_of_ln      : ↑integer;
    with AL do
    while RemoveRight(B) do                          try to obtain B's right neighbor rn
        rn := B + B↑.LeftSize;
        B↑.LeftSize := B↑.LeftSize + rn↑.LeftSize    append rn to B
    end while;
    while RemoveLeft(B) do                           try to obtain B's left neighbor ln
        size_of_ln := B – 2;                         location of ln's right size
        ln := B – size_of_ln↑;
        ln↑.LeftSize := ln↑.LeftSize + B↑.LeftSize;  append B to ln
        B := ln
    end while;
    B↑.RightSize := B↑.LeftSize;                     initialize B for insertion
    Linked_QInsert(B, Q[QIndex(B↑.LeftSize)])        Insert sets B's tags to Avail
    end with
end procedure ReturnBlock_GF;
```

As in all of the preceding parallel implementations, adjacent blocks may fail to be coalesced due to concurrency. For example, suppose neighboring blocks *A* and *B* are returned at the same time. If the ReturnBlock of *A* inspects *B*'s tag at the same time the ReturnBlock of *B* inspects *A*'s tag, then neither operation will attempt a merger.

4.2. Fixed Size Allocation

The preceding allocation methods honor requests for variable size blocks of contiguous memory. An alternative method is to partition memory into fixed size blocks and grant requests for arbitrary numbers of such blocks. Parallel implementations of this method follow. They are described, in part, using the terminology of paging (Denning [70]) as this type of memory management system could be one possible application for them. The first implementation represents memory with a concurrently accessible array and the second uses a concurrently accessible multiqueue.

Let the memory region be divided into *#PF* fixed size blocks called *page frames*. The status of each page frame is recorded in the integer array *PageFrameStatus*[0..*#PF*–1], where *PageFrameStatus*[*i*] is 1 if the *i*th page frame is available and 0 if the page frame is in use. The counter *#AvailPF* equals the number of available page frames. A request for *p* page frames locates *p* available entries (those with a value of one) in *PageFrameStatus*, marks them as in use (i.e., sets them to zero), and then places the indices of these entries (the allocated page frames) into an array or *page table* of length *p*. If *p* page frames can not be allocated, the request fails. The page frames in a page table are freed by simply updating their corresponding entries in *PageFrameStatus*.

We assume that page tables are themselves page frames.[17] Therefore, after the page frames in a table are released, the table is returned to the pool of available page frames as well.

```
AvailPages = record
                #PF             : integer;  number of page frames
                #AvailPF        : integer;  number of available page frames
                PageFrameStatus : array[0..#PF–1] of integer
             end

PageTable = record
               #Pages : integer;   number of entries in PageTable
               Value  : array[1..#Pages] of integer
            end;

function GetPages_V1 (PT              : PageTable;
                      #PagesRequested : integer;
                      AP              : AvailPages);
var i, j : integer;
   with AP do
   if TDR(#AvailPF, #PagesRequested) then
      i := 0;
      for j := 1 to #PagesRequested do
         repeat i := i+1 mod #PF until TDR(PageFrameStatus[i], 1);
         PT.Value[j] := i
      end for;
      PT.#Pages := #PagesRequested;
      return true
   else
      return false
   end if
   end with
end function GetPages_V1;
```

[17] Alternatively, the page tables themselves could consist of variable length segments.

```
procedure ReturnPages_V1 (PT : PageTable;
                          AP  : AvailPages);
var j : integer;
    with AP do
    for j := 1 to PT.#Pages do
        F&A(PageFrameStatus[PT.Value[j]], 1)
    end for;
    F&A(#AvailPF, PT.#Pages+1);
    F&A(PageFrameStatus[PageFrameIndex(PT)], 1)  return page table
    end with
end procedure ReturnPages_V1;
```

Notice that in the worst case the entire page frame status array is searched to locate an available page frame.

The pool of available page frames can also be represented as a sequentially or linked allocated queue (of chapter 2) or (linear or logarithmic) multiqueue (of chapter 3). If a queue is chosen, the allocation and deallocation of p page frames requires p deletions and p insertions. On the other hand, the same page frame operations can be done via one multi-deletion (section 3.1.1.3) and one insertion by using a multiqueue. Instead of items of multiplicity, the multiqueue contains arrays of items, i.e., arrays (page tables) of available page frame pointers. We give one such implementation below, where the multiqueue data structure and operations have been tailored to our paging application.

Let each page frame consist of an array of *PageSize* contiguous words. A page table is a page frame that contains an integer *#Pages*, an array *Value* of length *#Pages* of pointers to allocated page frames, and a component *Next* that is used to link returned page tables to one another. The available page frame multiqueue *AvailPageMQ* is a linked allocated linear multiqueue of page tables, where a page table's multiplicity is defined by its *#Pages* component.

A request for p page frames is a multi-deletion of size p, and a release of a page table containing q page frames is a multiqueue insertion of an item (the page table) of multiplicity q. In addition to the number of pages requested, the multi-deletion is passed a page table pointer in which to store the allocated page frames. If this pointer is nil, then one of the p page frames allocated will be set aside and used for the page table.

The final deletion of a page table T of multiplicity p is the deletion that allocates the last pointer from T's *Value* array. After this occurs, T itself becomes an available page frame. If the final deletion needs additional page frames (i.e., its request was only partially satisfied by the contents of T's *Value* array), it takes T as an available page frame and dequeues it from

AvailPageMQ. Otherwise, the final deletion leaves *T* on the queue, setting *T*'s multiplicity to one and initializing the first entry of *T*'s *Value* array to point to *T*. That is, *T* references itself as an available page frame. In this latter case, the ‘‘second’’ final deletion performed on *T* dequeues and uses *T* in satisfying its request. (In the algorithm below, a deletion is the ‘‘first’’ final deletion of a page table *T* if the first element of *T*'s *Value* array does not reference *T*.)

Since empty page tables are themselves available page frames, a multiqueue insertion of a page table of multiplicity *p* increments the count *#AvailPF* of available page frames in the queue by *p*+1. Also, if the insertion is passed an empty page table *T*, i.e., one having a multiplicity of zero, *T* is readied immediately for its second final deletion by setting the first entry in *T*'s *Value* array to point to *T* and setting *T*'s multiplicity to one.

```
AvailPageMQ = record    Linked Allocated Linear MultiQueue
                  #AvailPF, Qsize, Qfront, Qrear,
                  FrontMult : integer;
                  List      : array[0..Qsize–1] of ListElement
              end;

ListElement = record
                  HeadMult   : integer;
                  Head       : PageTable_Pointer;
                  Tail       : Pointer to ListElement or PageTable;
                  Access_Sem : Weak_Semaphore
              end;

ListElement_Pointer = ↑ListElement;

PageFrame = array[1..PageSize] of Words;

PageFrame_Pointer = ↑PageFrame;

PageTable = record
                #Pages : integer;
                Next   : PageTable_Pointer;
                Value  : array[1..#Pages] of PageFrame_Pointer
            end;

PageTable_Pointer = ↑PageTable;
```

```
function GetPages_V2 (PT                    : PageTable_Pointer;
                      #PagesRequested : integer;
                      Q                     : AvailPageMQ);
   return Linked_LinMulti-Delete(PT, #PagesRequested, Q)
end function GetPages_V2;


procedure ReturnPages_V2 (PT : PageTable_Pointer;
                          Q   : AvailPageMQ);
   Linked_LinMQInsert(PT, PT↑.#Pages, Q)
end procedure ReturnPages_V2;


procedure Linked_LinMQInsert (PT : PageTable_Pointer;
                              M  : integer;   Page Table's Multiplicity
                              Q  : AvailPageMQ);
var i : integer;
   with Q do
   i := F&A(Qrear, 1) mod Qsize;
   if i = Qsize–1 then F&A(Qrear, –Qsize) end if;
   PT↑.Next := nil;
   if M = 0 then                          table is empty
      PT↑.#Pages := 1;
      PT↑.Value[1] := PT                  prepare for second final deletion
   else
      PT↑.#Pages := M
   end if;
   Weak_P(List[i].Access_Sem);
      List[i].Tail↑.Next := PT;
      List[i].Tail := PT;
      if List[i].Head = PT then           list was empty
         List[i].HeadMult := PT↑.#Pages;   promote PT's multiplicity
         if Qfront = i then FrontMult := PT↑.#Pages end if
      end if;
   Weak_V(List[i].Access_Sem);
   F&A(#AvailPF, M+1)
   end with
end procedure Linked_LinMQInsert;
```

```
function Linked_LinMulti-Delete (PT : PageTable_Pointer;
                                 M  : integer;  number of pages requested
                                 Q  : AvailPageMQ);
var d, g, n, s : integer;
   with Q do
   if TDR(#AvailPF, M)   then
      repeat    keep taking items from the Qfront list item until satisfied
         s := 0;
         repeat
            if FrontMult > 0 then s := F&A(FrontMult, −M) end if
         until s > 0;
         d := Qfront;
         if PT = nil then  allocate page table
            PT := List[d].Head↑.Value[s];  PT↑.#Pages := 0;
            M := M – 1;  s := s – 1
         end if;
         g := s min M;   number of entries to copy from front page table
         PT↑.Value[PT↑.#Pages+1..PT↑.#Pages+g] := List[d].Head↑.Value[s–g+1..s];
         M := M – g;      remaining number to obtain
         PT↑.#Pages := PT↑.#Pages + g;
         if F&A(List[d].HeadMult, –g) = g then    final multi-delete
            Open_Next_List_Site_PV;
            Weak_P(List[d].Access_Sem);
               if M = 0 and List[d].Head ≠ List[d].Head↑.Value[1] then
                  init. for second final deletion
                  List[d].HeadMult := 1;
                  List[d].Head↑.Value[1] := List[d].Head
               else   dequeue front page table
                  if M > 0 and List[d].Head ≠ List[d].Head↑.Value[1] then
                     PT↑.Value[PT↑.#Pages+1] := List[d].Head;
                     M := M – 1;
                     PT↑.#Pages := PT↑.#Pages + 1
                  end if;
                  List[d].Head := List[d].Head↑.Next;
                  List[d].HeadMult := List[d].Head↑.Next↑.#Pages;
                  if List[d].Head = nil then
                     List[d].Tail := address(List[d])
                  end if
               end if;
               if Qfront = d then   queue wraparound
                  FrontMult := List[d].HeadMult
               end if;
            Weak_V(List[d].Access_Sem)
         end if
      until M = 0;
      return true
   else
      return false
   end if
   end with
end function Linked_LinMulti-Delete;
```

```
macro Open_Next_List_Site_PV;
    n := (d + 1) mod Qsize;
    Weak_P(List[n].Access_Sem);
        Qfront := n;
        FrontMult := List[n].HeadMult;
    Weak_V(List[n].Access_Sem);
end macro Open_Next_List_Site_PV;
```

4.3. Chapter Summary

In this chapter parallel algorithms for variable and fixed size block allocation of memory have been given. Many of the algorithms are parallel analogs to existing serial algorithms such as Knuth’s boundary tag method and the binary buddy system.

A key to parallelizing memory allocation is representing the pool of free blocks with concurrently accessible data structures; this is necessary to avoid a bottleneck in accessing shared data. As a consequence, our algorithms rely heavily on the data structure implementations of the preceding two chapters. Specifically, the parallel boundary tag method uses a concurrently accessible queue-like data structure, good fit and the second version of the buddy system use concurrently accessible queues, and the second paging scheme uses concurrently accessible multiqueues. The remaining parallel algorithms, the first versions of the buddy system and paging, represent the pool of free blocks with trees and arrays of counters (manipulated via fetch-and-add). Also, for our queue-based memory algorithms, the operation of interior removal plays a significant role in supporting block coalescing.

CHAPTER 5

Task and Resource Management

Task management for the Ultracomputer is based on a decentralized design that permits each task to service its own requests, thereby eliminating performance bottlenecks due to central servers. Of course, this strategy may introduce new serialization points, primarily when accessing shared data, and thus defeat the purpose of decentralization. Therefore, in this chapter a representative set of task and resource primitives that could form the nucleus of the task and resource management components of an Ultracomputer operating system is studied. The intention of this exercise is to demonstrate that serialization-free implementations of task and resource primitives are attainable. In addition, we present strategies that suggest how system activities may be parallelized when executed on multiprocessors such as the Ultracomputer. Task creation, execution, (normal) termination, destruction, and resource allocation are used as examples.

Parallelizing resource primitives may be of dubious value when the resource in question itself requires serial access. However, when a resource can be used concurrently by a group of tasks, being able to execute its primitives in parallel becomes more attractive. For example, if a resource is shared between readers and writers and the resource's request and release primitives are serialization-free, then reader activity becomes fully parallel in the absence of writer activity. Similarly, if *m* units of a resource are requested and released via serialization-free P/V operations on a counting semaphore (having a maximum value of *m*), then up to *m* units may be allocated and used in parallel. Finally, if we associate the occurrence of an event with the availability of a resource and the testing of an event's status is fully parallel, then accesses to the resource become serialization-free between the occurrence and resetting of the event.

We present serialization-free block-queue[1] implementations of semaphore,[2] readers/writers,

---

[1] That is, implementations that place a task on a wait queue if the task's request for a resource can not be granted immediately.

[2] The semaphore implementation is taken from Rudolph [82].

and event primitives, as well as a weak (i.e., busy-waiting) implementation for events, complementing the weak, fully parallel implementations of semaphore and readers/writers operations developed by Gottlieb et al. [83b]. The majority of the weak event primitives are serialization-free; notably, the important operation of testing an event's status is fully parallel. In general, the block-queue implementations exhibit the same degrees of parallelism as their weak counterparts.[3] An important feature of the block-queue readers/writers and event implementations is that tasks waiting (on a queue) for a resource can be resumed in parallel by multiple PEs. Also, for each block-queue implementation it is shown that blocked tasks may be destroyed without reducing an implementation's maximum level of parallelism.

To implement serialization-free task and resource primitives, we use distributed code and shared data structures, where the latter consists primarily of queues, lists, counters, and pointers. Bottlenecks can arise if these data structures are not concurrently accessible. Our overall strategy is therefore to represent shared queues and lists via concurrently accessible queues and multi-queues. In our implementations, shared counters and pointers do not, in general, require special representations as the operations of read, write, and fetch-and-add provide sufficient parallel access.

The chapter is divided into four parts: introduction, system data structures, task primitives, and resource primitives.

5.1. Introduction

In this section we introduce the basic concepts upon which our task and resource primitives and data structures are built, and the terminology used to describe them.

A task is *running* when it is being executed by a PE; it is *ready* when it is capable of running but there is no available PE; and, it is *blocked* when its execution cannot proceed until it acquires a resource (e.g., a critical code section). Ready tasks are kept on a queue called the *Ready Queue*. A blocked task is inserted onto a wait queue associated with the resource it has requested. When a task blocks or exits the system, it frees its PE by invoking a scheduling primitive that deletes a task from the Ready Queue, which is then run on the PE.

[3] The exception is one of the two implementations given for events, but even in this one case the testing of an event's status is serialization-free.

A task initiates a parallel activity by *creating* subtasks and inserting them onto the Ready Queue from where they will later be deleted and executed by available PEs.[4] The initiating task and its subtasks are referred to as the *parent* and its *children*, respectively. Children can in turn create their own offspring (subtasks). Thus, at any point in time, a task and all of its descendants can be represented by a tree, where each internal node N represents a parent task T, and N's immediate descendants represent T's children.

The creation of children may be done serially by one task or in parallel by multiple tasks. In the first case, a parent directly creates its children by allocating the resources needed for their execution. Then, to enable the children to begin execution, the parent inserts them, one by one, onto the Ready Queue. In the second case, a parent indirectly creates its children by requesting that available PEs, i.e., those attempting to delete tasks from the Ready Queue via the execution of scheduling primitives, be given the work of allocating the children's resources, i.e., of creating the children. PEs given this assignment then execute the children they create since they had been searching the Ready Queue for executable tasks in the first place. To support indirect creation, the Ready Queue consists of concurrently accessible queue and multiqueue components.[5] A parent inserts a ''*create M children*'' request, referred to as a (*task*) *template* with multiplicity M, onto a multiqueue component of the Ready Queue. Each time a deletion is performed on the template[6] by an available PE, the PE creates a child from the information contained in the template and then executes it. Direct creation is appropriate for a small number of children; otherwise, indirect creation is preferred.

For the block-queue readers/writers and event implementations, the unblocking of tasks, i.e., making blocked tasks ready, is another activity, like the creation of children, that may be done serially by one task or in parallel by multiple tasks. In the first case, one task deletes blocked tasks from a wait queue and inserts them onto a queue component of the Ready Queue. In the second case, one of two methods may be selected. One method permits recently unblocked tasks to delete other tasks from the wait queue and insert them onto a queue component of the Ready Queue. These operations can be done in parallel if the wait queue is concurrently

[4] Of course, the degree of concurrent execution attained depends on the actual number of available PEs.

[5] Gottlieb et al. [83b] state that parallelizing the creation of children was a primary motivation for the development of multiqueues.

[6] Recall from chapter 3 that up to M deletions may be done concurrently on this entry.

accessible. The second parallel method for unblocking tasks is similar to the method for indirectly creating *M* tasks, which inserts a template with multiplicity *M* onto a multiqueue component of the Ready Queue. A task unblocks *M* tasks from a wait queue *WQ* by inserting an ''*unblock M tasks from queue WQ*'' request onto a multiqueue component of the Ready Queue. One component of the request is either *WQ* or a pointer to *WQ*. The request is called a *queue tagged with multiplicity M* in the first case[7] and a *queue pointer tagged with multiplicity M* in the second. When an available PE performs a Ready Queue deletion on this request, it then performs a deletion from *WQ* to obtain a ready task, which it then executes.[8] Again, *WQ* should be a concurrently accessible queue to realize maximum concurrency.

Asymptotically, the parallel unblocking methods permit tasks to be unblocked and to begin execution in less time than the serial method when multiple PEs are available.

Note that a multiqueue component of the Ready Queue can contain objects of type template, queue tagged with multiplicity, or queue pointer tagged with multiplicity.

After a parent has inserted onto the Ready Queue either its children or a template that will be used to create them, the parent may continue with its execution or elect to *wait*, by blocking, for the children to *terminate*, i.e., to release their privately owned resources and exit the system. We prevent a parent from terminating before its children so that the parent can safely deallocate shared resources. In the event that the parent does wait, the *last* child to terminate will awaken it. Notice that determining which child is last represents a potential bottleneck when many children terminate simultaneously. As we shall show, the fetch-and-add operation provides a simple solution to this problem by enabling the children to concurrently decrement a counter that has been initialized to the number of created children. The last child is the one that lowers this counter to zero.

One task can direct another task to block itself by sending it a *suspend* signal. A suspended task can be *resumed* (unblocked) by another task (which need not be the one that caused the suspension). Also, one task can cause another task to terminate itself prematurely, along with the latter's entire subtree, by sending it a *destroy* signal. Suspend signals are cancelable (by resume

[7] We refer to the request as a queue tagged with multiplicity since the shorter term *queue with multiplicity* has been used by Gottlieb et al. [83b] as a synonym for multiqueue.

[8] To simplify the initial description of this activity, we have ignored task priorities. The unblocking of tasks that belong to the same wait queue but that have different priorities is discussed in section 5.3.2.6.

operations), but destroy signals are not. We show that the destruction of a task's subtree, like the creation of M children, is an activity that can be performed serially by one task or in parallel by multiple tasks. If multiple processors are available, the parallel method will complete in less time.

We wish to point out that under certain conditions a task will be permitted to disregard a suspend or destroy signal for a finite period of time. In particular, a task will not respond to either signal while it is executing an operation that might leave the system in an inconsistent state (if sent a destroy signal) or tie up a shared resource for an indefinite period of time (if sent a suspend signal). Such operations will be called *critical transactions*. These transactions will be dealt with in greater detail in section 5.3.

Based on the foregoing description, we will refer to a task's status as

> *running*: when it is being executed by a PE;
>
> *ready*: when it is capable of running but there is no free PE;
>
> *blocked*: when it is waiting (on a queue) for a resource (such as a critical code section);
>
> *suspended*: when it has been forced to block;
>
> *waiting-for-children*: when the task is blocked waiting for its children to terminate;
>
> *terminated*: when it has exited the system.

Suspended and waiting-for-children are special cases of blocked that will be treated separately from the latter to simplify the descriptions of the task primitives. The blocked status will be reserved for tasks on resource wait queues (excluding the Ready Queue); queues will not be used for tasks that are suspended or waiting-for-children.

5.2. Data Structures

Data structures for the Ready Queue, tasks, templates, queues tagged with multiplicity, and queue pointers tagged with multiplicity are defined in this section.

We assume that a task is assigned a priority from 0 to *MaxPrio*, where larger numbers represent higher priorities. The Ready Queue consists of concurrently accessible queue and multiqueue components $RQ[0..MaxPrio]$ and $RMQ[0..MaxPrio]$. $RQ[i]$ contains ready tasks of priority i and $RMQ[j]$ holds ready multiqueue items (e.g., templates, queues tagged with multiplicity) of priority *j*. Each queue and multiqueue component[9] is implemented as an array of linked lists and supports the operations of insertion, deletion, and interior access. The particular interior access operations used in this chapter are queue removal and the multiqueue operations of RaiseMult and LowerMult. The components *RQ* and *RMQ* are defined as (global) variables, not types, in order to simplify our implementations of the task and resource primitives:

```
var  RQ    : array[0..MaxPrio] of Linked_Queue;
     RMQ : array[0..MaxPrio] of Linked_MQueue;
```

To link an object (task or template) to a queue or multiqueue, it must contain a link component of corresponding type. The components were defined in chapters 2 and 3, respectively, and are repeated below:

```
QEntry = record     to link an object to a queue
             List_Loc : ListElement_Pointer;
             Next,
             Prev        : Pointer to QEntry or ListElement;
             #Missing : integer
         end;

MQEntry = record    to link an object to a multiqueue
             List_Loc : ListElement_Pointer;
             Mult        : integer;
             Next,
             Prev        : Pointer to MQEntry or ListElement;
             #Missing : integer
         end;
```

Each task and template is represented by a *descriptor* or *control block* defining its attributes. A task control block (*TCB*) provides sufficient information for a PE to begin the execution of a task.[10] A task template control block (*TTCB*) specifies the characteristics for the tasks that are to be created from it. The control blocks below contain the minimum number of attributes needed to support our primitives and queue operations. An actual implementation would likely contain additional attributes.

[9] The multiqueue components may be linear or logarithmic. See chapter 3 for details.

[10] Our *TCB*s have been modeled after one given by Shaw [74].

```
TCB =   record    Task Control Block
            ResourceQEntry,
            SiblingQEntry          : QEntry;
            Name                   : string;
            Status                 : (Ready, Running, Blocked, Suspended,
                                      WaitingForChildren, Terminated);
            PE_Id                  : integer;
            Priority               : 0..MaxPrio;
            MMap                   : MemMap;
            CPUState               : ProcessorState;
            Parent                 : TCB_Pointer;
            #Children              : integer;
            Progeny                : record
                                          DC  : ↑Linked_Queue;
                                          IC  : TTCB_Pointer
                                     end;
            MyDownElement          : ↑TCB_Pointer;
            TransCnt               : integer;
            Signals                : (None, ToBeDestroyed, ToBeSuspended);
            SysStack               : array[] of Word;
            Scheduling             : boolean;
            Lock                   : Weak_Semaphore
        end;

TCB_Pointer = ↑TCB;
```

A task is attached to (one of the concurrently accessible queue components of) the Ready Queue or to any other resource queue (e.g., a semaphore wait queue) through the resource queue entry of its control block. It is linked to its brothers through the sibling component.

Each task has a unique external name *Name*, which is recorded in a *Name Table*, and a unique internal name *I*, which is the address of its control block. The Name Table is a map from external to internal names. (The structure of this table will not be given.)

A task's current status is recorded in *Status*. When a task is running, *PE_Id* indicates which processor is executing it. The priority of a task ranges from 0 to *MaxPrio*. *MMap* defines a task's address space and provides any needed address translation (e.g., virtual to real). *CPUState* defines the initial processor state (the program counter, interrupt mask, etc.) for a task or a task's processor state when its PE was context-switched to another task.

The creator of a task, i.e., the task that (directly or indirectly) created the task, is recorded in *Parent*. *#Children* equals the number of created but not yet terminated subtasks belonging to a task. *Progeny* contains a task's offspring, allowing a task to locate its children quickly. More specifically, *Progeny* consists of a concurrently accessible queue *DC* of *TCB*s and a linked list *IC* of *TTCB*s. Children directly created by a task are inserted onto *DC*, and those indirectly created by it are attached to the template within *IC* from which they were formed. The function of the *MyDownElement* component will be explained when the template structure is described.

Each time a task starts or completes a critical transaction, *TransCnt* is incremented or decremented by one, respectively. *Signals* records the receipt of a signal sent to a task.

SysStack is the stack used by system primitives called by a task. The purpose of the *Scheduling* flag will be explained in section 5.3.1.4.

Finally, each *TCB* has a private binary semaphore *Lock* used when exclusive access is required.

A template contains sufficient context for a scheduling primitive to create a *TCB*. Many of its attributes are the same as for a task, those that differ are explained below.

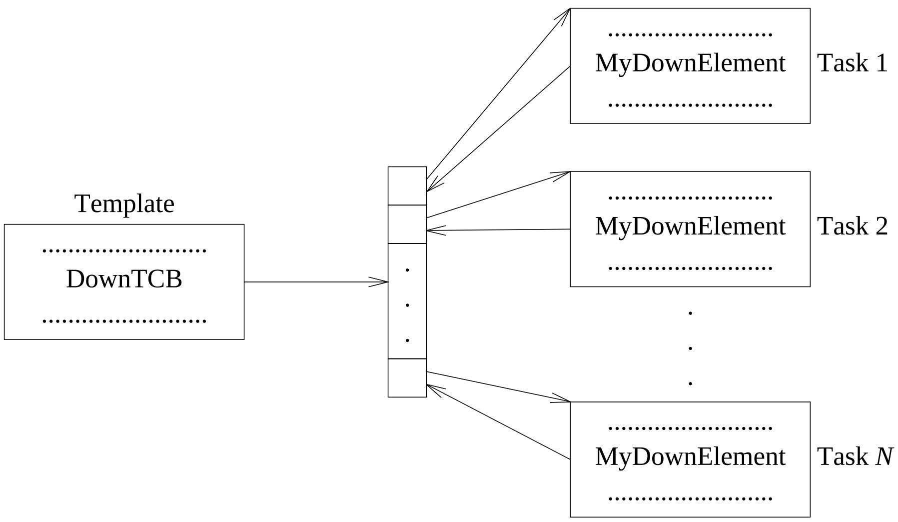
```
TTCB = record    Task Template Control Block
        ResourceMQEntry : MQEntry;
        NextTemplate    : TTCB_Pointer;
        Name            : string;
        Status          : (Ready, Suspended);
        Priority        : 0..MaxPrio;
        MMap            : MemMap;
        CPUState        : ProcessorState;
        Parent          : TCB_Pointer;
        DownTCB         : ↑array[] of TCB_Pointer;
        Index           : integer;
        #Created        : integer;
        Lock            : Weak_Semaphore
    end;

TTCB_Pointer = ↑TTCB;
```

All templates created by a task are linked together through their *NextTemplate* components. A template is always inserted onto a multiqueue component of the Ready Queue. It therefore has a multiqueue link component *ResourceMQEntry*. The multiplicity of a template is maintained in the *Mult* component of the resource multiqueue entry. *DownTCB* is (a pointer to) an array of size

Mult that contains (pointers to) the *TCB*s of the tasks created from a template.[11] When a scheduling primitive creates a task *T* from a template, it generates an index *i* into *DownTCB*, by incrementing *Index* by one, and then places a pointer to *T*'s *TCB* into *DownTCB*$[i]$. The primitive also sets *T*'s *MyDownElement* component to point to *DownTCB*$[i]$ as well.[12] Lastly, the scheduler increments *#Created*, indicating the completion of the task's creation.[13] Figure 5-1 illustrates the relationship between a template and *N* tasks created from it.

We note that the *DownTCB* structure is not essential. The *TCB*s of tasks created from a template could be inserted onto their parent's *DC* queue instead of being inserted into the *DownTCB* array associated with the template. Also, if the number of lists comprising the queue (i.e., the *Qsize* component) equals the maximum number of active tasks or tasks that a parent can have as children, then the queue supports the same level of concurrent access as the *DownTCB* arrays. An advantage of the latter over the former is that array operations require less time than queue operations. The arrays are presented only as an alternative to queues; we are not



N tasks attached to their template.

Figure 5-1.

[11] The elements of this array point *down* the task tree.

[12] When a task terminates it references this component to disassociate itself from its parent.

[13] As will be seen, this counter is required only to support task destruction.

necessarily advocating one over the other.

We will assume that there are a fixed number of task and template control blocks. Available blocks are found in the concurrently accessible queues *FreeTCB* and *FreeTTCB*. A block is allocated/deallocated by a deletion/insertion performed on its queue. Initially, all tasks and templates are on these queues and have their *Scheduling* and *Lock* attributes set to false and open,[14] respectively. These queues, like the Ready Queue, are assumed to be global variables.

Finally, recall from the preceding section that objects inserted onto multiqueue components of the Ready Queue are of type template, queue tagged with multiplicity, or queue pointer tagged with multiplicity. The first data type has already been defined. The second consists of a multiqueue link and a queue of tasks, while the third consists of a multiqueue link and a pointer to a queue of tasks. The multiplicity of an item of the second or third type is defined by the *Mult* component of the multiqueue link and equals the number of tasks on the queue that is associated with the item.

```
QTaggedWithMultiplicity = record
                              ResourceMQEntry : MQEntry;
                              Q               : Linked_Queue;
                          end;

QPointerTaggedWithMultiplicity = record
                                     ResourceMQEntry : MQEntry;
                                     QPtr            : ↑Linked_Queue;
                                 end;
```

5.3. Task Primitives

This section is divided into three parts. First, brief descriptions of the primitives and their interrelationships are given. Second, detailed implementations of the primitives are presented. Finally, the performances of different methods for creating and destroying tasks are compared.

5.3.1. Overview

Our task primitives are similar in style and power to those found in simple or model operating systems (e.g., Brinch Hansen [73b], Shaw[74], Madnick and Donovan [74]). It is assumed

[14] *Lock*s are included in templates to permit tasks and templates to be created by the same procedure.

that the primitives are implemented as subroutines. Our model, in which tasks execute primitives within their own address spaces is often referred to as a self-service approach. This should be contrasted with a server-based approach in which a task's PE is context-switched to a different (so called server) process each time a primitive needs to be performed. The primitives are listed below, accompanied by brief descriptions of their functions.

1. *Create Task/Template*–allocates and initializes a *TCB*/*TTCB*; sets the status to suspended.
2. *Initiate Template with Multiplicity M*–adds a template with multiplicity *M* to the Ready Queue.
3. *Create Task from Template*–creates a task from a template. The task's status is running since immediately after its creation it is executed.
4. *Resume*–cancels a task's pending suspend signal if present and if the task has a status of suspended, the task is inserted onto the Ready Queue.
5. *Signal Suspend*–signals a task to suspend itself.[15]
6. *Get Task*–deletes a task from the highest priority non-empty queue component $RQ[p]$ or deletes a multiqueue item i from the highest priority non-empty multiqueue component $RMQ[q]$ if $q > p$. In the latter case, if i is a template, then a task is created from it (via *CreateTaskFromTemplate*); otherwise, i is a queue or queue pointer tagged with multiplicity, and GetTask then performs a standard deletion from the queue associated with i to obtain a task. The task obtained from $RQ[p]$ or from item i of $RMQ[q]$ is then run.
7. *Signal Destroy*–signals a task to destroy itself.
8. *Destroy Tree*–signals each node within a given task *T*'s subtree to destroy itself and then terminates *T*.
9. *Process Signals*–suspends a task or begins the destruction of a task and its subtree (by calling *DestroyTree*) if the task has been appropriately signaled and all of the task's critical transactions have been completed.
10. *Wait for Children*–blocks a task until all its children have terminated.
11. *Child Terminated*–notifies a parent that one of its children has terminated. If all of the children have now terminated and the parent's status is waiting-for-children, the parent is inserted onto the Ready Queue.[16]
12. *Terminate*–marks a task as terminated.
13. *Start Critical Transaction*–returns false if a task has received a destroy signal and is not executing a critical transaction; otherwise, it increments the task's critical transaction count and returns true.

[15] A ''suspend template'' primitive is not given.

[16] This operation is not a true primitive, but a part of *Terminate*. However, it is treated separately to emphasize the relationship between a parent and its offspring.

14. *Complete Critical Transaction*–decrements a task's critical transaction counter and, if the count is now zero, processes its signals (via *ProcessSignals*).

15. *Block Task on Q*–marks the calling task as blocked and inserts it onto resource wait queue *Q*.

16. *Unblock Task from Q*–deletes a blocked task from resource wait queue *Q*, marks it ready, and inserts it onto the Ready Queue.

To gain a better understanding of the primitives' functions, interactions, and implementations, we examine four basic task activities: the normal life cycle of a task, i.e., its creation, execution, and termination; the abnormal termination or destruction of a task; the suspension of a task; and the coordination of accesses to a task's *TCB*, especially when a *TCB* is queued or its system stack is being used by a system primitive. Detailed implementations of the primitives themselves are then given.

5.3.1.1. The Task Life Cycle

We illustrate the life cycles of tasks by giving two possible implementations for the parallel loop statement

S : **forall** $i := 1$ **to** M **do** f,

which states that M evaluations of function f may be performed concurrently.[17]

Our first implementation consists of having P, the task that encounters the **forall** statement, create (directly) and then enqueue M subtasks onto the Ready Queue using the Create and Resume primitives.

for j := 1 **to** M **do**
    Create a task c_j *to execute* f;
    Resume task c_j
end for;

After P enqueues the subtasks, it has the option to continue executing or to wait for them to terminate (via a WaitForChildren call). Each time a task blocks, suspends, waits for its children, or terminates, it calls GetTask so that the PE that was executing the task may now execute a ready task. Therefore, as PEs become available, each subtask c_j is deleted from the Ready Queue and

[17] This construct, among others, is addressed in chapter 6.

executed.

Our second implementation is indirect in that a parent *P* creates a template *Temp* from which the children to execute *f* are later created. *P* inserts *Temp* with multiplicity *M* onto the Ready Queue. In short, *P* uses the Create and InitiateTemplate primitives:

Create a template Temp;
Initiate Temp with Multiplicity M;

After *P* initiates *Temp*, it may either continue to execute or wait for its children to terminate (again via a WaitForChildren call). Each time an available PE deletes *Temp* from the Ready queue (via GetTask), it creates a child (via CreateTaskFromTemplate) to evaluate *f* and then executes the resulting task.

In both implementations, after a child c_j of *P* executes *f*, it terminates (via Terminate). As part of c_j's termination, it notifies *P* (via ChildTerminated) that it is done. This results in the resumption of *P* when *P* is waiting and c_j is the last of its children to terminate.

As noted in Gottlieb [87], it may be preferable to assign children higher Ready Queue priorities than their parents to force a depth-first traversal of the overall task tree. For example, in evaluating nested **forall**s where parents (the initiators of **forall**s) wait for their children (the evaluators of loop bodies) to terminate, this approach would help to avoid an exponential generation of created but not yet terminated tasks, which could overwhelm the capacity of system tables or other resources.

5.3.1.2. Task Destruction

In keeping with the self-service design philosophy of the operating system, tasks are responsible for carrying out their own destruction (or suspension).

A task *D* initiates the destruction of another task *T* by ‘‘sending’’ it a *ToBeDestroyed* signal. That is, the *Signals* component within *T*’s *TCB* is set to *ToBeDestroyed*. *T*’s response to this signal will be first to signal its descendants to destroy themselves and then to terminate itself. When *T* is signaled by *D*, it will either be running or in a ‘‘wait’’ state, where wait$\in$\{blocked, suspended, waiting-for-children\}. Suppose *T* is running. Since (we assume that) a running task does not periodically check its signals component, *D* sends an interrupt[18] to *T*'s PE, in response to which *T* will check its signals. It is possible that *T*'s PE may have its interrupts disabled when the interrupt arrives. To ensure that *T* does not enter a wait state, and remain unaware of the signal, either when an interrupt is pending or after the *Signals* field has been set but before the interrupt arrives at the PE, we adopt the convention that a task checks its *TCB*'s *Signals* when performing a ‘‘running⟶waiting’’ transition. On the other hand, if *T* is already in a wait state when *D* marks *T*'s *TCB*, *D* will prematurely awaken *T* by placing it onto the Ready Queue, allowing the signal to be processed as soon as *T* runs. That is, after a task performs a ready⟶running transition, it will first process any signals that are pending before proceeding with its normal execution.

This strategy is sufficient if *T* can always abandon its current activity to process the destroy signal without leaving the system (i.e., its resources) in an inconsistent state. This may not be possible or could involve complex forward or backward error recovery schemes. We take a simpler tack by requiring a task to indicate when it has begun an operation, termed a critical transaction,[19] that must be completed before it can be destroyed. For example, the execution of a critical code section could be designated a critical transaction. Of course, this designation is unnecessary if a task can execute such an operation with interrupts disabled and remain a running task (since then the signal can be processed when the interrupts are re-enabled). If disabling interrupts is insufficient (e.g., if the task must enter a wait state when executing an operation), then its indication that it has started a critical transaction is needed to prevent it from being prematurely removed from a wait queue[20] and destroyed. We assume that a task cannot die or remain forever within a critical transaction.

When a task begins a critical transaction it invokes the StartCriticalTransaction primitive. The primitive fails (returns false) if it finds that the calling task has received a destroy signal and has not begun any other critical transactions. This check may seem unnecessary since a running task *T* that is to be destroyed has an interrupt sent to its PE. But without this check, *T* could begin a critical transaction (or disable its interrupts) before the interrupt sent by its destroyer

[18] This may be done with the *reflection* operation introduced in Edler et al. [85].

[19] In the model operating system given by Madnick and Donovan [74], the term ‘‘system must complete mode’’ is used to describe a task that has acquired a system lock that prevents the task from being killed until the lock is released. Our critical transactions denote a similar mode of operation.

[20] If the state is suspended or waiting-for-children, this is a virtual queue.

arrived at the PE, delaying *T*'s destruction until the transaction was completed. Therefore, a task's *Signals* component is checked before a critical transaction is begun.[21] A task that receives a destroy signal while executing a critical transaction will process it upon completing the transaction (via the CompleteCriticalTransaction primitive).

To illustrate how critical transactions are implemented, let us suppose that a shared resource *R* is requested and released as shown below, where Request and Release are atomic and a queue *Q* holds tasks waiting to use *R*.

```
Request(R): if R is unavailable then        Release(R): if Q is not empty then
                Block(Q)                                    Unblock(task, Q)
            else                                        else
                mark R as unavailable                       mark R as available
            end if;                                     end if;
```

If *R* has a status of *unavailable* when a task requests it, then the task blocks (i.e., inserts itself onto *Q*); otherwise, the task uses *R*. When a task *t* releases *R* and tasks are waiting for the resource (i.e., *Q* is not empty), *t* unblocks a waiting task *w* (i.e., *t* deletes task *w* from *Q* and inserts it onto the Ready Queue), thus granting *w*'s request.

Figure 5-2 gives two possible critical transaction implementations for accessing *R*.[22] Notice that the ''start'' of the critical transaction may be placed before or after the request for *R*. A difference between the two implementations is that in (b), but not (a), if a task receives a destroy signal at any time between its wait queue insertion (via Block) and its running, it will fail to start the transaction (assuming no other critical transactions have been begun); instead, it will release *R*, allowing other tasks to use the resource, and then destroy itself.

5.3.1.3. Task Suspension

A task *S* initiates the suspension of another task *T* by placing a *ToBeSuspended* signal in *T*'s *TCB*. If *T* is running and not executing a critical transaction, *S* interrupts *T*'s PE. In response, *T* releases its PE, but none of its other resources, and enters a suspended state from which it can later be resumed or destroyed. If *T* has begun a critical transaction or is in a blocked or waiting-

[21] Alternatively, this check could be omitted if we did not wish the signal to be processed as soon as possible. That is, the signal could be left pending until the transaction was completed.

[22] Our primitives support both implementations.

```
if StartCriticalTransaction then
    Request(R);
    use R;
    Release(R);
    CompleteCriticalTransaction
else
    DestroyTree
end if;
```

(a)

```
Request(R);
if StartCriticalTransaction then
    use R;
    Release(R);
    CompleteCriticalTransaction
else
    Release(R);
    DestroyTree
end if;
```

(b)

Critical Transaction Implementations.

Figure 5-2.

for-children state, its suspension is delayed until, respectively, the transaction is completed or the task runs. This avoids tying up shared resources that may be associated with the transaction and permits a ‘‘waiting’’ task to still acquire resources.

Task suspension is less urgent than task destruction. Therefore, unlike a destroy signal, a suspend signal is not detected by a task *T* when it undergoes a running⟶blocked or running⟶waiting-for-children transition or when it begins a critical transaction. In the first two cases, *T*’s suspend signal will not be processed until it runs, and in the last, it will not be processed until *T* completes all critical transactions.

5.3.1.4. TCB Access

In this section we address two *TCB* access implementation problems and their solutions. The first problem deals with *TCB* lock/queue deletion conflicts, and the second one concerns multiple primitives using the same *TCB* system stack concurrently.

In general, for a primitive to inspect or update a task’s attributes, it acquires exclusive access to the task’s *TCB* by setting the *TCB*’s *Lock*. However, as we shall show, this procedure cannot be followed when deleting a task from a queue, even though a deletion modifies a task’s resource queue entry attribute.[23]

[23] Recall from chapter 2 that a task’s (*TCB*’s) list location (*List_Loc*), which is a component of the resource queue entry (*ResourceQEntry*), is set to nil when the task (*TCB*) is deleted from a queue.

A primitive locks a task's *TCB* prior to inserting/removing the *TCB* onto/from a queue, and then unlocks it after the queue operation is completed. Unfortunately, deletions cannot be handled as easily: If a task's *TCB* had to be locked before being deleted from a queue, we shall show that deletions and removals can become deadlocked (causing primitives that invoke them, e.g., GetTask and SignalDestroy, to deadlock as well). For example, suppose a task *T* is at the front of a queue *Q*[24] when a task *A* locks *T*'s *TCB* and then executes a removal of *T* from *Q* at the same time that a task *B* executes a deletion from *Q*. The deletion locks the front list of *Q*, to obtain *Q*'s first task. It then waits to lock *T*'s *TCB* while the removal waits to lock *T*'s list, the front list of *Q*. Deadlock.

To preclude this deadlock scenario, we require a task primitive that deletes a *TCB* from a queue to set the *TCB*'s lock, instead of having the deletion do it. However, since deletion then neither sets nor inspects *TCB* locks, a primitive P_1 can delete a *TCB* that is locked by another primitive P_2: the setting of the lock does not prevent the *TCB*'s deletion. In addition, after a primitive P_3 deletes a *TCB* from a queue, another primitive P_4 can lock the *TCB* before P_3. In both of these cases, the primitive (P_1 or P_3) that deletes the *TCB* waits for the other primitive (P_2 or P_4) to unlock it. See Figure 5-3. We note that if the latter primitive (P_2 or P_4) inspects the *TCB* after it has locked the *TCB* and the former primitive (P_1 or P_2) has deleted the *TCB*, then it will find that the *TCB*'s resource queue entry indicates that the *TCB* is not queued[25] while the *TCB*'s status, which is either ready or blocked, indicates that the *TCB* is queued. In our implementations, this inconsistency occurs only when a SignalDestroy or DestroyTree primitive has locked a task's *TCB* as a prelude to destroying it. Both primitives respond by releasing the *TCB*. See sections 5.3.2.7 and 5.3.2.8 for details.

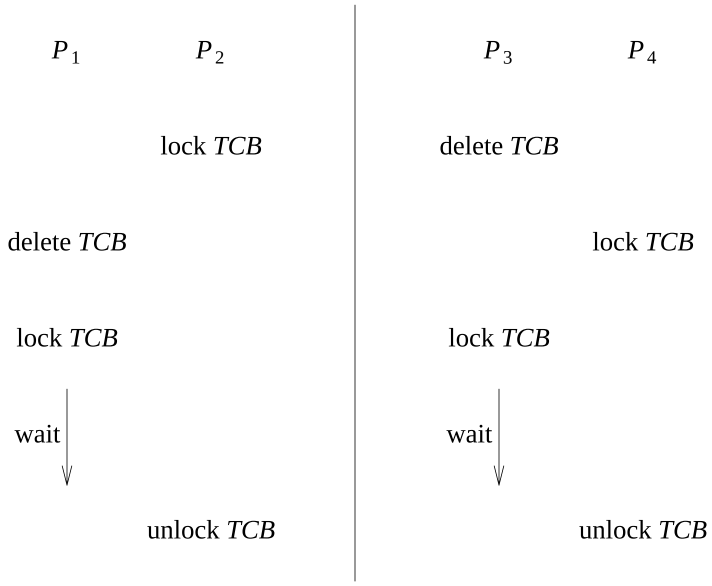
Another *TCB* access problem tied to queue deletion is that, without proper safeguards, two primitives can (erroneously) use a *TCB*'s system stack component simultaneously.[26] When a task *T* performs a running$\longrightarrow$waiting or running$\longrightarrow$terminated transition, its *TCB* is inserted onto a wait queue or the *FreeTCB* queue, respectively. *T* then calls GetTask, which uses the system stack of *T*'s *TCB*, to assign a ready task to its PE. Let us refer to *T*'s task control block as TCB_T.

---

[24] That is, *T* is at the head of a list *L*, which in turn, is as the front of *Q*.

[25] Specifically, the *TCB*'s list location component *List_Loc* is nil, which defines the *TCB* as dequeued.

[26] Recall that a system primitive invoked by a task uses the task's *TCB*'s system stack.



TCB lock/queue delete conflicts.

Figure 5-3.

First, suppose T calls GetTask because it has to wait and that while GetTask executes, the condition T is waiting on occurs, resulting in TCB_T being moved to the Ready Queue, where a PE then deletes it. The resumption of T by this new PE must be delayed until the GetTask initiated by T completes. Otherwise, if T invokes another system primitive before the GetTask is finished, two primitives will be using the same *TCB* stack concurrently. Second, suppose T calls GetTask because it is terminating and that while GetTask executes, a PE deletes TCB_T from the *FreeTCB* queue to create a new task U (via the Create primitive). U must not be permitted to invoke a system primitive until the GetTask operation invoked by T is done; otherwise, U's primitive and T's GetTask will use the same *TCB* stack concurrently.

Our solution for preventing two primitives from using the same *TCB* stack simultaneously is to raise a *TCB*'s scheduling flag prior to the invocation of a GetTask and then to lower this flag upon the completion of the GetTask (specifically, GetTask, just prior to returning, calls a function (Switch) that lowers the flag). A PE can not begin to execute a task obtained via a queue deletion until the task's *TCB* scheduling flag is down. We note that the function of the *TCB* scheduling flag could be performed via the *TCB Lock* component: raising and lowering the flag could be replaced with semaphore P and V operations on *Lock*. However, the code for a number of primitives would then contain unmatched P/V operations since the V would occur within GetTask. An explicit flag is used to indicate the temporary unavailability of a *TCB* to (hopefully) make the interactions of the primitives more understandable.

5.3.2. Implementations

Although not shown, it is assumed that the primitives run with interrupts disabled. The primitives perform minimal error checking; in particular, the possibility of a *TCB* or *TTCB* queue deletion or a table look-up failing, or the violation of a system constraint (such as the number of subtasks a task may create) has been ignored. We also assume that memory allocation for queues and arrays is performed via one of the methods given in chapter 4.

5.3.2.1. Create

Create allocates a task or template control block and then initializes it as follows: The control block is given a name, which is recorded in the Name Table, along with an initial priority, CPU state, and memory map. The control block's parent is *Current*, which is the identity of the task that invoked Create (i.e., associated with each PE is a pointer *Current* to the *TCB* of the task that the PE is currently executing). The task or template control block has an initial status of suspended, which can later be changed to ready through a Resume or an InitiateTemplate operation, respectively. If Create allocates a *TCB*, the *TCB* is inserted onto *Current*'s *DC* queue (which holds *TCB*s created directly by the parent), *Current*'s *#Children* counter is incremented by one, and the *TCB*'s *MyDownElement* pointer is set to nil. The *TCB* is treated as if it had been started and then immediately suspended. If a *TTCB* is formed, it is attached to *Current*'s *IC* list of templates.

As noted in section 5.3.1.4, it is possible that Create may allocate a *TCB* whose system stack is still being used by a GetTask. Therefore, before Create initializes a *TCB*, the *TCB*'s *Scheduling* flag must be down, indicating that no primitive is using the *TCB*'s stack.

```
function Create (BT : (Task, Template);  Block type
                 N   : string;           Name of task or template
                 P0  : integer;          Initial priority
                 M0  : MemMap;           Initial memory map
                 S0  : ProcessorState;   Initial processor state);
var t : Pointer to TCB or TTCB;
    if BT = Template then
        Linked_QDelete(t, FreeTTCB)
    else
        Linked_QDelete(t, FreeTCB);
        repeat until not t↑.Scheduling
    end if;
    Weak_P(t↑.Lock);
        t↑.Name := N;            RecordName(t);
        t↑.Priority := P0;       t↑.Parent := Current;
        t↑.CPUState := S0;       t↑.MMap := M0;
        t↑.Status := Suspended;
        case BT of
        Template  : ( t↑.Index := t↑.#Created := 0;
                      t↑.NextTemplate := Current↑.Progeny.IC;
                      Current↑.Progeny.IC := t   add t to list of templates )
        Task      : ( t↑.TransCnt := t↑.#Children := 0;
                      t↑.Progeny := nil;
                      t↑.MyDownElement := nil;
                      t↑.Signals := None;
                      F&A(Current↑.#Children, 1);
                      if Current↑.Progeny.DC = nil then
                          AllocateLinkedQueue(Current↑.Progeny.DC)
                      end if;
                      Linked_QInsert(address(t↑.SiblingQEntry), Current↑.Progeny.DC) )
      end case;
    Weak_V(t↑.Lock);
    return t
end function Create;
```

5.3.2.2. Initiate Template

A parent *P* has *M* children created (indirectly) from a template that it has previously allocated and (partially) initialized by first allocating a *DownTCB* array of size *M* that will hold pointers to the created children's *TCB*s, and then incrementing the *#Children* count by *M* to designate the impending creation of the tasks.[27] Lastly, the template with multiplicity *M* is inserted onto the Ready Queue.

Each step of InitiateTemplate can be accomplished in constant time.

```
procedure InitiateTemplate (Temp : TTCB_Pointer;
                            M    : integer);
    Weak_P(Temp↑.Lock);
        if Current = Temp↑.Parent and Temp↑.Status = Suspended then
            Temp↑.Status := Ready
        else
            Error   Current is not Temp's parent or Temp already initiated
        end if;
    Weak_V(Temp↑.Lock);
    Allocate(Temp↑.DownTCB, M);   allocate array of size M for offsprings' pointers
    F&A(Current↑.#Children, M);
    Linked_MQInsert(Temp, M, RMQ[Temp↑.Priority])
end procedure InitiateTemplate;
```

5.3.2.3. Create Task from Template

The CreateTaskFromTemplate primitive creates a task from a template by following the same sequence of operations as in Create except the template serves as the source of the assignments. A unique index *i* is generated for the task by performing a fetch-and-add on the template's *Index.* The task's external name is formed from the template's name and *i*. A task created from a template has a status of running since we only perform this operation when a PE is available. CreateTaskFromTemplate inserts a pointer to the task's *TCB* into *DownTCB*$[i]$ of the template and sets the task's *MyDownElement* component to point to *DownTCB*$[i]$. After the task is created, the template's *#Created* count is incremented by one. Notice that like Create, CreateTaskFromTemplate must wait until a *TCB*'s *Scheduling* flag is down before initializing the *TCB*.

M tasks can be created from a template in $\theta(\lceil M/N \rceil)$ time, where *N* is the minimum of the number of PEs and the number of lists forming the *FreeTCB* queue (i.e., the queue size *Qsize* of *FreeTCB*). References to the template by different PEs can be done in parallel.

[27] Notice that, instead, if each task formed from the template incremented *#Children* (by one) upon its creation (as part of CreateTaskFromTemplate), a parent could not determine from inspecting this counter whether all of its children had terminated or not. The WaitForChildren primitive (section 5.3.8) depends upon this information.

```
function CreateTaskFromTemplate (Temp : TTCB_Pointer);
var t : TCB_Pointer;
    i : integer;

    Linked_QDelete(t, FreeTCB);
    repeat until not t↑.Scheduling;
    Weak_P(t↑.Lock);
        i := F&A(Temp↑.Index, 1);
        t↑.Name := Temp↑.Name concat i;   RecordName(t);
        t↑.Status := Running;             t↑.PE_Id := ProcessorID;[28]
        t↑.Priority := Temp↑.Priority;    t↑.Parent := Temp↑.Parent;
        t↑.CPUState := Temp↑.CPUState;    t↑.MMap := Temp↑.MMap;
        t↑.#Children := 0;                t↑.TransCnt := 0;
        t↑.Progeny := nil;                t↑.Signals := None;
        Temp↑.DownTCB↑[i] := t;
        t↑.MyDownElement := address(Temp↑.DownTCB↑[i]);
        F&A(Temp↑.#Created, 1);
    Weak_V(t↑.Lock);
    return t
end function CreateTaskFromTemplate;
```

5.3.2.4. Resume

A suspended task externally named *N* is resumed by changing its status to ready and placing the task onto the Ready Queue; this is also how a task created directly by a parent begins execution. Notice that like Suspend, Resume verifies the external name of *N*'s *TCB* after the *TCB* is locked to guard against the possibility that *N* terminated and then had its *TCB* allocated to a new task between Resume's table loop-up of *N*'s internal name and the locking of the *TCB*.

It is possible for a task that has been signaled for suspension to be resumed before it has had an opportunity to process its signal. In this case, the signal need only be cleared. The task will have no knowledge that such a signal was ever sent.

[28] *ProcessorID* returns the unique processor identifier for its PE.

```
procedure Resume (N : string);
var t : TCB_Pointer;
    t := TaskNamed(N);
    Weak_P(t↑.Lock);
        if t↑.Name = N then
            if t↑.Signals = ToBeSuspended then
                t↑.Signals := None
            else  only take action if already suspended
                if t↑.Status = Suspended then
                    t↑.Status := Ready;
                    Linked_QInsert(t, RQ[t↑.Priority])
                end if
            end if
        end if;
    Weak_V(t↑.Lock)
end procedure Resume;
```

5.3.2.5. Signal Suspend

A task *S* signals another task externally named *N* to suspend itself by first finding the latter's internal name *t* (its *TCB* location) in the Name Table. Then, after locking *t*'s *TCB*, *S* checks whether the task has terminated, or has already suspended itself, or has a signal pending. *S* also checks whether the external name of the *TCB* is still *N*. If it isn't, then between *S*'s look-up of *N* in the Name Table and locking of the *TCB*, *N* terminated and its *TCB* was allocated to a new task. If any one of the above conditions holds, *S* releases the *TCB* and returns. Otherwise, *t* is signaled to suspend itself. *S* takes no further steps if *t* is not running or has begun a critical transaction; later, when *t* runs or completes the last of its transactions, it will process the signal and suspend itself (change its status to suspended). On the other hand, if *t* is running and has not started any critical transactions, *S* sends an interrupt to *t*'s PE, in response to which *t* calls ProcessSignals.

```
procedure SignalSuspend (N : string);
var t : TCB_Pointer;
    t := TaskNamed(N);   look up internal name of N in Name Table
    Weak_P(t↑.Lock);
        if t↑.Name = N and t↑.Signals = None and
          not (t↑.Status = Terminated or t↑.Status = Suspended) then
            t↑.Signals := ToBeSuspended;
            if t↑.TransCnt = 0 and t↑.Status = Running then
                StopPE(t↑.PE_Id)  send interrupt to t's PE
            end if
        end if;
    Weak_V(t↑.Lock)
end procedure SignalSuspend;
```

5.3.2.6. Get Task

GetTask tests, in order of descending priority, the queue and multiqueue components of the Ready Queue until a nonempty queue or multiqueue is found.[29] A deletion is then performed on this component. (If the Ready Queue is empty, GetTask cycles until an item can be deleted). Due to task destruction (see sections 5.3.2.7 and 5.3.2.8) or preemption (see section 5.4.2.3.2), GetTask may delete a nil item from the Ready Queue. If it does, the item is discarded and GetTask returns to the Ready Queue for another item.

Before proceeding further to describe the actions of GetTask, recall from section 5.1 that multiqueue components of the Ready Queue can contain templates, queues tagged with multiplicity, or queue pointers tagged with multiplicity.[30] The latter two structures allow an entire queue *WQ* of *M* blocked tasks to be inserted (either directly or indirectly) as a single item with multiplicity *M* onto a multiqueue component $RMQ[P]$ of the Ready Queue, where *p* is the priority of *WQ*.[31] The priority of *WQ* can differ from the priorities of its tasks. Hence, one of GetTask's functions is to transfer those tasks with priority lower than *P* from *WQ* to the queue components of the Ready Queue that correspond to their priorities.

---

[29] Appendix A reviews an algorithm, due to Gottlieb et al. [80], for finding the first nonempty queue that is more efficient when the number of priorities is large.

[30] All three structures are defined in section 5.2.

[31] See sections 5.4.2.2.3, 5.4.2.3.2, 5.4.3.2.3, and 5.4.3.2.3.1 for details.

Suppose GetTask deletes a non-nil item I from a multiqueue component $RMQ[P]$. If I is a template, GetTask creates a task t from I (via CreateTaskFromTemplate) and executes t (as explained below). Otherwise, GetTask performs a deletion[32] from the queue associated with I to obtain an item t. If t is nil, again due to task destruction, GetTask discards it and returns to the Ready Queue for the next priority P item. Otherwise, t is a task. If t's priority p is lower than P, then GetTask inserts t onto $RQ[p]$ and again returns to the Ready Queue for the next priority P item. If t's priority is at least as high as P, t is executed as described below.

Suppose GetTask obtains a task t either from $RQ[P]$ or from a queue of blocked tasks that is a member of the Ready Queue, and t is of priority P or higher. GetTask changes t's status to running and sets t's PE identifier to that of the processor executing GetTask. A context-switch from *Current* to t is then performed via the Switch primitive.[33] This primitive lowers *Current*'s *Scheduling* flag[34] and modifies *Current*'s CPU state to enable *Current*'s task to resume execution (when it later runs) at the point following the call to Switch at the end of GetTask.

Observe that after GetTask performs a Ready Queue deletion of a task t, it waits until t's *Scheduling* flag is down before executing t. As previously noted in section 5.3.1.4, this is to prevent two GetTasks from using t's *TCB* system stack concurrently.

---

[32] Actually, as a queue Q of blocked tasks can be transferred to the Ready Queue before all of Q's tasks have completed their insertions, GetTask will cycle performing deletions until one returns with an item. See section 5.4.2.2.3 for details.

[33] As the details of Switch are highly machine dependent, its implementation is omitted and its function outlined only.

[34] Recall from section 5.3.1.4 that a task raises this flag prior to calling GetTask.

```
procedure GetTask;
var empty : boolean;
     j        : integer;
     q        : ↑Linked_Queue;
     t        : Pointer to TCB, TTCB, QPointerTaggedWithMultiplicity, or
                QTaggedWithMultiplicity;
    j := MaxPrio;  t := nil;
    repeat
        repeat until not Linked_QDelete(t, RQ[j]) or t ≠ nil;
        if t = nil then  RQ[j] is empty
            repeat  try multiqueue RMQ[j]
                empty := Linked_MQDelete(t, RMQ[j]);
                if not empty and t ≠ nil and type(t) ≠ TTCB then
                    if type(t) = QPointerTaggedWithMultiplicity then
                        q := t↑.QPtr
                    else  queue tagged with multiplicity
                        q := address(t↑.Q)
                    end if;
                    repeat until Linked_QDelete(t, q↑);
                    if t ≠ nil then  t is a task
                        if t↑.Priority < j then
                            Linked_QInsert(t, RQ[t↑.Priority];
                            t := nil
                        end if
                    end if
                end if;
            until empty or t ≠ nil
        end if;
        j := j+1 mod (MaxPrio+1)
    until t ≠ nil;
    if type(t) = TTCB then
        t := CreateTaskFromTemplate(t)
    else  t is a task
        repeat until not t↑.Scheduling;
        Weak_P(t↑.Lock);
            t↑.PE_Id := ProcessorID;
            t↑.Status := Running;
        Weak_V(t↑.Lock)
    end if;
    Switch(t)
end procedure GetTask;
```

5.3.2.7. Signal Destroy

A task externally named *N* is signaled to destroy itself via SignalDestroy. Once the task accepts the signal, it will initiate the destruction of its task tree. Notice that here, as in

SignalSuspend and Resume, SignalDestroy verifies the external name of *N*'s *TCB* after the *TCB* is locked to guard against the possibility that *N* terminated and then had its *TCB* allocated to a new task.

```
procedure SignalDestroy (N : string);
var t : TCB_Pointer;
    t := TaskNamed(N);
    Weak_P(t↑.Lock);
        if t↑.Name = N then Mark(t) end if;
    Weak_V(t↑.Lock)
end procedure SignalDestroy;
```

The procedure Mark, after ensuring that its task has neither received a destroy signal nor terminated, sets the task's *Signals* field and raises the task's priority to *MaxPrio* to speed it out of the system. In addition, if the task has not begun any critical transactions, other steps are taken, based on its status, to accelerate its recognition of the signal. Keep in mind while reading through the following cases that the *TCB* is locked while Mark executes. During this time the only other primitives that may access the *TCB* are GetTask and Unblock through queue deletions (which do not lock *TCB*s), and ChildTerminated, which decrements a task's *#Children* count (without locking the *TCB*).

(1) A running task *T*'s PE is interrupted by issuing a StopPE interrupt. In general, if the PE's interrupts are disabled (e.g., because the task is executing a primitive), the StopPE interrupt will be processed when they are enabled. However, if they are disabled and *T* is attempting to start a critical transaction or to perform a running⟶waiting transition, then the primitive *P* involved with this action will detect and process *T*'s destroy signal. Recall that the first two steps taken by *P* is to lock *T*'s *TCB* and then check for any pending signals. As Mark's caller has already locked the *TCB*, *P* cannot proceed until Mark returns and its caller unlocks the *TCB*. Thus, *T* can not enter a wait state, one from which it might never awaken, if its destroy signal is set.

(2) A ready task *T* is removed[35] from its current queue and inserted onto the highest priority one. If the removal fails, *T* has been deleted by a GetTask, which cannot update *T*'s

[35] Recall from chapter 2 that interior removal substitutes a nil item for an item that is removed from a queue to maintain the FIFO queue property. The operation fails if it does not remove the specified item.

status to running until Mark's caller unlocks *T*'s *TCB*. Regardless of whether Mark removes *T* from the Ready Queue, *T*'s destroy signal will be processed during its ready$\longrightarrow$running transition.[36]

(3) A blocked task *T* is removed from its wait queue and inserted onto the Ready Queue. If the removal fails, *T* has been deleted via an Unblock, which will update *T*'s status to ready and insert *T* onto the highest priority queue component of the Ready Queue once Mark's caller releases *T*'s *TCB*. In either case, as in (2), when *T* makes the ready$\longrightarrow$running transition, it will process the signal.

We note that a task *T* that is a member of a wait queue *WaitQ* that has been inserted onto the Ready Queue (as described in section 5.1) has a status of blocked even though it is ready to be run. It would be both costly and unnecessary to update the statuses of all tasks on *WaitQ* before inserting the latter onto the Ready Queue. To destroy *T*, it must be removed from *WaitQ*, not a queue or multiqueue component of the Ready Queue. By not updating *T*'s status, i.e., leaving it as blocked, we ensure that *T* will be removed from the correct queue.

(4) A task *T* waiting for its children to terminate is, if possible, prematurely awakened. Recall that, in general, *T*'s *#Children* counter equals how many children it has. This counter is decremented by one by each child of *T* when terminating and by *T* when undergoing a running$\longrightarrow$waiting-for-children transition via the WaitForChildren primitive. Hence, to prematurely awaken *T*, Mark increments the *#Children* count by 1 to, in effect, undo the WaitForChildren operation. If the count is not -1 when the increment is performed, the last child has not yet begun its termination, and mark inserts *T* onto the Ready Queue. Otherwise, the last child has begun its termination and is waiting to awaken *T*, which it will do once *T*'s *TCB* is released by Mark's caller. Regardless of which task awakens *T*, the destroy signal will be processed when *T* makes the ready$\longrightarrow$running transition.

Notice that when Mark processes a task *T* that has a status of waiting-for-children, the last child of *T* could not have already terminated: When *T* is waiting and the last child of *T* drives *#Children* to -1 (via the ChildTerminated primitive), the child then locks *T*'s *TCB*, changes *T*'s status to ready, sets *#Children* to zero, and inserts *T* onto the Ready

[36] Recall from section 5.3.1.2 that during this transition a task processes any pending signals.

Queue. As *T*'s *TCB* is locked by Mark's caller and has a status of waiting-for-children, the last child of *T* must still be executing.

(5) A suspended task is (prematurely) resumed. The destroy signal will be processed when the task makes the ready⟶running transition.

```
procedure Mark (T : TCB_Pointer);
var l      : ListElement_Pointer;
    oldp : integer;

    T's TCB is locked by caller
    if T↑.Signals = ToBeDestroyed or T↑.Status = Terminated then
        return
    end if;
    T↑.Signals := ToBeDestroyed;
    oldp := T↑.Priority;
    T↑.Priority := MaxPrio;
    if T↑.TransCnt > 0 then return end if;

    case T↑.Status of
    Running    : ( StopPE(T↑.PE_Id) )

    Ready      : ( if Linked_QRemove(T, RQ[T↑.oldp]) then
                       Linked_QInsert(T, RQ[MaxPrio])
                   end if)

    Blocked    : ( l := T↑.QEntry.List_Loc;
                   if l ≠ nil and Linked_QRemove(T, l↑.Queue_Id) then
                       T↑.Status := Ready;
                       Linked_QInsert(T, RQ[MaxPrio])
                   end if )

    WaitingForChildren
               : ( if F&A(T↑.#Children, 1) ≠ −1 then awaken T
                       T↑.Status := Ready;
                       Linked_QInsert(T, RQ[MaxPrio])
                   end if )

    Suspended : ( T↑.Status := Ready;
                   Linked_QInsert(T, RQ[MaxPrio]) )
    end case
end procedure Mark;
```

As mentioned in section 5.3.1.4, if SignalDestroy finds that a task's status and queue entry are inconsistent, it releases the *TCB*. This can happen when SignalDestroy locks a *TCB* that has a status of blocked or ready and before SignalDestroy (i.e., Mark) can remove the *TCB* from the queue, another task deletes it, causing the *TCB*'s status to be at odds with its queue entry: the entry is nil while the status is blocked or ready.

5.3.2.8. Destroy Tree

When a task *P* recognizes a destroy signal, it signals each of its immediate children to destroy themselves and then waits for them to terminate. The children do likewise, signaling their own offspring to destroy themselves and then waiting for them to terminate, etc. The destruction of the tree then starts at the root and descends towards the leaves, drawing more and more tasks into the activity.

Each template *t* of *P* that is on the Ready Queue is removed by *P* to prevent additional children from being created from it.[37] Recall that *t*'s initial multiplicity M_0 was added to *P*'s *#Children* count when the template was enqueued on the Ready Queue (see InitiateTemplate, section 5.3.2.2). Therefore, having intercepted *t* before all of its tasks are created, *P* decrements its *#Children* count by *t*'s remaining multiplicity *M* to adjust the count for those tasks not created (or equivalently, to simulate their termination). Otherwise, *P* would wait forever for nonexistent children to terminate (see Terminate and WaitForChildren, sections 5.3.2.12 and 5.3.2.10, respectively). The total number of tasks created from *t* then equals $M_0 - M$. *P* waits until *t*'s *#Created* count equals this value, and then signals each task in *t*'s *DownTCB* array to destroy itself.

Tasks that *P* has created directly, i.e., those in the *DC* queue, are signaled in a similar manner to that described above. There will be nil entries in the queue for any of these tasks that have already terminated.

[37] This is done via the multiqueue primitive LowerMult. Recall from chapter 3 that MQLowerMult(*I*, *MaxMult*, *MQ*, *M*) returns true if item *I* is removed from multiqueue *MQ* (and false otherwise) and sets *M* to *I*'s multiplicity at the time of the removal.

```
procedure DestroyTree;
var temp   : TTCB_Pointer;
    child  : TCB_Pointer;
    j, k, m : integer;

    mark indirectly created tasks, i.e., those formed from templates
    temp := Current↑.Progeny.IC;
    while temp ≠ nil do
        if temp↑.Status = Ready then
            if MQLowerMult(temp, MaxMult, RMQ[temp↑.Priority], m) then
                F&A(Current↑.#Children, –m)    m tasks not created
            end if;
            k := temp↑.ResourceMQEntry.Mult – m;  k tasks to be created
            repeat until k = temp↑.#Created;
            for j := 0 to k–1 do  process temp's DownTCB array
                child := temp↑.DownTCB↑[j];
                if child ≠ nil then  child has not yet terminated
                    Weak_P(child↑.Lock);
                        if child↑.Parent = Current then
                            Mark(child)
                        end if;
                    Weak_V(child↑.Lock)
                end if
            end for
        end if;
        temp := temp↑.NextTemplate
    end while;

    mark tasks created directly by parent
    while Linked_QDelete(child, Current↑.Parent↑.Progeny.DC) do
        if child ≠ nil then
            Weak_P(child↑.Lock);
                if child↑.Parent = Current then
                    Mark(child)
                end if;
            Weak_V(child↑.Lock)
        end if
    end while;
    Terminate
end procedure DestroyTree;
```

Note in the procedure above that after *P* locks a child *C*'s *TCB*, *P* checks that it is still *C*'s parent before proceeding. This is necessary since *C* can terminate and have its *TCB* allocated to another task between the time *P* inspects the *DownTCB* array or *DC* queue and locks the *TCB*.

5.3.2.9. Process Signals

When a task performs a ready⟶running transition following a suspension or a wait for its children to terminate, or when its PE is interrupted, it processes any pending signals (subject to our critical transaction constraint). The task kills itself if it has received a destroy signal. If it has received a suspend signal, it marks itself as suspended, clears the signal, and reschedules its PE. When a suspended task is resumed, it processes any new signals that arrived between its suspension and resumption.

```
procedure ProcessSignals;
L:   Weak_P(Current↑.Lock);
         if Current↑.TransCnt = 0 then
             if Current↑.Signals = ToBeDestroyed then
                 Weak_V(Current↑.Lock);
                 DestroyTree
             elseif Current↑.Signals = ToBeSuspended then
                 Current↑.Status := Suspended;
                 Current↑.Signals := None;
                 Current↑.Scheduling := true;
                 Weak_V(Current↑.Lock);
                 GetTask;
                 go to L  check for new signals
             end if
         end if
end procedure ProcessSignals;
```

5.3.2.10. Wait for Children

WaitForChildren coordinates a parent with its children. Since this primitive can cause a running⟶waiting transition, it first ensures that the parent has not received a destroy signal.

A parent's *#Children* count equals the number of children it has. As each child terminates, it decrements this count by one. When the parent wishes to delay its execution until all of its offspring have terminated, it too decrements *#Children* by one. If the parent drives the count to −1, all of the children have terminated, so the parent continues; otherwise, the parent's status becomes waiting-for-children, and its PE is rescheduled.[38] When the parent eventually runs, it

[38] Brinch Hansen [73b] performed this operation by having the parent inspect the *#Children* counter and set a flag to true if it was waiting. The counter and flag had to be exclusively accessed by parents and their offspring. Rudolph [82] encoded the function of the flag into the range of values assumed by the counter and eliminated the need for mutual exclusion by adjusting the counter via fetch-and-adds.

checks for any new signals that may have arrived while it was waiting.

```
procedure WaitForChildren;

    Weak_P(Current↑.Lock);
        if Current↑.Signals = ToBeDestroyed and Current↑.TransCnt = 0 then
            Weak_V(Current↑.Lock);
            DestroyTree
        elseif F&A(Current↑.#Children, −1) = 0 then  children done
            Current↑.#Children := 0;
            Weak_V(Current↑.Lock)
        else
            Current↑.Status := WaitingForChildren;
            Current↑.Scheduling := true;
            Weak_V(Current↑.Lock);
            GetTask;
            ProcessSignals
        end if
end procedure WaitForChildren;
```

5.3.2.11. Child Terminated

As stated above, as each task terminates, it lowers its parent's *#Children* count by one. If a task lowers *#Children* to −1, it is the *last* child to terminate and its parent is already waiting for the children. This child awakens its parent by placing it onto the Ready Queue and reinitializing *#Children* to zero.[39] If all the children terminate before the parent waits, there is no *last* child.

A parent's offspring modify its *#Children* count without locking the *TCB*, permitting multiple offspring to terminate concurrently. The parent, however, locks its *TCB* (in WaitForChildren) before decrementing *#Children* to preclude coordination problems between itself and the last terminating child. If this were not done, the last child could lock the *TCB* before its parent and would then have to release the *TCB* and cycle until the parent had entered its waiting-for-children state.

---

[39] An alternative scheduling policy is to have the last terminating child yield its PE to its parent (Brinch Hansen [73b]).

```
procedure ChildTerminated;
    if F&A(Current↑.Parent↑.#Children, –1) = 0 then
        Weak_P(Current↑.Parent↑.Lock);
            Current↑.Parent↑.Status := Ready;
            Current↑.Parent↑.#Children := 0;
            Linked_QInsert(Current↑.Parent, RQ[Current↑.Parent↑.Priority]);
        Weak_V(Current↑.Parent↑.Lock)
    end if
end procedure ChildTerminated;
```

5.3.2.12. Terminate

When a task wishes to terminate, it first waits until all of its children have terminated[40] and then deletes its own name from the Name Table, severs its connection with its parent (by removing itself from its parent's *IC* or *DC* component, depending upon how it was created), and notifies its parent that it has terminated. Resources acquired by the task are then released (e.g., templates, private memory, etc.), and its *TCB* is marked terminated and returned to the *FreeTCB* queue. Finally, the task's PE is rescheduled.

```
procedure Terminate;
    WaitForChildren;
    EraseName(Current);  erase name from Name Table
    if Current↑.MyDownElement ≠ nil then  created from a template
        Current↑.MyDownElement↑ := nil    remove TCB from template's DownTCB
    else    directly created, remove entry from DC queue
        Linked_QRemove(address(Current↑.SiblingQEntry), Current↑.Parent↑.Progeny.DC)
    end if;
    ChildTerminated;
    DeallocateOwnedResources;   e.g., templates, local memory, etc.
    Weak_P(Current↑.Lock);
        Current↑.Status := Terminated;
        Current↑.Scheduling := true;
        Linked_QInsert(Current, FreeTCB);
    Weak_V(Current↑.Lock);
    GetTask
end procedure Terminate;
```

[40] The task can force the children to terminate rapidly by destroying them.

5.3.2.13. Start Critical Transaction

The StartCriticalTransaction primitive raises a task's critical transaction count and returns true if either the task has not received a destroy signal or a critical transaction is under way. Otherwise, the primitive returns false and does not increment the count. In the latter case, it is assumed that the task performs any housekeeping operations that are necessary to ensure that other tasks' critical transactions can proceed and then destroys itself. An example of such a housekeeping chore is the release of a resource as illustrated in Figure 5-2b.

```
function StartCriticalTransaction;

    Weak_P(Current↑.Lock);
        if Current↑.TransCnt = 0 and Current↑.Signals = ToBeDestroyed then
            StartCriticalTransaction := false
        else
            F&A(Current↑.TransCnt, 1);
            StartCriticalTransaction := true
        end if;
    Weak_V(Current↑.Lock)
end function StartCriticalTransaction;
```

5.3.2.14. Complete Critical Transaction

When a task completes a critical transaction, it lowers its transaction count. When the last transaction is finished, the task processes any pending signals.

```
procedure CompleteCriticalTransaction;

    Weak_P(Current↑.Lock);
        if F&A(Current↑.TransCnt, −1) = 1 then
            Weak_V(Current↑.Lock);
            ProcessSignals
        else
            Weak_V(Current↑.Lock)
        end if
end procedure CompleteCriticalTransaction;
```

5.3.2.15. Block

A task *T* blocks by adding itself to a resource queue (other than the Ready Queue), permitting its PE to execute another task. Since this is a running ⟶ waiting transition, the block operation will not proceed if *T* has received a destroy signal (and no critical transactions have been started). Instead, a nil item will be added to the queue in place of *T*.[41]

```
procedure Block (Q : Linked_Queue);
    Weak_P(Current↑.Lock);
        if Current↑.TransCnt = 0 and Current↑.Signals = ToBeDestroyed then
            Weak_V(Current↑.Lock);
            InsertNilItem(Q);
            DestroyTree
        else
            Current↑.Status := Blocked;
            Current↑.Scheduling := true;
            Linked_QInsert(Current, Q);
            Weak_V(Current↑.Lock);
            GetTask
        end if
end procedure Block;
```

A task that was blocked does not immediately process its signals upon returning from GetTask, unlike a task that had been suspended or waiting for its children to terminate (cf., ProcessSignals and WaitForChildren, sections 5.3.2.9 and 5.3.2.10). As illustrated in Figure 5-2b, it may be necessary for a task that was blocked to process its signals in the environment that led to its block, e.g., to allow it to release a resource. Therefore, it is assumed that a blocked task's signals will be processed after the task returns from Block.

5.3.2.16. Unblock

Unblock deletes an item from a resource wait queue and inserts it onto the Ready Queue, unless the item is nil (representing a removed task), in which case no insertion is done.[42] Unblock returns true if it deletes an item and false otherwise. Unblock also identifies (via its first parameter) the item that is deleted from the queue to permit its caller to respond, if necessary, to the ''unblocking'' of a nil item (e.g., see section 5.4.1.2).

---

[41] This is done by executing the standard queue insertion procedure that supports interior removal (see chapter 2), but, instead of inserting a *TCB* onto the rear list *L* of the queue, the missing item count of the last item on *L* is incremented by one.

[42] Block, SignalDestroy, and DestroyTree can place a nil item onto a wait queue.

```
function Unblock (T : TCB_Pointer;
                  Q : Linked_Queue);
    Unblock := Linked_QDelete(T, Q);
    if Unblock then
        if T ≠ nil then
            Weak_P(T↑.Lock);
                T↑.Status := Ready;
                Linked_QInsert(T, RQ[T↑.Priority]);
            Weak_V(T↑.Lock)
        end if
    end if
end function Unblock;
```

5.3.3. A Comparison of Serial and Parallel Implementations

In the preceding section parallel implementations for creating and destroying tasks have been given. We will now compare these implementations with serial ones. A coarse measurement of the performances of the various implementations suggests that, indeed, the parallel variants should, in general, be chosen over the serial ones.

5.3.3.1. Creation

Direct and indirect methods for creating subtasks were given in section 5.3.1.1. The former takes a straightforward, serial approach to implementing this activity whereas the latter is (potentially) more parallel due to its use of a multiqueue data structure. In Figure 5-4, steps D1–D2 and I1–I2 list the task primitives that are used by each method. These steps describe only the work that is required to place tasks or templates onto the Ready Queue. However, to properly compare the two methods, the actions taken by PEs to execute these tasks or to create tasks from these templates and then execute them should be considered as well. Therefore, steps D3 and I3 list the primitives that are necessary for the tasks to begin execution. The major queue and/or memory allocation operations performed by each primitive are shown as well.

For each method we compute the minimum total time for N tasks to be created and to begin execution when one and N PEs are available (corresponding to heavy and light system loads). We make the following simplifying assumptions: N is not greater than *#PEs*, the total number of processors; each queue/multiqueue/memory allocation operation is done in unit time; these operations are assumed to be the dominant components of each primitive shown in Figure 5-4; for our

Direct Method	Indirect Method
(D1) Create a *TCB T* – delete *T* from *FreeTCB* queue – insert *T* onto *DC* queue	(I1) Create a *TTCB Temp* – delete *Temp* from *FreeTTCB* queue
(D2) Resume *T* – insert *T* onto Ready Queue	(I2) Initiate *Temp* with Multiplicity *N* – allocate a *DownTCB* array of size *N* – insert *Temp* with multiplicity *N* onto Ready Queue
(D3) Get Task – delete *T* from Ready Queue	(I3) Get Task – delete *Temp* from Ready Queue – delete a *TCB* from *FreeTCB* queue[43]

Task Creation and Execution

Figure 5-4.

comparisons the time to execute tasks is ignored.

For the direct method the creation and initiation of N subtasks requires $4N$ time units when one PE is available: each step (D1 through D3) is performed serially N times. When N PEs are available, this same activity can be done in $1+3N$ time units: after a parent executes D1 and D2, another task executes D3 at the same time that the parent begins its next execution of D1. Thus, $3N$ time units are required for the parent to create and enqueue the children, and one additional time unit is needed for the Nth child to be deleted once the parent is done.

For the indirect method, the minimum time to create and initiate N subtasks is $3+2N$ time units when one PE is available: as above, each of the first two steps is performed once, and the last step is performed N times. When N PEs are available, the minimum time is $4+\lceil N/\min(Qsize, \#PEs)\rceil$, where *Qsize* is the number of lists comprising the *FreeTCB* queue. The first two steps, I1 and I2, are done in 3 time units. The third step requires $1+\lceil N/\min(Qsize, \#PEs)\rceil$ time units: a template with multiplicity N can be concurrently deleted by N tasks in unit time, and a concurrently accessible queue consisting of *Qsize* serially accessible lists can support M concurrent deletions, where M is the minimum of *Qsize* and *#PEs*.

[43] GetTask allocates a *TCB* via CreateTaskFromTemplate.

The timings for both methods are summarized in Table 5-1. Assuming that the queue and memory allocation operations can be used to estimate each method's performance, the direct method should be selected when the size of the creation is relatively small; otherwise, the indirect method should be chosen.

Minimum Total Time to Create and Initiate N Tasks*		
Method	Number of Available PEs	
	1	N
Direct	$4N$	$1+3N$
Indirect	$3+2N$	$4+\lceil N/\min(Qsize, \#PEs)\rceil$

* (measured in units of queue, multiqueue and memory allocation operations)

Table 5-1.

5.3.3.2. Destruction

We have given a method for destroying a task and its descendants in which subtrees of the task tree are destroyed in parallel. Let us consider an alternative serial strategy: Let a task *D* destroy a task tree by visiting (e.g., using a depth-first traversal) and signaling each node to kill itself. Unlike the parallel method, a node *N*'s offspring are signaled by *D*, not *N*.

In general, the parallel method is faster than this serial version. However, there is at least one scenario (albeit somewhat contrived) where the latter may be faster than the former. Suppose an internal node *N* of the tree is in a wait state (i.e., its corresponding task has a status of blocked, waiting-for-children, or suspended). The parallel method marks *N* and places *N*'s corresponding task *n* onto the Ready Queue. During the time that *n* resides on the Ready Queue, its children create tasks, which in turn create other tasks. Thus, when *n* finally runs, its subtree has increased in size between the times of its Ready Queue insertion and deletion. A task *D* using the serial method above destroys *n* and its descendants first by signaling *n* and then by proceeding immediately to mark *n*'s children (and their subtrees). *D* prevents *n*'s children from creating new tasks. It processes fewer nodes than *n* does under the parallel method. For this scenario then the serial method can be faster than the parallel one.

We could incorporate part of the serial method into the parallel one. Namely, when a task *T* found that a child *C* was blocked, *T* could destroy *C*'s children, instead of waiting for *C* to do it. To measure the impact of this modification on the complexity of the parallel method, we would have to consider what action *C* would take after it was unblocked: terminate, aid *T* in the destruction of the children, or take over the destruction of the children from *T*.

5.4. Resource Primitives

The Ultracomputer implementations of Gottlieb et al. [83b] for semaphore and readers/writers coordination operations require a task to busy-wait (cycle on its PE) if its resource request cannot be granted immediately. Idling PEs in this manner for extensive periods of time is wasteful when there are tasks waiting to be run. In such cases higher processor utilization can be achieved by using block-queue synchronization implementations in place of those based on busy-waiting. In these implementations a task denied access to a resource is inserted onto a wait queue associated with the resource and its PE is then switched to another task. When the resource is released, the first task on this queue is deleted and transferred to the Ready Queue to be run. Thus, tasks (and their PEs) do not cycle waiting for resources. However, a block-queue implementation may surreptitiously introduce another form of busy-waiting. Specifically, concurrent resource requests and releases can lead to congestion in accessing a wait queue. This is prevented by using concurrently accessible queues, where the operations of insertion, deletion, and interior access are serialization-free. Resource requests and releases that access these queues will then be serialization-free as well.

Block-queue implementations of semaphore, readers/writers, and event operations, based on concurrently accessible wait queues, as well as a weak implementation of event primitives, are covered in the following sections, The block-queue implementations allow blocked tasks to be destroyed (i.e., removed from wait queues) without serializing resource requests or releases. Also, two parallel methods for unblocking tasks from reader/writer and event wait queues are given. The first one utilizes recently unblocked tasks to aid in the unblocking of other tasks. Specifically, after a task is unblocked from a wait queue, it deletes other tasks from the wait queue and inserts them onto the Ready Queue. The second method unblocks tasks from a wait queue in constant time by transferring the entire wait queue to the Ready Queue as a single object: the wait queue is treated as a single item with multiplicity and inserted onto a multiqueue component of the Ready Queue. These two methods are compared in section 5.4.4.

5.4.1. Semaphores

After reviewing Rudolph's [82] block-queue implementation of the P/V semaphore operators, a modified implementation that supports the destruction of blocked tasks is given. A version of P is also presented that permits a task to specify whether it wishes to wait for or to continue to execute without a resource that is currently unavailable.

5.4.1.1. A Review of Block-queue P/V

A block-queue counting semaphore *S* associated with a resource *R* is represented by a wait queue *WaitQ* and an integer value *Val*. Initially, *WaitQ* is empty and *Val* equals the number of available units of *R*. For example, when *R* is a hardware resource such as tape drives, *Val* equals the number of unallocated drives, and when *R* is a software resource such as a critical code section, *Val* is one as only one ''unit'' of *R* may be used at a time.

The block-queue semaphore operators P and V function as follows: A task *A* executing P(*S*) decrements *Val* by one and accesses *R* if *Val*'s predecrement value was positive. Otherwise, *A* is blocked and is inserted onto the wait queue of *S*. When a task *B* performs a V(*S*), *Val* is incremented by one. If *Val*'s preincrement value was nonnegative, the V is complete since no tasks are waiting to use *R*. Otherwise, *B* unblocks the first task on *S*'s wait queue (i.e., a task is deleted from the wait queue and inserted onto the Ready Queue). The Unblock operation may have to be tried a (bounded) number of times before its queue deletion is successful since *B* can execute its V between another task's decrement of *Val* and insertion onto the wait queue.

If *Val* has an initial value of $i \geq 0$, then after *p* P operations and *v* V operations,

$$Val = i - p + v.$$

Furthermore, the number of tasks on the wait queue equals $-Val$ or zero, whichever one is greater.

The block-queue P and V operations are given below, where the Block and Unblock primitives are as defined in sections 5.3.2.15 and 5.3.2.16. In this initial implementation it is assumed that tasks on the wait queue can not be destroyed. Consequently, the queue will never contain nil items.

```
BQ_Semaphore = record   Block-queue Semaphore
                   Val          : integer;
                   WaitQ        : Linked_Queue
               end;

Initially:
   Val ≥ 0;
   WaitQ = ∅;

procedure BQ_P (S : BQ_Semaphore);
   if F&A(S.Val, –1) < 1 then
      Block(S.WaitQ)
   end if
end procedure BQ_P;

procedure BQ_V (S : BQ_Semaphore);
var t : TCB_Pointer;
   if F&A(S.Val, 1) < 0 then
      repeat until  Unblock(t, S.WaitQ)
   end if
end procedure BQ_V;
```

The use of a resource (e.g., a critical code section) that is managed with semaphore operations can be designated a critical transaction by following either implementation of Figure 5-2: the figure's generic request and release operations need only be replaced with P and V operations on the semaphore.

5.4.1.2. Destroying a Blocked Task

Suppose a task *T* is on a semaphore wait queue when another task *D* executes SignalDestroy(*T*). *D* signals *T* to destroy itself, and, assuming that *T* has not begun any critical transactions, *D* attempts to take *T* off of the wait queue using the remove queue primitive. If the removal succeeds, a nil item is left in *T*'s place on the queue; if it fails, *T* has already been deleted by a V.[44]

When a V operation deletes a nil item from a wait queue, it responds by discarding the item and performing another V to simulate the release of the resource by the previously destroyed task.

[44] *T* could not have been removed by another task executing SignalDestroy(*T*) since the removal performed by SignalDestroy may only be attempted once, by the task that marks *T* to destroy itself.

Thus, a V operation cycles until the wait queue is emptied or a task is found. Only V is affected by the destruction of blocked tasks; P remains unchanged.

```
procedure BQ_V (S : BQ_Semaphore);
var t : TCB_Pointer;
    repeat
        if F&A(S.Val, 1) < 0 then
            repeat until Unblock(t, S.WaitQ)
        else
            return   WaitQ is empty
        end if
    until t ≠ nil
end procedure BQ_V;
```

Notice that when a task blocked on *S* is destroyed, *S*'s value *Val* is not altered. Thus, when *Val* is negative, its absolute value equals the total number of nil plus non-nil items on the wait queue.

In the above implementation, Vs are responsible for performing resource releases for destroyed tasks to maintain the correct value of a semaphore. Let us consider an alternative method (suggested and implemented by Edler [84]) that shifts the onus of this maintenance to the SignalDestroy primitive. This frees Vs from having to do additional releases, but requires V and SignalDestroy operations to be mutually exclusive.

To illustrate the need for mutual exclusion, we first present simple but flawed implementations for these operations, designated Flawed_SignalDestroy and Flawed_V, that attempt to carry out the alternative method. Flawed_SignalDestroy differs from our previous SignalDestroy only in that upon removing a task *T* from a semaphore *S*'s wait queue, it increments *S*'s value by one to undo *T's* request via

L: **if** Remove(*T, S.WaitQ*) **then** F&A(*S.Val*, 1) **end if**.

The result of this modification to SignalDestroy is that *Val* equals the number of tasks on the queue. Flawed_V then cycles until a task is deleted or *Val* becomes positive:

```
procedure Flawed_V (S : BQ_Semaphore);
var t : TCB_Pointer;
    if F&A(S.Val, 1) < 0 then
        repeat
            if S.Val > 0 then return end if    WaitQ is empty
        until Unblock(t, S.WaitQ) and t ≠ nil
    end if
end procedure Flawed_V;
```

Unfortunately, these versions of V and SignalDestroy can bring about unexpected and unwanted results. Specifically, they permit a resource to be overallocated, e.g., a critical code section could be concurrently executed. As an illustration of this, consider the following sequence of operations on a binary semaphore *S*, whose value *Val* is initially one.

(1) Tasks *A* and *B* execute P(*S*), lowering *Val* to −1. *A* runs and *B* blocks.

(2) *A* then begins a Flawed_V(*S*). Its fetch-and-add makes *Val* zero. *A*'s deletion of a task from the wait queue (via Unblock) is then delayed long enough for the next two steps to occur.

(2.1) Another task *C* signals *B* to destroy itself. *C* removes *B* from the wait queue and increments *Val* by one (via statement *L*).

(2.2) Two more tasks, *D* and *E*, perform Ps on *S*, lowering *Val* once more to −1. *D* runs and *E* blocks.

(3) *A* now executes its deletion (via Unblock) and obtains a nil item (representing the destroyed task *B*) from the wait queue. *A* then performs a second deletion (since *Val* is still negative) and succeeds in unblocking *E*, which then runs. At this point, two tasks, *D* and *E*, have acquired the critical resource. Therefore, our initial implementation of this method is indeed faulty.

The flaw in the implementation is that the signaling of a blocked task, specifically its removal from the wait queue and the subsequent modification of the semaphore's value, can be interwoven with the execution of a V. A correct implementation of this method is obtained by requiring Flawed_SignalDestroy (or merely statement *L*) and the Flawed_V procedure to be mutually exclusive. More precisely, either multiple Flawed_SignalDestroys or multiple Flawed_Vs can be executed in parallel, but a Flawed_SignalDestroy and a Flawed_V must be executed serially.[45]

Although correct, this implementation is inferior to the original presented at the beginning of this section since the latter is serialization-free.

5.4.1.3. P with Optional Blocking

The P/V implementation presented in the preceding section is easily extended to give a task the option of blocking within P or immediately exiting from P (allowing the task to perform another activity) when the resource in question is unavailable. Specifically, P returns true if it has obtained the resource and false otherwise. The SignalDestroy and V primitives are unaffected by this modification to P.

As before, a task *T* determines if a semaphore *S* is closed by decrementing *S*'s value. However, if *T* does not wish to be blocked and *S* is closed, a nil item *N* is inserted onto the wait queue in *T*'s place. To the V operation that deletes *N*, it will appear as if a task had been inserted onto the queue and then destroyed, so V will release the resource on *N*'s behalf and delete the next task from the queue.

```
function BQ_P (S         : BQ_Semaphore;
               Blockable : boolean);
    BQ_P := true;
    if F&A(S.Val, –1) < 1 then        S is closed
        if Blockable then             task can be blocked
            Block(S.WaitQ)
        else
            InsertNilTask(S.WaitQ);   place nil task on queue
            BQ_P := false
        end if
    end if
end function BQ_P;
```

Alternatively, P could re-increment *Val* (by one) if the resource was already in use and P was not to block. This implementation is more complex and requires P and V to be mutually exclusive (just as V and SignalDestroy would have to be if the latter had to modify a semaphore's

[45] These two primitives are an example of a variation on the readers/writers problem (dubbed the readers/readers problem by Edler [84]) in which two groups of operations (or tasks) are mutually exclusive, but operations from either group can be executed concurrently. A solution to the readers/readers problem containing serial sections appears in Brinch Hansen [73b]. A weak (i.e., busy-waiting) solution to this problem in which the activity of one reader group is fully parallel in the absence of readers from the other group has also been developed by Freudenthal [87].

value). If they were not, a V that incremented *Val* between P's decrement and re-increment would then expect to find a task on the wait queue, one which might never appear.

5.4.2. Readers/Writers Coordination

In this section, fair and writer priority readers/writers block-queue implementations are given. Both implementations, like their weak counterparts, are serialization-free in the absence of writer activity. Also presented are methods that allow a group of blocked readers to be unblocked concurrently.

After reviewing the overall mechanics of the block-queue implementations, the fair version is given and then extended to derive the more complex writer priority version. The granularity of parallelism in both implementations is very fine, with many combinations of interleaving between reader and writer operations being possible. In light of this, our explanations of the implementation is descriptive and informal. Chapter 7 conducts a more rigorous examination of a simplified version of the fair implementation. For the implementation we prove that readers and writers are mutually exclusive and free from deadlock, and that multiple readers can use the resource at the same time.

5.4.2.1. Overview

The readers/writers problem deals with two types of tasks, readers and writers, sharing a resource that may be concurrently accessed by either an arbitrary number of readers or by a single writer. Readers and writers *request*, *use*, and then *release* the resource. We shall refer to tasks as follows: a task that has requested but not yet released the resource is an *operating* task; a task that is using the resource is a *running* task; a task whose resource request has been denied and not yet granted is a *blocked* or *waiting* task.[46] When a reader requests the resource and there are no operating writers, the request is granted and the reader uses the resource; otherwise, the request is denied and the reader is inserted onto a *read wait queue*. Similarly, when a writer requests the resource and there are no operating tasks, the request is granted and the writer uses the resource; otherwise, the request is denied and the writer is inserted onto a *write wait queue*. When a reader releases the resource, it unblocks (grants the resource to) a waiting writer if one exists (thus

[46] These task states will be more rigorously defined in chapter 7.

making the writer a running writer). On the other hand, when a writer releases the resource, it unblocks (grants the resource to) all waiting readers in the fair implementation but preferentially unblocks (grants the resource to) a waiting writer in the writer priority implementation. Observe that if a single wait queue were used to hold both waiting readers and writers, then a task that released the resource would have to *search* the queue for readers or writers, depending upon which type of task was supposed to use the resource next.

For each implementation, two parallel methods for unblocking a set *G* of *M* readers are given. For the first method, a writer unblocks the *M* tasks using the standard Unblock primitive, i.e., by deleting them from the wait queue and then inserting them onto the Ready Queue. This activity is accelerated by having readers that belonged to *G* aid the writer in unblocking the remaining tasks in *G*.[47] The method's effectiveness depends upon the size of *M* and the availability of PEs to run the recently unblocked readers. In the second method, a writer unblocks *G*'s tasks in constant time by transferring the read wait queue to the Ready Queue as a single object: the read wait queue is treated as an item with multiplicity *M* and inserted onto a multiqueue component of the Ready Queue. These two methods are compared in section 5.4.4.

In the following implementations, note that the resource request and release operations are serialization-free. In contrast, conventional readers/writers implementations require these operations to be mutually exclusive (e.g., see Brinch Hansen [73b]).

Lastly, the use of a resource by a reader or a writer can be designated a critical transaction by following either one of the implementations given in Figure 5-2.

5.4.2.2. A Fair Implementation

Readers and writers request and release a resource by performing fetch-and-adds on a shared activity counter *Val* associated with the resource.[48] The implementation assumes that the maximum number of readers is less than the constant *MaxR*. We shall show that *Val* equals $\#r + \#w * MaxR$, where $\#r$ and $\#w$ are, respectively, the number of operating readers and writers.

---

[47] An implementation of this strategy was given by Edler [84]; however, the implementations in this section have the advantage of being serialization-free.

[48] Operations on this counter are similar to those performed on the activity counter used in the weak implementation of Gottlieb et al. [83b].

Initially, *Val* is zero. Readers and writers request the resource by incrementing *Val* by one and *MaxR*, respectively. A reader becomes a running reader (uses the resource) if there are no operating writers when its request is made, i.e., if *Val* is less than *MaxR*. A writer becomes a running writer if there are no operating tasks at the time of its request, i.e., if *Val* equals zero. If a task's request is denied, it blocks. Readers and writers release the resource by decrementing *Val* by one and *MaxR*, respectively. If *rr* readers are using the resource and writers are blocked, then no new readers (or writers for that matter) can use the resource until a blocked writer is granted the resource. In this case, the *last* of the *rr* readers to release the resource unblocks a writer. Notice that there is no last reader if there are no operating writers when readers release the resource. When a running writer releases the resource, all blocked readers are run; if there are none, then the first blocked writer is run. Thus, the implementation is fair since readers unblock waiting writers and writers unblock waiting readers in preference to waiting writers, and the presence of a waiting writer (reader) forces subsequent readers (writers) that request the resource to block, which prevents readers (writers) from starving writers (readers).

A major complexity in the implementation is determining which running reader is last when a writer is waiting to use the resource. For example, suppose there are *rr* running readers and no blocked writers: $Val = rr$. A writer then requests the resource, making $Val = MaxR + rr$, and blocks. Each running reader *R* releases the resource via

$$i_R := \text{F\&A}(Val, -1),$$

where i_R is local to reader *R*. For each *R*, i_R is greater than *MaxR* since there is an operating writer, which is blocked. We could consider the reader *L* that lowers *Val* to *MaxR* (i_L = $MaxR + 1$) to be last and have it unblock the writer. Unfortunately, between the writer's request and *L*'s release, another reader could request the resource, increment *Val* by one ($Val = MaxR + 2$), and block. *L* would then fail to unblock the writer since its release would indicate that there were other running readers (i_L would equal $MaxR + 2$, not $MaxR + 1$). All future requests would also block (as $Val > MaxR$), and no task would use the resource.

The problem above is that *Val* reflects the *current* state of the resource with respect to read and write requests; it does not indicate how many tasks are running, i.e., using the resource. Therefore, a second counter *Cnt* is introduced to identify the last running reader. When there are no waiting writers or there is a running writer, *Cnt* equals zero (which is therefore its initial value).

Let us return to the previous example to illustrate how *Cnt* permits the last running reader to be identified. Initially there are *rr* running readers and no operating writers: $Cnt = 0$ and $Val = rr$. A writer *W* arrives and requests the resource via

$$v := \text{F\&A}(Val,\ MaxR),$$

where *v* is local to *W*. Then, $Val = MaxR + rr$ and $v = rr$. Since *v* lies between 0 and *MaxR*, the writer determines that the resource is being used by *v* running readers. *W* therefore increments *Cnt* by $MaxR - v$. Each running reader *R*, upon releasing the resource, detects that there is an operating writer since i_R is greater than *MaxR* (indicating that *Val* is greater than or equal to *MaxR*). In response, *R* increments *Cnt* by one. Thus, there are *rr* increments by 1, performed by the running readers, as well as one increment by $MaxR - v$, performed by *W*. After all of these increments have been performed, $Cnt = rr + MaxR - v = MaxR$. If *W* drives *Cnt* to *MaxR*, it reinitializes *Cnt* to zero and accesses the resource since all of the *rr* readers have terminated. If not, *W* blocks and the last running reader, i.e., the one driving *Cnt* to *MaxR*, reinitializes *Cnt* to zero and unblocks a writer. It is important to recall that any readers (or writers) that request the resource while *W* is an operating writer will block since $Val \geq MaxR$. Thus, no new running readers will be introduced into the system until a writer is run, preventing writers from starving.

Next, consider when a running writer *W* releases the resource by executing

$$v := \text{F\&A}(Val,\ -MaxR).$$

Just prior to *W*'s release

$$Val = br + (bw + 1) * MaxR,$$

where *br* and *bw* are the number of blocked readers and writers, respectively. If there are blocked readers and no blocked writers ($MaxR < v < 2 * MaxR$) when *W* releases the resource, all *br* readers will be run (i.e., the read wait queue will be emptied), and the resource requests of incoming readers will be granted until the next write request is issued. If there are blocked writers and no blocked readers ($v \geq 2 * MaxR$ and *v* modulo *MaxR* = 0) when *W* releases the resource, one blocked writer will be run. Finally, if there are both blocked readers and blocked writers ($v > 2 * MaxR$ and *v* modulo $MaxR \neq 0$), *W* will initialize *Cnt* to $MaxR - br$ (where $br = v$ modulo *MaxR*, the number of readers that will run) and then unblock *br* readers. *Cnt* is initialized to this value to enable the last running reader to be identified. It is as if one of the *bw* blocked writers, say *X*, had requested the resource when there were *br* running readers and no operating writers, and then had incremented *Cnt* by $MaxR - br$ before any one of the *br* readers had incremented it by one, causing *X* to block.

Let us summarize the various references to *Cnt*. The first writer to request the resource when there are *rr* running readers and no operating writers increments *Cnt* by $MaxR{-}rr$. A writer initializes *Cnt* to $MaxR{-}br$, where $br = v$ modulo *MaxR*, if it releases the resource and finds there are blocked writers and *br* blocked readers. A running reader increments *Cnt* only if there is an operating writer when it releases the resource. If the last running reader releases the resource after the first writer to request it has blocked, the reader will initialize *Cnt* to zero and then unblock a writer.

In our implementations, tasks that request the resource and block will not necessarily be unblocked in first-come, first-served order (FCFS) relative to the time their resource requests were made. Instead, they will be unblocked in the order of their insertions onto the read wait queue. The two orderings may differ due to possible timing delays between a task *T*'s request and its insertion onto the wait queue. Other tasks requesting the resource after *T* can be inserted onto the queue before it, permitting them to use the resource ahead of *T*. For example, suppose that a number of writers and a set *G* of *br* readers have requested a resource while a writer *W* is using it. After *W*'s release, a reader $R \notin G$ requests the resource and is inserted onto the resource's read wait queue *Q* (it must block since there are waiting writers) *before* all of the tasks in *G* have been similarly inserted. *R* is then queued ahead of one or more of the tasks belonging to *G*. *W* then deletes the first *br* tasks from *Q*. This leaves one of the tasks of *G* on *Q* where it will remain (at least) until a blocked writer is run. Such a timing scenario applies as well to the first writer *W* to request the resource when there are running readers. *W* will not be deleted from the write wait queue by the last running reader if another writer *X* arrives after *W* and manages to complete its block before *W*'s. *W* will remain blocked (at least) until *X* and all waiting readers are run.

One way to ensure a FCFS service discipline among tasks that request the resource and block is by time-stamping their resource requests. A task's insertion onto a wait queue can then be based on the time stamp (similar to the queue implementation given in section 2.2.2.2.) Since this introduces a potential bottleneck into task blocking, we elect not to enforce such an ordering.

Our implementations also fail to enforce a FCFS service discipline between blocked readers and blocked writers. For example, suppose there are no operating tasks when a writer *W* requests a resource *R*. While *W* runs, let *R* be requested by *j* readers, a writer *X*, and *k* additional readers, in that order. All of the readers and *X* block. When *W* releases *R*, it determines that $j{+}k$ readers have requested *R*, based on the current value of *Val*, and unblocks them all. However, this does not follow the order in which the resource requests were made since *X* inquired about the resource before *k* of the readers. The ordering of requests is not honored because *Val* indicates the total number of waiting readers, not how many have arrived between different write requests. One way to enforce a FCFS ordering between readers and writers would be to have each blocked writer state how many readers are to be run ahead of it. But, this would then require writers to be placed onto their wait queue based on the order in which they requested the resource. Once again, we elect not to enforce such an ordering between readers and writers to avoid bottlenecks.

The following implementation is based on the first method for unblocking readers, namely, a writer and recently unblocked readers delete tasks from a read wait queue and transfer them to the Ready Queue. It is assumed that a blocked reader or writer will not be destroyed, an issue that is addressed in section 5.4.2.4.

5.4.2.2.1. Method 1: Readers Unblocking Readers

The data structure for this implementation consists of an activity counter *Val* that records resource requests and releases, an integer *Cnt* that identifies the last running reader, an integer *Awaken* that controls the number of readers to be unblocked, and separate read and write wait queues *RWaitQ* and *WWaitQ* that hold, respectively, blocked readers and writers. For convenience this data structure is referred to as a *block-queue readers/writers structure*.

```
BQ_RW = record    Block-queue Readers/Writers Structure
            Val,
            Cnt,
            Awaken                : integer;
            RWaitQ,
            WWaitQ                : Linked_Queue
        end;

Initially:
    Val     = 0;
    Cnt     = 0;
    Awaken  = 0;   number of readers to unblock
    RWaitQ  = ∅;  read and write wait queues are empty
    WWaitQ = ∅;
```

The procedure *UnblockReaders* given below moves *Awaken* tasks from a read wait queue to the Ready Queue. It may be concurrently executed by writers[49] and newly unblocked readers. As previously mentioned, a task may have to (busy) wait for a reader to appear on a read wait queue (due to possible delays between a reader's request for the resource and its subsequent blocking when its request is denied).

[49] As we will show in the next section, multiple writers may concurrently execute this procedure.

```
procedure UnblockReaders (S : BQ_RW);
var reader : TCB_Pointer;
    while F&A(S.Awaken, −1) > 0 do
        repeat until Unblock(reader, S.RWaitQ)
    end while
end procedure UnblockReaders;


procedure Read_Request (S : BQ_RW);
    if F&A(S.Val, 1) ≥ MaxR  then  writer has requested resource
        Block(S.RWaitQ);           place caller on read wait queue
        UnblockReaders(S)          help awaken other readers
    end if
end procedure Read_Request;


procedure Read_Release (S : BQ_RW);
var writer : TCB_Pointer;
    if F&A(S.Val, −1) > MaxR then         writer has requested resource
        if F&A(S.Cnt, 1) = MaxR−1 then  last running reader
            S.Cnt := 0;
            repeat until Unblock(writer, S.WWaitQ)
        end if
    end if
end procedure Read_Release;
```

```
procedure Write_Request (S : BQ_RW);
var v : integer;
    v := F&A(S.Val, MaxR);                      request resource
    if v ≠ 0 then                                resource in use
        if v < MaxR then                         first writer, ∃v running readers
            if F&A(S.Cnt, MaxR−v) = v then   readers done
                S.Cnt := 0                       become running writer
            else
                Block(S.WWaitQ)
            end if
        else
            Block(S.WWaitQ)
        end if
    end if
end procedure Write_Request;
```

```
procedure Write_Release (S : BQ_RW);
var  v       : integer;
     writer  : TCB_Pointer;
    v := F&A(S.Val, −MaxR);                     release resource
    if v mod MaxR ≠ 0 then                      ∃data blocked readers
        if v > 2*MaxR then                       ∃blocked writers too
            S.Cnt := MaxR − (v mod MaxR)
        end if;
        S.Awaken := v mod MaxR;
        UnblockReaders
    else
        if v ≥ 2*MaxR then                       ∃blocked writers only
            repeat until Unblock(writer, S.WWaitQ)
        end if
    end if
end procedure Write_Release;
```

5.4.2.2.2. A Flawed Solution

To illustrate the complexity and fine level of parallelism present in the readers/writers implementation, consider the seemingly simpler method given below for unblocking *Awaken* readers, which was used by the author in an earlier attempt at solving this problem.

```
procedure Flawed_UnblockReaders (S : BQ_RW);
var reader : TCB_Pointer;
    while TDR(S.Awaken, 1) do
        repeat until Unblock(reader, S.RWaitQ)
    end while
end procedure Flawed_UnblockReaders;
```

The TDR function is expanded below with *S.Awaken* and 1 substituted for the function's *V* and *Delta* parameters (see chapter 1 for a complete description of this function):

```
L1:  TDR := false;
L2:  if S.Awaken ≥ 1 then
L3:      if F&A(S.Awaken, −1) ≥ 1 then
L4:          TDR := true
L5:      else
L6:          F&A(S.Awaken, 1)
L7:      end if
L8:  end if;
```

Substituting Flawed_UnblockReaders for UnblockReaders can lead to a reader and a writer using the resource, a direct violation of a readers/writers constraint. The basic problem is that the TDR function is not atomic: an assignment to *Awaken* by one task can occur during a TDR on *Awaken* by another task. To illustrate the flaw in the above procedure, suppose a writer *W* is running and there are *br* blocked readers and two blocked writers *X* and *Y*.

(1) Writer *W* releases the resource, initializes *Awaken* to *br* and *Cnt* to $MaxR - br$ and begins to execute Flawed_UnblockReaders.

(2) A reader *R* unblocked by *W* joins the latter in executing Flawed_UnblockReaders. Eventually, both *R* and *W* execute TDRs on *Awaken* when its value is one. Specifically, they test *Awaken* at the same time (at L2), and then *R* followed by *W* decrements it (at L3), lowering its value to −1. *W's* increment of *Awaken* (at L6) is then delayed until these three steps occur:

(2.1) The *br* readers that were unblocked run, release the resource, and the last of these readers unblocks writer *X*.

(2.2) While *X* runs, *k* readers request the resource and block. *X* then releases the resource and initializes *Awaken* to *k* and *Cnt* to $MaxR - k$.

(2.3) Another reader requests the resource and blocks.

(3) *W* increments *Awaken* by one (at L6) to a value of k+1.

(4) As a result of the preceding step, k+1 readers are unblocked.

(5) The kth reader releases the resource and unblocks writer *Y* (since $Cnt = MaxR$).

(6) Writer *Y* and the k+1st reader run.

Unlike the flawed version just discussed, UnblockReaders as given in section 5.4.2.2.1 correctly preserves readers/writers exclusion given the above scenario since no reincrement of *Awaken* is ever performed: Let *Awaken* have an initial value of br>0 when a writer *W* executes UnblockReaders. *W* cycles, first decrementing *Awaken* by one, and second, if the old value was positive, deleting a task from the read wait queue. After *br* iterations of these two steps, all of the readers that were to be unblocked have been deleted from the queue. Suppose, as above, that these readers release the resource, start another writer *X* that runs, releases the resource, and initializes *Awaken* to k>0. If this all happens before *W* decrements *Awaken* for the *br*+1st time, *W* and *X* will unblock tasks until *Awaken* reaches zero, but no more than *k* readers will be deleted.[50] Thus, a reader will not resume a blocked writer while other readers are still running.

The above scenario also points out that since a writer, such as *W*, may be about to decrement *Awaken* after another writer *X* has reinitialized it, it is imperative that *Cnt* be initialized before *Awaken* within the write release procedure. Otherwise, between *X*'s initializations of *Awaken* and *Cnt*, a reader *R* can be unblocked by *W* and increment *Cnt* upon its release of the resource. The completion of *R* would then be missed, preventing the last running reader from being identified.

5.4.2.2.3. Method 2: Queueing the Read Wait Queue

By utilizing a multiqueue component of the Ready Queue, the unblocking of readers by a writer can be performed in constant time. As noted in section 5.4.2.1, a read wait queue of *N* tasks can be inserted directly onto the Ready Queue by inserting an item *I* with multiplicity *N* onto multiqueue *RMQ*[*P*], where *P* is the priority chosen for the reader unblocking activity and *I* is a queue tagged with multiplicity whose queue component is the read wait queue. Recall from

[50] As mentioned in the previous section, this illustrates how two writers may end up unblocking readers at the same time.

section 5.3.2.6 that when GetTask performs a multiqueue deletion from $RMQ[P]$ and obtains a reference to I, it then performs a deletion from the read wait queue to obtain a task T. If T is of priority $p \geq P$, where larger values represent higher priorities, then T is executed. Otherwise, GetTask inserts T onto $RMQ[p]$ (and returns to the Ready Queue for another task). We note that GetTask's read wait queue deletion, like the Unblock operation in the UnblockReaders procedure, may have to be executed a (bounded) number of times before a task is finally obtained: I may be made a member of the Ready Queue between a reader's increment of *Val* and its Block operation (refer to the procedures below for details).

Readers/writers procedures based on this method are identical to those given for the first unblocking method except that readers do not unblock readers (the UnblockReaders procedure is eliminated) and a writer, upon releasing the resource, transfers a read wait queue (i.e., I above) to the Ready Queue (in place of deleting tasks one by one from a read wait queue and inserting them onto the Ready Queue). Within the write release procedure, the expression v modulo *MaxR* gives the number of readers to be unblocked and thus defines the multiqueue insertion multiplicity for a read wait queue.

In the following implementation, rather than associating an explicit queue tagged with multiplicity variable with a read wait queue, the former is represented by the *REntry* and *RWaitQ* components of the block-queue readers/writers structure.

```
BQ_RW = record    Block-queue Readers/Writers Structure
            Val,
            Cnt                        : integer;
            REntry                     : MQEntry;
            RWaitQ,
            WWaitQ                     : Linked_Queue
        end;

Initially:
    Val             = 0;
    Cnt             = 0;
    REntry.List_Loc = nil;
    RWaitQ          = ∅;    read and write wait queues are empty
    WWaitQ          = ∅;
```

```
procedure Read_Request (S : BQ_RW);
    if F&A(S.Val, 1) ≥ MaxR then  writer has requested resource
        Block(S.RWaitQ)           place caller on read wait queue
    end if
end procedure Read_Request;


procedure Read_Release (S : BQ_RW);
var writer : TCB_Pointer;
    if F&A(S.Val, −1) > MaxR then          writer has requested resource
        if F&A(S.Cnt, 1) = MaxR−1 then  last running reader
            S.Cnt := 0;
            repeat until Unblock(writer, S.WWaitQ)
        end if
    end if
end procedure Read_Release;
```

```
procedure Write_Request (S : BQ_RW);
var v : integer;
    v := F&A(S.Val, MaxR);                 request resource
    if v ≠ 0 then                          resource in use
        if v < MaxR then                   first writer, ∃v running readers
            if F&A(S.Cnt, MaxR–v) = v then   readers done
                S.Cnt := 0                 become running writer
            else
                Block(S.WWaitQ)
            end if
        else
            Block(S.WWaitQ)
        end if
    end if
end procedure Write_Request;


procedure Write_Release (S : BQ_RW);
var  v       : integer;
     writer  : TCB_Pointer;
    v := F&A(S.Val, –MaxR);                    release resource
    if v mod MaxR ≠ 0 then                     ∃blocked readers
        if v > 2*MaxR then                     ∃blocked writers too
            S.Cnt := MaxR – (v mod MaxR)
        end if;
        Linked_MQInsert(address(S.REntry), v mod MaxR, RMQ[P])
    else
        if v ≥ 2*MaxR then                     ∃blocked writers only
            repeat until Unblock(writer, S.WWaitQ)
        end if
    end if
end procedure Write_Release;
```

5.4.2.3. A Writer Priority Implementation

In this section we consider a readers/writers implementation that gives priority to writers over readers. It is quite similar to the fair readers/writers implementation with the exception that when a running writer releases the resource, it unblocks a waiting writer in preference to waiting readers. Our implementation also gives a writer that is requesting the resource priority over readers that have already been granted the resource but have not yet begun to use it (as suggested by Edler [84]). That is, the readers' permission will be rescinded to allow the writer to proceed. A simpler priority scheme that requires the requesting writer to wait until all readers granted the resource have used it will be described as well.

We begin with a writer priority implementation based on the first method for unblocking readers followed by an implementation based on the second unblocking method. Again, the destruction of blocked tasks is addressed in section 5.4.2.4.

5.4.2.3.1. Method 1: Readers Unblocking Readers

The main difficulty encountered in this implementation is in giving a writer requesting the resource priority over those readers that have been granted the resource, but have not yet begun to use it. Generally, when a writer *W* releases the resource and there are blocked readers and no blocked writers, all the readers are granted the resource. That is, they will be unblocked (deleted from a read wait queue and inserted onto the Ready Queue), and run (when deleted from the latter queue). Now, however, if another writer *X* requests the resource before all of the readers have been unblocked, the unblocking activity will be suspended until all write requests have been honored. Furthermore, to minimize *X*'s waiting time, any of the readers that have been unblocked but have not yet begun to use the resource by the time *X* arrives will be required to release the resource and then reissue their requests.

The suspension of the unblocking activity may be done by the first writer to request the resource or by a reader that was unblocked. The latter case is needed in a scenario to be described shortly whereby the first writer fails to halt the unblocking activity. First, let us consider how a writer can suspend the unblocking of readers.

Suppose there are *br* blocked readers and no blocked writers when a writer *W* releases the resource. *W* then initializes the counter *Awaken* to *br* and begins to delete tasks from the read wait queue (via the UnblockReaders procedure). Since there are no operating writers[51] ($Val < MaxR$), any new readers that request the resource run, without regard to *W*'s progress in deleting the blocked readers. Let a writer *X* then request the resource via

$$v := \text{F\&A}(Val,\ MaxR).$$

By making *Val* greater than or equal to *MaxR*, all read requests that follow *X*'s will block until all write requests are satisfied. At the time of *X*'s request, there are *v* operating readers. They are

[51] Recall from section 5.4.2.1 that an operating task is a task that has requested but not yet released the resource.

either running[52] or blocked. X suspends the unblocking of any more readers by executing

$$m := \text{F\&A}(Awaken, -MaxR),$$

where the current value of *Awaken* equals the number of readers that remain to be unblocked. Thus, of the v operating readers that were granted permission to access the resource before X, m of the original br readers remain blocked (and will remain so until all writers are run). The number of running readers then equals $v{-}m$. As in the fair readers/writers implementation, a counter *Cnt* determines when the last of these running readers has terminated: each running reader increments *Cnt* by one, and X increments it by $MaxR{-}v{+}m$. X accesses the resource if it raises *Cnt* to *MaxR*; otherwise, it blocks, to be resumed by the last running reader (which is the reader that raises *Cnt* to *MaxR*).

Each running reader that was blocked first checks for the existence of operating writers (i.e., if $Val > MaxR$) before using the resource. If such writers exist, the reader releases the resource and reissues its request to allow the writers faster access to the resource. In effect, the reader's permission to access the resource is temporarily removed.

Observe that in the scenario above, X can decrement *Awaken* and increment *Cnt* before W initializes *Awaken* to br. In this case it will appear to X that the reader unblocking activity has been completed (i.e., the value of the variable m above is zero) and that each reader that was blocked when X requested the resource has since become a running reader. X will therefore block (since its increment of *Cnt* will not raise *Cnt*'s value to *MaxR*), and W will initialize *Awaken* to br and start to unblock readers, ignorant of X's presence.[53] Each of the readers unblocked by W will release the resource as soon as it runs since a writer (X) is waiting. Thus, none of these readers makes any real progress until X uses the resource. Therefore, running readers that were blocked will attempt to suspend the unblocking of additional readers when they find that a writer is waiting. The actions of the readers and writers under this scenario are as follows: Suppose there are br blocked readers when a writer W releases the resource. Let the resource then be requested by rr readers, which become running readers, followed by a writer X. X's resource request (i.e., a fetch-and-add of *MaxR* on *Val*) indicates that there are v operating readers, composed of running

---

[52] For the purposes of this implementation reader tasks that are ready (i.e., on the Ready Queue) will be treated as if they were already running.

[53] The time X remains blocked could possibly be decreased by having W reexamine Val (to determine if any writers have arrived) before initializing *Awaken*. However, we suspect that this optimization would rarely be effective in practice.

and/or blocked readers, where $v = rr + br$. X decrements *Awaken* before W initializes it to br. Since *Awaken* was nonpositive, X assumes that the reader unblocking activity has been completed and that the v operating readers are running readers. Hence, X increments *Cnt* by $MaxR - v$ and blocks (since it does not drive *Cnt* to *MaxR*). After X's decrement of *Awaken*, W initializes *Awaken* to br and unblocks a reader R. Upon running, R inspects *Val* and determines that a writer (X) has arrived ($Val > MaxR$) and is waiting to use the resource. R then prevents the unblocking of further reader tasks by executing

$$m := \text{F\&A}(Awaken, -MaxR),$$

just as the first writer does in the previous scenario; R then increments *Cnt* by m to simulate m read releases. The net effect of R's and X's increments of *Cnt* is the same as when a writer halts the unblocking activity and increments *Cnt* by $MaxR - v + m$.

The last case to consider is that of a reader R that is unblocked between the first writer X's increment of *Val* and decrement of *Awaken*. If R decrements *Awaken* before X, R and X follow the second scenario. That is, they proceed as if X had decremented *Awaken* before the writer releasing the resource could initialize it. Otherwise, the first scenario applies whereby the writer suspends the unblocking activity.

In summary, each reader that is unblocked and finds that a writer is waiting will attempt to halt the unblocking of additional readers. Whether it succeeds or fails, the reader will release the resource to allow a writer to acquire it, and then repeat the read request.

```
BQ_RW = record    Block-queue Readers/Writers Structure
            Val,
            Cnt,
            Awaken                  : integer;
            RWaitQ,
            WWaitQ                  : Linked_Queue
        end;

Initially:
    Val     = 0;
    Cnt     = 0;
    Awaken  = 0;  number of readers to unblock
    RWaitQ  = ∅;  read and write wait queues are empty
    WWaitQ  = ∅;
```

```
procedure WP_Read_Request (S : BQ_RW);
var m : integer;
L:  if F&A(S.Val, 1) ≥ MaxR then                  ∃write requests
        Block(S.RWaitQ);                          yield to writers
        if S.Val > MaxR then                      ∃new write requests
            m := F&A(S.Awaken, −MaxR);
            if m > 0 then                         halted UnblockReaders
                F&A(S.Cnt, m)                     m readers not unblocked
            end if;
            WP_Read_Release(S);                   yield resource to a writer
            go to L                               repeat request
        else
            UnblockReaders(S)
        end if
    end if
end procedure WP_Read_Request;
```

```
procedure WP_Read_Release (S : BQ_RW);
var v      : integer;
    writer : TCB_Pointer;
   if F&A(S.Val, −1) > MaxR then                  waiting writers
       if F&A(S.Cnt, 1) = MaxR−1 then             last running reader
           S.Cnt := 0;
           repeat until Unblock(writer, S.WWaitQ)
       end if
   end if
end procedure WP_Read_Release;
```

```
procedure WP_Write_Request (S : BQ_RW);
var m, v : integer;
    v := F&A(S.Val, MaxR);                  request resource
    if v ≠ 0 then                           resource in use
        if v < MaxR then                    ∃running/blocked readers
            m := F&A(S.Awaken, –MaxR);      stop UnblockReaders
            if m < 0 then m := 0 end if;    m readers not unblocked
            if F&A(S.Cnt, MaxR–v+m) = v–m then
                S.Cnt := 0
            else
                Block(S.WWaitQ)
            end if
        else
            Block(S.WWaitQ)
        end if
    end if
end procedure WP_Write_Request;


procedure WP_Write_Release (S : BQ_RW);
var  v       : integer;
     writer  : TCB_Pointer;
    v := F&A(S.Val, –MaxR);         release resource
    if v ≥ 2*MaxR then              ∃blocked writers
        repeat until Unblock(writer, S.WWaitQ)
    elseif v mod MaxR ≠ 0 then      ∃blocked readers
        S.Awaken := v mod MaxR;
        UnblockReaders
    end if
end procedure WP_Write_Release;
```

The implementation can be modified easily to ensure that all readers that have been granted permission to access the resource do indeed use it before any subsequent write requests are honored. The read request procedure need only be altered so that a reader that was blocked does not check for waiting writers when it runs, but proceeds directly to use the resource (i.e., the **if** statement following Block in the read request procedure is replaced with a call to UnblockReaders). The write request procedure is changed so that a writer does not attempt to suspend the unblocking of readers. Instead, it is assumed that all operating readers are running readers when the writer requests the resource (i.e., in the write request procedure, the variable *m* is eliminated and *Awaken* is not decremented).

5.4.2.3.2. Method 2: Queueing the Read Wait Queue

In the following implementation, a read wait queue is inserted directly onto a multiqueue component of the Ready Queue. More precisely, an item *I* with multiplicity *N* is added to the

Ready Queue, where *I* is a queue tagged with multiplicity whose queue component is the read wait queue and *N* equals the number of tasks to be unblocked from this queue.

A reader or writer halts the unblocking of readers not by making *Awaken* negative, as in the first implementation, but by removing *I* from the Ready Queue via

MQLowerMult(**address**(*I*), *MaxMult*, *RMQ*[*P*], *M*),[54]

where *P* is the priority of the unblocking activity. If MQLowerMult succeeds, $M \le N$ tasks will not be unblocked, and the task that removes the read wait queue from the Ready Queue must therefore increment *Cnt* as described in the previous section: If a reader removes the read wait queue from the Ready Queue, then it increments *Cnt* by *M*, and if a writer does so, then it increments *Cnt* by $MaxR - (v - M)$, where *v* equals the number of operating readers at the time the writer requested the resource.

In the following implementation, the *REntry* and *RWaitQ* components of a block-queue readers/writers structure represent the queue tagged with multiplicity variable that is associated with the read wait queue.

```
BQ_RW = record    Block-queue Readers/Writers Structure
            Val,
            Cnt                     : integer;
            REntry                  : MQEntry;
            RWaitQ,
            WWaitQ                  : Linked_Queue
        end;

Initially:
   Val             = 0;
   Cnt             = 0;
   REntry.List_Loc = nil;
   RWaitQ          = ∅;    read and write wait queues are empty
   WWaitQ          = ∅;
```

[54] Recall from chapter 3 that if MQLowerMult(*I*, *MaxMult*, *MQ*, *M*) is successful, *I* is removed from multiqueue *MQ* and *M* is set to the remaining multiplicity of *I*. Otherwise, *M* is set to zero.

```
procedure WP_Read_Request (S : BQ_RW);

var m : integer;  multiplicity of read wait queue

L:  if F&A(S.Val, 1) ≥ MaxR then
        Block(S.RWaitQ);
        if S.Val > MaxR then                      writer waiting - remove RWaitQ
            if Linked_MQLowerMult(address(S.REntry), MaxMult, RMQ[P], m) then
                F&A(S.Cnt, m)                     simulate m read releases
            end if;
            WP_Read_Release(S);                   yield resource to writer
            go to L
        end if
    end if
end procedure WP_Read_Request;


procedure WP_Read_Release (S : BQ_RW);

var writer : TCB_Pointer;

    if F&A(S.Val, −1) > MaxR then          writer waiting
        if F&A(S.Cnt, 1) = MaxR−1 then     last running reader
            S.Cnt := 0;
            repeat until Unblock(writer, S.WWaitQ)
        end if
    end if
end procedure WP_Read_Release;
```

```
procedure WP_Write_Request (S : BQ_RW);
var m, v : integer;
     v := F&A(S.Val, MaxR);                          write request
     if v ≠ 0 then                                   resource in use
          if v < MaxR then                           ∃running/blocked readers
               Linked_MQLowerMult(address(S.REntry), MaxMult, RMQ[P], m);
               if F&A(S.Cnt, MaxR–v+m) = v–m then
                    S.Cnt := 0
               else
                    Block(S.WWaitQ)
               end if
          else
               Block(S.WWaitQ)
          end if
     end if
end procedure WP_Write_Request;
```

```
procedure WP_Write_Release (S : BQ_RW);
var  v       : integer;
     writer  : TCB_Pointer;
     v := F&A(S.Val, –MaxR);          release resource
     if v ≥ 2*MaxR then               ∃blocked writers
          repeat until Unblock(writer, S.WWaitQ)
     else
          if v mod MaxR ≠ 0 then      ∃v modulo MaxR blocked readers
               Linked_MQInsert(address(S.REntry), v mod MaxR, RMQ[P])
          end if
     end if
end procedure WP_Write_Release;
```

The following modifications can be made to prevent readers granted permission to use the resource from being preempted by a writer that arrives before they have actually begun to access it: Readers are always run after being unblocked (i.e., the **if** statement following Block in the read request procedure is omitted), and a writer is not allowed to suspend the unblocking of readers (i.e., the multiqueue LowerMult operation and the variable *m* are eliminated from the write request procedure).

5.4.2.4. Destroying a Blocked Reader/Writer Task

The procedures for the readers/writers implementations assume that a blocked reader or writer will not be destroyed. In order to support the destruction of blocked reader (writer) tasks, other tasks must perform read (write) releases on their behalf.[55] For the implementations based on the first method for unblocking readers this means that a reader or writer that deletes a nil item from a read (write) wait queue executes a read (write) release.[56] For example, the statement

Unblock(*T*, *S.RWaitQ*)

within the UnblockReaders procedure (section 5.4.2.2.1) needs to be replaced with

Unblock(*T*, *S.RWaitQ*);
if *T* = **nil then**
    Read_Release(*S*)
end if.

Similar substitutions have to be made for the unblocking of writers.

When employing the second method for unblocking readers, tasks on a read wait queue attached to the Ready Queue are not deleted by other reader or writer tasks, but by GetTask. Therefore to implement this method the GetTask procedure of section 5.3.2.6 would have to be modified as follows: Upon deleting a nil item from a read or write wait queue, GetTask performs a read or write release, respectively, on behalf of the destroyed task. Basically, this requires wait queues to be typed as reader, writer, or non-reader/writer.

5.4.3. Events

An event is a task synchronization mechanism commonly associated with the occurrence (or completion) of an operation such as I/O. An event *E* has either *occurred* or *not occurred*. Tasks *await* the occurrence of an event *E* by either busy-waiting or blocking. Once a task *causes E* to occur, all tasks waiting on *E* resume their execution. The event may be *reset* (to *not occurred*) at any time. After an event has been *caused* (*reset*), subsequent *causes* (*resets*) will be ignored.

In this section, one weak and two block-queue event implementations are given. For each implementation, await, cause, and reset primitives are defined. Awaits are fully parallel in all implementations. Notably, within this primitive, the testing of an event's status and busy-waiting or blocking for its occurrence are serialization-free. The cause primitive is serialization-free in the busy-waiting implementation, as well as in one of the block-queue implementations. Resets are serialized in all implementations. Fortunately, many more tasks are likely to await the

[55] This is analogous to the actions taken by a V operation when a blocked task on a semaphore queue is destroyed. See section 5.4.1.2.

[56] Recall that *T* is replaced with a nil item on the wait queue when it is signaled to destroy itself.

occurrence of an event than cause or reset it.

5.4.3.1. A Weak Implementation

An event is represented by an integer value *Val* and two semaphores, *R_Sem* and *A_Sem*. If *Val* is negative, the event has occurred. Otherwise, *Val* equals the number of tasks awaiting the event. *R_Sem* is a binary semaphore that serializes resets: concurrent resets are prohibited to eliminate coordination problems in the management of *Val*.[57] However, concurrent causes and resets are supported. *A_Sem* is a general (counting) semaphore that signals how many tasks busy-waiting on an event can be released (i.e., informed that the event has occurred). Initially, *R_Sem* is one (to enable Resets), *A_Sem* is zero (no tasks are awaiting the event), and *Val* is zero if the event has not occurred and negative if it has.

A task *T* causes an event *E* to occur by making *E*'s value negative via

$$v := \text{F\&A}(E.Val, -MaxR),$$

where *MaxR* bounds the number of tasks that can await an event. When *E* is caused, there are *v* tasks waiting on it. *T* releases them by incrementing *A_Sem* by *v*.

An event is reset by reinitializing *Val* to zero.

When a task awaits an event *E*, it first determines if *E* has been caused by testing *Val*. If *E* has occurred ($Val<0$), the task continues. Otherwise, the task increments *Val* by one to indicate that it is waiting on *E* and then busy-waits on *A_Sem*. It is possible that between the time the task tests for the occurrence of *E* and increments *Val*, *E* is caused. In this case, the task will detect the occurrence of *E* (since $Val<0$) and not busy-wait.

The initial test of *Val* (within the await procedure) is necessary to prevent a task from erroneously flipping an event's status from occurred to not occurred. Suppose this test were omitted. Let a task then perform an await on an event *E* by incrementing *Val* and (busy) waiting if *Val*'s old value is nonnegative (*E* has not occurred) or continuing if it is negative (*E* has occurred). Now, if a task repeatedly executes awaits on an event *E* that has occurred, *Val* will eventually become nonnegative, changing the event's status to not occurred, although no reset has been performed. By pretesting an event's status at the start of an await, a task increments the event's value only if the event has not occurred. Since the total number of tasks that can await an

[57] We will return to this issue after the primitives have been presented.

event is less than *MaxR*, an event's status can not be switched by the execution of awaits.

The timing of a cause and a reset does not affect the release of busy-waiting tasks. For example, suppose an event *E* is caused when there are *v* tasks busy-waiting on its *A_Sem*, and the event is then reset before any one of these tasks tests the semaphore. The tasks do not ‘‘miss’’ the occurrence of *E* since the task that causes *E* increments *A_Sem* by *v*, independently of any resets that may later take place.

```
Weak_Event = record
                Val     : integer;
                R_Sem,
                A_Sem : Weak_Semaphore
             end;


Initially:
    Val     = 0;  if the event has not occurred
    Val     < 0;  if the event has occurred
    R_Sem = 1;  Resets may be done
    A_Sem = 0;  No tasks are awaiting the event


procedure Cause (E : Weak_Event);
var v : integer;
    if E.Val ≥ 0 then                    E hasn't occurred
        v := F&A(E.Val, −MaxR);   cause E
        if v > 0 then                    v tasks waiting
            Weak_VC(E.A_Sem, v)
        end if
    end if
end procedure Cause;


procedure Reset (E : Weak_Event);
    Weak_P(E.R_Sem);
        if E.Val < 0 then   reset E
            E.Val := 0
        end if;
    Weak_V(E.R_Sem)
end procedure Reset;
```

```
procedure Await (E : Weak_Event);
    if E.Val ≥ 0 then                      E hasn't occurred
        if F&A(E.Val, 1) ≥ 0 then  wait for E to occur
            Weak_P(E.A_Sem, 1)
        end if
    end if
end procedure Await;
```

Reset tests an event *E*'s value *Val* before setting it to zero to avoid erasing the count of tasks busy-waiting on *E*. Suppose this test were not done before *Val* is zeroed. Let *v* tasks be waiting (*Val* = *v*) for *E* to occur when a reset is executed. *Val* is set to zero. As a result, it appears to the next cause that zero (instead of *v*) tasks are waiting. By pretesting *Val* before its assignment, the count of waiting tasks is not lost.

Resets are serialized to preserve the awaiting task count. Suppose they were not. Let two resets be executed *concurrently* on an event *E* that has a status of occurred (i.e., *Val* is negative). Assume both resets test *Val* at the same time. Since *Val* is negative, each reset sets it to zero. However, between the executions of these two assignments, a task can perform an await of *E* and set *Val* to one. The second reset's assignment of zero to *Val* would then erase the fact that a task was waiting.

On the other hand, tasks may simultaneously attempt to cause an event without ill effects. Multiple tasks may test an event's value *Val* when *Val* is nonnegative and decrement it by *MaxR*.[58] The task that does so first actually causes the event. The other decrements have no effect on the status of the event (they only make its value ''more'' negative).

As in all of the previous implementations containing weak semaphores, a task can starve while performing an await if P/V are not fair. Notice that even if the semaphore primitives are FCFS, tasks waiting for an event to occur may not be released in the order that they began to execute their awaits. For example, assume an event *E* has not occurred and a task *T* does an await on *E*. Suppose *T* increments *Val* to one (it is the first to await *E*) and before executing a P on *A_Sem* (i) *E* is caused, (ii) *A_Sem* is set to one, (iii) *E* is then reset, and (iv) another task *U* executes an await of *E* and does a P on *A_Sem*. In this case, *U* continues to execute and *T* must wait for *E* to re-occur.

[58] If a cause finds *Val* to already be negative (the event has occurred), no decrement is performed in order to prevent integer underflow.

5.4.3.2. A Block-queue Implementation

Our block-queue implementation of events replaces the busy-waits performed on an event's weak await semaphore with operations performed on a wait queue associated with the event. As in the readers/writers implementations, two methods are given for unblocking tasks from a wait queue.

5.4.3.2.1. Method 1: Waiters Unblocking Waiters

In this implementation, as tasks are resumed they aid the causer of an event in unblocking other tasks awaiting the event. To coordinate the unblocking activity, we associate with an event an integer *Awaken* that equals the number of tasks that are to be unblocked from the event's wait queue. (The procedure UnblockWaiters below is analogous to the UnblockReaders procedure used in the readers/writers implementations.)

Unlike the weak event implementation, multiple causes are not fully parallel: access to *Awaken* and the unblocking of readers by different causers are serialized to ensure the proper management of *Awaken.* Hence, these operations are placed into a critical code section protected by a binary semaphore *C_Sem.* After the primitives are presented, a justification for the critical code section will be given in the next section.

As in the weak implementation, the timing of a cause and a reset does not affect the number of tasks unblocked: If v tasks are awaiting an event when it is caused, then v tasks will be unblocked regardless of any subsequent resets that occur. However, the wait queue ordering of tasks may not correspond to the order in which the tasks began to execute their awaits. Thus, the v tasks that are unblocked may not be the first v tasks to have tested the event's status.

```
BQ_Event = record
              Val, Awaken : integer;
              C_Sem,
              R_Sem       : Weak_Semaphore;
              AwaitQ      : Linked_Queue
           end;
```

Initially:

```
  Val     = 0;   if event has not occurred
  Val     < 0;   if event has occurred
  Awaken = 0;    number of tasks to be unblocked
  C_Sem   = 1;   Causes may be done
  R_Sem   = 1;   Resets may be done
  AwaitQ  = ∅;
```

```
procedure UnblockWaiters (E : BQ_Event);
var t                    : TCB_Pointer;
  while F&A(E.Awaken, –1) > 0 do
    repeat until Unblock(t, E.AwaitQ)
  end while
end procedure UnblockWaiters;
```

```
procedure Cause (E : BQ_Event);
var v : integer;
  if E.Val ≥ 0 then                 E hasn't occurred
    v := F&A(E.Val, –MaxR);         try to cause E
    if v > 0 then                   successful
      Weak_P(E.C_Sem);
        E.Awaken := v;              v tasks waiting
        UnblockWaiters(E);
      Weak_V(E.C_Sem)
    end if
  end if
end procedure Cause;
```

```
procedure Reset (E : BQ_Event);
  Weak_P(E.R_Sem);
    if E.Val < 0 then reset E
      E.Val := 0
    end if;
  Weak_V(E.R_Sem)
end procedure Reset;
```

```
procedure Await (E : BQ_Event);
    if E.Val ≥ 0 then                   E hasn't occurred
        if F&A(E.Val, 1) ≥ 0 then       wait for E
            Block(E.AwaitQ);
            UnblockWaiters(E)
        end if
    end if
end procedure Await;
```

Observe that if tasks causing events are the only ones permitted to unblock tasks awaiting events (i.e., tasks unblocked by the causer of an event do not help to unblock other waiting tasks), then the cause procedure above can be made serialization-free: Multiple causers of an event can unblock tasks in parallel (obviating the need for the critical code section) since the event's wait queue is concurrently accessible and the number of tasks each causer has to unblock is kept in a local variable (*v*). This modification is appropriate when the number of tasks on the await queue is small and causes are likely to overlap. Otherwise, the original procedure should be preferred as it allows multiple tasks to unblock tasks on the await queue.

5.4.3.2.2. A Flawed Version of Method 1

Let us consider an implementation that permits tasks resumed by the causer of an event to unblock other tasks awaiting the event, as in the original implementation of the preceding section, and that allows causes to be serialization-free, as in the modification suggested above. The following procedure is an attempt at such a strategy, but as its name suggests, a correct implementation is not obtained.

```
procedure Flawed_Cause (E : BQ_Event);
var v : integer;
    if E.Val ≥ 0 then                   E hasn't occurred
        v := F&A(E.Val, –MaxR);         cause E
        if v > 0 then
            F&A(E.Awaken, v);           v tasks waiting
            Flawed_UnblockWaiters(E)
        end if
    end if
end procedure Flawed_Cause;
```

```
procedure Flawed_UnblockWaiters (E : BQ_Event);
var t                              : TCB_Pointer;
      while TDR(E.Awaken, 1) do
           repeat until Unblock(t, E.AwaitQ)
      end while
end procedure Flawed_UnblockWaiters;
```

In the Flawed_Cause procedure, a fetch-and-add is used in place of a simple assignment to modify *Awaken* since one cause could occur before all of the tasks being unblocked by a previous cause had been deleted and *Awaken* driven to zero. Thus, each cause adds to *Awaken* the number of additional tasks to be unblocked. This implies that *Awaken* must equal zero after the cessation of all cause and await activity (unlike the original implementation which permits *Awaken* to have a negative value). Otherwise, after a group of tasks had been unblocked, the next cause performing a fetch-and-add of *v* on *Awaken*, when *Awaken* was negative, would unblock fewer than *v* tasks. Flawed_UnblockWaiters differs from UnblockWaiters in that *Awaken* is decremented via a TDR, instead of simply a fetch-and-add, to ensure that *Awaken* is restored to zero; it cannot be left at some unknown negative value.

Flawed_UnblockWaiters then resembles Flawed_UnblockReaders (see section 5.4.2.2.2). Not surprisingly, a scenario similar to the one given for the flawed readers/writers solution (of section 5.4.2.2.2) can lead to a waiting task not being unblocked by a cause: Suppose an event *E*'s value of *Awaken* equals 1 when two tasks *A* and *B* execute Flawed_UnblockWaiters and perform TDRs on *Awaken*. Both tasks perform tests and decrements, driving *Awaken* to −1. *A* unblocks a task since its decrement occurred first. *B*, having referenced *Awaken* after *A* drove it to zero, increments *Awaken* by 1 to undo its decrement and thus complete the TDR. Assume that before *B*'s increment is performed (i) *A* performs another TDR and returns (as *Awaken* is nonpositive), (ii) *E* is reset, (iii) another task *C* awaits *E* and blocks, and (iv) a Flawed_Cause of *E* is executed by a task *D*. *D* increments *Awaken* by one, but since this only raises the value to zero, the Flawed_Cause operation will not unblock *C* (i.e., this operation's call to Flawed_UnblockWaiters will not result in *C* being unblocked since *Awaken* is zero). *B* then increments *Awaken* after (iv), raising the count to one, and returns. Belatedly, *Awaken* equals the number of tasks to be unblocked. The result is that *C* misses the occurrence of *E* and must wait (possibly forever) for the event to re-occur.

Unlike the flawed version just presented, the original implementation does not allow blocked tasks to miss the occurrence of events. However, it is possible that a task causing an event and/or a task unblocked as a result of one cause to end up unblocking tasks awaiting the next occurrence of the event: Suppose *Awaken* is initialized to $v > 0$ by a cause. Eventually, v tasks are unblocked and *Awaken* equals zero. The task executing the cause primitive and these v tasks then each decrement *Awaken* once more to determine if any tasks remain to be unblocked. If all of these tasks exit from UnblockWaiters before the next cause takes place, *Awaken* will have a final value of $-(v+1)$. But, if the next cause occurs and assigns a value of j to *Awaken* before k of these $v+1$ tasks have decremented *Awaken*, up to $\min(j, k)$ of them can help unblock the tasks waiting for the event's occurrence.

5.4.3.2.3. Method 2: Queueing the Await Queue

In this section we give two implementations that allow tasks awaiting an event to be transferred en masse onto the Ready Queue, as opposed to deleting and inserting them one by one from an await queue to the Ready Queue. The implementations differ in that the first one inserts an await queue indirectly onto the Ready Queue whereas the second one does the insertion directly. In both implementations, causes are fully parallel, as are awaits (note that they do not unblock other tasks), and resets are unaffected (they remain serial).

5.4.3.2.3.1. Indirect Version

Let v tasks be awaiting an event E. Instead of unblocking these tasks one at a time, a task causing E places an item I with multiplicity v onto $RMQ[P]$, where I is a queue pointer tagged with multiplicity[59] whose *QPtr* component points to E's await queue and P is the await queue's priority. I is passed as a second parameter to the cause primitive. We assume that the proper safeguards are in place to prevent I from being prematurely deallocated or used by another cause.

[59] This data structure was defined in section 5.2 and explained with respect to GetTask in section 5.3.2.6.

```
BQ_Event = record
                Val       : integer;
                R_Sem  : Weak_Semaphore;
                AwaitQ : Linked_Queue
           end

Initially:
   Val       = 0;   if event has not occurred
   Val       < 0;   if event has occurred
   R_Sem  = 1;   Resets may be done
   AwaitQ = ∅;

procedure Cause (E : BQ_Event;
                 I  : QPointerTaggedWithMultiplicity);
var v : integer;
   if E.Val ≥ 0 then                                E hasn't occurred
      v := F&A(E.Val, −MaxR);                      try to cause E
      if v > 0 then                                 v tasks waiting
         I.QPtr := address(E.AwaitQ);
         Linked_MQInsert(address(I), v, RMQ[P])
      end if
   end if
end procedure Cause;

procedure Reset (E : BQ_Event);
   Weak_P(E.R_Sem);
      if E.Val < 0 then reset E
         E.Val := 0
      end if;
   Weak_V(E.R_Sem)
end procedure Reset;

procedure Await (E : BQ_Event);
   if E.Val ≥ 0 then               E hasn't occurred
      if F&A(E.Val, 1) ≥ 0 then  wait for E
         Block(E.AwaitQ)
      end if
   end if
end procedure Await;
```

An implementation is given in the next section that permits await queues to be inserted directly onto the Ready Queue, i.e., queues tagged with multiplicity instead of queue pointers tagged with multiplicity are used to add await queues to the Ready Queue. The implementation above would be incorrect if we were to substitute the former data structure for the latter. Such a modification would require a single queue tagged with multiplicity item *I* to be associated with an event *E*'s await queue. Each time *E* was caused, *I* would be inserted onto the Ready Queue. However, certain sequences of causes and resets would result in *I* occupying multiple (linked list) positions of a multiqueue component of the Ready Queue (which is a physical impossibility).[60] Indirectly inserting an await queue onto the Ready queue via queue pointers tagged with multiplicity avoids this problem. For example, suppose I_1 and I_2 are queue pointers tagged with multiplicity that reference an event *E*'s await queue. A cause of *E* inserts I_1 with multiplicity *M* onto the Ready Queue. Then, a reset, an await, and then another cause of *E* take place before I_1 is taken off of the Ready Queue, i.e., before *M* deletions are performed on I_1. The second cause does a Ready Queue insertion of I_2 with multiplicity one that also references *E*'s await queue.

5.4.3.2.3.2. Direct Version

This implementation is direct in that an await queue is placed onto the Ready Queue by performing a Ready Queue insertion of an item *I* with multiplicity *M*, where *I* is a queue tagged with multiplicity[61] whose queue component is the await queue and *M* is the number of tasks to be deleted from this queue. For simplicity the implementation is described in terms of await queues rather than queue tagged with multiplicity items.

A cause inserts an event's await queue onto the Ready Queue if the await queue is not already a member of the Ready Queue. Otherwise, the cause increments the await queue's multiplicity by the number of tasks that are awaiting the event via the RaiseMult primitive of chapter 3.[62]

The major complexity of the implementation lies in preventing temporally overlapping causes of an event *E* from each inserting *E*'s await queue onto the Ready Queue: Suppose *v* tasks are awaiting *E* and *E*'s await queue is not a member of the Ready Queue. A cause C_1 sets *E*'s

---

[60] In contrast, the readers/writers implementations permit a read wait queue to be directly inserted onto the Ready Queue each time a writer releases a resource and has to unblock readers. Recall that only writers insert read wait queues onto the Ready Queue. Since writers are mutually exclusive and a writer cannot be run until the associated read wait queue is deleted or removed from the Ready Queue, at most one reference to a read wait queue is ever on the Ready Queue.

[61] In the implementation *I* is represented by the *AEntry* and *AWaitW* components of the block-queue event data structure.

[62] Recall that MQRaiseMult(*I*, *K*, *Q*) increments item *I*'s multiplicity by *K* and returns true if *I* belongs to multiqueue *Q* and returns false otherwise.

status to occurred, and before it determines that *E*'s await queue must be inserted onto the Ready Queue, *E*'s status is reset to not occurred, *w* tasks perform awaits on *E*, and another cause C_2 sets *E*'s status to occurred. If C_1 and C_2 simultaneously test the await queue's status with respect to the Ready Queue (enqueued or dequeued), both will find that the former is not a member of the latter. However, only one of these causes must be allowed to insert the await queue onto the Ready Queue.

To prevent multiple insertions of an await queue onto the Ready Queue, we associate a flag with the await queue. Initially, the flag is raised (i.e., set to one). If a cause C_1 succeeds in lowering the flag via a TDR of one, it inserts the await queue with multiplicity *M*, where *M* is the number of tasks awaiting the event, onto the Ready Queue, raises the flag, and then returns (to its caller). Otherwise, C_1 assumes that the await queue is a member of the Ready Queue and attempts to increment the await queue's multiplicity via the RaiseMult primitive. If successful, C_1 returns. If not, then the await queue is not attached to the Ready Queue because either the cause C_0 that lowered the flag has not yet inserted the await queue onto the Ready Queue or it has and enough deletions have been performed on the await queue to drive the multiplicity to zero, resulting in the await queue being taken off of the Ready Queue. In either case, C_1 repeats the procedure described above, namely, it first tries to insert the await queue onto the Ready Queue, and if it cannot, then it attempts to increment the await queue's multiplicity.

It is possible for one cause C_1 to raise an await queue's flag after it has inserted the await queue onto the Ready Queue, and then for another cause C_2 to lower the flag before the await queue has been deleted from the Ready Queue. In this case we do not want C_2 to do an insertion of the await queue onto the Ready Queue (as this would result in two instances of the await queue being on the Ready Queue). Therefore, after C_2 lowers the flag, it attempts to increment the multiplicity of the await queue. If the await queue is still a member of the Ready Queue, it will succeed and no insertion will be done. Otherwise, it will insert the await queue onto the Ready Queue.

The event data structure is extended to include a multiqueue entry and a flag for the await queue. Await queues are assumed to have a priority of *P*. The await and reset procedures are the same as those in the preceding section.

```
BQ_Event = record
        Val, Flag : integer;
        R_Sem     : Weak_Semaphore;
        AEntry    : MQEntry;
        AwaitQ    : Linked_Queue
    end;
```

```
Initially:
  Val              = 0;  if event has not occurred
  Val              < 0;  if event has occurred
  Flag             = 1;  Flag is raised
  R_Sem            = 1;  Resets may be done
  AEntry.List_Loc = nil;
  AwaitQ           = ∅;
```

```
procedure Cause (E : BQ_Event);
var v : integer;
  if E.Val ≥ 0 then                                E hasn't occurred
    v := F&A(E.Val, −MaxR);                        try to cause E
    if v > 0 then                                  v tasks waiting
      repeat
        if TDR(E.Flag, 1) then                     lower flag
          if not Linked_MQRaiseMult(address(E.AEntry), v, RMQ[P]) then
            Linked_MQInsert(address(E.AEntry), v, RMQ[P])
          end if;
          F&A(E.Flag, 1);                          raise flag
          return
        end if
      until Linked_MQRaiseMult(address(E.AEntry), v, RMQ[P])
    end if
  end if
end procedure Cause;
```

```
procedure Reset (E : BQ_Event);
  Weak_P(E.R_Sem);
    if E.Val < 0 then reset E
      E.Val := 0
    end if;
  Weak_V(E.R_Sem)
end procedure Reset;
```

```
procedure Await (E : BQ_Event);
    if E.Val ≥ 0 then                    E hasn't occurred
        if F&A(E.Val, 1) ≥ 0 then        wait for E
            Block(E.AwaitQ)
        end if
    end if
end procedure Await;
```

5.4.4. Unblocking Methods Compared

Two basic methods for unblocking *N* tasks have been given, where the tasks in question can be either readers waiting to use a resource or processes awaiting the occurrence of an event. Both methods are reviewed and compared below in the context of the readers/writers problem. Note however that the following discussion could be just as easily based on event processing by substituting tasks awaiting and causing an event for readers and writers of a resource, respectively.

The first unblocking method allows readers that were blocked to help a writer unblock additional readers: A writer deletes readers from a wait queue and inserts them onto the Ready Queue. When a reader is deleted from the Ready Queue and executed, it, like the writer, deletes other readers from the wait queue and inserts them onto the Ready Queue. This unblocking activity continues until a total of *N* readers have been moved from the wait queue to the Ready Queue. It is possible for a writer to unblock all *N* readers by itself, e.g., when there are no available PEs to execute the recently unblocked readers or when *N* is small. Let us refer to the case where a writer performs all of the wait queue deletions as as *M1W* and to the other case, where readers do aid a writer, as *M1RW*.

The second method for unblocking *N* readers transfers a read wait queue to the Ready Queue through a single multiqueue insertion. More specifically, a writer inserts a read wait queue with multiplicity *N* onto a priority *P* multiqueue component $RMQ[P]$ of the Ready Queue. When a PE performs a $RMQ[P]$ deletion and obtains a reference to a queue, it then performs a read wait queue deletion to obtain a task *T*. If *T*'s priority *p* is lower than *P*, i.e., $p < P$, where larger values represent higher priorities, then *T* is inserted onto $RMQ[p]$. Otherwise, the PE executes *T*. (See GetTask, section 5.3.2.6, for details.) The fewest number of queue operations are performed when all tasks on a read wait queue are of priority *P* or higher, and the greatest number are performed when the tasks' priorities are all less than *P*. Let us refer to these two cases as *M2HP* and *M2LP*, respectively.

To compare the cases, we compute the minimum time for N blocked readers to commence execution when one and N PEs are available (corresponding to heavy and light system loads), based on the total time required to execute each case's queue and/or multiqueue operations. We make the following simplifying assumptions: N is not greater than *#PEs*, the total number of processors; each queue/multiqueue operation is done in unit time; for our comparisons the time to execute tasks is ignored; the read wait queues and the queue and multiqueue components of the Ready Queue are of size $S = 2^h \leq \#PEs$, i.e., each one contains S, a power of two, linked lists, and S does not exceed the number of PEs.

For *M1W*, a minimum of $3N$ time units is required to unblock and resume N tasks when only one PE is available: N deletions are performed on the wait queue, followed by N insertions and deletions from the Ready Queue. When N PEs are available but only one task, the writer, accesses the wait queue, then the minimum time for N readers to be unblocked and to begin execution is $2N$+1 time units: the writer requires time $2N$ to delete the readers from the read wait queue and insert them onto the Ready Queue. A task added to the Ready Queue at a time t is deleted from the queue at time t+1. Therefore, after the writer unblocks the tasks, one additional step is needed to delete the Nth reader from the Ready Queue.

For *M1RW*, the minimum time for N blocked readers to begin execution is $3N$ time units when there is one available PE (this is identical to the one PE case of *M1W*). When N PEs are available the minimum time for N readers to be unblocked and resumed is at least $2+2\lceil \log \min(N, 2S-1)\rceil + \lceil (\max(N, 2S-1)-(2S-1))/S \rceil$ time units. Let us begin by explaining the presence of the logarithmic term. Suppose the number of tasks that perform deletions from a queue Q of size $S = 2^h$ doubles every time step. Then, at step t=1+log S, S tasks are deleting from Q. At this point having additional tasks perform Q deletions becomes ineffective as the queue can only support S concurrent operations. In other words, the number of deletions performed on Q doubles each time step until a total of $2^t - 1 = 2S - 1$ deletions have been performed. From this point on, the maximum number of deletions performed in a given time step equals S. In the case of *M1RW* this means that the number of readers deleted from the read wait queue roughly doubles every 2 time steps[63] until M readers have been deleted from the wait queue, where M is the minimum of

[63] This is as actually an upper bound as readers deleted from the wait queue must still be inserted and deleted from the Ready Queue before they can help unblock other readers.

N and $2S-1$. More precisely, letting $D(t)$ equal the number of read wait queue deletions performed during a time step t, we have the recurrence

$$
\begin{aligned}
&D(0) = 0\\
&D(1) = 1\\
&D(2) = 0\\
&D(t) = D(t-2) + D(t-3) \quad t \geq 3
\end{aligned}
$$

This is arrived at as follows: A writer *W* deletes a task *A* from a read wait queue at a time *t*, inserts *A* onto the Ready Queue at time t+1, deletes a task *B* from the read wait queue at time t+2, inserts *B* onto the Ready Queue at time t+3, etc. *A* is deleted from the Ready Queue at time t+2, and it then deletes a task *C* from the read wait queue at time t+3, inserts *C* onto the Ready Queue at time t+4, etc. From here on, *A* acts exactly like *W*. Thus, after a task (e.g., *W*) does a deletion from the read wait queue at a time *t*, it does another one at time t+2. Each task deleted from the read wait queue at time *t* (e.g., *A*) deletes another task from this queue at time t+3.

When $N > 2S-1$, the first $2S-1$ readers are deleted from the read wait queue as described above. The remaining $N-(2S-1)$ readers are deleted in $\lceil (N-(2S-1))/S \rceil$ time steps, where each step consists of *S* concurrent deletions (except the last, which consists of $N-(2S-1)$ modulo *S* concurrent deletions).

Regardless of whether *N* is greater than $2S-1$ or not, tasks that are deleted from the read wait queue during a time *t* are inserted onto the Ready Queue at time $t+1$, and then deleted from the Ready Queue at time $t+2$. Thus, after the *N*th task has been deleted from the wait queue, two additional queue operations must be executed for it to resume execution.

For *M2LP*,[64] the minimum time for *N* blocked readers to begin execution is 1+4*N* units when one PE is available. In this case all of the queue and multiqueue operations are performed serially. When *N* PEs are available, $4+\lceil N/S \rceil$ time units are required for *N* readers to be resumed: the read wait queue is inserted onto a multiqueue of the Ready Queue as an item with multiplicity *N*; it is then concurrently accessed by *N* multiqueue deletions; like *M1RW*, $\lceil N/S \rceil$+2 time steps are required to delete the *N* readers from the wait queue and then insert and delete them from the Ready Queue. That is, a set of readers deleted from the read wait queue at a time *t* are inserted

[64] Recall that for this case the read wait queue is inserted onto a multiqueue component of priority *P* of the Ready Queue and that each task on the wait queue has a priority lower than *P*. Thus, the tasks have to be deleted from the wait queue and reinserted onto queue components of the Ready Queue that correspond to their priorities.

onto queue components of the Ready Queue that correspond to their priorities at time $t+1$ and then deleted from the Ready Queue at time $t+2$. Thus, after the Nth reader is deleted from the wait queue, two additional queue operations must be performed for it to be resumed.

For *M2HP*,[65] the minimum time for N blocked readers to be resumed when only one PE is available is $1+2N$ units: all of the queue and multiqueue operations are executed serially. When N PEs are available $2+\lceil N/S \rceil$ time units are required: the read wait queue is inserted onto a multiqueue of the Ready Queue as an item with multiplicity N; it is then concurrently accessed by N multiqueue deletions; finally, $\lceil N/S \rceil$ sets of S[66] concurrent wait queue deletions are done.

Table 5-2 summarizes the timings. *M1RW* requires time logarithmic in N or the size of the wait queue (which ever is smaller) before the maximum number of readers that can aid a writer is reached. This ‘‘start-up’’ cost is the major difference between this case and the two cases of the second method. For all three of these cases the size of the wait queue is the main bottleneck.

The average performance of the second method, i.e., when roughly half of the tasks on a wait queue have to be reinserted onto the Ready Queue, appears to be no worse than that of the first, and could possibly be better, especially when the size of the wait queue is larger than N. More exact comparisons of these methods could be obtained by parameterizing the number of

Minimum Times to Unblock and Resume N Tasks*		
Method	Number of PEs	
	1	N
M1W	$3N$	$1+2N$
M1RW	$3N$	$2+2\lceil \log \min(N, 2S-1) \rceil + \lceil (\max(N, 2S-1)-(2S-1))/S \rceil$†
M2LP	$1+4N$	$4+\lceil N/S \rceil$
M2HP	$1+2N$	$2+\lceil N/S \rceil$

* (measured in units of queue operations, where S is the queue size)
† (lower bound)

Table 5-2.

[65] Recall that unlike *M2LP*, each task of the read wait queue has a priority of P or higher. Thus, they do not have to be reinserted onto the Ready Queue after they are deleted from the wait queue.

[66] The last set consists of N modulo S deletions.

available PEs and the average time required to execute each task.

5.5. Chapter Summary

In this chapter we have presented parallel implementations for a number of task and resource activities. Examples of the former included task creation, scheduling, (normal) termination, and destruction. Examples of the latter included P/V, resource request/release, and cause/await/reset operations for, respectively, semaphores, reader/writers coordination, and events. Also presented were algorithms for the parallel unblocking of multiple tasks waiting to use a resource.

Our strategy for parallelizing the above activities is to use a self-service approach with respect to the PEs and to place shared data into concurrently accessible data structures. We permit each PE to service its own requests, instead of using central servers, to avoid service bottlenecks. Data shared among the PEs are held in data structures that can be accessed in parallel to prevent access to such information from becoming a new source of bottlenecks. For example, the system Ready Queue is constructed from concurrently accessible queues and multiqueues, and other resource wait queues are constructed from concurrently accessible queues. In addition, to support the parallelization of certain Ready Queue operations, three types of Ready Queue multiqueue objects have been defined: templates, queues tagged with multiplicity, and queue pointers tagged with multiplicity.

The PE self-service approach in conjunction with the concurrently accessible Ready Queue and wait queue representations, and the multiqueue objects permit activities such as task creation, scheduling, and unblocking to be done concurrently: First, to create M tasks in parallel, a template with multiplicity M is inserted onto a multiqueue component of the Ready Queue. Each deletion performed on this item by an available PE results in the creation of a task which may then be executed by the deleting PE. Second, since the system Ready Queue is concurrently accessible, task scheduling can be done in parallel by multiple PEs. Third, the unblocking of N tasks on a wait queue can be accomplished either by having multiple tasks delete tasks from the wait queue and insert them onto the Ready Queue or by having one task insert the entire wait queue as a single item (of type queue tagged with multiplicity or queue pointer tagged with multiplicity) with multiplicity N onto the Ready Queue.

Certain activities such as task termination are parallelized, in part, via fetch-and-adds on shared counters. Another characterization of our parallelization techniques is that the tasks that are associated with an activity participate in its execution. For example, a task to be destroyed recursively signals its children to destroy themselves. Another example is the first method cited above for unblocking tasks, where tasks that were blocked unblock other tasks. We also comment that in the case of destruction, the queue remove operation and the multiqueue LowerMult operation of, respectively, chapters 2 and 3, are used to remove tasks and templates from queues and multiqueues (and without reducing the concurrency of either queue structure).

We have also compared our parallel methods for creating, destroying, and unblocking tasks against serial alternatives. The comparisons suggest that for large numbers of tasks, the parallel methods should be chosen.

CHAPTER 6

High-level Parallel Constructs

In this chapter the implementations of high-level parallel programming constructs are considered. We believe that stating parallelism and synchronization through *closed constructs*, specifically parallel analogs of loops and compound statements, facilitates the production and maintenance of reliable programs. While this style should be adequate for most applications, explicit programming with the task primitives (e.g., Create, InitiateTemplate, WaitForChildren) may be necessary to obtain more precise control of parallelism.

First, the closed constructs are presented and translated into the task primitives of chapter 5. Next, shared variables are discussed in the context of block-structured languages. Then, performance issues are considered, and finally a method for dynamically optimizing the number of tasks assigned to the execution of a construct is given.

6.1. The Parallel Compound Statement

The parallel compound statement consists of a set of inhomogeneous statements that may be executed in parallel. It has appeared often in the past (e.g., Dijkstra [68a], Brinch Hansen [73b], Wijngaarden et al. [76]). One possible syntax for this construct is:

$$\textbf{parbegin}\ s_1;\ s_2;\ \ldots\ s_N\ \textbf{parend}$$

Statements s_1 through s_N may be executed concurrently. After all of the statements within the compound statement have executed, the statement following the **parbegin** is performed.

A task executes a compound statement of N statements by creating N subtasks, initiating them, and then waiting for them to terminate. We first show how a task indirectly creates its children, and then how a direct creation is done. Recall from chapter 5 that the first method has available PEs create a parent's children from a template and that the second method has a parent create each child itself.

Prior to executing the compound statement, a parent task *P* initializes a compiler generated variable that is used by the subtasks (the children of *P*) performing the compound statement to determine which of the construct's statements are executed. *P* then creates a template *t* whose *CPUState* includes a pointer to a code block (described below) that each task created from *t* is to execute. After *P* inserts *t* with multiplicity *N* onto the Ready Queue (via InitiateTemplate), it waits for the children to be created and executed. After all children have terminated, *P* continues its execution at the point following the **parend**. The code segment for *P* is:

```
j := 1;          j is shared among the children of P
t := Create(Template, Name, Priority, MMap, CPUState);
InitiateTemplate(t, N);
WaitForChildren;
```

The code block executed by each child created from *t* is given below. The compound statement is translated into a sequential **case** statement, where the *i*th clause of the **case** corresponds to statement s_i of the **parbegin**. A child selects a clause by incrementing a shared index.

```
ProcessSignals;
Allocate(localmem);   generic memory allocation procedure
case F&A(j, 1) of
1                                         :  s1;
2                                         :  s2;
                                              .
                                              .
                                              .
N                                         :  sN
end case;
Terminate;
```

Each child can allocate its own local memory (for its private variables, etc.) via one of the (bottleneck-free) memory acquisition methods given in chapter 4. Alternatively, the children can subdivide concurrently a contiguous region *R* of the parent's address space.[1] For example, as in the parallel first-fit method, a child could allocate a subregion *r* of size *s* via

$$r := \mathrm{F\&A}(V, s),$$

where shared variable *V* points to *R*. One policy not advocated is for the parent to serially allocate a separate memory region for each child before enqueueing the template onto the Ready

[1] In this case the parent would have to allocate *R* prior to the children's initiation if it would otherwise lack the necessary space.

Queue since this requires execution time linear in N.

Following the tasking conventions of chapter 5, a child, prior to executing the **case**, processes any pending signals, and, after evaluating a clause, terminates (at which time its local memory is deallocated). The Nth child to terminate will attempt to awaken the parent.[2]

If instead a direct creation is desired, a parent P executes:

```
j := 1;        j is shared among the children of P
for k := 1 to N do
    t := Create(Task, Name, Priority, MMap, CPUState);
    Resume(t)
end for;
WaitForChildren;
```

The code block for the children remains the same.

6.2. The Parallel Loop

The parallel loop consists of a set of homogeneous statements that may be executed in parallel. Like the parallel compound statement, it has often been proposed as a basic structure for parallel programming languages (e.g., Gosden [66], Droughon et al. [67], Lundstrom and Barnes [80], and Davies [81]). A possible syntax for this loop is

$$\textbf{forall } j := lb \textbf{ to } ub \textbf{ do } s;$$

The number of iterations n performed over s equals $ub - lb + 1$. (We assume that j cannot be modified within s.) Thus, n instances of s may be concurrently executed, where each instance uses a different value of j (ranging from lb to ub). After all of the instances have been executed, the statement following the **forall** is begun.

The parallel loop's implementation is very similar in style to that of the parallel compound statement. To do an indirect creation of size n, a parent task P initializes a shared index to the loop's lower bound, creates and enqueues a template t with multiplicity n (where n equals the number of loop iterates) onto the Ready Queue, and then waits for its children to terminate. P's actions are summarized below:

---

[2] Recall from chapter 5 that it is possible for all N children to terminate before the parent executes its wait. In this case the parent immediately returns from WaitForChildren and is not awakened by the Nth child.

```
k := lb;   k is shared by P's children
n := ub – lb + 1;
t := Create(Template, Name, Priority, MMap, CPUState);
InitiateTemplate(t, n);
WaitForChildren;
```

Each child created from *t* executes the following code block.

```
ProcessSignals;
Allocate(localmem);
j := F&A(k, 1);
s;
Terminate;
```

For a parent to perform a direct creation of its children it executes

```
k := lb;
n := ub – lb + 1;
for i := 1 to n do
    t := Create(Task, Name, Priority, MMap, CPUState);
    Resume(t)
end for;
WaitForChildren;
```

The code block for the children is the same as the one for indirect creation.

6.3. Shared Variables

Support for completely unstructured sharing of data segments between arbitrary subsets of k processes could require 2^k separate segments. In practice, sharing patterns will probably exhibit more structure. Let us assume a conventional block-structured language augmented with closed constructs for parallelism (e.g., loops and compound statements), in which the usual scope rules apply: a variable can be accessed by any statement within the scope of its declaration. In particular, if that statement is a parallel statement, the variable will be shared by all tasks executing the statement. It follows that the status of a variable (private or shared) changes through the execution of the code.

As an example, consider the following code fragment (assuming function arguments are passed by reference):

```
var x : integer;
parbegin
    var y : integer;
    f(x, y); g(x, y)
parend;
```

Prior to the **parbegin**, x is private; while the functions f and g are executing, x is shared (by f and g); at the termination of the parallel clause x becomes private again.[3] The variable y is private. Note also that the status of x and y (private or shared) is implicitly defined by static scoping rules and can be computed at compile time.

6.4. Performance

The parallel constructs described above each represent an n-way fork in control. Potentially all of the subtasks generated from such constructs could be concurrently evaluated. Whether or not this actually occurs at run-time is a function of the availability of processors, the scheduling policy, the presence of synchronization, and other specific implementation details of the programming language and the operating system.

The basic unit of parallelism provided by the closed constructs is the program statement. A task could consist of just one simple assignment, but this would be extremely inefficient: the overhead incurred in scheduling such a task is large compared to its execution time. An efficient system must not initiate processes that do not execute a significant amount of code, and the system implementor must be careful to minimize overhead. In many cases, the compiler can detect cases where the granularity of parallelism is too small for efficient execution, and perform straightforward code transformations, e.g., short parallel blocks may be replaced with serial blocks. More interesting is the *chunking* of a parallel loop (Edler et al. [85]): a loop specifying the parallel execution of n iterations is replaced by a nested loop that specifies the parallel execution of n/k iterations, each consisting of the serial execution of k iterations. Such transformations may be delayed until run-time where the degree of chunking can be set according to the number of available processors.[4]

[3] This dynamic behavior impacts the paging mechanism (if present) and cache scheme as discussed, respectively, in Teller [88] and McAuliffe [86].

[4] The performance of such chunking policies has been examined by Kruskal and Weiss [85], who argue that even very naive schemes can perform acceptably well.

6.5. Dynamic Task Partitioning of a Construct

Let us consider a chunking variation where the number of tasks assigned to a construct *and* the number of clauses within the construct that each task evaluates is determined while the construct is being executed.

The parallel implementations above initiate the maximum number N of tasks that may be able to execute a construct concurrently. Each task executes one statement (a clause or iteration) of a construct and then terminates. The degree of parallelism actually realized can vary considerably, depending upon the structure and size (execution time) of these statements. If a construct is free of busy-waiting synchronization (i.e., the construct's statements do not coordinate their execution through busy-waits), then one task could serially execute all N statements, or N tasks could concurrently execute one statement each, or some number of tasks in between these two extremes could each serially execute a number of statements in parallel with one another. On the other hand, if $M \leq N$ statements of a construct do perform busy-waiting synchronization, at least M tasks must be created to execute it.

In the absence of busy-waiting within the statements, the number of tasks assigned to a construct's execution can be done dynamically. Let the maximum number N of tasks be initiated. However, when a task completes a statement, it tries to evaluate another one, instead of terminating. A construct may then be executed by from 1 to N tasks.

One criticism of this solution is that the execution of a construct can be completed while the associated template t, with a remaining multiplicity of $M < N$, is still on the Ready Queue. In this case, we wish to avoid having M tasks created needlessly from t to execute the already completed construct. Therefore, when a task finds that all of a construct's clauses have been assigned to other tasks, it removes the template from the Ready Queue and deducts the template's remaining multiplicity M from the *#Children* count of the template's parent to simulate the termination of M children. The code block for the parallel compound statement is amended below to include this optimization. The parallel loop statement may be similarly modified.

```
ProcessSignals;
Allocate(localmem);
i := F&A(j, 1);
while i ≤ N do
    case i of
    1                                                    :  s1;
    2                                                    :  s2;
                                                             .
                                                             .
                                                             .
    N                                                    :  sN
    end case
    i := F&A(j, 1)
end while;
if i = N+1 then   all case statements initiated
    t := InternalNameOfTemplate;[5]
    if Linked_MQLowerMult(t, MaxMult, RMQ[t↑.Priority], M) then
        F&A(t↑.Parent↑.Children, –M)
    end if
end if;
Terminate;
```

6.6. Chapter Summary

In this chapter we have shown how high-level parallel constructs that form *N*-way forks in control (e.g., loops and compound statements) can be implemented via the task primitives of chapter 5. Two different implementations have been given. One implementation places *N* tasks onto the Ready Queue in O(*N*) time; whereas, the other implementation does so in constant time by placing a task template for the construct with multiplicity *N* onto the Ready Queue. We have also considered a chunking variation where the number of tasks assigned to a construct and the number of clauses within the construct that each task evaluates is determined while the construct is being executed. The efficiency of this method is aided by our ability to remove a construct's template from the Ready Queue (via the LowerMult primitive).

---

[5] Although not included in the *TCB* definition of chapter 5, a task can locate the template from which it was created by locating the template's internal name via the Name Table (as the template's name is a prefix of the task's own external name). Alternatively, the template's internal name could be included in the *TCB* definition.

CHAPTER 7

A Proof of a Readers/Writers Implementation

In this chapter we prove that a fair block-queue implementation of the readers/writers problem (similar to the one given in section 5.4.2.2.1) satisfies the following criteria:

P1: readers and writers are mutually exclusive;

P2: writers are mutually exclusive;

P3: readers and/or writers can not deadlock: if readers are waiting to use the resource then either a writer is using the resource or one will be using it within a finite time, and if writers are waiting to use the resource then either a reader or a writer is using the resource or one will be using it within a finite time.

P4: if a reader requests the resource when there are no writers active (i.e., no writer has requested but not yet released the resource), then it uses the resource within a finite time.

The first three criteria are *safety properties*, i.e., they state that something bad cannot happen, while the last criteria is a *liveness property*, i.e., it states that something good must eventually happen (Owicki and Lamport [82]). Note that we do not prove that the implementation satisfies the liveness property of fairness, although we do believe it holds.

In section 7.1, a state transition model and terminology describing this problem are given. In section 7.2, the block-queue implementation is presented. A direct proof of this implementation is not attempted due to the large number of reader/writer operations that may be concurrently and serially executed. Instead, in section 7.3, we augment the original implementation with additional variables that record resource state information and with critical code sections that ensure that updates to these variables are indivisible. The criteria given above are then restated as assertions defined over these new variables. These assertions are then shown to hold for the augmented implementation. Finally, in section 7.4, through a series of transformations, the criteria

P1–P4 for the original implementation are derived from the assertions on the augmented implementation.

7.1. Problem Model and Terminology

Let us begin by reviewing the basic block-queue readers/writers problem as described by Brinch Hansen [73b]. Two groups of tasks, readers and writers, share a resource. Writers require exclusive access to the resource, whereas multiple readers may use it simultaneously. In general, tasks *request*, *use*, and then *release* the resource. The request operation grants or denies a task use of the resource. If denied the resource, a task waits (on a queue) until permission is granted. The release operation indicates that a task has completed using the resource, and, in addition, if tasks are waiting, one or more of them are granted permission to use the resource (and are dequeued from a wait queue), provided that neither one of the mutual exclusion constraints (**P1** nor **P2**) is violated.

The readers/writers implementation of section 5.4.2.2.1 is fair, i.e., starvation-free: If readers are using the resource and writers are waiting, then when the readers are done, one of the waiting writers is granted the resource. Similarly, if a writer is using the resource and readers are waiting, then when the writer is done, all of the waiting readers are granted the resource. If there are only waiting writers when the writer releases the resource, then one of them is granted the resource.

Since writers have exclusive access, when a writer releases the resource, the resource becomes free. If other tasks are waiting, the writer grants permission to use the resource to a subset of these tasks. Multiple readers may use the resource simultaneously. Hence, when a reader i releases the resource and writers are waiting, i grants permission to use the resource to one of these writers only if i is the *last* reader to release the resource, i.e., if no other reader is using it. This ‘‘last reader’’ restriction is dictated by constraints **P1** and **P2**: First, if readers are using the resource, a waiting writer must not be given permission to use it too. Second, if all readers using the resource release it and more than one writer is waiting, only one may be granted permission to use the resource.

In our implementations, the read release and the test by which readers determine who is last are two separate atomic operations. The test, which in the following is referred to as the *last scheduler test* (the origin of the name will become apparent shortly), is performed by readers only if there exist writers whose resource requests have not yet been granted. A last reader grants one of these writers permission to use the resource. The first writer that requests the resource when readers are using it is responsible for reinitializing the last scheduler test, after which it waits for permission to use the resource. Between a writer's resource request and reinitialization of the test, all of the readers might release the resource and perform the test, which will then fail to identify a last reader since the test is uninitialized. If this occurs, there is no last reader, and the writer uses the resource without delay (i.e., it does not wait). We therefore refer to the writer's reinitialization of the test as a *rerequest* for the resource that is either granted or denied depending upon the progress of readers that are using the resource.

Reader/writer resource state transitions are illustrated in Figure 7-1, where the operations (shown as arc labels) are atomic.[1] Initially, all tasks are *inactive* vis-à-vis the resource: none have requested it. A task becomes *active* after it requests the resource. A writer becomes inactive after it releases the resource. A reader becomes inactive either by releasing the resource when there are no active writers, or, if writers are active at the time of the release, by performing the last scheduler test.

An inactive task becomes a *running* task (i.e., a task that uses the resource) if it is granted permission to use the resource when it requests the resource. An inactive task whose resource request is denied becomes a *waiting* task. A waiting task *t* becomes a running task after it is given permission to use the resource as a consequence of another task's resource release or last scheduler test (in the latter case, *t* must be a writer).

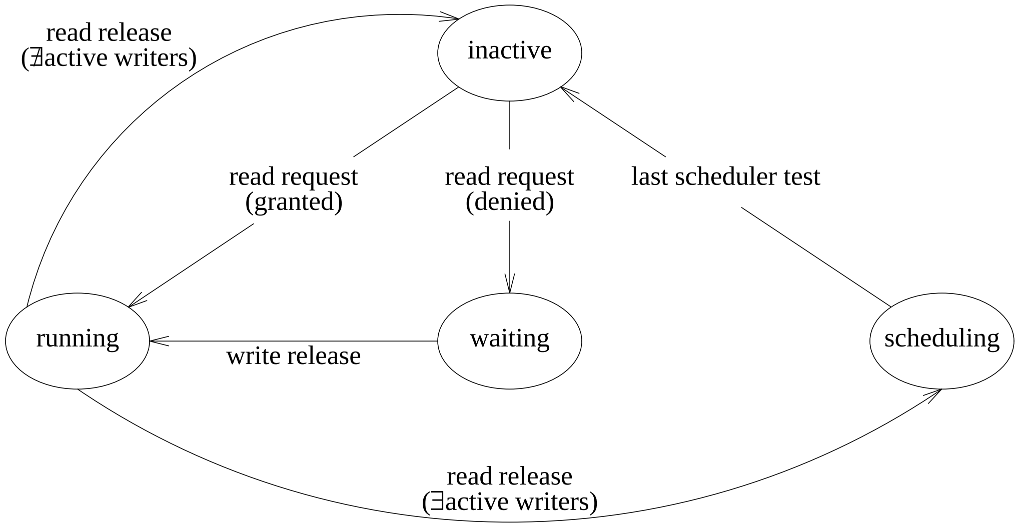
An inactive writer becomes a *migrating* writer when it requests the resource if the request is neither granted nor denied. A migrating writer becomes a waiting writer after it rerequests the resource and permission to access the resource is denied. Otherwise, it becomes a running writer.

A running reader becomes a *scheduling* reader when it releases the resource if there are migrating or waiting writers at the time of the release.[2] A scheduling reader becomes inactive after performing the last scheduler test.

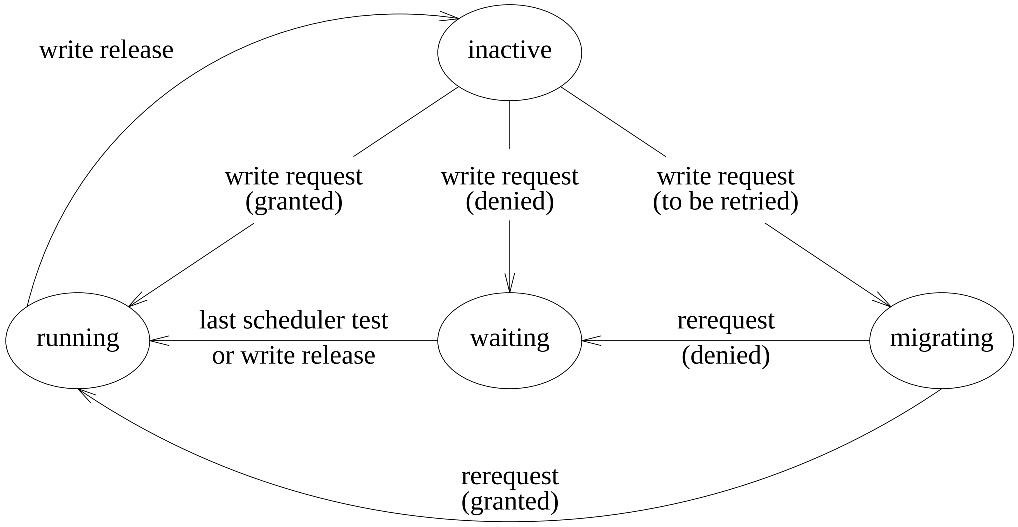
A task is an *operating* task if it has requested but not yet released the resource. Thus, active writers (i.e., running, waiting, and migrating writers) and running and waiting readers are

[1] The definitions of our model's task states are borrowed in large part from Brinch Hansen [73b].

[2] By property **P1**, there can not be any running writers.



a) Reader Transition Diagram



b) Writer Transition Diagram

Figure 7-1.

operating tasks, whereas inactive tasks and scheduling readers are not operating tasks.

Let us now describe in greater detail when a task is granted or denied permission to use the resource and the relationship between migrating writers and scheduling readers. An inactive reader's resource request is granted if there are no operating writers when the request is made;

otherwise, the request is denied and the reader becomes a waiting reader. An inactive writer's resource request is granted if there are no operating tasks when the writer makes its request. If there are operating writers when an inactive writer issues its request, then it becomes a waiting writer. Otherwise, if there are operating readers and no operating writers when an inactive writer requests the resource, then it becomes a migrating writer. Actually, when an inactive writer becomes a running or migrating writer, the set of operating readers consists only of running readers; there are no waiting readers. To be more precise,

Proposition 1. If there are no active (operating) writers, then there are neither scheduling nor waiting readers.[3]

At this point it is important to restate the functions of the last scheduler test and the rerequest operation, which are actually opposite sides of the same coin, in terms of task states. The last scheduler test indicates to a scheduling reader whether a migrating writer has rerequested the resource (which enables the test to identify a last scheduling reader) and become a waiting writer, and, if so, whether this reader is last, i.e., whether there are no running readers and this reader is the last scheduling reader. The rerequest operation tells a migrating writer whether there are running or scheduling readers.

A migrating writer *j*'s resource rerequest is granted if there are neither running nor scheduling readers when the rerequest is made. That is, for *j* to become a running writer, the rerequest must occur after all readers that were running at the time of *j*'s initial request have become inactive (by performing the last scheduler test). Otherwise, the rerequest is denied and the writer becomes a waiting writer.

When a scheduling reader performs the last scheduler test and finds that it is last, it grants a waiting writer permission to use the resource. In this case, a writer *j* requested the resource when there were running readers (and no active writers) and then rerequested it when there were running and/or scheduling readers.[4] As a result, *j* became a waiting writer. In summary, a running reader releases the resource and becomes inactive if there are no active writers. Otherwise, it becomes a scheduling reader, and if it is the last such reader, it grants the resource to a waiting

[3] This proposition is proved in section 7.3.2. See corollary 1.

[4] Recall that running readers become scheduling readers upon releasing the resource if there are migrating or waiting writers.

writer.

If there are waiting readers when a running writer *j* releases the resource, then *j* grants all of them permission to use it. If there are also waiting writers at the time of *j*'s release, then prior to granting the readers permission to use the resource, *j* reinitializes the last scheduler test as if readers were using the resource when a migrating writer had rerequested it and then had become one of the waiting writers. If there are only waiting writers when *j* releases the resource, then *j* grants one of them permission to use it.

7.2. The Algorithm

Algorithm 7-1 is a fair block-queue readers/writers implementation. It differs from the algorithm given in section 5.4.2.2.1 in that only a writer that releases the resource dequeues readers from the read wait queue, i.e., readers that were on the queue do not help dequeue other readers. Moreover, Algorithm 7-1, unlike the algorithm in section 5.4.2.2.1, does not permit two writers to unblock readers at the same time. These restrictions produce a solution to the readers/writers problem that is simpler but less parallel than the one given in section 5.4.2.2.1. We will prove that Algorithm 7-1 satisfies criteria **P1** through **P4**, and only conjecture that a similar proof may be constructed for the algorithm in section 5.4.2.2.1.

Algorithm 7-1 assumes that the maximum number of concurrent readers and writers is less than *MaxR* and that the set of task identifiers (or indices) is $\{1, 2, \ldots, MaxR{-}1\}$. Note that a task cannot be simultaneously a reader and a writer: if task *i* is a reader (i.e, is executing the reader code) and task *j* is a writer (i.e., is executing the writer code), then $i \neq j$.

RWaitQ and *WWaitQ* are read and write wait queues that contain, respectively, reader and writer tasks that are waiting for permission to use the resource. Tasks are inserted onto and deleted from these queues via Block and Unblock, which were defined in chapter 5: Block inserts a task onto a wait queue, and Unblock deletes a task from such a queue and inserts it onto the Ready Queue.[5] Operations on *RWaitQ* and *WWaitQ* are fair (i.e., insertions and deletions are performed within a finite time).[6] This is accomplished by using the ticket numbering method given

[5] We assume that tasks on the Ready Queue begin execution within a finite time.

[6] The readers/writers implementations of chapter 5 do not specify whether queue operations are fair or not. For Algorithm 7-1 we assume that they are fair to simplify our proofs of the readers/writers properties (**P1** through **P4**). We believe that one can prove that the fair readers/writers implementations of this chapter and of chapter 5 are starvation-free regardless of which queue implementation is used.

```
reader i:  L1:  v_i := F&A(Val, 1);                    request resource
           L2:  if v_i ≥ MaxR then                     waiting
           L3:      Block(RWaitQ)                      reader i blocks
           L4:  end if;
           L5:  ‘‘read’’                               use resource
           L6:  v_i := F&A(Val, –1);                   release resource
           L7:  if v_i > MaxR then                     ∃active writers
           L8:      c_i := F&A(Cnt, 1);                last scheduler?
           L9:      if c_i = MaxR – 1 then             yes
           L10:         Cnt := 0;
           L11:         repeat until Unblock(WWaitQ)
           L12:     end if
           L13: end if;

writer j:  M1:  v_j := F&A(Val, MaxR);                 request resource
           M2:  if v_j > 0 then                        resource in use
           M3:      if v_j < MaxR then                 ∃running readers
           M4:          c_j := F&A(Cnt, MaxR – v_j);   rerequest
           M5:          if c_j = v_j then              readers done
           M6:              Cnt := 0                   writer j runs
           M7:          else
           M8:              Block(WWaitQ)              writer j blocks
           M9:          end if
           M10:     else                               ∃a running writer k≠j
           M11:         Block(WWaitQ)                  writer j blocks
           M12:     end if
           M13: end if;
           M14: ‘‘write’’                              use resource
           M15: v_j := F&A(Val, –MaxR);                release resource
           M16: if v_j mod MaxR > 0 then               ∃waiting readers
           M17:     b_j := v_j mod MaxR;               # of waiting readers
           M18:     if v_j > 2*MaxR then               ∃waiting writers
           M19:         Cnt := MaxR–b_j
           M20:     end if;
           M21:     for 1 to b_j do
           M22:         repeat until Unblock(RWaitQ)
           M23:     end for
           M24: else
           M25:     if v_j ≥ 2*MaxR then
           M26:         repeat until Unblock(WWaitQ)
           M27:     end if
           M28: end if;
```

Initially: Cnt = *Val* = 0; *RWaitQ* = *WWaitQ* = ∅;

A Fair Block-queue Implementation of the Readers/Writers Problem
Algorithm 7-1.

in section 2.2.2.2 to generate ticket numbers (within the insertion and deletion procedures) that define the order in which items are inserted onto and deleted from a queue; specifically, tickets are numbered 0 through $MaxT - 1$, where $MaxT = MaxR^2$ and the wait queues (i.e., *RWaitQ* and *WWaitQ*) are of (queue) size *MaxR*. Also, Algorithm 7-1 assumes that blocked tasks (i.e., tasks on a wait queue) cannot be destroyed.[7]

Val and *Cnt* are shared variables that control the use of the resource. All subscripted variables in the algorithm are local to a given reader or writer.

The resource request and release operations described in the previous section are implemented as fetch-and-adds on *Val*. Specifically, L1, L6, M1, and M15 correspond to the read request, read release, write request, and write release operations. Similarly, the last scheduler test and the rerequest operation are performed as fetch-and-adds on *Cnt* at L8 and M4, respectively.

Val has an initial value of zero. It is incremented and decremented by 1 (*MaxR*) upon each read (write) request and release, respectively. It determines the total number of active writers ($\lfloor (Val/MaxR) \rfloor$) and running plus waiting readers (*Val* modulo *MaxR*).

Cnt is used to implement the last scheduler test and the rerequest operation and thereby to coordinate scheduling readers and migrating writers. When reading the following one should recall proposition 1, which states that if there are no active writers, then there are neither scheduling nor waiting readers. *Cnt* has an initial value of zero. If there exist $v_j > 0$ running readers when an inactive writer *j* requests the resource and becomes a migrating writer, then writer *j* rerequests the resource by incrementing *Cnt* by $MaxR - v_j$. If *Cnt* then equals *MaxR*, all running and scheduling readers are done, and writer *j* reinitializes *Cnt* to zero and becomes a running writer. Otherwise, writer *j* becomes a waiting writer. If there are active writers when a running reader releases the resource, it becomes a scheduling reader and performs the last scheduler test by incrementing *Cnt* by 1. If this raises *Cnt* to *MaxR*, this task *is* a last scheduling reader in which case it sets *Cnt* to zero and grants a waiting writer permission to use the resource. Finally, if there exist $b_k > 0$ waiting readers and at least one waiting writer when a writer *k* releases the resource, then *Cnt* is set to $MaxR - b_k$ (as if a migrating writer had rerequested the resource when b_k readers were using it and then had become a waiting writer) and b_k waiting readers are granted access to

[7] The task parameter specified for Unblock in chapter 5 is therefore unnecessary and has been omitted.

the resource.

Tasks undergo state transitions, as depicted in Figure 7-1, via fetch-and-adds on *Val* and *Cnt* as these two variables effectively define the state of the resource. Between a task's execution of any two fetch-and-adds on *Val* or *Cnt*, its flow of control is based (solely) on the (old) value of *Val* or *Cnt* returned by the most recent fetch-and-add executed. In particular, the actions of other tasks cannot alter this flow of control.

As a result of a task's flow of control being determined by the latest fetch-and-add it has executed, we make the following observations (assuming that each PE executes with only a finite delay between instructions): within a finite time of a task executing a read request (at L1) and becoming either a running reader or a waiting reader, it executes ''*read*'' (at L5) or blocks (at L3), respectively; within a finite time of a task executing a write request (at M1) and becoming either a running writer or a waiting writer, it executes ''*write*'' (at M14) or blocks (at M11), respectively; within a finite time of a task executing a write rerequest (at M4) and becoming either a running writer or a waiting writer, it executes ''*write*'' (at M14) or blocks (at M8), respectively; within a finite time of a task performing a last scheduler test (at L8) and being identified as a last scheduling reader, it executes Unblock (at L11); within a finite time of a task executing a write release (at M15) when there are waiting readers or waiting writers, it executes Unblock (at M22 or M26, respectively).[8]

When a task makes a waiting$\longrightarrow$running transition as the result of another task *i*'s resource release or last scheduler test, the waiting task is unblocked by *i* within a finite time. Observe that it is not known explicitly which waiting task makes this transition. To manage the resource, it is only necessary to know that some task is running, not which one.

Algorithm 7-1 does not permit us to identify (easily) the states of specific tasks. For example, as noted above, when a writer releases the resource and grants a waiting task permission to use the resource, it is unknown which waiting task has become a running task. The task's identity is not known until the task is deleted from the read wait queue. Also, the number of read and write requests can be determined from *Val*, but not which group (if any) is running. Therefore, before proving this algorithm correct, we will study another fair readers/writers algorithm,

[8] Unblock may have to be tried a (bounded) number of times before it is successful since this operation can be attempted between a task's resource request (e.g., at L1) and Block (e.g., at L3).

Algorithm 7-2, that has the same overall structure as Algorithm 7-1, but which explicitly manages the resource based on the states of the tasks.

7.3. The Augmented Algorithm

In this section we introduce Algorithm 7-2, a fair block-queue readers/writers implementation, and prove that it satisfies criteria **P1** through **P4**. Then, in the following section, through a series of transformations, Algorithm 7-2 is converted into Algorithm 7-1.

7.3.1. Overview

Algorithm 7-2 differs from Algorithm 7-1 in that the resource is managed using new shared state variables that explicitly record the states of tasks.[9] Task state transitions occur via fetch-and-adds on *Val* and *Cnt* in Algorithm 7-1 and via assignments to state variables in Algorithm 7-2. Basically, each fetch-and-add in the former is represented by a group of assignments in the latter. To keep state transitions atomic and to ensure that tasks do not act on inconsistent state variable values, all state variable references are performed within code segments that are defined to be mutually exclusive of one another. These segments are called *critical regions* (Brinch Hansen [73a]) and are delimited by **region–end region** pairs.[10] The fetch-and-adds of Algorithm 7-1 are retained in Algorithm 7-2 even though they could be deleted without affecting the correctness of the latter. They remain since it will later be proved that, with respect to managing the resource, *Val* and *Cnt* serve the same function as the state variables.

In Algorithm 7-2, subscripted scalar variables are local to tasks (e.g., v_i is local to task *i*). All other variables are shared. Queues (e.g., *RWaitQ*) and arrays (e.g., *ar*) are of size *MaxR*. Initially, the queues are empty and the remaining shared variables are zero.

In the algorithm and throughout the rest of the chapter, for an array *a*, we use the expressions $a = 0$ and $a \neq 0$ as shorthands for $(\forall i)\ (a[i] = 0)$ and $(\exists i)\ \ (a[i] \neq 0)$, respectively.

RIdQ and *WIdQ* are non-blocking queues of reader and writer task identifiers, respectively. That is, the queues contain values representing task indices. An *RIdQ* (*WIdQ*) insertion always

[9] Later, when Algorithm 7-2 is converted into Algorithm 7-1, these state variables will be replaced with auxiliary variables (Owicki [75]) that record the state of the resource but are not required for its management.

[10] Brinch Hansen [73a] associates a critical region with a specific shared variable. Regions referencing the same variable are mutually exclusive. In this context, we treat the set of array variables as a single shared variable; thus, all of our critical regions are mutually exclusive.

```
reader i:  R1:  region ‘‘read request’’
           R2:      if rw=ww=mw=0 then
           R3:          rr[i] := 1;  action_i := run
           R4:      else
           R5:          wr[i] := 1;  action_i := wait;
           R6:          my#_i := r#;  r# := r#+1 mod MaxT;
           R7:          QInsert(i, RIdQ)
           R8:      end if;
           R9:      ar[i] := 1;
          R10:      v_i := F&A(Val, 1)
          R11:  end region;
          R12:  if action_i = wait then
          R13:      Block(RWaitQ)
          R14:  end if;
          R15:  ‘‘read’’
          R16:  region ‘‘read release’’
          R17:      rr[i] := 0;
          R18:      if ww ≠ 0 or mw ≠ 0 then
          R19:          sr[i] := 1;  action_i := schedule
          R20:      else
          R21:          ar[i] := 0;  action_i := inactive
          R22:      end if;
          R23:      v_i := F&A(Val, –1)
          R24:  end region;
          R25:  if action_i = schedule then
          R26:      region ‘‘last scheduler test’’
          R27:          c_i := F&A(Cnt, 1);
          R28:          sr[i] := 0;  ar[i] := 0;
          R29:          if sr=rr=mw=0 and ww≠0 then ‘‘last’’
          R30:              action_i := grant_writing;
          R31:              QDelete(writerid_i, WIdQ);
          R32:              ww[writerid_i] := 0;  rw[writerid_i] := 1
          R33:          end if
          R34:      end region;
          R35:      if action_i = grant_writing then
          R36:          Cnt := 0;
          R37:          repeat until Unblock(WWaitQ)
          R38:      end if
          R39:  end if;
```

Algorithm 7-2a.

```
writer j:
region ''write request''
    if rr=wr=rw=ww=mw=0 then
        rw[j] := 1;  action_j := run
    elseif rw=ww=mw=0 then
        mw[j] = 1;  action_j := migrate;
    else
        ww[j] := 1;  action_j := wait;
        my#_j := w#;  w# := w#+1 mod MaxT;
        QInsert(j, WIdQ)
    end if;
    aw[j] := 1;
    v_j := F&A(Val, MaxR)
end region;
if action_j ≠ run then
    if action_j = migrate then
        region ''rerequest resource''
            c_j := F&A(Cnt, MaxR–v_j);
            if sr=rr=0 then
                rw[j] := 1;   action_j := run
            else
                ww[j] := 1;   action_j := wait;
                my#_j := w#;  w# := w#+1 mod MaxT;
                QInsert(j, WIdQ)
            end if;
            mw[j] := 0
        end region;
        if action_j = run then
            Cnt := 0
        else
            Block(WWaitQ)
        end if
    else
        Block(WWaitQ)
    end if
end if;
''write''
```

Algorithm 7-2b.

(*writer j continued*)

```
region ''write release''
    rw[j] := 0;  aw[j] := 0;
    if wr≠0 then
        action_j := grant_reading;
        a_j := ∑ wr[i];
        for 1 to a_j do
            QDelete(readerid_j, RIdQ);
            wr[readerid_j] := 0;
            rr[readerid_j] := 1
        end for
    else
        if ww≠0 then
            action_j := grant_writing;
            QDelete(writerid_j, WIdQ);
            ww[writerid_j] := 0;
            rw[writerid_j] := 1
        end if
    end if;
    v_j := F&A(Val, –MaxR)
end region;
if action_j = grant_reading then
    b_j := a_j;
    if v_j > 2*MaxR then
        Cnt := MaxR–b_j
    end if;
    for 1 to b_j do
        repeat until Unblock(RWaitQ)
    end for
else
    if action_j = grant_writing then
        repeat until Unblock(WWaitQ)
    end if
end if;
```

Initially: $ar = aw = rr = rw = wr = ww = mw = sr = 0;$
$RWaitQ = WWaitQ = RIdQ = WIdQ = \emptyset;$
$Cnt = Val = r\# = w\# = 0;$

Algorithm 7-2c.

succeeds. An *RIdQ* (*WIdQ*) deletion on a nonempty queue returns true and supplies a task index. If the queue is empty, the deletion returns false. As explained below, these queues bind together task state variables and waiting⟶running task state transitions.

It is assumed that if a task executes a non-blocking statement then the next statement will be executed by that task after a finite delay. No such assumption is made if the first statement is blocking.

RWaitQ and *WWaitQ* are queues of blocked tasks. Tasks are inserted onto and deleted from these queues via the Block and Unblock primitives, respectively. As in Algorithm 7-1, insertions and deletions are fair and are again based on a ticket numbering method. Deletion ticket numbers are still generated within the deletion procedure. Insertion ticket numbers, however, are generated prior to Block (which invokes the insertion procedure). Specifically, (at R6) a reader i assigns the current *RWaitQ* ticket number $r\#$ to $my\#_i$ and then increments $r\#$ by one (which is maintained modulo *MaxT*[11]); similarly, (at W8 or W22) a writer j assigns the current *WWaitQ* ticket number $w\#$ to $my\#_j$ and then increments $w\#$ by one (which is also kept modulo *MaxT*). Note that when a task k generates an insertion ticket number for a wait queue *WQ*, it inserts its identifier onto the corresponding identifier queue (e.g., at R7) and sets $action_k$ to *wait* (e.g., at R5). Within a finite time k performs Block (e.g., at R13) as $action_k$ equals *wait*. Task k will be inserted onto *WQ* when the ticket number for the next task to be added to this queue equals $my\#_k$.

Assigning wait queue insertion ticket numbers and performing identifier queue insertions as described above allows us to identify which task on a wait queue is given permission to use the resource (i.e., which one has become a running task) when the resource is released (and an identifier queue deletion is performed). In contrast, Algorithm 7-1 does not perform this identification until a wait queue deletion is performed. (See claim 6 below.)

Regions correspond to reader/writer operations. In the following, a reader/writer task is said to be *using* the resource if and only if it is performing ‘‘read’’/‘‘write’’ (at R15/W36). The first reader region (at R1) corresponds to a read request operation. That is, after executing this region, the reader either uses the resource (at R15) or blocks (at R13). Similarly, a writer requests the resource via the first writer region (at W1). The region (at R16 or W37) executed by a task immediately after using the resource corresponds to a resource release operation. The region (at W16) labeled ‘‘rerequest’’ represents a write rerequest. It is executed only if a writer neither uses the resource (at W36) nor blocks (at W33) following its initial request (at W1). After a writer performs this region, it either uses the resource or blocks (at W30). Finally, the last scheduler

[11] Recall from section 7.2 that tickets are numbered 0 through $MaxT - 1$, where $MaxT = MaxR^2$.

test, which determines whether a reader should unblock a writer, is represented by the region (at R26) labeled ‘‘last scheduler test’’.

We now define task states with respect to the regions in much the same manner as task states were described in section 7.1 and depicted in Figure 7-1. However, our definitions apply only when all tasks are outside of regions: when a task executes a region, its state changes. Even when a task *i* is not executing a region, *i*'s state may not be defined since another task may be within a region that can alter *i*'s state. Specifically, when a reader (writer) task is in state waiting, its state can be changed to running via another task's execution of a write release region (last scheduler test or write release region).

Initially, all tasks are *inactive* with respect to the resource: none have requested it. An inactive task becomes *active* after it requests the resource (at R1 or W1). A writer becomes inactive after it releases the resource (at W37). A reader *i* becomes inactive by releasing the resource (at R16) if it will not then perform the last scheduler test (at R26), i.e., if the reader has set its local variable $action_i$ to *inactive* when doing the release. Otherwise, the reader becomes inactive when performing the test.

An inactive reader/writer *i* becomes a *running* reader/writer if $action_i$ has a value of *run* after the resource is requested (at R1/W1). An inactive reader/writer *i* becomes a *waiting* reader/writer if its identifier *i* is inserted onto *RIdQ*/*WIdQ* when it requests the resource. In addition, $action_i$ is set to *wait*. A waiting reader becomes a running reader when its identifier is deleted from *RIdQ* (at W43) as part of a write release. A waiting writer becomes a running writer when its identifier is deleted from *WIdQ* (at either R31 or W50) as part of a reader's last scheduler test or another writer's resource release.

A running reader *i* becomes a *scheduling* reader when releasing the resource (at R16) if $action_i$ equals *schedule*. A scheduling reader becomes inactive when the last scheduler test is performed (at R26).

An inactive writer *j* becomes a *migrating* writer when requesting the resource if $action_j$ equals *migrate*. A migrating writer *j* becomes a waiting writer when it rerequests the resource (at W16) if $action_j$ equals *wait*. Otherwise, it becomes a running writer as $action_j$ equals *run*.

Claim 1. Arrays *ar*, *aw*, *rr*, *rw*, *wr*, *ww*, *sr*, and *mw* are binary valued.

Proof. By inspection of Algorithm 7-2. □

Claim 2. *RIdQ*/*WIdQ* deletions always succeed, i.e., an identifier queue is never empty when a deletion is performed on it.

Proof. Initially, all queues are empty and all variables are zero. Consider *WIdQ*. Observe (1) that when a task j inserts its identifier onto *WIdQ* (at W9 or W23), it also sets $ww[j]$ to one (at W7 or W21, respectively), and (2) that no other task but j can set $ww[j]$ to one. A task i performs a *WIdQ* deletion (at R31 or W50) only if there exists a k such that $ww[k]$ is one (which it determines by testing *ww* at R29 or W48), which implies that identifier k is on *WIdQ*. After i deletes an identifier l from *WIdQ*, $ww[l]$ is set to zero. Hence, *WIdQ* deletions always succeed. By a similar argument it can be shown *RIdQ* deletions always succeed. □

The following claim gives the relationship between task states and the state variables of Algorithm 7-2. The claim is valid only when all tasks are executing outside of regions since within regions state variables and task states can change.

Claim 3. The following state array elements reflect task states (when no region is being executed):

task i is an active reader iff $ar[i]=1$;
task i is an active writer iff $aw[i]=1$;
task i is a running reader iff $rr[i]=1$;
task i is a running writer iff $rw[i]=1$;
task i is a waiting reader iff $wr[i]=1$;
task i is a waiting writer iff $ww[i]=1$;
task i is a scheduling reader iff $sr[i]=1$;
task i is a migrating writer iff $mw[i]=1$.

Proof. The claim is clearly valid when execution begins since all tasks are inactive and all variables are zero. Thus we must show that when a region is executed, the claim remains valid. There are many cases to consider but most are simple: a task i changes state and sets its state variables appropriately. The exceptions occur when a task i removes a task j from *RIdQ* or *WIdQ*. In these cases task i sets the state variables of task j. However, the waiting$\longrightarrow$running transitions are defined to occur precisely when a task identifier is removed from an identifier queue so the claim remains valid for these region executions as well.[12] □

Claim 4. Within a finite time after a task becomes a waiting reader/writer, the task itself (i.e., the corresponding *TCB*) is inserted onto *RWaitQ*/*WWaitQ*.

Proof. When a task *i* inserts its identifier onto an identifier queue *IdQ* (e.g., at R7), making it a waiting task, it also sets $action_i$ to *wait* (e.g., at R5). Hence, within a finite time, *i* performs Block (e.g., at R13) and inserts itself onto the wait queue that corresponds to *IdQ*. □

Claim 5. The order in which tasks become waiting readers/writers corresponds to the order in which they (i.e., their *TCB*s) are inserted onto *RWaitQ*/*WWaitQ*.[13]

Proof. When a task *i* generates an insertion ticket number for a wait queue *WQ* (e.g., at R6), it also inserts its identifier onto the identifier queue *IdQ* that corresponds to *WQ* (e.g., at R7), making *i* a waiting task. These two operations are executed only within regions. Hence, *i* receives ticket number *n* if and only if its identifier is the *n*th item inserted onto *IdQ*. Wait queue insertions are based on ticket numbers. Therefore, if identifier *i* is the *n*th item inserted onto the *IdQ*, task *i* will be the *n*th item inserted onto *WQ*. □

Claim 6. Within a finite time after a waiting task becomes a running task, the task itself (i.e., its *TCB*) is deleted from *RWaitQ*/*WWaitQ*.

Proof. First recall that when a task identifier *k* is inserted onto or deleted from a wait queue, task *k* becomes a waiting or running task, respectively. Now suppose that *N WIdQ* deletions have been performed (at R31 and/or W50) by *N* tasks[14] and that the *N*th deletion removed an identifier *j* from *WIdQ*. This implies that *N* writers, including writer *j*, have inserted their identifiers onto *WIdQ* and, within a finite time of their insertions, these *N* writers block in the order that they became waiting writers (by claims 4 and 5). Within a finite time after each task *i* performs a *WIdQ* deletion, *i* cycles performing *WWaitQ* deletions via Unblock (at R35 or W67) until one succeeds, as $action_i$ was set previously to *grant_writing* (at R30 or W49). Therefore, within a

[12] As indicated, the definition of the waiting$\longrightarrow$running transitions makes claim 3 easy. But we must remember that a task may be in state running even though it is blocked. The key point (claim 6 below) is that such a task will become unblocked within a finite time.

[13] While not needed to prove that Algorithm 7-2 satisfies criteria **P1** through **P4**, we note that one can prove (using claims 4, 5, and 6) that the order in which waiting tasks become running tasks is the same as the order in which the corresponding *TCB*s are deleted from wait queues.

[14] For the proof these *N* tasks need not be distinct, i.e., a task may execute more than one pair of *WIdQ*/*WWaitQ* deletions.

finite time after the Nth task performs a *WIdQ* deletion, the first N writers inserted onto *WWaitQ*, one being task j, are deleted.[15] An argument similar to the one above can be given to prove that within a finite time after an identifier l is deleted from *RIdQ*, task l is deleted from *RWaitQ*. □

Let us return briefly to the last scheduler test and rerequest regions. A last scheduling reader is identified by inspecting the scheduling reader, running reader, migrating writer, and waiting writer arrays, which is consistent with the description of this test given in section 7.1.[16] Running writers are not checked for, as the test does not concern itself with such tasks, and their absence is assumed when there are scheduling readers.[17] A write rerequest is granted if there are no running or scheduling readers, as indicated by the running and scheduling reader arrays. The absence of running writers and other migrating writers is assumed.[18]

Recall from section 7-1 that if there exist $n>0$ waiting readers when a writer releases the resource, then all n readers are granted permission to use the resource. In Algorithm 7-2, when a writer j releases the resource, *RIdQ* contains the identifiers of all waiting readers (by definition), and the number of identifiers on this queue equals $\sum wr[i]$ (by claim 3). Therefore, from an inspection of the write release region we see that j grants all waiting readers permission to use the resource. Similarly, the last scheduler test and the write release regions grant permission to use the resource to the writer that has been waiting the longest. In contrast, Algorithm 7-1 grants the resource to the task(s) at the front of a wait queue, which may not be the oldest waiting task(s). In converting Algorithm 7-2 into Algorithm 7-1 we show that this difference does not cause Algorithm 7-1 to violate any one of the four criteria stated in the beginning of this chapter.

The following claims will be used below to establish that if certain assertions hold, then Algorithm 7-2 satisfies criteria **P1** through **P4**. Also, certain claims (specifically, 7, 8, 11, and 12) refer to a task as being *at* a particular region. By this we mean that the task is *outside* of the

---

[15] We believe that one can prove that a task that performs the Nth *WIdQ*/*RIdQ* deletion also performs the Nth *WWaitQ*/*RWaitQ* deletion. We show that this identifier queue–wait queue deletion sequence is true for a task that performs a *WIdQ* deletion via a last scheduler test or a *RIdQ* deletion via a write release. See the the proofs of claims 23 and 25.

[16] The check for waiting writers is actually unnecessary. In section 7.3 we prove that if there are scheduling readers, then there are migrating or waiting writers (see assertion **A3**). A last scheduling reader is identified, in part, by the absence of migrating writers. Therefore, there must be a waiting writer when there is a last scheduling reader.

[17] This assumption is proved in section 7.3. See assertion **A4**.

[18] This assumption is also proved in section 7.3. See assertion **A5**.

specified region and that the next statement to be executed by the task is within the region.

Claim 7. If a task i is at the last scheduler test (i.e., at R26), then it is a scheduling reader and $sr[i]=1$.

Proof. By inspection and claim 3. □

Claim 8. If a task j is at the write rerequest (i.e., at W16), then it is a migrating writer and $mw[j]=1$.

Proof. By inspection and claim 3. □

Claim 9. If a reader is using the resource, then it is a running reader.

Proof. Before a reader i uses the resource (at R15), it must first request the resource (at R1), which sets $action_i$ to either *run* or *wait*. As $action_i$ is local to reader i, its value is not altered (at R19 or R21) until reader i uses and then releases the resource.

CASE 1. Reader i is using the resource and $action_i = run$. Reader i became a running reader when it requested the resource since $action_i = run$. As i is using the resource, it has not yet performed the read release. Therefore, i is still a running reader.

CASE 2. Reader i is using the resource and $action_i = wait$. Since $action_i = wait$, reader i became a waiting reader when its identifier was inserted onto *RIdQ* (at R7) at the time it requested the resource. By claim 4, within a finite time of the request, i blocked (at R13), i.e., inserted itself onto *RWaitQ*. Now, suppose i was the Nth task to insert its identifier onto *RIdQ*. By claim 5, i was then the Nth task to be inserted onto *RWaitQ*. For i to be using the resource after being inserted onto *RWaitQ*, i had to have been deleted from *RWaitQ* (at W63). From an examination of Algorithm 7-2 we see that each *RWaitQ* deletion is preceded by an *RIdQ* deletion, and the latter is always successful (by claim 2). Therefore, when i was deleted from *RWaitQ*, N *RIdQ* deletions must have already been performed, and one of those deletions had to have removed identifier i from *RIdQ*. After i's identifier was deleted from *RIdQ*, it became a running reader. After i was deleted from *RWaitQ*, it returned from its block (at R13) and began using the resource. Since i has not yet released the resource, it is still a running reader. □

Claim 10. If a writer is using the resource, then it is a running writer.

Proof. Similar to claim 9. □

Claim 11. If a task i is at the read release (i.e., at R16), then it is a running reader and $rr[i]=1$.

Proof. Suppose a task i is using the resource (at R15). By claim 9, i is a running reader, and by claim 3, $rr[i] = 1$. After i uses the resource, it is at the read release. □

Claim 12. If a task j is at the write release (i.e., at W37), then it is a running writer and $rw[j] = 1$.

Proof. Similar to claim 11. □

Claim 13. A task uses the resource within a finite time of becoming a running reader.

Proof. A reader i becomes a running reader by making either an inactive⟶running or a waiting⟶running transition.

CASE 1. The inactive⟶running transition. Let an inactive reader i request the resource (at R1) and become a running reader; $action_i$ equals *run* (it is assigned this value at R3). Following the request, i does not block (at R13), as $action_i$ does not equal *wait*, but proceeds to use the resource.

CASE 2. The waiting⟶running transition. Let an inactive reader i request the resource and become a waiting reader. In making the waiting⟶running transition, i sets $action_i$ to *wait* (at R5) and inserts its identifier onto *RIdQ* (at R7). Within a finite time of requesting the resource, i blocks (at R13), i.e., inserts itself onto *RWaitQ*, by claim 4. Task i then becomes a running reader when its identifier is deleted from *RIdQ* via Unblock (at W63) by a writer. By claim 6, within a finite time after this *RIdQ* deletion, i itself is deleted from *RWaitQ*. Task i then returns from its block and uses the resource. □

Claim 14. A task uses the resource within a finite time of becoming a running writer.

Proof. Similar to claim 13. □

Claim 15. If there are waiting writers, running and/or scheduling readers, and neither migrating writers nor running writers (when no region is being executed), then, within a finite time, the waiting writer whose task identifier is at the front of *WIdQ* becomes a running writer via the last scheduler test, and, furthermore, this writer is the *next* task to become a running writer.

Proof. First note that after a task becomes a waiting writer, it either remains a waiting writer or becomes a running writer. Now, suppose that there are waiting writers, M running readers, N scheduling readers, and neither migrating nor running writers. Let task j's identifier be at the front of *WIdQ*, i.e., task j is the next waiting writer to become a running writer.

Let us assume that reader i is the last of the M running readers to release the resource. Note that each one of the M running readers that releases the resource while j is a waiting writer becomes a scheduling reader (as $ww \neq 0$ at R18). Now, during the time T_1 that j is a waiting writer and i is a running reader, no writer can become a migrating writer or a running writer: While there are waiting writers, all inactive tasks that request the resource become waiting tasks as $ww \neq 0$, and while there are running readers, no waiting writer can become a running writer via the last scheduler test as $rr \neq 0$. Also, during T_1, no task can become a running reader and, consequently, no task (other than one of the original M running readers) can become a scheduling reader: Since there are no running writers during T_1, no waiting reader can become a running reader, and any inactive reader that requests the resource will become a waiting reader since there is a waiting writer. Therefore, when task i performs the read release, it becomes one of $L \leq M+N$ scheduling readers, there is a waiting writer j, and there are no migrating writers, running writers, or running readers.

Let us assume that task k is the last of the L scheduling readers to perform the last scheduler test. During the time T_2 that j is a waiting writer and k is a scheduling reader, no writer can become a migrating writer or a running writer: While there are waiting writers, all inactive tasks that request the resource become waiting tasks as $ww \neq 0$, and while there are scheduling readers, no waiting writer can become a running writer via the last scheduler test as $sr \neq 0$. Also, during T_2, no task can become a running reader and, consequently, no task can become a scheduling reader: Since there are no running writers during T_2, no waiting reader can become a running reader, and any inactive reader that requests the resource will become a waiting reader since there is a waiting writer. Therefore, when task k performs the last scheduler test, it deletes writer identifier j from *WIdQ* (at R31) as there are waiting writers ($ww \neq 0$) and there are neither migrating writers ($mw = 0$), running readers ($rr = 0$), nor scheduling readers ($sr = 0$). Thus, waiting writer j becomes a running writer after the test is performed by k. Finally, we note that both T_1 and T_2 are finite: running reader i uses the resource within a finite time (by claim 13), after which it becomes a scheduling reader when it releases the resource; scheduling reader k, by inspection of the algorithm, becomes inactive within a finite time. □

Claim 16. If there is a migrating writer and there are neither scheduling readers nor running tasks (when no region is being executed), then, within a finite time, the migrating writer becomes a running writer.

Proof. Suppose that a task j is a migrating writer and there are neither scheduling readers nor running tasks. While j is a migrating writer, no inactive task can become a running task, and, consequently, no task can become a scheduling reader since all inactive tasks that request the resource become waiting tasks (as $mw \neq 0$). Hence, when j rerequests the resource, $action_j$ is set to *run* (at W19), as there are neither running readers ($rr = 0$) nor scheduling readers ($sr = 0$), and j becomes a running writer. □

Claim 17. If there are running and/or scheduling readers, a migrating writer, and no running writers[19] (when no region is being executed), then, within a finite time, there are either (a) running and/or scheduling readers, waiting writers, and neither migrating writers nor running writers or (b) a migrating writer and neither scheduling readers nor running tasks.

Proof. Suppose there exists a migrating writer j, M running readers, N scheduling readers, and no running writers. During the time T that j is a migrating writer, no inactive or waiting task can become a running task: While there is a migrating writer, any inactive task that requests the resource becomes a waiting task (as $mw \neq 0$), and no scheduling reader can cause a waiting writer to become a running writer via the last scheduler test (again since $mw \neq 0$). Also, since we have assumed that there are initially no running writers, no waiting reader can become a running reader during T. Consequently, while j is a migrating writer, no new reader can become a running reader, and only readers from among the M running readers can become scheduling readers (when releasing the resource). Clearly, j, the M running readers, and the N scheduling readers perform, respectively, the rerequest (at W16), the read release (at R16), and the last scheduler test (at R26) regions within a finite time. The regions are executed serially as they are mutually exclusive. If j performs its region before all $M + N$ readers perform their respective regions, then j's identifier is inserted onto *WIdQ* (at W23), as $rr \neq 0$ and/or $sr \neq 0$, and j becomes a waiting writer, giving (a). On the other hand, if the M running readers perform read releases, becoming scheduling readers, and they and the original N scheduling readers perform the last scheduler test before j performs its rerequest, then there is no last scheduler, as $mw \neq 0$, giving (b). □

Property **P4** states that if a reader requests the resource when there are no writers (i.e., no writer has requested but not yet released the resource), then it uses the resource within a finite time. Observe in Algorithm 7-2 that when an inactive reader requests the resource, it becomes a running reader (and, by claim 13, uses the resource within a finite time), if there are no running, waiting, or migrating writers, i.e., if no writer has requested but not yet released the resource.

[19] In section 7.3.3, we prove that migrating writers and running writers are mutually exclusive (see assertion **A5**).

Hence, Algorithm 7-2 satisfies **P4**. Proving that the algorithm also satisfies the remaining readers/writers properties (**P1** through **P3**) is harder and is the subject of the following sections.

7.3.2. The Assertions

Claim 18. If the following assertions hold whenever no region is being executed, then Algorithm 7-2 satisfies criteria **P1**, **P2**, and **P3**.

(1) *Running readers and running writers are mutually exclusive*:

$$\mathbf{I1} \equiv \sim(\sum_i rr[i]>0 \quad \sum_i rw[i]>0)$$

(2) *Running writers are mutually exclusive*:

$$\mathbf{I2} \equiv \sum_i rw[i] \leq 1$$

(3) *The system is deadlock-free*:

$$\mathbf{I3} \equiv \mathbf{[}\sum_i wr[i]>0 \qquad \sum_i rw[i]>0 \quad \sum_i mw[i]>0$$
$$((\sum_i rr[i]>0 \quad \sum_i sr[i]>0) \quad \sum_i ww[i]>0)\mathbf{]}$$
$$\mathbf{[}\sum_i ww[i]>0 \quad \sum_i rw[i]>0 \quad \sum_i mw[i]>0 \quad \sum_i sr[i]>0 \quad \sum_i rr[i]>0\mathbf{]}$$

Proof. Recall from the preceding section that task states are defined to be valid only when all tasks are outside regions. As no region contains a blocking statement or an unbounded loop, it follows that if there exists a task *i* executing a region *R*, then one possible execution sequence is for all other tasks to temporarily delay their executions until *i* exits *R*. We shall use this observation in proving the claim.

(a) **P1** states that readers and writers can not use the resource at the same time. Assume **P1** is false. This means that there exists a reader *i* and a writer *j* such that the first is reading, i.e., performing ‘‘*read*’’ (at R15), and the second is writing, i.e., performing ‘‘*write*’’ (at W36). Note that *i* and *j* are outside of regions and by the observation above we may assume the other tasks are also outside regions.

$$\sim\mathbf{P1} \equiv (\exists i,\ j \mid i \text{ is reading and } j \text{ is writing})$$
$$(\exists i,\ j \mid i \text{ is a running reader and } j \text{ is running writer}) \quad \text{(by claims 9, 10)}$$
$$(\sum_i rr[i]>0 \quad \sum_i rw[i]>0) \quad \text{(by claim 3)}$$
$$\sim\mathbf{I1}$$

(b) Similarly, ~**P2**, the mutual exclusion criterion for writers, implies ~**I2**.

(c) **P3** states that if readers are waiting to use the resource then either a writer is using the resource or one will be using it within a finite time, and if writers are waiting to use the resource then either a reader or a writer is using the resource or one will be using it within a finite time. In other words, as long as whenever tasks are waiting, the resource is being used or will be used within a finite time, there is no deadlock.[20] The waiting readers and waiting writers implications differ because the resource can be shared simultaneously by readers. Hence, readers should not be waiting if there are no writers even if other readers are using the resource.

Let us define a task as ‘‘waiting for the resource’’ if its resource request has been denied and the task has not yet been granted permission to use the resource. This is equivalent to a task being in state waiting: If a task requests (or rerequests) the resource and becomes a waiting task, it blocks on a wait queue and does not use the resource; its request is denied. When a waiting task becomes a running task, it uses the resource[21] (by claims 13 and 14); its request is granted.

Let us recall certain sets of task states from which it follows that the resource is in use or will be in use within a finite time. First, by claims 13 and 14, if there are running tasks, then the resource is being used or will be used within a finite time. Second, by claim 15, if there are waiting writers, running and/or scheduling readers, and neither migrating writers nor running writers, then, within a finite time, a waiting writer becomes a running writer. Third, by claim 16, if there is a migrating writer and there are neither scheduling readers or running tasks, then, within a finite time, the migrating writer becomes a running writer. Fourth, by claim 17, if there are running and/or scheduling readers and a migrating writer and no running writers, then, within a finite time, either the migrating writer becomes a waiting writer, which maps into the second case, or the running and scheduling readers become inactive, which maps into the third case. In summary then, the resource is being used or will be used within a finite time if there are running tasks, or migrating writers, or running and/or scheduling readers and waiting writers.

$$\mathbf{I3} \equiv \ \mathbf{[}\sum_i wr[i]>0 \quad \sum_i rw[i]>0 \quad \sum_i mw[i]>0$$

$$((\sum_i rr[i]>0 \quad \sum_i sr[i]>0) \quad \sum_i ww[i]>0)\mathbf{]}$$

$$\mathbf{[}\sum_i ww[i]>0 \quad \sum_i rw[i]>0 \quad \sum_i mw[i]>0 \quad \sum_i sr[i]>0 \quad \sum_i rr[i]>0\mathbf{]}$$

$$\mathbf{[}(\exists i \mid i \text{ is a waiting reader})$$

$$(\exists j \mid j \text{ is a running or migrating writer})$$

[20] This does not preclude starvation.

[21] Recall that a task is using the resource if it is performing either ‘‘*read*’’ (at R15) or ‘‘*write*’’ (at W36).

($\exists k,\ l \mid k$ is a running or scheduling reader and l is a waiting writer)**]**

[($\exists j \mid j$ is a waiting writer)

($\exists k \mid k$ is a running task, or a migrating writer, or a scheduling reader)**]**

[($\exists i \mid$ reader i is waiting to use the resource)

($\exists j \mid j$ is writing or will be writing within a finite time)**]**

[($\exists j \mid$ writer j is waiting to use the resource)

($\exists k \mid k$ is reading or writing or k will be writing within a finite time)**]**

P3 □

It will facilitate the inductive proof of these three assertions to prove at the same time that the following auxiliary assertions hold when no region is being executed:

(1) *The number of active readers equals the number of running readers plus the number of waiting readers plus the number of scheduling readers*:

$$\mathbf{A1} \ \equiv \ \sum_i ar[i] = \sum_i (rr[i] + wr[i] + sr[i])$$

(2) *The number of active writers equals the number of running writers plus the number of waiting writers plus the number of migrating writers*:

$$\mathbf{A2} \ \equiv \ \sum_i aw[i] = \sum_i (rw[i] + ww[i] + mw[i])$$

(3) *If there are scheduling readers then there exist waiting or migrating writers:*

$$\mathbf{A3} \ \equiv \ \sum_i sr[i]>0 \quad \sum_i ww[i]>0 \quad \sum_i mw[i]>0$$

$$\equiv \ \sum_i sr[i]=0 \quad \sum_i ww[i]>0 \quad \sum_i mw[i]>0$$

(4) *Scheduling readers and running writers are mutually exclusive*:

$$\mathbf{A4} \ \equiv \ \sim\!(\sum_i sr[i]>0 \quad \sum_i rw[i]>0)$$

(5) *Migrating and running writers are mutually exclusive*:

$$\mathbf{A5} \ \equiv \ \sum_i mw[i]\le 1 \quad \sim\!(\sum_i mw[i]>0 \quad \sum_i rw[i]>0)$$

The first two auxiliary assertions require no explanation. The remainder characterize the function and underlying implications of the last scheduler test/rerequest operation as described in section 7.1. The third assertion **A3** states that if there exists a scheduling reader, then there must also exist a migrating or waiting writer. Otherwise, when there is a last scheduling reader, there would not be a waiting writer (as assumed) for the reader to grant the resource to. As a last scheduling reader grants a waiting writer the resource, the fourth assertion **A4** states that running writers and scheduling readers can not coexist. Otherwise, multiple writers could use the resource simultaneously. For the last assertion we note that an inactive writer becomes a migrating writer if there exist running readers and no active writers at the time of the request. Since running and migrating writers are also active writers, **A5** states that migrating writers and running writers are mutually exclusive.

In sections 7.1 and 7.2 when describing several state transitions (e.g., inactive writer$\longrightarrow$migrating writer) we cited proposition 1, which states that if there are no active writers then there are no waiting or scheduling readers. We now remark that this last fact follows from **I3**, **A2**, and **A3**:

Corollary 1. **I3** **A2** **A3** $\sum_i aw[i]=0 \quad \sum_i wr[i]=0 \quad \sum_i sr[i]=0.$

7.3.3. The Proof

The assertions above reference only the auxiliary variables of Algorithm 7-2. All assignments and references to these variables are within critical regions that have been defined as indivisible computational units. Therefore, for our purposes, an execution of Algorithm 7-2 can be represented by the sequence or history H of executed regions, where H=[($region_1$, $task_{j_1}$), ($region_2$, $task_{j_2}$), ... , ($region_n$, $task_{j_n}$)]; $region_i \in$ {the read request region (at R1), the read release region (at R16), the last scheduler test region (at R26), the write request region (at W1), the write rerequest region (at W16), the write release region (at W37)}; and $1 \le j_i < MaxR$. An assertion **A** then holds over Algorithm 7-2 if for every execution history H, **A** is true before and after each region in H is executed.

For convenience we repeat all of the assertions below and define **I** to be their conjunction.

$\mathbf{A1} \equiv \quad \sum ar[i] = \sum (rr[i] + wr[i] + sr[i])$

$\mathbf{A2} \equiv \sum aw[i] = \sum (rw[i] + ww[i] + mw[i])$

$\mathbf{A3} \equiv \sum sr[i] = 0 \quad \sum ww[i] > 0 \quad \sum mw[i] > 0$

$\mathbf{A4} \equiv \sim(\sum sr[i] > 0 \quad \sum rw[i] > 0)$

$\mathbf{A5} \equiv \sum mw[i] \leq 1 \quad \sim(\sum mw[i] > 0 \quad \sum rw[i] > 0)$

$\mathbf{I1} \equiv \sim(\sum rr[i] > 0 \quad \sum rw[i] > 0)$

$\mathbf{I2} \equiv \sum rw[i] \leq 1$

$\mathbf{I3} \equiv [\sum wr[i] > 0 \quad \sum rw[i] > 0 \quad \sum mw[i] > 0$

$\qquad ((\sum rr[i] > 0 \quad \sum sr[i] > 0) \quad \sum ww[i] > 0)]$

$\quad [\sum ww[i] > 0 \quad \sum rw[i] > 0 \quad \sum mw[i] > 0 \quad \sum sr[i] > 0 \quad \sum rr[i] > 0]$

$\mathbf{I} \equiv \mathbf{A1} \quad \mathbf{A2} \quad \mathbf{A3} \quad \mathbf{A4} \quad \mathbf{A5} \quad \mathbf{I1} \quad \mathbf{I2} \quad \mathbf{I3}$

Also, in the claims that follow, we continue to use the notation $a = 0$ and $a \neq 0$ as shorthands for $(\forall i)\ (a[i] = 0)$ and $(\exists i)\ (a[i] \neq 0)$, respectively.

Claim 19. **I** holds over Algorithm 7-2: For each history H=[($region_1$, $task_{j_1}$), ($region_2$, $task_{j_2}$), ... , ($region_n$, $task_{j_n}$)] of Algorithm 7-2 and $\forall$($region_i$, $taskj_i$)$\in H$, **I** is true before and after $region_i$ is executed by $taskj_i$.

Proof. Before $region_1$ is executed, **I** is true since all state variables are initially zero. Inductively assume that **I** holds over the first k–1 regions of H and that task i executes $region_k$. We must show that **I** holds after $region_k$.

(1) Let $region_k$ be the read request region.

  (a) **A1** holds after the read request region: Task i becomes an active reader (and sets $ar[i]$ to one at R9) and either a running or a waiting reader (and sets $rr[i]$ or $wr[i]$ to one at R3 or R5, respectively) upon executing the read request region.

  (b) **A2**, **A3**, **A4**, **A5**, and **I2** hold after the read request region: no writer state variable or scheduling reader state variable is modified within this region.

  (c) **I1** holds after the read request region: Task i becomes a running reader (and sets $rr[i]$ to 1 at R3) only if there are no running writers ($rw = 0$ at R2).

  (d) **I3** holds after the read request region: Suppose that there are no waiting readers when i requests the resource and becomes a waiting reader. Then from an examination of the code it is evident that before i executes the region, there are running, migrating, or waiting writers. If there are running or migrating writers, then **I3** holds after i exits the region. If there are only waiting writers, then (using the induction hypothesis that **I3** holds before $region_k$), there are also running or scheduling readers since

$$\textbf{I3} \quad rw = 0 \quad mw = 0 \quad ww \neq 0 \quad sr \neq 0 \quad rr \neq 0.$$

Therefore, if there are only waiting writers when i requests the resource, then after it completes the request, it is a waiting reader and there are running or scheduling readers, and, consequently, **I3** is true. For all other task state scenarios, **I3** is clearly true after i completes the region.

(2) Let $region_k$ be the read release region.

(a) **A1** holds after the read release region: Task i becomes either an inactive reader (and sets $rr[i]$ and $ar[i]$ to zero at R17 and R21, respectively) or a scheduling reader (and sets $rr[i]$ and $sr[i]$ to zero and one, respectively, at R17 and R19) upon executing this region.

(b) **A2**, **A5**, and **I2** hold after the read release region: no writer state variable is modified within this region.

(c) **A3** holds after the read release region: Task i becomes a scheduling reader (and sets $sr[i]$ to one at R19) only if there exists a migrating or waiting writer ($mw \neq 0$ or $ww \neq 0$ at R18) when it releases the resource.

(d) **A4** and **I1** hold after the read release region: When task i is at the read release region, by claim 11, it is a running reader ($rr[i] = 1$), and by the induction hypothesis, **I1** holds, which implies that there are no running writers as **I1** $rr[i] = 1$ $rw = 0$.

(e) **I3** holds after the read release region: Task i becomes an inactive reader (and sets $rr[i]$ and $ar[i]$ to zero at R17 and R21, respectively) if there are neither waiting nor migrating writers ($ww = 0$ and $mw = 0$ at R18) when it releases the resource; otherwise, it becomes a scheduling reader (and sets $rr[i]$ and $sr[i]$ to zero and one at R17 and R19, respectively).

(3) Let $region_k$ be the last scheduler test region.

(a) **A1** holds after the last scheduler test region: Task i becomes an inactive reader (and sets $sr[i]$ and $ar[i]$ to zero at R28) when performing this region.

(b) **A2** holds after the last scheduler test region: When task *i* performs this region, it either causes a waiting writer *j* to become a running writer (and sets $ww[j]$ and $rw[j]$ to zero and one, respectively, at R32) or does not affect the state of any writer.

(c) **A3**, **A4**, **A5**, **I1**, and **I2** hold after the last scheduler test region: When task *i* is at this region, by claim 7, it is a scheduling reader ($sr[i]=1$), and by the induction hypothesis, **A3** and **A4** hold. This implies that there exists a waiting or a migrating writer and no running writer when *i* is at the last scheduler test as

$$\mathbf{A3}\quad \mathbf{A4}\quad sr[i]=1\quad (ww\neq 0\quad mw\neq 0)\quad rw=0.$$

Task *i* causes a waiting writer *j* to become a running writer (and sets $ww[j]$ and $rw[j]$ to zero and one, respectively, at R32) only if *i* is the last scheduling reader and there are neither running readers nor migrating writers ($sr=rr=mw=0$ at R29). Thus, **A3**, **A4**, **A5**, **I1**, and **I2** are true after *i* executes this region.

(d) **I3** holds after the last scheduler test region: As in (c) we have that when task *i* is at the last scheduler test,

$$\mathbf{A3}\quad \mathbf{A4}\quad sr[i]=1\quad (ww\neq 0\quad mw\neq 0)\quad rw=0.$$

If there is a migrating writer, then **I3** is obviously true after *i* executes this region. On the other hand, suppose there is a waiting writer and no migrating writer. If *i* is the last scheduling reader to perform the last scheduler test and there are no running readers ($sr=rr=mw=0$ at R29), a waiting writer *j* becomes a running writer ($rw[j]=1$ at R32), which implies that **I3** holds after the region. Otherwise, no such transition occurs as there are other running or scheduling readers, but as there are waiting writers after *i* completes the region, **I3** must hold.

(4) Let $region_k$ be the write request region.

(a) **A1** holds after the write request region: this region does not affect the states of reader tasks.

(b) **A2** and **A3** hold after the write request region: Task *i* becomes an active writer (and sets $aw[i]$ to one at W11) and either a running, migrating, or waiting writer (and sets $rw[i]$, $mw[i]$, or $wr[i]$ to one at W3, W5, or W7, respectively) upon executing the write request region.

(c) **A4** holds after the write request region: When task i is at this region, by the induction hypothesis, **A3** holds, and if a task j is a scheduling reader ($sr[j]=1$), then there are waiting or migrating writers as

$$\mathbf{A3}\quad sr[j]=1 \quad ww \neq 0 \quad mw \neq 0.$$

In this case, i cannot become a running writer (see the test at W2).

(d) **A5** holds after the write request region: Task i becomes a migrating writer (and sets $mw[i]$ to one at W5) only if there are no running, waiting, or migrating writers ($rw = ww = mw = 0$ at W4).

(e) **I1** and **I2** hold after the write request region: Task i becomes a running writer (and sets $rw[i]$ to one at W3) only if there are no running or waiting readers and no running, waiting, or migrating writers ($rr = wr = rw = ww = mw = 0$ at W2).

(f) **I3** holds after the write request region: Suppose that i requests the resource when there are no waiting writers and it becomes one. This implies (from an examination of the write request region) that when the request is made, there are running or waiting writers, and, therefore, after the request is completed, **I3** is true. For all other task state scenarios, **I3** is clearly true after i completes the region.

(5) Let $region_k$ be the write rerequest region.

(a) **A1** holds after the write rerequest region: this region does not affect the states of reader tasks.

(b) **A2** holds after the write rerequest region: When task i as at this region, by claim 8, $mw[i]=1$. Upon executing this region, i becomes either a running writer (and sets $rw[i]$ and $mw[i]$ to one and zero at W19 and W26, respectively) or a waiting writer (and sets $ww[i]$ and $mw[i]$ to one and zero at W21 and W26, respectively).

(c) **A3**, **A4**, and **I1** hold after the write rerequest region: Task i becomes a running writer if there are neither scheduling or running readers ($sr = rr = 0$ at W18); otherwise, i becomes a waiting writer.

(d) **A5** and **I2** hold after the write rerequest region: When task i is at this region, $mw[i]=1$, and, by the induction hypothesis, **A5** holds. This implies that i is the only migrating writer and there are no running writers as

$$\mathbf{A5}\quad mw[i]=1 \quad \sum mw[k]=1 \quad \sum rw[k]=0.$$

After i executes this region, it is a waiting or running writer. Hence, **A5** and **I2** hold after the write rerequest region.

(e) **I3** holds after the write rerequest region: Suppose there are no waiting writers when i rerequests the resource and becomes one. This implies (from an examination of the region) that there are scheduling or running readers before and after the rerequest is made. Hence, **I3** holds after i exits the region. For all other task state scenarios, **I3** is clearly true after i completes the region.

(6) Let $region_k$ be the write release region.

(a) **A1** holds after the write release region: By definition, a task is a waiting reader if and only if its task identifier is on *RIdQ*. When i executes this region, each task j whose identifier is on this queue becomes a running reader (and $wr[j]$ and $rr[j]$ are set to zero and one at W44 and W45, respectively).

(b) **A2** holds after the write release region: When i is at this region, by claim 12, $rw[i]=1$. By performing this region, i becomes inactive (and sets $rw[i]$ and $aw[i]$ to zero at W38). Also, if there are no waiting readers and there are waiting writers when i executes this region, then the first task j whose identifier is on *WIdQ* becomes a running writer (and $ww[j]$ and $rw[j]$ are set to zero and one at W51 and W52, respectively).

(c) **A3** and **A4** hold after the write release region: When i is at this region, $rw[i]=1$, and by the induction hypothesis, **A4** holds. This implies that there are no scheduling readers before i executes this region as

$$\textbf{A4}\;\; rw[i]=1 \;\; sr=0$$

and thus no task can become a scheduling reader via this region's execution.

(d) **A5**, **I1**, and **I2** hold after the write release region: When i is at this region, $rw[i]=1$, and, by the induction hypothesis, **A5**, **I1**, and **I2** hold. Thus, i is the only running writer and there are no running readers or migrating writers as

$$\textbf{A5}\;\;\textbf{I1}\;\;\textbf{I2}\;\; rw[i]=1 \;\; \sum rw[k]=1 \;\; \sum rr[k]=0 \;\; \sum mw[k]=0.$$

In performing this region, i becomes inactive and either all waiting readers become running readers or the first waiting writer becomes a running writer (or neither occurs if there are no waiting tasks).

(e) **I3** holds after the write release region: As in (d), after i executes the region, i becomes inactive and either all waiting readers become running readers or the first waiting writer becomes a running writer (or neither occurs if there are no waiting tasks). If there were both waiting readers and waiting writers before i released the resource, then immediately afterwards there are running readers, waiting writers, and no waiting readers. If there were only waiting readers before the release, then immediately afterwards there are no waiting tasks. If there are only waiting writers, say M in number, before the release, then immediately afterwards there is a running writer and $M-1$ waiting writers. In each case, **I3** holds after i completes the release. □

Recall that the goal of this chapter is to prove the correctness of Algorithm 7-1, i.e., that it satisfies the four basic readers/writers criteria given at the beginning of the chapter. Up to this point we have shown that a related algorithm (7-2) satisfies various other properties. Now, using these later results as tools, we proceed toward the original objective.

7.4. The Transformations

In this section we establish relationships between the state variables of Algorithm 7-2 and the original variables, *Val* and *Cnt*, of Algorithm 7-1. Then, based on these relationships, Algorithm 7-2 is converted into Algorithm 7-1 without invalidating the already proven readers/writers properties of the former.

Recall that Algorithms 7-1 and 7-2 as well as the block-queue implementations of chapter 5 assume that the maximum number of tasks is bounded by *MaxR*:

Assumption 1. When no region is being executed,

$$MaxR > \sum (rr[i] + wr[i] + sr[i] + rw[i] + ww[i] + mw[i]).$$

The relationship between *Val* and the state variables of Algorithm 7-2 is given by the following claim.

Claim 20. When no region is being executed,

$$Val = \sum (rr[i] + wr[i]) + (\sum (rw[i] + ww[i] + mw[i])) * MaxR.$$

Proof. Initially, *Val* and all state variables are zero. When an inactive task becomes a running or waiting reader (running, waiting, or migrating writer) via a read (write) request, the task sets the corresponding state variables and increments *Val* by one (*MaxR*). When a running task releases the resource, it again sets the corresponding state variables and decrements *Val* by one (*MaxR*). When a task executes a last scheduler test or a write rerequest, neither *Val* nor the sums of the state arrays change. □

Claim 21. When no region is being executed,

$$MaxR > Val > 0 \quad (rr \neq 0 \quad wr \neq 0) \quad rw = ww = mw = 0.$$

Proof. By assumption 1 and claim 20. □

Claim 22. When no region is being executed,

$$Val \geq MaxR \quad rw \neq 0 \quad ww \neq 0 \quad mw \neq 0.$$

Proof. By assumption 1 and claim 20. □

The following claims (23 through 29) establish the relationship between the variable *Cnt* and the task state variables of Algorithm 7-2. In the following we refer to a reinitialization of *Cnt* (at R36, W28, and W60) by the region that immediately precedes it. For example, statement R36 is the last scheduler test reinitialization of *Cnt*.

Claim 23. If a task is at the last scheduler test reinitialization of *Cnt* (at R36) when no region is being executed, then there is a running writer.

Proof. Suppose a task i performs the last scheduler test and sets $action_i$ to *grant_writing*, as this is the only way that a task can reach R36. Furthermore, let us assume that i deletes an identifier j from *WIdQ* (at R31), making j a running writer. Just after i completes the last scheduler test, j is the only running writer (by claim 19 and assertion **I2**) and it cannot use the resource (and then become an inactive writer via a write release) until after it is deleted from *WWaitQ*.

Suppose that after i performs the last scheduler test but before it performs a *WWaitQ* deletion via Unblock (at R37), another task k deletes j from *WWaitQ* via Unblock (at either R37 or W67). This implies that prior to k's Unblock, k must have performed a last scheduler test or a write release and deleted an identifier l from *WIdQ* (at R31 or W50, respectively), making l a running writer. However, this implies that both j and l are running writers just after k completes the last scheduler test or write release region, which is a contradiction (by claim 19 and assertion **I2**). Therefore, no such k exists. Thus, when i is at R36, running writer j has not yet been deleted from *WWaitQ*. □

Claim 24. If a task is at the write rerequest reinitialization of *Cnt* (at W28) when no region is being executed, then there is a running writer.

Proof. Observe that if a writer j is at W28, then $rw[j]=1$. □

Claim 25. If a writer j is at the write release reinitialization of *Cnt* (at W60) when no region is being executed, then $a_j = \sum rr[i]$, there are running readers and waiting writers, and there are neither scheduling readers, migrating writers, nor running writers.

Proof. First observe that a task j performs the write release reinitialization of *Cnt* (at W60) only if $action_j$ equals *grant_reading* and v_j exceeds $2*MaxR$. Hence, when j was at W37, there were $N>0$ waiting readers and *Val* exceeded $2*MaxR$. Since at W37 j was the only running writer and there were no migrating writers (by claim 19 and assertions **I2** and **A5**), $Val>2*MaxR$ implies there were $M>0$ waiting writers (by claim 20 and assumption 1). Moreover at W37, $rw[j]=1$ (by claim 12) and there are no running or scheduling readers (by claim 19 and assertions **I1** and **A4**).

Immediately after j completes the write release, $v_j>2*MaxR$, $action_j = grant\_reading$, $a_j = N$, the N waiting readers have become running readers (as their identifiers have been deleted from *RIdQ*), there are $M>0$ waiting writers, and there are neither scheduling readers, migrating writers, nor running writers. Now j will proceed to reinitialize *Cnt* (at W60) and unblock a reader (at W63). We will now show that when j reinitializes *Cnt*, none of the N running readers has been unblocked, none of the M waiting writers has become a running writer, and no other task has become a running or scheduling reader or a migrating or running writer.

Let us assume that after j releases the resource but before it performs the unblock, another writer k unblocks a reader (at W63). This implies that when k released the resource, $a_k>0$ waiting readers became running readers. Suppose k's release followed j's release (releases are performed serially). Then, after j released the resource and just before k's release, k was a running writer and there were N running readers, which is a contradiction (by claim 19 and assertion **I1**). If the order of the releases were reversed, we would obtain the same contradiction (but with j and a_k running readers instead of with k and N running readers). Thus, no such k exists.

Hence, none of the N waiting readers that became running readers when j released the resource can be removed from *RWaitQ* until j performs an unblock. Furthermore, after j releases the resource but before it unblocks a reader

(1) none of the M waiting writers can become a running writer: If a task were to execute a region that caused a waiting writer to become a running writer, then immediately after this region was executed, there would be N running readers and a

running writer, which is a contradiction (by claim 19 and assertion **I1**);

(2) an inactive writer cannot become a running or migrating writer: A writer that requests the resource becomes a waiting writer since, by (1), there are (at least) M waiting writers;

(3) an inactive reader cannot become a running reader: A reader that requests the resource becomes a waiting reader since, by (1), there are (at least) M waiting writers;

(4) a waiting reader cannot become a running reader: By the argument just given above regarding writers j and k, there does not exist a writer k that can perform a write release and cause a waiting reader to become a running reader before j unblocks a reader;

(5) none of the N running readers can become a scheduling reader: Since j has not yet unblocked any one of these readers, none of them can perform a read release and become a scheduling reader.

We conclude that when j is at W60 (i.e., before it unblocks a reader) and all tasks are outside of regions, there are (at least) M waiting writers and neither scheduling readers nor migrating or running writers, and $a_j = N = \sum rr[i]$, that is, the set of running readers consists of exactly the N tasks that were waiting readers when j released the resource. □

Corollary 2. If a task is at the write release reinitialization of *Cnt* (at W60), then no other task may perform a write request region and become a running writer.

Proof. We have just shown in the proof above that after a running writer j releases the resource but before it unblocks a reader (at W63), an inactive writer cannot request the resource and become a running writer (by part (2) of claim 25's proof). The write release reinitialization statement of *Cnt* is performed by j after the release but before the unblock. □

Claim 26. *Cnt* is always nonnegative. Indeed, whenever *Cnt* is updated it is either assigned a nonnegative value or incremented (via fetch-and-add) by a positive value.

Proof. Initially, *Cnt* is zero. Three of the five updates to *Cnt* either increment it by one (at R27) or assign it a value of zero (at R36 and W28). Let us consider the remaining two updates. (1) When an inactive task j requests the resource and becomes a migrating writer, there are no other active writers and there are running or waiting readers, and therefore $0 < Val < MaxR$ (by claim 21). Then, when j rerequests the resource, it increments *Cnt* (at W17) by a positive value as $0 < v_j < MaxR$. (2) When a task performs a write release reinitialization of *Cnt* (at W60), b_j (which equals a_j) equals the number of running readers (by claim 25), which is less than *MaxR* (by assumption 1). Hence, at the write release reinitialization statement of *Cnt*, *Cnt* is assigned a positive value. □

Claim 27. If a writer is at the resource release (at W37), then *Cnt* is zero.

Proof. We begin by making the following observation. The claim is true if the following assertions hold for each region *R* that causes a task *i* to become a running writer and for the interval *I* beginning when *R* ends and ending when *i* arrives at the write release:

(1) *Cnt* is never assigned a positive value in *I*.

(2) Either *Cnt* is zero upon exiting *R* or *Cnt* is assigned a value of zero in *I*.

We shall now present subclaims that prove (1) and (2). Before proceeding, recall that an inactive or migrating writer becomes a running writer by performing the write request or write rerequest region, respectively, and that a waiting writer becomes a running writer by another task performing a write release or last scheduler test region.

Subclaim 27.1. For each region *R* that causes a task *i* to become a running writer and for the interval *I* beginning when *R* ends and ending when *i* arrives at the write release, *Cnt* is never assigned a positive value in *I*, i.e., (from the observations made in proving claim 26) neither the last scheduler test region nor the write rerequest region nor the write release reinitialization of *Cnt* is performed in *I*.

Proof of subclaim. Observe that running writer *i* cannot perform a last scheduler test, a write rerequest, or a write release reinitialization of *Cnt* prior to arriving at the write release, i.e., while it is a running writer. We must therefore show that no other task can perform such an assignment in *I*. Let us start by assuming that *i* becomes a running writer via the write request region. In (a), we prove that a task cannot perform the last scheduler test region or the write rerequest region in *I*, and in (b), we prove that a task cannot perform the write release reinitialization statement in *I*.

(a) Suppose that after *i* performs the write request region and becomes a running writer but before it releases the resource, another task *j* performs the last scheduler test or write rerequest region. Then, *j* is, respectively, a scheduling reader (by claim 7) or a migrating writer (by claim 8) when at the region, which is a contradiction since a running writer (i) excludes both scheduling readers and migrating writers (by claim 19 and assertions **A4** and **A5**). Therefore, no such j exists.

(b) Now suppose that after i performs the write request region and becomes a running writer but before it releases the resource, another task j performs a write release reinitialization of *Cnt*. Then, j performed the write release region (it must execute this region in order to reach the reinitialization statement) before or after i performed the write request region. Regardless of the relative order in which these two regions were executed, immediately after the second one was performed, there was a running writer (i) and a running reader (by the proof of claim 25, j caused a waiting reader r to become a running reader when it performed the write release and r cannot be unblocked until after j reinitializes *Cnt*), which is a contradiction (by claim 19 and assertion **I1**). Hence, no such j exists.

In a similar manner it can be shown that if i enters interval I by performing the write rerequest region or by another task k performing the write release or last scheduler test region, then no task can assign *Cnt* a positive value in I. *End of subclaim proof.*

Subclaim 27.2. For each region R that causes a task i to become a running writer and for the interval I beginning when R ends and ending when i arrives at the write release, either *Cnt* is zero upon exiting R or *Cnt* is assigned a value of zero (at R36 or W28) in I.

Proof of subclaim. Let us consider each region R that can cause i to become a running writer, beginning with those regions that assign a positive value to *Cnt*.

(a) Let R be the write rerequest region. A migrating writer i rerequests the resource and becomes a running writer. In this transition, i increments *Cnt* (at W17) and before it uses the resource, it reinitializes *Cnt* to zero (at W28).

(b) Let R be the last scheduler test region. A scheduling reader r performs a last scheduler test and causes a waiting writer i to become a running writer. Task r increments *Cnt* (at R27), reinitializes *Cnt* to zero (at R36), and then unblocks i (at R37).[22]

[22] Recall from the proof of claim 23 that i can be unblocked only by r.

(c) Let *R* be the write request region. An inactive task *i* performs a write request and becomes a running writer. While *i* is performing the region, no task *j* can assign *Cnt* a positive value via a write release reinitialization of *Cnt* (at W60) (*j* cannot perform the last scheduler test or write rerequest region while *i* is in the write request region since regions are mutually exclusive): Assume not. Then at some point in time *j* is at W60 when *i* is performing the write request region. But this is a contradiction by corollary 2. Hence, no such *j* exists. Also observe that *Cnt* is not referenced by *i* during the transition (i.e., within the write request region).

We will now show that when *i* exits the write request region, it can be a running writer only if *Cnt* is zero. Suppose not. Then *Cnt* must have been positive when *i* entered the write request region since we have just shown *Cnt* cannot be assigned a positive value while *i* is in the region. As *Cnt* is initially zero (i.e., before any task requests the resource), the write rerequest region, or the last scheduler test region, or the write release reinitialization of *Cnt* had to have been performed before *i* entered the region. Let us consider each one of these cases.

(i) Suppose another task *k* rerequested the resource, incrementing *Cnt* to a positive value, before *i*'s write request. First assume *k* became a running writer upon exiting the rerequest region. While *k* is a running writer, *i* cannot perform the write request region and become a running writer (otherwise, when *i* exited the region, *i* and *k* would both be running writers, which is a contradiction by claim 19 and assertion **I2**), and by (a) *Cnt* is set to zero before *k* arrives at the write release. Second, assume *k* became a waiting writer upon exiting the rerequest region. Then the next task that becomes a running writer does so via another task executing the last scheduler test: First note that when *k* is at the write rerequest region, there are no running writers and *k* is the only migrating writer (by claim 19 and assertion **A5**). Immediately after *k* performs the region and becomes a waiting writer, there are running and/or scheduling readers, a waiting writer (*k*), and neither migrating nor running writers. Therefore, by claim 15, the next task *t* that becomes a running writer does so via the last scheduler test region, and by (b) *Cnt* is set to zero before *t* arrives at the write release.

(ii) Now suppose task *k* performed the last scheduler test region, incrementing *Cnt* to a positive value, before *i*'s write request. First assume *k* caused a task *t* to become a running writer via the test. As in (i), *i* cannot become a running writer while *t* is a running writer, and by (b) *Cnt* is set to zero before *t* releases the resource. Second assume that *k* did not cause a task to become a running writer upon exiting the last scheduler test region. When *k* was at the last scheduler test, there were no running writers and there was a migrating writer and/or a waiting writer (by claim 19 and assertions **A4** and **A3**). Immediately after the test was completed, there was still a migrating writer and/or a waiting writer. First, suppose there was a migrating writer. Note, by inspecting the write request region, that *i* cannot become a running writer via a write request while there is a migrating writer. If a migrating writer rerequests the resource and becomes a running writer then it will set *Cnt* to zero before another task can become a running writer (see (a)). If the migrating writer becomes a waiting writer upon exiting the rerequest region, then (by claim 19 and assertion **I3**) there must also be running and/or scheduling readers. It then follows by claim 15 that the next task that becomes a running writer will do so via another task performing the last scheduler test, and before this task releases the resource, *Cnt* is set to zero (see (b)). Next, suppose there were only waiting writers when *k* exited the last scheduler test region. By claim 19 and assertion **I3**, there were also running readers and/or scheduling readers upon *k*'s exit. Once again, by claim 15, the next task that becomes a running writer will do so via another task performing the last scheduler test, and before this task releases the resource, *Cnt* is set to zero.

(iii) Finally, suppose task *k* performed the write release reinitialization of *Cnt* before *i* entered the write request region. By the proof of claim 25, this implies that *k* arrived at the resource release when there were $M > 0$ waiting writers and $N > 0$ waiting readers (if there were not both waiting writers and waiting readers, *k* would not perform the reinitialization of *Cnt*). Moreover, immediately after *k*'s release was completed, there were *M* waiting writers, *N* running readers, no scheduling readers, and no migrating or running writers. But in this case, by claim 15, the next task *t* that becomes a running writer will do so via another task performing a last scheduler test, and as shown in (b), *Cnt* is assigned a value of zero before *t* releases the resource.

Hence, when *i* exits the write request region as a running writer, *Cnt* can only be zero.

(d) Let *R* be the write release region. A running writer *w* releases the resource when there are waiting writers and no waiting readers, and a waiting writer *i* becomes a running writer. As in (c), no task (including *w*) can assign a positive value to *Cnt* while *w* performs the region, and upon *w*'s exit from the region, *Cnt* must be zero. *End of subclaim proof.* □

Claim 28. If a reader is at the read release when there are waiting writers and no migrating writers, then *Cnt* is positive.

Proof. We begin by making the following observation. The claim is true if the following assertions hold for each region *R* that causes a task *i* to become a running reader and for the interval *I* beginning when *R* ends and ending when *i* arrives at the read release:

(1) *Cnt* is never assigned a nonpositive value in *I*.

(2) If there are waiting writers and no migrating writers when *i* is at the read release, then *Cnt* was assigned a positive value in *I*.

We shall now present subclaims that prove (1) and (2). Before proceeding, recall that an inactive reader becomes a running reader by performing a read request region, and that a waiting reader becomes a running reader by another task performing a write release region.

Subclaim 28.1. For each region *R* that causes a task *i* to become a running reader and for the interval *I* beginning when *R* ends and ending when *i* arrives at the read release, *Cnt* is never assigned a value of zero in *I*, i.e., (from the observations made in claim 26) neither the last scheduler test reinitialization statement (at R36) nor the write rerequest reinitialization statement (at W28) of *Cnt* is performed in *I*.

Proof of subclaim. Observe that a running reader cannot assign *Cnt* a value of zero via either a last scheduler test reinitialization of *Cnt* or a write rerequest reinitialization of *Cnt*. Therefore we must show that no other task can perform such an assignment in *I*. Let us start by assuming that *i* becomes a running reader via the read request region.

Suppose that after *i* performs the read request region and becomes a running reader but before it releases the resource, another task *j* reinitializes *Cnt* to zero. First, let us assume that *j* does so via the last scheduler test reinitialization of *Cnt*. Then, *j* must have previously performed the last scheduler test region and caused a waiting writer *k* to become a running writer (by the proof of claim 23) before or after *i* performed the read request region. Regardless of the relative order in which these two regions were executed, immediately after the second one was performed, there was a running reader (*i*) and a running writer (*k*),[23] which is a contradiction (by claim 19 and assertion **I1**). Hence, *j* cannot reinitialize *Cnt* to zero via the last scheduler test reinitialization statement. Therefore, let us assume that *j* sets *Cnt* to zero via the write rerequest reinitialization statement. In this case, *j* must have previously performed the write rerequest region and become a running writer (by claim 24) before or after *i* performed the read request region. Again, regardless of the relative order in which these two regions were executed, immediately after the second one was performed, there was a running reader (*i*) and a running writer (*j*), which is a contradiction (by claim 19 and assertion **I1**). Hence, no such *j* exists.

Similarly, after a task *k* performs the write release region and causes a waiting reader *i* to become a running writer but before *i* releases the resource, no task can perform a last scheduler test reinitialization of *Cnt* or a write rerequest reinitialization of *Cnt*. *End of subclaim proof.*

Subclaim 28.2. For each region *R* that causes a task *i* to become a running reader and for the interval *I* beginning when *R* ends and ending when *i* arrives at the read release, if there are waiting writers and no migrating writers when *i* is at the read release, then *Cnt* is assigned a positive value in *I*.

Proof of subclaim.

(a) Let *R* be the read request region. Suppose an inactive task *i* becomes a running reader upon requesting the resource. Then, by inspection of the region, when *i* requested the resource, there were no active writers. Assume that when *i* later releases the resource (at R15), there are waiting writers and no migrating writers. Then, after *i* became a running reader, an inactive writer *j* first requested the resource, becoming a migrating writer, and then rerequested the resource, becoming a waiting writer and assigning *Cnt* a positive value (by the proof of claim 26).

(b) Let *R* be the write release region. There are two cases two consider. First, suppose a running writer *j* performs a write release when there are no waiting writers and there are $N > 0$ waiting readers. When *j* is at W37, it is the only running writer and there are no migrating writers (by claim 19 and assertions **I2** and **A5**). Immediately

[23] Recall from the proof of claim 23 that *k* can be unblocked only by *j* and this cannot occur until after *j* reinitializes *Cnt* to zero.

after j's release, all of the waiting readers are running readers and there are no active writers; this state is indistinguishable from the one produced when N tasks become running readers by requesting the resource when there are no active writers. Therefore, as in (a), if following j's write release there are waiting writers and no migrating writers when a reader arrives at the read release, then *Cnt* is positive.

Second, suppose a running writer j performs a write release when there are $M > 0$ waiting writers and $N > 0$ waiting readers. Immediately after the release, there are M waiting writers, N running readers, and no scheduling readers or migrating or running writers (by the proof of claim 25). Before any one of these readers is unblocked by j,[24] j reinitializes *Cnt* (at W60) to a positive value (by the proof of claim 25). After j completes the write release but before the N readers release the resource, none of the M waiting writers can become a running writer nor can a task become a migrating writer (by the proof of claim 15). Hence, *Cnt* is positive when each of the N readers releases the resource. *End of subclaim proof.*

□

Our next claim defines the relationship between the variable *Cnt* and the state and local variables of Algorithm 7-2. Within the claim a number of assertions are made regarding these variables. To help motivate these assertions, we make the following observations without formal proof. The proofs that follow do not rely on these observations.

First, note (by inspection of the write request region) that a task j becomes a migrating writer when making a write request only if there are no active writers and there are running or waiting readers. Corollary 1 (see section 7.3.2) states that if there are no active writers, then there are neither waiting readers nor scheduling readers. Hence, when j becomes a migrating writer, there are $N > 0$ running readers and all other tasks are inactive. From this and claim 20, it follows that $v_j = N$ after j completes the write request.

Second, we note, by the proof of claim 17 and the absence of scheduling readers when j becomes a migrating writer, that while j is a migrating writer, the set of running and scheduling readers consists of those tasks that were running readers when j requested the resource and that have not yet become inactive.

---

[24] By the proof of claim 25, only j can unblock these readers.

Finally, observe that when a migrating or waiting writer becomes a running writer via a write rerequest or last scheduler test, respectively, *Cnt* is positive as a result of a fetch-and-add performed at W17 or R27 (see claim 26 for details). *Cnt* is then set to zero following the running writer's or last scheduling reader's reinitialization of *Cnt* at W28 or R37, respectively. Also, when a writer *k* is at the write release, *Cnt* is zero (by claim 27). If *k* releases the resource when there are waiting writers and waiting readers, in which case the waiting readers become running readers, then *Cnt* is assigned a positive value when *k* reaches W60 (see claim 26 for details). Hence, while a task is a running writer (reader), *Cnt* can be positive (zero) and then zero (positive). We comment that the following claim will enable us to show that when a task becomes a running writer via a write rerequest or last scheduler test, *Cnt* is not only positive, but equals *MaxR*.

Claim 29. When no region or reinitialization statement of Algorithm 7-2 is being executed, the following assertions hold:[25]

(1) *If there are no active writers (*$\mathbf{B_1}$*), then Cnt is zero (*$\mathbf{C_1}$*)*:

$\mathbf{B_1} \quad \mathbf{C_1}$, where

$\mathbf{B_1} \equiv rw = ww = mw = 0$

$\mathbf{C_1} \equiv Cnt = 0$

(2) *If there are migrating writers (*$\mathbf{B_2}$*), then there exists a task j such that j is a migrating writer and Cnt equals the number of running readers that have become inactive since j requested the resource (*$\mathbf{C_2}$*), i.e., Cnt equals the number of tasks that were running readers when j requested the resource (*v_j*) less the number of running and scheduling readers*:

$\mathbf{B_2} \quad \mathbf{C_2}$, where

$\mathbf{B_2} \equiv mw \neq 0$

$\mathbf{C_2} \equiv (\exists j \mid mw[j] = 1 \quad Cnt = v_j - \sum (rr[k] + sr[k]))$

(3) *If there is a running writer (*$\mathbf{B_3}$*), then Cnt equals zero or MaxR* ($\mathbf{C_3}$):

$\mathbf{B_3} \quad \mathbf{C_3}$, where

$\mathbf{B_3} \equiv rw \neq 0$

$\mathbf{C_3} \equiv Cnt = 0 \quad Cnt = MaxR$

(4) *If there are running readers and waiting writers and neither scheduling readers nor migrating writers (*$\mathbf{B_4}$*), then Cnt equals either zero or MaxR less the number of running readers* ($\mathbf{C_4}$):

$\mathbf{B_4} \quad \mathbf{C_4}$, where

$\mathbf{B_4} \equiv rr \neq 0 \quad ww \neq 0 \quad sr = mw = 0$

$\mathbf{C_4} \equiv Cnt = 0 \quad Cnt = MaxR - \sum rr[k]$

[25] The reader will note that the fourth and fifth implications below are very similar. However, combining them would result in the loss of certain task state information that is necessary for the proof of the claim.

(5) *If there are waiting writers and scheduling readers and no migrating writers (*$\mathbf{B_5}$*), then Cnt equals MaxR less the number of running and scheduling readers* ($\mathbf{C_5}$):

$\mathbf{B_5} \quad \mathbf{C_5}$, where

$\mathbf{B_5} \equiv ww \neq 0 \;\; sr \neq 0 \;\; mw = 0$

$\mathbf{C_5} \equiv Cnt = MaxR - \sum (rr[k] + sr[k])$

(6) *Exactly one of the above assertions holds because both of its propositions are true*:

$(\exists! \mathbf{i})\ (\mathbf{B_i})$

Proof. First recall that state variables are modified only within regions, which, by definition, are mutually exclusive. In addition, observe that the fetch-and-adds and reinitializations of *Cnt* are atomic (as each one involves only local variables and one reference to *Cnt*), and that each region contains at most one reference to *Cnt*. Therefore, in proving the claim, an execution of Algorithm 7-2 can be represented as the sequence or history of executed regions and reinitialization statements, i.e., overlapping executions of regions and reinitialization statements have the same effect as some (unspecified) serial execution of the regions and reinitialization statements in question.

Second, observe that $(\forall \mathbf{i} \neq \mathbf{j})\ \sim(\mathbf{B_i} \;\; \mathbf{B_j})$. For example,

$$\mathbf{B_1} \;\; \mathbf{B_2} \;\; mw = 0 \;\; mw \neq 0 \;\; \text{contradiction.}$$

Similarly,

$$\mathbf{B_2} \;\; \mathbf{B_3} \;\; mw \neq 0 \;\; rw \neq 0 \;\; \text{contradiction (by claim 19 and assertion } \mathbf{A5}\text{).}$$

Thus, (6) simplifies to $\underset{\mathbf{i}}{}\mathbf{B_i}$. From this it follows that we need only show $(\exists \mathbf{i})\ (\mathbf{B_i} \;\; \mathbf{C_i})$ to prove the claim. That is, let

$\mathbf{D_i} \equiv \mathbf{B_i} \;\; \mathbf{C_i}$, i.e.,

$\mathbf{D_1} \equiv rw = ww = mw = 0 \;\; Cnt = 0$

$\mathbf{D_2} \equiv mw \neq 0 \;\; (\exists j \mid mw[j] = 1 \;\; Cnt = v_j - \sum (rr[k] + sr[k]))$

$\mathbf{D_3} \equiv rw \neq 0 \;\; (Cnt = 0 \;\; Cnt = MaxR)$

$\mathbf{D_4} \equiv rr \neq 0 \;\; ww \neq 0 \;\; sr = mw = 0 \;\; (Cnt = 0 \;\; Cnt = MaxR - \sum rr[k])$

$\mathbf{D_5} \equiv ww \neq 0 \;\; sr \neq 0 \;\; mw = 0 \;\; Cnt = MaxR - \sum (rr[k] + sr[k])$.

Then, to prove the claim it is sufficient to show that for each history H=[(s_1, $task_{j_1}$), (s_2, $task_{j_2}$), ... , (s_n, $task_{j_n}$)] of Algorithm 7-2, where s_i is either a reinitialization statement of *Cnt* or a region, and $\forall (s_i,\ task_{j_i}) \in H$, $\exists \mathbf{x},\ \mathbf{y}$ such that $\mathbf{D_x}$ is true before s is executed by $task_{j_i}$ and $\mathbf{D_y}$ is true immediately afterwards.

Before s_1 is executed, $\mathbf{D_1}$ is true since all variables are initially zero. Inductively assume that ${}_{\mathbf{j}}\mathbf{D_j}$ holds over s_1 through s_{k-1} of H and that task i executes s_k. We show that ${}_{\mathbf{j}}\mathbf{D_j}$ holds after $region_k$.

(1) Let s_k be the read request region. When task i performs this region, it becomes a running reader only if there are no active writers ($rw = ww = mw = 0$ at R2), in which case only $\mathbf{D_1}$ can be true before and after the region is executed. Otherwise, i becomes a waiting reader and either $\mathbf{D_2}$, $\mathbf{D_3}$, $\mathbf{D_4}$, or $\mathbf{D_5}$ is true before and after the region is executed.

(2) Let s_k be the read release region. When task i is at this region, it is a running reader (by claim 11). $\mathbf{D_3}$ cannot hold (by claim 19 and assertion **I1**). If $\mathbf{D_1}$ holds before i releases the resource, then it holds afterwards. If any one of the remaining assertions holds before the release, then there are waiting writers or migrating writers when i releases the resource, and i therefore becomes a scheduling reader (setting $rr[i]$ and $sr[i]$ to zero and one, respectively, at R17 and R19). Thus, if $\mathbf{D_2}$ or $\mathbf{D_5}$ holds before the release, then $\mathbf{D_2}$ or $\mathbf{D_5}$ holds, respectively, afterwards. If $\mathbf{D_4}$ holds prior to the release, then $Cnt = MaxR - \sum rr[k]$ (since, by claim 28, Cnt must be positive), and after the release, as i is a scheduling reader, $\mathbf{D_5}$ holds.

(3) Let s_k be the last scheduler test region. When task i is at this region, it is a scheduling reader (by claim 7), and there are migrating or waiting writers and no running writers (by claim 19 and assertions **A3** and **A4**). Therefore, $\mathbf{D_2}$ or $\mathbf{D_5}$ holds. First, suppose it is $\mathbf{D_2}$. Then, when i is at R26, there exists a migrating writer j such that $Cnt = v_j - \sum (rr[k] + sr[k])$. In performing this test, i increments Cnt by one (at R27) and sets $sr[i]$ to zero (at R28). Hence, $\mathbf{D_2}$ holds after this region. Second, suppose $\mathbf{D_5}$ holds when i is at R26. If i is not a last scheduling reader (i.e., it does not perform a *WIdQ* deletion at R31), then $\mathbf{D_5}$ holds after i makes the test (just as $\mathbf{D_2}$ holds above). Otherwise, if i is a last scheduling reader, then (when it is at R26) there are waiting writers and neither running readers nor migrating writers, and

$$Cnt = MaxR - \sum sr[k] = MaxR - 1.$$

In performing the test, i increments Cnt by one (at R27) and causes a waiting writer to become a running writer (at R31), after which $\mathbf{D_3}$ holds.

(4) Let s_k be the last scheduler test reinitialization of *Cnt*. When task i is at this statement, there is a running writer (by claim 23). Therefore, only $\mathbf{D_3}$ can hold. After the reinitialization is done, *Cnt* is zero and there is a running writer, which implies that $\mathbf{D_3}$ still holds.

(5) Let s_k be the write request region. If $\mathbf{D_2}$, $\mathbf{D_3}$, $\mathbf{D_4}$, or $\mathbf{D_5}$ holds when i performs this region, then i becomes a waiting writer and $\mathbf{D_2}$, $\mathbf{D_3}$, $\mathbf{D_4}$, or $\mathbf{D_5}$, respectively, holds afterwards. Now, suppose $\mathbf{D_1}$ holds when i is at W1. Then, *Cnt* equals zero, and there are no waiting or scheduling readers as

$$\mathbf{I3}\ \ \mathbf{A2}\ \ \mathbf{A3}\ \ \mathbf{D_1}\ \ wr = sr = 0.$$

If i requests the resource when there are no running readers, it becomes a running writer (as $rr = sr = rw = ww = mw = 0$ at W2) and $\mathbf{D_3}$ will hold upon i's completion of the request. Otherwise, i sets v_i (at W12) to the current value of *Val*, which equals $\sum rr[k]$ (by claim 20), and i becomes a migrating writer. Then, after i makes the request,

$$mw[i] = 1\ \ Cnt = 0\ \ sr = 0\ \ v_j = \sum rr[k]\ \ \mathbf{D_2}.$$

(6) Let s_k be the write rerequest region. When i performs this region, it is a migrating writer (by claim 8). Hence, only $\mathbf{D_2}$ can hold at W16. First, suppose that there are no scheduling or running readers when i issues a rerequest. Then, at W16, $Cnt = v_i$. In performing this region, i increments *Cnt* by $MaxR - v_i$ (at W17) and becomes a running writer. Thus, $\mathbf{D_3}$ holds when i exits the region. Second, suppose there are scheduling or running readers when i makes a rerequest. Then, at W16, $Cnt = v_i - \sum (rr[k] + sr[k])$. Task i then increments *Cnt* by $MaxR - v_i$ and becomes a waiting writer, which implies that $\mathbf{D_4}$ or $\mathbf{D_5}$ holds when i completes the rerequest.

(7) Let s_k be the write rerequest reinitialization of *Cnt*. When i is at this statement (W22), it is a running writer (by claim 24). Since *Cnt* is set to zero, $\mathbf{D_3}$ holds after this statement is executed by i.

(8) Let s_k be the write release region. When i is at this region, it is a running writer (by claim 12), and therefore, $\mathbf{D_3}$ holds. In addition, there are no migrating writers (by claim 19 and assertion **A5**) or scheduling readers (by claim 19 and assertion **A4**), and *Cnt* equals zero (by claim 27). First, suppose either that there are no waiting tasks or that there are waiting readers and no waiting writers when i makes the release. Then, after the release is done, there are no active writers, and $\mathbf{D_1}$ holds. Second, suppose there are waiting writers and no waiting readers when the release is performed. Then one of the waiting writers

becomes a running writer and $\mathbf{D_3}$ holds upon the release. Lastly, suppose there are waiting readers and waiting writers at the time of *i*'s release. Then, all of the waiting readers become running readers and $\mathbf{D_4}$ holds afterwards: Recall from the proof of claim 25 that until *i* reinitializes *Cnt* (at W60) none of these readers can use the resource and, aside from these readers, no other task can become a running reader (and then possibly a scheduling reader).

(9) Let s_k be the write release reinitialization of *Cnt*. When *i* reinitializes *Cnt* at W60, there are running readers and waiting writers, and neither scheduling readers nor migrating writers, and a_i equals the number of running readers (by claim 25). Therefore, $\mathbf{D_4}$ holds. After *i* performs the reinitialization, no task's state has changed and

$$Cnt = MaxR - b_i = MaxR - a_i = MaxR - \sum rr[k] \quad \mathbf{D_4}. \ \square$$

Claim 30. When a task *i* is at one of the following statements of Algorithm 7-2 (and no other task is performing either a reinitialization statement of *Cnt* or a region), the relationship between the state variables, local variables, *Val*, and *Cnt* is as stated:

(1) At R2, $rw{=}ww{=}mw{=}0 \quad Val{<}MaxR$
(2) At R12, $action_i{=}wait \quad v_i{\geq}MaxR$
(3) At R18, $ww{\neq}0 \quad mw{\neq}0 \quad Val{>}MaxR$
(4) At R25, $action_i{=}schedule \quad v_i{>}MaxR$
(5) At R29, $sr = rr = mw = 0 \quad ww{\neq}0 \quad c_i = MaxR - 1$
(6) At R35, $action_i = grant\_writing \quad c_i = MaxR - 1$
(7) At W2, $rr{=}wr{=}rw{=}ww{=}mw{=}0 \quad Val{=}0$
(8) At W4, $rw{=}ww{=}mw{=}0 \quad 0{<}Val{<}MaxR$
(9) At W14, $action_i{\neq}run \quad v_i{>}0$
(10) At W15, $action_i{=}migrate \quad v_i{<}MaxR$
(11) At W18, $sr = rr = 0 \quad c_i = v_i$
(12) At W27, $action_i = run \quad c_i = v_i$
(13) At W39, $wr{\neq}0 \quad Val$ modulo $MaxR{>}0$
(14) At W42, $a_i{=}Val$ modulo $MaxR$
(15) At W48, $ww{\neq}0 \quad Val{\geq}2*MaxR$
(16) At W57, $action_i{=}grant\_reading \quad v_i$ modulo $MaxR > 0$
(17) At W58, $a_i{=}v_i$ modulo $MaxR$
(18) At W66, $action_i = grant\_writing \quad v_i \geq 2*MaxR$

Proof. First note that when a task i executes a region, only task i may modify *Val* and task state variables since regions are mutually exclusive and *Val* and state variables are modified only within regions. Second, recall that all variables of the form v_i are local to task i and therefore may not be altered by any other task.

(1) At R2, $rw{=}ww{=}mw{=}0 \;\; Val{<}MaxR$: When i is at R1, $rw = ww = mw = 0 \;\; Val < MaxR$ (by claim 22 and assumption 1). This is also true when i is at R2.

(2) At R12, $action_i{=}wait \; v_i{\geq}MaxR$:

$$(action_i{=}wait) \text{ at R12 } \sim(rw = ww = mw = 0) \text{ at R1}$$
$$(Val{\geq}MaxR) \text{ at R1 (by claim 22)}$$
$$(v_i{\geq}MaxR) \text{ at R12}$$

(3) At R18, $ww{\neq}0 \;\; mw{\neq}0 \;\; Val{>}MaxR$: () Suppose that when i is at R16 there are waiting or migrating writers. Then, $rr[i] = 1$ (by claim 11), $ww{\neq}0$ or $mw{\neq}0$ (by claim 3), and $Val > MaxR$ (by claim 20). This implies that when i is at R18, $ww{\neq}0 \;\; mw{\neq}0 \; Val{>}MaxR$.

() Suppose that when i is at R16, $Val > MaxR$. Since i is a running reader (by claim 11), there are no running writers (by claim 19 and assertion **I1**). Hence, there are waiting or migrating writers (by claim 22). This is still true when i is at R18, i.e., $Val{>}MaxR \; ww{\neq}0 \;\; mw{\neq}0$.

(4) At R25, $action_i{=}schedule \; v_i{>}MaxR$: This follows from (3).

(5) At R29, $sr = rr = mw = 0 \;\; ww{\neq}0 \;\; c_i = MaxR - 1$: When i is at R26, it is a scheduling reader (by claim 7), and there are either waiting or migrating writers (by claim 19 and assertion **A3**). Hence, at R29 the test for waiting writers can be eliminated. Also note that when i performs a last scheduler test, $\mathbf{D_2}$ or $\mathbf{D_5}$ holds (by the proof of claim 29, part (3)).

() Suppose when i is at R26, there are neither running readers nor migrating writers and i is a last scheduling reader. Then only $\mathbf{D_5}$ can hold, implying $Cnt = MaxR - 1$, and thus, when i is at R29, $c_i = MaxR - 1$.

() Assume that when i is at R26, $Cnt = MaxR - 1$. Suppose $\mathbf{D_2}$ holds. Then there exists a migrating writer j such that $Cnt = v_j - \sum (rr[k] + sr[k]) = MaxR - 1$. Since i is a scheduling reader ($sr{\neq}0$) and v_j is always nonnegative (being assigned only (old) values of *Val*, which are always nonnegative by claim 20), it must be that there are no running readers ($rr = 0$)

and $v_j = MaxR$. But the latter is a contradiction since by the proof of claim 26 (part 1), $v_j < MaxR$. Hence, $\mathbf{D_5}$ must hold at R26, which implies that $Cnt = MaxR - \sum (rr[k] + sr[k])$. Since i is a scheduling reader and $Cnt = MaxR - 1$, there can be no running readers. Therefore, when i is at R29, $sr = rr = mw = 0 \;\; ww \neq 0$.

(6) At R35, $action_i = grant\_writing \;\; c_i = MaxR - 1$: This follows from (5).

(7) At W2, rr=wr=rw=ww=mw=0 Val=0: By claim 20.

(8) At W4, rw=ww=mw=0 0<Val<$MaxR$: When i is at W1, $0 < Val < MaxR$ if and only if there are no active writers ($rw = ww = mw = 0$) and there are running or waiting readers ($rr \neq 0$ or $wr \neq 0$) (by claim 21). Hence, when i is at W4, rw=ww=mw=0 0<Val<$MaxR$.

(9) At W14, $action_i \neq run \;\; v_i > 0$: When i is at W1, (by claim 20) $Val > 0$ if and only if there are active writers ($rw \neq 0$ or $ww \neq 0$ or $mw \neq 0$), running readers ($rr \neq 0$), or waiting readers ($wr \neq 0$). In these cases, i sets $action_i$ to either *migrate* (at W5) or *wait* (at W7), depending upon the states of the tasks. Then, when i completes its request and is at W14, $v_i > 0$ (due to the fetch-and-add on Val at W12) and $action_i \neq run$. In the other case, ($Val = 0$ and $rw = ww = mw = rr = wr = 0$), $action_i$ is set to *run* and v_i is set to zero.

(10) At W15, $action_i = migrate \;\; v_i < MaxR$: Consider the value of Val when i is at W1. First, if $Val = 0$, i does not get to W15 (by (9)). Second, (by claim 21) $0 < Val < MaxR$ if and only if there are no active writers ($rw = ww = mw = 0$) and there are running readers ($rr \neq 0$) or waiting readers ($wr \neq 0$). In this case, i sets $action_i$ to *migrate* (at W5). Then, when i completes its request and is at W15, $0 < v_i < MaxR$ (due to the fetch-and-add on Val at W12) and $action_i = migrate$. Finally, if $Val \geq MaxR$ at W1, then i sets $action_i$ to *wait* (at W7) and when i is at W15, $v_i \geq MaxR$ (due to the fetch-and-add on Val at W12). Hence, at W15, $action_i = migrate \;\; v_i < MaxR$.

(11) At W18, $sr = rr = 0 \;\; c_i = v_i$: When i is at W16, it is a migrating writer (by claim 8), and therefore $\mathbf{D_2}$ holds. Hence, $Cnt = v_i$ if and only if there are neither scheduling nor running readers, which implies that at W18, $c_i = v_i \;\; sr = rr = 0$.

(12) At W27, $action_i = run \;\; c_i = v_i$: This follows from (11).

(13) At W39, $wr \neq 0 \;\; Val$ modulo $MaxR > 0$: Consider i at W37. Since Val modulo $MaxR = \sum rr[j] + wr[j]$ (by claim 20) and $rr = 0$ (by claim 19 and assertion **I1**), Val modulo $MaxR = \sum wr[j]$. But neither Val nor wr is modified from W37 to W39.

(14) At W42, $a_i = Val$ modulo $MaxR$: When i is at W39, Val modulo $MaxR = \sum wr[j]$ (as shown in (13)). But neither *Val* nor *wr* is modified from W39 to W42, and at W41 a_i is assigned $\sum wr[j]$.

(15) At W48, $ww \neq 0$ $Val \geq 2*MaxR$: When i is at W37, $rw[i] = 1$ (by claim 12), $mw = 0$ (by claim 19 and assertion **A5**), and therefore $Val \geq 2*MaxR$ if and only if $ww \neq 0$ (by claim 20 and assumption 1). But neither *Val* nor *ww* is modified from W37 to W48.

(16) At W57, $action_i$=*grant_reading* v_i modulo $MaxR>0$: This follows from (13), the assignment at W40, and the assignment of *Val* (via fetch-and-add) to v_i at W55.

(17) At W58, $a_i = v_i$ modulo $MaxR$: This follows from (14) and the assignment of *Val* (via fetch-and-add) to v_i at W55.

(18) At W66, $action_i$ = *grant_writing* $v_i \geq 2*MaxR$: This follows from (15), the assignment at W49, and the assignment of *Val* (via fetch-and-add) to v_i at W55. □

Using the preceding claim, Algorithm 7-2 is converted into Algorithm 7-3 by replacing expressions based on state variables with expressions based on *Val* and *Cnt*. The state variable based expressions of Algorithm 7-2 are represented as comments (delimited by {}) within Algorithm 7-3. Clearly, Algorithm 7-3, like Algorithm 7-2, satisfies properties **P1–P4**.

Next, by eliminating state variables from Algorithm 7-3, we produce Algorithm 7-4. This algorithm also satisfies properties **P1–P4** since state variables do not affect how 7-3 manages the resource. Removing identifier queues from 7-4 gives Algorithm 7-5. The four readers/writers properties hold for Algorithm 7-5 since identifier queues are used only in conjunction with state variables (which are not present in 7-4).

```
reader i:  A1:  region ‘‘read request’’
           A2:      if Val < MaxR then  {rw=ww=mw=0}
           A3:          rr[i] := 1;  action_i := run
           A4:      else
           A5:          wr[i] := 1;  action_i := wait;
           A6:          my#_i := r#; r# := r#+1 mod MaxT;
           A7:          QInsert(i, RIdQ)
           A8:      end if;
           A9:      ar[i] := 1;
           A10:     v_i := F&A(Val, 1)
           A11: end region;
           A12: if v_i ≥ MaxR then  {action_i = wait}
           A13:     Block(RWaitQ)
           A14: end if;
           A15: ‘‘read’’
           A16: region ‘‘read release’’
           A17:     rr[i] := 0;
           A18:     if  Val > MaxR then  {ww ≠ 0  mw ≠ 0}
           A19:         sr[i] := 1;  action_i := schedule
           A20:     else
           A21:         ar[i] := 0;  action_i := inactive
           A22:     end if;
           A23:     v_i := F&A(Val, −1)
           A24: end region;
           A25: if v_i > MaxR then  {action_i = schedule}
           A26:     region ‘‘last scheduler test’’
           A27:         c_i := F&A(Cnt, 1);
           A28:         sr[i] := 0;  ar[i] := 0;
           A29:         if  c_i = MaxR − 1 then  {sr=rr=mw=0  ww≠0}
           A30:             action_i := grant_writing;
           A31:             QDelete(writerid_i, WIdQ);
           A32:             ww[writerid_i] := 0;  rw[writerid_i] := 1
           A33:         end if
           A34:     end region;
           A35:     if c_i = MaxR − 1 then  {action_i = grant_writing}
           A36:         Cnt := 0;
           A37:         repeat until Unblock(WWaitQ)
           A38:     end if
           A39: end if;
```

Algorithm 7-3a. Control based on *Val* and *Cnt*.

```
writer j:
region ‘‘write request’’
    if Val = 0 then {rr=wr=rw=ww=mw=0}
        rw[j] := 1; action_j := run
    elseif Val < MaxR then {rw=ww=mw=0}
        mw[j] = 1; action_j := migrate;
    else
        ww[j] := 1; action_j := wait;
        my#_j := w#; w# := w#+1 mod MaxT;
        QInsert(j, WIdQ)
    end if;
    aw[j] := 1;
    v_j := F&A(Val, MaxR)
end region;
if v_j > 0 then {action_j ≠ run}
    if v_j < MaxR then {action_j = migrate}
        region ‘‘rerequest resource’’
            c_j := F&A(Cnt, MaxR–v_j);
            if c_j = v_j then {sr=rr=0}
                rw[j] := 1; action_j := run
            else
                ww[j] := 1; action_j := wait;
                my#_j := w#; w# := w#+1 mod MaxT;
                QInsert(j, WIdQ)
            end if;
            mw[j] := 0
        end region;
        if c_j = v_j then {action_j = run}
            Cnt := 0
        else
            Block(WWaitQ)
        end if
    else
        Block(WWaitQ)
    end if
end if;
‘‘write’’
```

Algorithm 7-3b. Control based on *Val* and *Cnt*.

(*writer j continued*)

```
region ‘‘write release’’
    rw[j] := 0;  aw[j] := 0;
    if Val mod MaxR > 0 then  {wr≠0}
        action_j := grant_reading;
        a_j := Val mod MaxR;  {∑wr[i]}
        for 1 to a_j do
            QDelete(readerid_j, RIdQ);
            wr[readerid_j] := 0;
            rr[readerid_j] := 1
        end for
    else
        if Val≥2*MaxR then  {ww≠0}
            action_j := grant_writing;
            QDelete(writerid_j, WIdQ);
            ww[writerid_j] := 0;
            rw[writerid_j] := 1
        end if
    end if;
    v_j := F&A(Val, –MaxR)
end region;
if v_j mod MaxR > 0 then  {action_j = grant_reading}
    b_j := v_j mod MaxR;  {a_j}
    if v_j > 2*MaxR then
        Cnt := MaxR–b_j
    end if;
    for 1 to b_j do
        repeat until Unblock(RWaitQ)
    end for
else
    if v_j ≥2*MaxR then  {action_j = grant_writing}
        repeat until Unblock(WWaitQ)
    end if
end if;
```

Initially: $ar = aw = rr = rw = wr = ww = mw = sr = 0$;
$RWaitQ = WWaitQ = RIdQ = WIdQ = \emptyset$;
$Cnt = Val = r\# = w\# = 0$;

Algorithm 7-3c. Control based on *Val* and *Cnt*.

```
reader i:  C1:  region ‘‘read request’’
           C2:      if Val≥MaxR then {become a waiting reader}
           C3:          my#_i := r#; r# := r#+1 mod MaxT
           C4:          QInsert(i, RIdQ)
           C5:      end if;
           C6:      v_i := F&A(Val, 1)
           C7:  end region;
           C8:  if v_i ≥MaxR then
           C9:      Block(RWaitQ)
           C10: end if;
           C11: ‘‘read’’
           C12: region ‘‘read release’’
           C13:     v_i := F&A(Val, –1)
           C14: end region;
           C15: if v_i > MaxR then
           C16:     region ‘‘last scheduler test’’
           C17:         c_i := F&A(Cnt, 1)
           C18:         if c_i = MaxR – 1 then
           C19:             QDelete(writerid_i, WIdQ)
           C20:         end if
           C21:     end region;
           C22:     if c_i = MaxR – 1 then
           C23:         Cnt := 0;
           C24:         repeat until Unblock(WWaitQ)
           C25:     end if
           C26: end if;
```

Algorithm 7-4a. State variables eliminated.

```
writer j:  region ‘‘write request’’
               if Val≥MaxR then {become a waiting writer}
                   my#_j := w#; w# := w#+1 mod MaxT
                   QInsert(j, WIdQ)
               end if;
               v_j := F&A(Val, MaxR)
           end region;
           if v_j > 0 then
               if v_j < MaxR then
                   region ‘‘rerequest resource’’
                       c_j := F&A(Cnt, MaxR–v_j);
                       if c_j ≠ v_j then {become a waiting writer}
                           my#_j := w#; w# := w#+1 mod MaxT
                           QInsert(j, WIdQ)
                       end if
                   end region;
                   if c_j = v_j then
                       Cnt := 0
                   else
                       Block(WWaitQ)
                   end if
               else
                   Block(WWaitQ)
               end if
           end if;
           ‘‘write’’
```

Algorithm 7-4b. State variables eliminated.

(*writer j continued*)

```
region ''write release''
    if Val mod MaxR > 0 then  {wr≠0}
        for 1 to Val mod MaxR do  {∑wr[i]}
            QDelete(readerid_j, RIdQ);
        end for
    else
        if Val≥2*MaxR then  {ww≠0}
            QDelete(writerid_j, WIdQ);
        end if
    end if;
    v_j := F&A(Val, –MaxR)
end region;
if v_j mod MaxR > 0 then
    b_j := v_j mod MaxR;
    if v_j > 2*MaxR then
        Cnt := MaxR–b_j
    end if;
    for 1 to b_j do
        repeat until Unblock(RWaitQ)
    end for
else
    if v_j ≥2*MaxR then
        repeat until Unblock(WWaitQ)
    end if
end if;
```

Initially: $RWaitQ = WWaitQ = RIdQ = WIdQ = \emptyset$;
$Cnt = Val = r\# = w\# = 0$;

Algorithm 7-4c. State variables eliminated.

```
reader i:   region ‘‘read request’’
                if Val≥MaxR then {become a waiting reader}
                    my#_i := r#; r# := r#+1 mod MaxT
                end if;
                v_i := F&A(Val, 1)
            end region;
            if v_i ≥ MaxR then
                Block(RWaitQ)
            end if;
            ‘‘read’’
            region ‘‘read release’’
                v_i := F&A(Val, −1)
            end region;
            if v_i > MaxR then
                region ‘‘last scheduler test’’
                    c_i := F&A(Cnt, 1)
                end region;
                if c_i = MaxR − 1 then
                    Cnt := 0;
                    repeat until Unblock(WWaitQ)
                end if
            end if;
```

Algorithm 7-5a. Identifier queues eliminated.

```
writer j:  F1:  region ''write request''
           F2:      if Val≥MaxR then {become a waiting writer}
           F3:          my#_j := w#; w# := w#+1 mod MaxT
           F4:      end if;
           F5:      v_j := F&A(Val, MaxR)
           F6:  end region;
           F7:  if v_j > 0 then
           F8:      if v_j < MaxR then
           F9:          region ''rerequest resource''
           F10:             c_j := F&A(Cnt, MaxR–v_j);
           F11:             if c_j ≠ v_j then {become a waiting writer}
           F12:                 my#_j := w#; w# := w#+1 mod MaxT
           F13:             end if
           F14:         end region;
           F15:         if c_j = v_j then
           F16:             Cnt := 0
           F17:         else
           F18:             Block(WWaitQ)
           F19:         end if
           F20:     else
           F21:         Block(WWaitQ)
           F22:     end if
           F23: end if;
           F24: ''write''
           F25: region ''write release''
           F26:     v_j := F&A(Val, –MaxR)
           F27: end region;
           F28: if v_j mod MaxR > 0 then
           F29:     b_j := v_j mod MaxR;
           F30:     if v_j > 2*MaxR then
           F31:         Cnt := MaxR–b_j
           F32:     end if;
           F33:     for 1 to b_j do
           F34:         repeat until Unblock(RWaitQ)
           F35:     end for
           F36: else
           F37:     if v_j ≥ 2*MaxR then
           F38:         repeat until Unblock(WWaitQ)
           F39:     end if
           F40: end if;

        Initially:  RWaitQ = WWaitQ = ∅;
                    Cnt = Val = r# = w# = 0;
```

Algorithm 7-5b. Identifier queues eliminated.

Algorithm 7-5 is transformed into Algorithm 7-6 by modifying the read request region as follows: the fetch-and-add on *Val* is moved from the end to the start of the region, the test $Val \geq MaxR$ is replaced with the test $v_i \geq MaxR$, and the *RWaitQ* ticket number calculation is done via a fetch-and-add (the new calculation is identical to the one used in the queue implementation of section 2.2.2.2, where *r#*, *MaxT*, and $my\#_i$ correspond to *Qrear*, *L*, and *iticket#*, respectively). The read request regions of the two algorithms appear in Figure 7-2, where blank lines are used to align corresponding statements of the two regions; the remainder of Algorithm 7-6 is identical to Algorithm 7-5 and therefore is not shown.

Claim 31. Algorithm 7-6 satisfies properties **P1–P4**.

Proof. In Algorithm 7-5, all references to *Val* and *r#* occur within regions and regions are mutually exclusive. Hence, the read request region of Algorithm 7-5 can be considered to be a serial code segment. *Val* is modified exactly once in the region, by a fetch-and-add. The fetch-and-add on *Val* is also the only assignment to v_i in the region, and v_i is not used (i.e., referenced) in the region. Therefore, it is safe to move the fetch-and-add on *Val* to the start of the region provided that all other references to *Val* within the region are replaced with references to v_i (as v_i then equals the old value of *Val*). Thus, in the read request region of Algorithm 7-6, since the fetch-and-add on *Val* is at the start of the region, the test $v_i \geq MaxR$ is used in place of the test $Val \geq MaxR$. By inspection of Figure 7-2, it is clear that both regions generate the same sequence

---

(a)	(b)
region ‘‘*read request of Alg. 7-5*’’	**region** ‘‘*read request of Alg. 7-6*’’
	v_i := F&A(*Val*, 1);
if $Val \geq MaxR$ **then**	**if** $v_i \geq MaxR$ **then**
$my\#_i$:= *r#*;	$my\#_i$:= F&A(*r#*, 1) **mod** *MaxT*;
r# := *r#*+1 **mod** *MaxT*	**if** $my\#_i = MaxT - 1$ **then**
	F&A(*r#*, $-MaxT$)
	end if
end if;	**end if**
v_i := F&A(*Val*, 1)	
end region;	**end region**;

Read request regions of Algorithms 7-5 and 7-6.

Figure 7-2.

of *RWaitQ* ticket numbers, although by slightly different procedures. Therefore, while the two algorithms differ with respect to their read request regions, the regions map a given (*Val*, *r#*) input into the same (*Val*, *r#*, v_i, $my\#_i$) output. We conclude then that Algorithm 7-6, like Algorithm 7-5, satisfies properties **P1–P4**. □

We convert Algorithm 7-6 into Algorithm 7-7 by modifying the write request and write rerequest regions in the same manner as Algorithm 7-5's read request region was modified to produce Algorithm 7-6; in the case of the write rerequest region, only the *WWaitQ* ticket number calculation is affected.

```
reader i:
region ‘‘read request’’
    v_i := F&A(Val, 1);
    if v_i ≥ MaxR then {become a waiting reader}
        my#_i := F&A(r#, 1) mod MaxT;
        if my#_i = MaxT − 1 then F&A(r#, −MaxT) end if
    end if;
end region;
if v_i ≥ MaxR then
    Block(RWaitQ)
end if;
‘‘read’’
region ‘‘read release’’
    v_i := F&A(Val, −1)
end region;
if v_i > MaxR then
    region ‘‘last scheduler test’’
        c_i := F&A(Cnt, 1)
    end region;
    if c_i = MaxR − 1 then
        Cnt := 0;
        repeat until Unblock(WWaitQ)
    end if
end if;
```

Algorithm 7-7a. Modified read request, write request, and rerequest regions.

```
writer j:
region ''write request''
    v_j := F&A(Val, MaxR);
    if v_j ≥ MaxR then {become a waiting writer}
        my#_j := F&A(w#, 1) mod MaxT;
        if my#_j = MaxT − 1 then F&A(w#, −MaxT) end if
    end if
end region;
if v_j > 0 then
    if v_j < MaxR then
        region ''rerequest resource''
            c_j := F&A(Cnt, MaxR−v_j);
            if c_j ≠ v_j then {become a waiting writer}
                my#_j := F&A(w#, 1) mod MaxT;
                if my#_j = MaxT − 1 then F&A(w#, −MaxT) end if
            end if
        end region;
        if c_j = v_j then
            Cnt := 0
        else
            Block(WWaitQ)
        end if
    else
        Block(WWaitQ)
    end if
end if;
''write''
region ''write release''
    v_j := F&A(Val, −MaxR)
end region;
if v_j mod MaxR > 0 then
    b_j := v_j mod MaxR;
    if v_j > 2*MaxR then
        Cnt := MaxR−b_j
    end if;
    for 1 to b_j do
        repeat until Unblock(RWaitQ)
    end for
else
    if v_j ≥ 2*MaxR then
        repeat until Unblock(WWaitQ)
    end if
end if;
```

Initially: $RWaitQ = WWaitQ = \varnothing$; $Cnt = Val = r\# = w\# = 0$;

Algorithm 7-7b. Modified read request, write request, and rerequest regions.

Claim 32. Algorithm 7-7 satisfies properties **P1–P4**.

Proof. Similar to claim 31. □

Next, Algorithm 7-7 is transformed into Algorithm 7-8 by moving the *RWaitQ* ticket number calculation from within the read request region to immediately before the *RWaitQ* Block statement, as depicted in Figure 7-3, where blank lines are used to align corresponding statements of the two code segments. Obviously, both algorithms generate identical sequences of *RWaitQ* insertion ticket numbers. The remainder of Algorithm 7-8 is the same as 7-7 and is not shown.

Tasks in Algorithm 7-8, unlike those in Algorithm 7-7, may be inserted onto, and thereby deleted from, wait queues in an order that differs from the order in which their read requests were denied. This is because the read request and the *RWaitQ* insertion ticket number calculation are separate atomic operations in Algorithm 7-8 and a single atomic operation in Algorithm 7-7. However, observe in both algorithms that a reader whose resource request is denied blocks within a finite time (as the insert routines of the algorithm are fair). In all other respects the code of the

---

(a)

```
region ''read request of Alg. 7-7''
    v_i := F&A(Val, 1);
    if v_i ≥ MaxR then
        my#_i := F&A(r#, 1) mod MaxT;
        if my#_i = MaxT − 1 then
            F&A(r#, −MaxT)
        end if
    end if
end region;
if v_i ≥ MaxR then




    Block(RWaitQ)
end if;
```

(b)

```
region ''read request of Alg. 7-8''
    v_i := F&A(Val, 1);
    if v_i ≥ MaxR then




    end if
end region;
if v_i ≥ MaxR then
    my#_i := F&A(r#, 1) mod MaxT;
    if my#_i = MaxT − 1 then
        F&A(r#, −MaxT)
    end if;
    Block(RWaitQ)
end if;
```

Reader code fragments of Algorithms 7-7 (a) and 7-8 (b).

Figure 7-3.

two algorithms is identical. In particular, they use the same control logic to implement task state transitions. For example, when a task *i* performs the read request of Algorithm 7-8 (7-7), it becomes a waiting reader if *Val*≥*MaxR* and a running reader otherwise.

As the two algorithms can differ (at least) with respect to the *RWaitQ* insertion and deletion of readers, we will compare the algorithms with respect to the execution of regions and *RWaitQ* (*WWaitQ*) insertions and deletions; recall that the latter operations are performed within *RWaitQ* (*WWaitQ*) Block and Unblock. We will show that for each execution history of Algorithm 7-8, there exists a corresponding execution history of Algorithm 7-7 such that if Algorithm 7-8 fails to satisfy the four readers/writers properties, then so will Algorithm 7-7.

Without loss of generality, let *RWaitQ* and *WWaitQ* insertion and deletion be atomic. Then, for our purposes, an execution history of Algorithm 7-7 or 7-8 can be represented by the sequence of executed regions and wait queue insertions and deletions: Since *RWaitQ* and *WWaitQ* are not referenced within regions, overlapping executions of regions and queue operations have the same effect as some (unspecified) serial execution of the regions and queue operations in question.

Claim 33. For each execution history I=[(s_1, $task_{i_1}$), (s_2, $task_{i_2}$), ... , (s_n, $task_{i_n}$)] of Algorithm 7-8, where s_k is either a region or a wait queue operation and $task_k$ is at s_k, there exists a corresponding execution history J=[(s_1, $task_{j_1}$), (s_2, $task_{j_2}$), ... , (s_n, $task_{j_n}$)] of Algorithm 7-7. Furthermore, before and after each s_a is executed, there is a one-to-one mapping between the task states of *I* and *J*, i.e., when $task_{i_a}$ and $task_{j_a}$ are at s_a, for each task *p* executing Algorithm 7-8, there exists a task *q* executing Algorithm 7-7 such that *p* and *q* are in the same task state.[26]

Proof. Before s_1 is executed, all tasks are inactive. Hence, both algorithms start from the same initial task configuration. Inductively assume that the claim is true over s_1 through s_{k-1} and that task *i* executes s_k of Algorithm 7-8. We show that the claim holds after s_k.

(1) Let s_k be the read request region. When *i* is at this region it is an inactive reader, and, by the induction hypothesis, there exists a task *j* that is an inactive reader in Algorithm 7-7. Since we can specify the execution of 7-7, we let *j* execute until it is at the read request region. By claim 20, *Val* is determined by the number of operating tasks (i.e., running

[26] Note that the claim does not state that when *p* is on *RWaitQ* (*WWaitQ*) in Algorithm 7-8, then *q* is on the corresponding queue in Algorithm 7-7. We believe this stronger condition holds, but it is not needed to prove that Algorithm 7-1 satisfies properties **P1**–**P4**.

readers, waiting readers, running writers, waiting writers, migrating writers). Hence, by the induction hypothesis, when inactive reader *i* and inactive reader *j* are at s_k in their respective algorithms, the value of *Val* is the same in each algorithm. Thus, when *i* requests the resource and becomes either a running reader ($Val < MaxR$) or a waiting reader ($Val \geq MaxR$), *j* performs the same transition when it requests the resource. Note that if *i* and *j* become waiting readers, each performs an *RWaitQ* insertion (when it reaches the *RWaitQ* Block) within a finite time.

(2) Let s_k be the *RWaitQ* insertion within *RWaitQ* Block. Suppose *i* is the *M*th task at the *RWaitQ* insertion (i.e., the *M*th task inserted onto *RWaitQ*) of Algorithm 7-8. Then over s_1 to s_{k-1}, $N \geq M$ tasks executing Algorithm 7-8 and an equal number of tasks executing Algorithm 7-7 had their read requests denied[27]: By (1), over s_1 to s_{k-1}, for each inactive task of Algorithm 7-8 that had its resource read request denied, there exists an inactive task of Algorithm 7-7 that also had its resource request denied; furthermore, when a task has its resource request denied, it performs an *RWaitQ* insertion within a finite time.

As *i* is the *M*th task at the *RWaitQ* insertion, exactly $M-1$ *RWaitQ* insertions have been performed over s_1 to s_{k-1} in Algorithm 7-8. By the induction hypothesis, an equal number of *RWaitQ* insertions have been performed over the same interval in Algorithm 7-7. This implies that the *M*th task *j* of Algorithm 7-7 that has had a read request denied has not yet blocked. We now let *j* execute in 7-7. Since $v_j = v_i$ and $v_j \geq MaxR$, *j* can reach s_k. Since *RWaitQ* insertions do not alter task states, the claim holds after the *RWaitQ* insertions of *i* and *j*.

The two algorithms differ only with respect to the code segments that appear in Figure 7-3, and we have just shown that the claim holds over these code segments. Clearly, the claim holds over the remaining regions and wait queue operations. □

Observe that in Algorithms 7-7 and 7-8, the states of individual tasks are not known. Resource management is based on the existence or nonexistence of (unidentified) tasks in various states. This information is inferred from the values of the variables *Val* and *Cnt*. Similarly, the readers/writers criteria **P1–P4** stated at the beginning of the chapter address only the numbers (or

[27] Since *i* calculates an *RWaitQ* insertion ticket number after performing the read request region, more than *M* read requests may have been denied over s_1 to s_{k-1}.

existence) of readers and writers that are using or waiting for the resource; they are not concerned with the policy by which waiting tasks are chosen to become running tasks. This is the only point over which Algorithms 7-7 and 7-8 differ.

Claim 34. Algorithm 7-8 satisfies properties **P1–P4**.

Proof. Suppose not. Then there exists an execution history *I* of Algorithm 7-8 that fails to satisfy the properties. By claim 33, there exists an execution history *J* of Algorithm 7-7, corresponding to *I*, such that there is a one-to-one mapping between the task states of *I* and *J*. Since *I* fails to satisfy the four readers/writers properties, so must *J*. But this is a contradiction by claim 32. □

Algorithm 7-8 is transformed into Algorithm 7-9 by moving the *WWaitQ* ticket number calculation from the write request and write rerequest regions to immediately before the *WWaitQ* Block operations.

```
reader i:  I1:  region  ‘‘read request’’
           I2:      v_i := F&A(Val, 1);
           I3:      if v_i ≥ MaxR then
           I4:      end if
           I5:  end region;
           I6:  if v_i ≥ MaxR then
           I7:      my#_i := F&A(r#, 1) mod MaxT;
           I8:      if my#_i = MaxT − 1 then F&A(r#, −MaxT) end if;
           I9:      Block(RWaitQ)
          I10:  end if;
          I11:  ‘‘read’’
          I12:  region  ‘‘read release’’
          I13:      v_i := F&A(Val, −1)
          I14:  end region;
          I15:  if v_i > MaxR then
          I16:      region  ‘‘last scheduler test’’
          I17:          c_i := F&A(Cnt, 1)
          I18:      end region;
          I19:      if c_i = MaxR − 1 then
          I20:          Cnt := 0;
          I21:          repeat until Unblock(WWaitQ)
          I22:      end if
          I23:  end if;
```

Algorithm 7-9a. *RWaitQ* ticket calculation at I7-I8.

```
writer j:  J1:  region ‘‘write request’’
           J2:      v_j := F&A(Val, MaxR);
           J3:      if v_j ≥ MaxR then
           J4:      end if
           J5:  end region;
           J6:  if v_j > 0 then
           J7:      if v_j < MaxR then
           J8:          region ‘‘rerequest resource’’
           J9:              c_j := F&A(Cnt, MaxR–v_j);
          J10:              if c_j ≠ v_j then
          J11:              end if
          J12:          end region;
          J13:          if c_j = v_j then
          J14:              Cnt := 0
          J15:          else
          J16:              my#_j := F&A(w#, 1) mod MaxT;
          J17:              if my#_j = MaxT – 1 then F&A(w#, –MaxT) end if;
          J18:              Block(WWaitQ)
          J19:          end if
          J20:      else
          J21:          my#_j := F&A(w#, 1) mod MaxT;
          J22:          if my#_j = MaxT – 1 then F&A(w#, –MaxT) end if;
          J23:          Block(WWaitQ)
          J24:      end if
          J25:  end if;
          J26:  ‘‘write’’
          J27:  region ‘‘write release’’
          J28:      v_j := F&A(Val, –MaxR)
          J29:  end region;
          J30:  if v_j mod MaxR > 0 then
          J31:      b_j := v_j mod MaxR;
          J32:      if v_j > 2*MaxR then
          J33:          Cnt := MaxR–b_j
          J34:      end if;
          J35:      for 1 to b_j do
          J36:          repeat until Unblock(RWaitQ)
          J37:      end for
          J38:  else
          J39:      if v_j ≥ 2*MaxR then
          J40:          repeat until Unblock(WWaitQ)
          J41:      end if
          J42:  end if;
```

Initially: $RWaitQ = WWaitQ = \emptyset$; $Cnt = Val = r\# = w\# = 0$;

Algorithm 7-9b. *WWaitQ* ticket calculations at J16-J17 and J21-J22.

Claim 35. Algorithm 7-9 satisfies properties **P1–P4**.

Proof. Similar to claims 33 and 34. □

Finally, Algorithm 7-9 is transformed into Algorithm 7-1 by first eliminating the (vacuous) **if** statements from the read request, write request, and write rerequest regions. Each region then consists of a single, atomic fetch-and-add. The region delimiters, being redundant, are therefore removed. Lastly, each *RWaitQ* (*WWaitQ*) Block operation is modified so that the *RWaitQ* (*WWaitQ*) ticket number calculation is performed within each Block, instead of immediately before it. Clearly, Algorithm 7-1, like all of the preceding algorithms, satisfies properties **P1–P4**.

7.5. Chapter Summary

In this chapter we presented a fair block-queue readers/writers implementation and set out to prove that it satisifies the following correctness criteria: readers and writers are mutually exclusive, writers are mutually exclusive, readers and/or writers do not deadlock, and multiple readers can use the resource when there are no active writers.

Rather than proving these properties directly, we first considered a second implementation in which fetch-and-adds were replaced with critical code sections called regions. Within each region we introduced additional variables that recorded the states of all tasks in the system. We then restated our four correctness criteria as assertions over these state variables and proved that the second implementation does satisfy these criteria. Through a series of transformations on the second implementation, we produced the first implementation. In addition, the four correctness criteria were proved to hold over these transformations; therefore, we concluded that our first implementation satisfies the four stated readers/writers properties.

CHAPTER 8

Conclusion

In this thesis we addressed serialization-free algorithms to be used in an operating system for a shared-memory MIMD machine that supports the fetch-and-add coordination primitive. A major design objective was to enable the operating system to be largely bottleneck-free. To meet this goal we advocated a self-service paradigm for the operating system: First, each PE serviced its own requests to avoid central server bottlenecks. Second, data shared among the PEs were held in concurrently accessible data structures to prevent access to such information from becoming a new source of bottlenecks. We indicated how this two-part strategy may be applied to the memory and task components of the operating system. For this application the types of requests that a PE was allowed to service for itself included memory allocation and task creation and coordination. Two data structures that proved central to parallelizing many memory and task activities were concurrently accessible queues and multiqueues, the subject of our second and third chapters.

First, we will summarize our results. Second, we will indicate the current status of our work vis-à-vis an actual shared-memory MIMD machine, the NYU Ultracomputer. Finally, we will discuss possible extensions to the thesis.

8.1. Thesis Summary

A key to obtaining many of our parallel algorithms (e.g., memory allocation) is the concurrently accessible queue data structure introduced by Gottlieb et al. [83b]. Recall that they presented concurrent insertion and deletion algorithms for sequentially allocated queues and that Rudolph [82] outlined an implementation for insertion and deletion on linked allocated queues. We completed Rudolph's work by providing fully defined linked allocated queue implementations. Then, for both sequentially and linked allocated queues, we defined interior access operations. An important feature of interior access operations is that their implementations do not reduce the concurrent accessibility of a queue.

The most notable interior access operation is removal. We showed that removing an interior item from a sequentially allocated queue is a relatively simple task. Performing a removal on a linked allocated queue is complicated by the ordering required on each linked list. To maintain the FIFO queue property, it is necessary to record how many times a given item's list successor has been removed. As we will note below, the removal operation is required by many of our algorithms.

Our algorithms also rely on another concurrently accessible data structure, the multiqueue. Recall that multiqueues (as defined by Gottlieb et al. [83b]) permit multiple copies of the same item to be inserted with far greater efficiency than can be achieved with concurrently accessible queues. For each type of multiqueue, linear and logarithmic, we gave implementations for both sequentially and linked allocated variants. We improved the performance of the original sequentially allocated multiqueue insertion and deletion algorithms of Gottlieb et al. [80] by eliminating the need for non-final deletions to execute readers/writers protocols. We also showed how an array or queue of items can be treated as a single multiqueue entry.

Four new multiqueue operations were defined: multi-deletion, generic type one interior access, and the specific type two interior access primitives of RaiseMult and LowerMult. The first takes advantage of the multiqueue structure to efficiently implement the deletion of multiple items. The second allows an interior item's application dependent information to be accessed. The third increments an item's multiplicity, and the fourth decrements an item's multiplicity or, if this is not possible, removes the item.

In general, supporting interior access on a multiqueue raises the cost of insertion and deletion, but does not reduce the maximum concurrency of the multiqueue (the exception is when an interior access is being done on an active deletion site, in which case concurrent deletion activity is curtailed). We presented readers/writers and writer priority readers/writers implementations of type one and type two interior access operations. We noted that while the writer priority variants admit starvation even in the absence of interior access activity, non-final deletions are as efficient as when interior access is not supported.

We presented a number of parallel memory allocation algorithms that use concurrently accessible queues and multiqueues to represent the pool of free blocks. Many of the algorithms are parallel analogs to existing serial algorithms such as Knuth's boundary tag method and the binary buddy system. The specific algorithms and their data structures are as follows: the parallel boundary tag method uses a concurrently accessible queue-like data structure, good fit (a cross between first fit and the binary buddy system) and the second version of the buddy system use concurrently accessible queues, and the second paging scheme uses concurrently accessible multiqueues. The remaining parallel algorithms, the first versions of the buddy system and paging, represent the pool of free blocks with trees and arrays of counters (manipulated via fetch-and-add). For our queue-based memory algorithms, the operation of interior removal is needed to support block coalescing. Also, the multiqueue-based paging scheme is an example of a multiqueue that consists of arrays (page tables) of items (pages); it uses the multi-delete operation to obtain multiple pages at a time.

Concurrently accessible queues and multiqueues play significant roles in our parallel implementations of various task activities; these include creation, scheduling, destruction, and coordination (via semaphore, readers/writers, and event primitives). For example, the system Ready Queue is constructed from concurrently accessible queues and multiqueues, and other resource wait queues are constructed from concurrently accessible queues. In addition, to support the parallelization of certain Ready Queue operations, three types of Ready Queue multiqueue objects were defined: templates, queues tagged with multiplicity, and queue pointers tagged with multiplicity.

To illustrate how the queue and multiqueue data structures were used, let us review the implementations of task creation, task scheduling, and the unblocking of tasks that are on a (read or event) wait queue: First, to create M tasks in parallel, a template with multiplicity M is inserted onto a multiqueue component of the Ready Queue. Each deletion performed on this item by an available PE results in the creation of a task which may then be executed by the deleting PE. Second, since the system Ready Queue is concurrently accessible, task scheduling can be done in parallel by multiple PEs. Third, the unblocking of N tasks on a wait queue can be accomplished either by having multiple tasks delete tasks from the wait queue and insert them onto the Ready Queue or by having one task insert the entire wait queue as a single item with multiplicity N onto the Ready Queue.

Certain activities such as task termination are parallelized, in part, via fetch-and-adds on shared counters. Another characterization of our parallelization techniques is that tasks associated with an activity participate in its execution. For example, a task to be destroyed recursively signals its children to destroy themselves. Another example is the first method cited above for unblocking tasks, where tasks that were blocked, unblock other tasks. We also comment that in the case of destruction, the queue remove operation and the multiqueue LowerMult operation are used to remove tasks and templates from queues and multiqueues (and without reducing the concurrency of either queue structure).

We also compared our parallel methods for creating, destroying, and unblocking tasks against serial alternatives. The comparisons suggest that for large numbers of tasks, the parallel methods should be chosen.

We showed how high-level parallel constructs (e.g., loops and compound statements) can be implemented with the task primitives and concurrently accessible queues and multiqueues previously described. Two different implementations for a parallel construct that forms an N-way fork in control have been given. One implementation places N tasks onto a concurrently accessible queue component of the Ready Queue in $O(N)$ time; whereas, the other implementation does so in constant time by placing a task template for the construct with multiplicity N onto a concurrently accessible multiqueue component of the Ready Queue. We also considered a chunking variation where the number of tasks assigned to a construct and the number of clauses within the construct that each task evaluates is determined while the construct is being executed. The efficiency of this method is aided by our ability to remove a construct's template from the Ready Queue (via the LowerMult primitive).

Our block-queue readers/writers implementations are arguably the most complicated algorithms in the thesis. Still, we were able to prove that a variant of one of our fair block-queue readers/writers implementations satisfies the following correctness criteria: readers and writers are mutually exclusive, writers are mutually exclusive, readers and/or writers do not deadlock, and multiple readers can use the resource when there are no active writers.

8.2. Current Status

We presented numerous algorithms for the shared-memory MIMD machine model described in chapter 1. A number of these algorithms have been implemented and are currently being used in the operating system for the NYU Ultracomputer prototypes. Each prototype consists of 8 PEs connected via a bus to a shared memory of 8MB (see Gottlieb [87] for details). The Ultracomputer's operating system, Symunix (Edler et al. [88a, 88b]), is based on the self-service paradigm. Of the algorithms given in the thesis, Symunix uses the linked allocated queues, linked allocated linear multiqueues, and the queue-based version of the binary buddy system.

8.3. Future Research

We presented six different parallel algorithms for performing memory allocation: a boundary tag method, good fit, two versions of the binary buddy system, and two versions of (non-demand) paging. Of these, only the two versions of the buddy system have been implemented on the Ultracomputer prototype and compared (Berke [85]). The comparison indicated that the queue-based version of the buddy system is approximately 60% faster than the tree-based version and hence was chosen for Symunix. The remaining memory allocation algorithms still need to be examined.

Several of our parallel memory allocation implementations (e.g., good fit, the second versions of both the buddy system and paging) represent the pool of free memory blocks with concurrently accessible queues and multiqueues. These data structures impose an unnecessary FIFO ordering over the processing of block requests. Below we describe various modifications to the queue and multiqueue data structures and algorithms that may be considered when the FIFO property need not be supported. A goal of these modifications is to produce simpler data structures and faster operations than the corresponding queue and multiqueue data structures and operations.

Let us refer to a repository for unordered items, i.e., items that need not be processed in FIFO order, as a *pile*. Analogously, a pile with multiplicity or *multipile* is a pile that consists of items tagged with multiplicity.

Let us begin by considering how a sequentially allocated queue may be converted into a sequentially allocated pile. Initially, let us assume that only insertions and deletions are done on a pile. First, since piles need not enforce the FIFO property, pile site protocols can be implemented via weak (fair or unfair) P/V operations (a global protocol, such as the ticket numbering scheme described in chapter 2 is unnecessary). Second, we need to consider how operations should be assigned to sites. Suppose that we were to use the following site calculation scheme for piles. Let a pile operation be assigned an initial site i either deterministically (e.g., via a fetch-and-add on a counter) or randomly. If the operation finds that site i is locked by another operation, it picks a new site j (e.g., by moving to the next consecutive site) and sees if site j is free. The operation continues to visit sites until one is found that is unlocked and either empty, if the operation is an insertion, or full, if the operation is a deletion; note that this strategy can starve an operation if the operation always chooses locked sites. We claim (without proof) that piles should use the queue site calculation procedure (which sends consecutive operations to consecutive sites and does not permit operations to move to new sites) in preference to the above scheme. First, the queue site calculation scheme can be made starvation-free (by using fair P/V operations). Second, we believe that the average time a task busy-waits to lock a site should be no longer than in the above scheme since the queue calculation site procedure evenly distributes operations over pile sites.

We note that interior access operations on piles may be implemented in an identical manner to their queue counterparts (as interior access operations act on sites).

In summary, we expect that sequentially allocated piles will be no easier to implement than sequentially allocated queues.

Next, let us consider how a linked allocated queue may be converted into a linked allocated pile. Again, as with sequentially allocated piles, operations may use weak P/V site protocols to gain exclusive access to lists. First, assume that only insertions and deletions are performed on piles. Then, at each pile site, instead of using a FIFO linked list, we may use a stack, i.e., a singly linked list processed in last-in, first-out (LIFO) order. Pile insertion and deletion site calculations may follow either one of the two methods described above: use the standard queue site calculation or choose a site (e.g., randomly) and if it is locked visit other sites until a free one is found. The latter method can lead to unbalanced lists and the starvation of operations. Hence, we believe that the queue site calculation method should be chosen when implementing piles.

Now suppose interior access operations are to be done on linked allocated piles. Consider interior removal. First, the stack associated with each site should be represented by a doubly linked list (to facilitate removal). Second, we may wish to eliminate the missing item counter scheme that is found in the queue implementation. Recall that for queues, associated with each site and queued item is a counter that records the presence of holes in the queue that occur as a result of interior removals. The counters are referenced by deletions and used to preserve the FIFO property. Suppose we eliminate these counters from the pile implementation. As a consequence of removals, the lists of the piles may become unbalanced. Hence, if deletions are assigned to lists using the standard queue site calculation procedure, then a deletion could be sent to an empty list. In this case, the deletion should not be permitted to remain at that site since a subsequent insertion might never be performed there. Thus, with no missing item counts, a deletion must be permitted to visit multiple pile sites until a nonempty one is found. Such a policy can starve individual deletions (if they always choose lists that are already locked). Also, deletions may starve if we allow a removal to merely take an item from a pile without indicating in some way that the pile contains one less item. That is, assume that *PCount* equals the total number of items in a pile (just as *Qlower* equals the total number of items in a queue) and that a deletion enters the pile by performing a TDR of 1 on *PCount*. If a removal does not decrement *PCount* by 1 when removing an item, a deletion could enter the pile and fail to find an item; it would starve. Hence, removals should follow a policy similar to the one used by the multiqueue LowerMult operation: When a removal takes an item from a pile, let it perform a TDR of 1 on *PCount*. If the TDR succeeds, the removal is done; otherwise, the removal adds one (via fetch-and-add) to a nil pool counter. A deletion visits pile sites until it either obtains an item or has visited all sites. In the latter case, the deletion then performs a TDR of 1 on the nil pool counter. If successful, the deletion returns a value of nil to its caller; otherwise, the deletion repeats the above procedure until it finds a nonempty site or can take an item from the nil pool.

In summary, we believe that linked allocated piles will be, at best, marginally easier to implement than linked allocated queues.

Next, let us consider how a linear multiqueue, either sequentially or linked allocated, may be streamlined when only multipiles need be supported. Recall from chapter 3 that only one site of a linear multiqueue can be accessed by deletions at a time. This restriction is needed to support the FIFO queue property. In implementing a linear multipile, we may therefore wish to eliminate this constraint. Whether this is advisable depends in part on the level of deletion activity. We believe that having only one active deletion site should be adequate when the average number of deletions that wait for a new site to be opened is relatively small; otherwise, we may permit deletions to visit multiple sites of a multipile in search of a nonempty location. Again, as in the case of piles, the latter policy may subject an operation to starvation.

Interior access operations on a linear multipile may be implemented in essentially the same manner as their multiqueue counterparts. In particular, raising an item's multiplicity via a RaiseMult operation should involve the same steps as in the multiqueue case. To lower an item's multiplicity via a LowerMult operation, we believe that a nil pool counter is still needed, just as in the multiqueue implementation. The counter guarantees that deletions that enter the multipile are always assured of being able to obtain an item (either from a nonempty site or from the nil pool).

Several optimizations may be made specifically for linked allocated linear multipiles. First, stacks can be used in place of FIFO lists at multipile sites. Second, missing item counts need not be maintained to support the removal of items.

In summary, we expect that sequentially allocated linear multipiles will be no easier to implement than sequentially allocated linear multiqueues and that linked allocated linear multipiles will be, at best, marginally easier to implement than linked allocated linear multiqueues.

Finally, let us consider how we may convert a logarithmic multiqueue into a logarithmic multipile. Here is probably our best source of optimizations. Recall that to support the FIFO queue property a logarithmic multiqueue requires two trees and two arrays (of cells or serially accessible linked lists). To implement a logarithmic multipile, we need only one tree and one array. In the logarithmic multiqueue, once deletions have emptied a tree, incoming deletions have to wait until the next tree is opened (by a ‘‘last’’ deletion) before proceeding. For a logarithmic multipile implemented via one tree and array, this wait (and the requirement that the last deletion on a tree open up the next one) is therefore eliminated. Unlike the other queue and multiqueue implementations, deletions on a multipile are always directed to nonempty sites since they are guided by the counters of the multipile’s tree. However, since multipile sites may be emptied in non-serial order, an insertion on a sequentially allocated logarithmic multipile may have to visit multiple sites until a nonempty one is found. Note that insertions could be directed to empty sites (just as deletions are directed to non-empty sites) by maintaining a second tree of counters, where each counter recorded the number of empty sites associated with the leaves of its subtree; deletions that emptied sites could ascend this tree, incrementing each counter on the path from the leaf associated with the site to the root of the tree. For a linked allocated logarithmic multipile, insertions never have to visit multiple sites: an insertion can always be done at any given site. Hence, unlike the other pile and multipile variants, a linked allocated logarithmic multipile does not pose problems with respect to directing deletions and insertions to nonempty and empty sites. Interior access operations on logarithmic multipiles can of course be done along the lines of their multiqueue counterparts. Also, as in each of the previous linked allocated implementations, a linked allocated logarithmic multipile may substitute stacks for the LIFO lists of the logarithmic multiqueue and missing item counts may be eliminated.

In summary, we expect that sequentially allocated logarithmic multipiles will be easier to implement than sequentially allocated logarithmic multiqueues: the memory requirements of the former are half that of the latter and because of this difference, on average, deletions from the former should be faster than deletions from the latter. An implementation of a sequentially allocated logarithmic multipile will have to deal with the problem of insertions locating empty sites, but we do not anticipate that this will be difficult to solve. A linked allocated logarithmic multipile has the same advantages over a linked allocated logarithmic multiqueue as a sequentially allocated logarithmic multipile has over a sequentially allocated logarithmic multiqueue. In addition, insertions into a linked allocated logarithmic multipile do not have to search for empty sites. Therefore, we believe that a linked allocated logarithmic multipile should be significantly easier to implement than a linked allocated logarithmic multiqueue and the former data structure should be considered as a future research topic.

APPENDIX A

A Review of Queues With Priority

Gottlieb et al. [80] give a highly parallel implementation of a ‘‘queue with priority,’’ i.e., a queue consisting of items tagged with priorities, that supports two operations: a ‘‘priority insertion’’ that adds an item with priority to the queue, and a ‘‘priority deletion’’ that removes the first item of highest priority. In reviewing this implementation we assume that the reader is already familiar with the contents of chapters 2 and 3.

The set S of priorities is assumed to be finite and fixed in advance. Let

$$S = \{0, 1, \ldots, \#P-1\},$$

where $\#P=2^h$ and 0 is the highest priority. A queue with priority is then represented by an array of concurrently accessible queues $Q[0..\#P-1]$. A priority insertion of an item I with priority p adds I to $Q[p]$. A priority deletion deletes an item from the highest priority nonempty queue. When $\#P$ is large and many of the highest priority queues are empty, a linear search by deletions over the queue array to find a non-empty queue is expensive. Gottlieb et al. [80] reduce this cost by presenting an implementation that guides deletions directly to the highest priority non-empty queue in the same manner that deletions on logarithmic multiqueues are directed to the first non-empty multiqueue site.

A queue with priority, QP, is augmented with a complete binary tree T of height h. The ith leaf of T is paired with the ith queue of QP. For each node N of T there is a counter that records the number of items in the queues associated with the leaves of N’s subtree. An insertion of an item I with priority p adds I to $Q[p]$ and then increments the counters of the nodes on the path from leaf p to the root. A deletion starts at the root and is guided by the counters to the highest priority non-empty queue (i.e., the leftmost leaf with a non-zero count), where a standard queue delete operation is then performed.

The simplest queue with priority implementation is given below. The array of queues consists of sequentially allocated concurrently accessible queues (as defined in chapter 2). It is assumed that these queues are not subject to interior access. More complex queues with priority may be implemented by constructing the queue array from any of the other types of queues given in chapters 2 and 3. Also, if desired, interior access operations (including removal) can be supported on the queues.

```
QueueWithPriority = record
                        #P  : integer;
                        Cnt : array[1..2*#P−1] of integer;
                        Q   : array[0..#P−1] of Seq_Queue
                    end;

Object = record
             Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
    Cnt[] := 0;   all queues are empty

function QPrioInsert (Item     : Object_Pointer;
                      Priority : integer;
                      QP       : QueueWithPriority);
var n : integer;
   with QP do
   if Seq_QInsert(Item, Q[Priority]) then
      n := leaf corresponding to Q[Priority];
      while n ≠ root do
         F&A(Cnt[n], 1);
         n := parent(n)
      end while;
      F&A(Cnt[root], 1);
      return true
   else
      return false  Q[Priority] is full
   end if
   end with
end function QPrioInsert;
```

```
function QPrioDelete (Item : Object_Pointer;
                      QP   : QueueWithPriority);
var n, p : integer;
    with QP do
    n := root;
    if TDR(Cnt[n], 1) then      QP is not empty
        while n is not a leaf do
            if TDR(Cnt[leftson(n)], 1) then
                n := leftson(n)
            elseif TDR(Cnt[rightson(n)], 1) then
                n := rightson(n)
            end if
        end while;
        p := queue index corresponding to leaf n;
        Seq_QDelete(Item, Q[p])
        return true
    else
        return false  QP is empty
    end if
    end with
end function QPrioDelete;
```

APPENDIX B

A Review of Multiqueue Insertion and Deletion

We review the insertion and deletion algorithms of Gottlieb et al. [80] for the sequential implementations of linear and logarithmic multiqueues. In these algorithms non-final and final deletions explicitly coordinate their references to the queue with one another. In the presentation below, familiarity with the contents of chapter 3 is assumed.

B.1. Linear Multiqueues

The sequentially allocated multiqueue is implemented as an array *Cell*[0..*Qsize*−1]. Associated with each cell are two fields, *Remaining* and *Removed*, that are initialized to the multiplicity of the item that is inserted into the cell.

An insertion calculates its site using the standard technique described in chapter 2. It then sets its site's *Remaining* and *Removed* counts to the inserted item's multiplicity. It also increments the count *Qlower* of items in the queue by this value. Thus, multiqueue insertion is almost identical to its concurrently accessible queue counterpart (see section 2.1.1).

A deletion cycles decrementing (via fetch-and-add) the *Remaining* component of the front queue cell until it obtains a positive value. It then copies the item in the cell and decrements the cell's *Removed* component. The *final* deletion of the cell, the one driving *Removed* to zero, removes the item from the queue and advances the front queue pointer *Qfront* to the next cell.

Qfront is referenced by incoming deletions to determine which cell is the current deletion site and by final deletions to advance the site to the next cell. These two activities are mutually exclusive. If they were not, an incoming deletion could decrement a positive *Remaining* component of an item that was not at the front of the queue. The exclusion is performed through readers/writers code where the incoming deletions are the readers and the final deletions are the writers. Once a site is emptied by a final deletion, subsequent deletions cannot make any further progress until a new site is opened by the final deletion. Therefore, final deletions are given priority over non-final deletions (i.e., writer priority readers/writers coordination is used).

```
Seq_LinMQueue =     record    Sequentially Allocated Linear MultiQueue
                        Qlower, Qupper, Qsize,
                        Qfront, Qrear, #W : integer;
                        RW_Act : integer;
                        Cell : array[0..Qsize–1] of CellElement
                    end;

CellElement = record
                  Remaining,
                  Removed   : integer;
                  Value     : Object_Pointer
              end;

Object = record
             Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;

Initially:
  Qlower, Qupper, Qfront, Qrear = 0;
  #W = 0;                     number of active writers
  RW_Act = MaxR;
  for 0≤j<Qsize,
    Cell[j].Remaining = 0;
```

```
function Seq_LinMQInsert (Item : Object_Pointer;
                          M    : integer;   item's multiplicity
                          Q    : Seq_LinMQueue);
var i : integer;
  with Q do
  if TIR(Qupper, 1, Qsize) then
    i := F&A(Qrear, 1) mod Qsize;
    if i = Qsize−1 then F&A(Qrear, −Qsize) end if;
    Cell[i].Value := Item;
    Cell[i].Removed  := M;
    Cell[i].Remaining  := M;
    F&A(Qlower, M);
    return true
  else
    return false
  end if
  end with
end function Seq_LinMQInsert;
```

```
function Seq_LinMQDelete (Item : Object_Pointer;
                          Q    : Seq_LinMQueue);
var d, s : integer;
  with Q do
  if TDR(Qlower, 1) then
    s := 0;
    repeat
      WP_Read_Request(RW_Act, #W);
        d := (Qfront + 1) mod Qsize;
        if Cell[d].Remaining > 0 then
          s := F&A(Cell[d].Remaining, −1)
        end if;
      WP_Read_Release(RW_Act)
    until s > 0;
    Item := Cell[d].Value;
    if F&A(Cell[d].Removed, −1) = 1 then    final delete
      WP_Write_Request(RW_Act, #W);
        Qfront := (Qfront + 1) mod Qsize;
      WP_Write_Release(RW_Act, #W);
      F&A(Qupper, −1)
    end if;
    return true
  else
    return false
  end if
  end with
end function Seq_LinMQDelete;
```

Observing that the maximum number of deletions performed on a cell is bounded by the item's multiplicity enables us to eliminate the readers/writers code from the deletion algorithm. Instead of having each incoming deletion read *Qfront* and test a cell's *Remaining* count, the number of deletions is restricted to the multiplicity of the front item (by initializing *RW_Act* to an item's multiplicity instead of to the constant *MaxR*). Thus, a final deletion does not have to explicitly block incoming deletions from referencing these queue components.

This observation leads to the modified implementation given in section 3.1.1.2 in which a final deletion moves *Qfront* to the next cell *c* and sets *FrontMult* (the equivalent of *RW_Act* above) to *c*'s multiplicity. Insertions update *FrontMult* when their site is also the deletion site. Thus, references to these variables by final deletions and insertions are contained in critical code sections. When the average item multiplicity $\overline{M}$ is large, the majority of queue operations consist of non-final deletions. Eliminating the need for these operations to execute readers code lowers the overall cost of moving items on and off the queue.

B.2. Logarithmic Multiqueues

The logarithmic multiqueue is implemented as an array of cells augmented with an auxiliary tree of counters as described in section 3.2.1.1. An insertion places an item with multiplicity M into a cell and then ascends the tree, adding M to each node's count that lies on the path from the leaf associated with the cell to the root. Insertion queue wraparound is handled by using two queues (each of size *Qsize*), each with its own count tree. Even phased insertions are performed on one queue and odd phased insertions on the other one.

The queue for the current deletion phase is defined by *CurrentQ*. A deletion cycles decrementing the current queue's root count via a TDR until a positive value is found. It then descends the queue's tree, guided by the counters, to a leaf node where it makes a copy of the item associated with that node. The *last* deletion of a phase switches *CurrentQ* to the other queue to open the next phase. Incoming deletions and last deletions are readers and writers, respectively, of *CurrentQ*. The implementation of Gottlieb et al. [80] once again employs readers/writers protocols to coordinate deletion activities.

```
Seq_LogMQueue = Sequentially Allocated Logarithmic MultiQueue
                record
                    Qlower, Qupper, Qsize, Qrear,
                    CurrentQ, #W, #Done : integer;
                    RW_Act : integer;
                    Queue : array[0..1] of
                              record
                                  Cnt  : array[1..2*Qsize−1] of integer;
                                  Cell : array[0..Qsize−1] of CellElement
                              end
                end;

CellElement = record
                  Mult  : integer;
                  Value : Object_Pointer
              end;

Object = record
             Application_Defined_Attributes
         end;

Object_Pointer = ↑Object;
```

Initially:

```
Qlower, Qupper, Qrear, CurrentQ, #W = 0;
#Done = 1;
RW_Act = MaxR;
Queue[0..1].Cnt[] = 0;
```

```
function Seq_LogMQInsert (Item : Object_Pointer;
                          M    : integer;   Item's multiplicity
                          Q    : Seq_LogMQueue);
var i, index, n : integer;
    with Q do
    if TIR(Qupper, 1, Qsize) then
        i := F&A(Qrear, 1) mod 2*Qsize;
        if i = 2*Qsize–1 then F&A(Qrear, –2*Qsize) end if;
        index := i div Qsize;          select queue
        i := i mod Qsize;              index into cell array
        Queue[index].Cell[i].Value := Item;
        Queue[index].Cell[i].Mult  := M;

        n := leaf corresponding to cell i of Queue[index];
        while  n ≠ Root do   ascend tree
            F&A(Cnt[n], M);
            n := parent(n)
        end while;
        F&A(Cnt[n], M);

        F&A(Qlower, M);
        return true
    else
        return false
    end if
    end with
end function Seq_LogMQInsert;
```

```
function Seq_LogMQDelete (Item : Object_Pointer;
                          Q    : Seq_LogMQueue);
var d, index, n    : integer;
    success        : boolean;

    with Q do
    if TDR(Qlower, 1) then
        repeat
            WP_Read_Request(RW_Act, #W);
                index := CurrentQ;
                success := TDR(Queue[index].Cnt[root], 1);
            WP_Read_Release(RW_Act)
        until success;
        n := root of Q[index];
        while n is not a leaf do
            if TDR(Cnt[leftson(n)], 1) then
                n := leftson(n)
            elseif TDR(Cnt[rightson(n)], 1) then
                n := rightson(n)
            end if
        end while;

        d := cell index corresponding to leaf n;
        Item := Queue[index].Cell[d].Value;

        if F&A(Queue[index].Cell[d].Mult, –1) = 1 then    final delete
            F&A(Qupper, –1);
            if F&A(#Done, 1) = Qsize then   last delete in phase
                #Done = 1;
                WP_Write_Request(RW_Act, #W);
                    CurrentQ := (CurrentQ + 1) mod 2;
                WP_Write_Release(RW_Act, #W)
            end if
        end if;
        return true
    else
        return false
    end if
    end with
end function Seq_LogMQDelete;
```

The modified logarithmic multiqueue implementation given in section 3.2.1.2 eliminates the readers/writers code from the above algorithms by applying the same strategy used in the linear case. Note the correspondence between the queue components *CurrentQ* and *RW_Act* defined above to the variables *DPhase* and *PhaseMult* of the modified queue algorithms.

The sequential and linked allocated logarithmic multiqueue implementations given in chapter 3 could have represented a multiqueue as two arrays and two trees instead of as one array and one tree, each of dimension two. The former representation was not chosen since its space requirements exceed those of the latter by approximately a factor of two when implementing linked allocated multiqueues. The additional space is necessary to maintain the FIFO queue property. Let us assume that two arrays and two trees are used in implementing a linked list allocated multiqueue, where the number of active deletion and insertion phases is limited to one and two, respectively. Even phased operations are performed on one queue Q_E and odd phased operations on the other queue Q_O. Suppose consecutive insertion phases p_0, p_1, and p_2 have been completed, where p_0 is even, and a final phase p_0 deletion is performed on the head item of a list L of Q_E. The new head item of L belongs to phase p_2. The deletion cannot ascend Q_E's tree with this new item's multiplicity until all of the items belonging to phase p_0 have been taken. Otherwise, the FIFO queue property could be violated. However, if Q_E's tree had two counters per node for phases $2j$ and $2(j+1)$, and similarly, Q_O's tree had two counters per node for phases $2j+1$ and $2(j+1)+1$, then the deletion could ascend the tree with the multiplicity of the phase p_2 item. In chapter 3, equivalent results are obtained by using one array and one tree with two counters per node.

Bibliography

Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ''Mach: A New Kernel Foundation for Unix Development,'' *Proc. USENIX Conf.*, pp. 93-112, June 1986.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Com- puter Algorithms, Addison-Wesley, Reading, MA, 1974.

Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, ''The Eden System: A Technical Review,'' *IEEE Trans. SW. Eng.*, **SE-11**, pp. 43-59, January 1985.

Anderson, J. P., S. A. Hoffman, J. Shifman, and R. J. Williams, ''D-825–A Multiple Computer System for Command and Control,'' *Proc. AFIPS Conf.*, **22**, pp. 86-96, 1962.

Anderson, D. W., F. J. Sparacio, and R. M Tomasulo, ''The IBM System/360 Model 91: Machine Philosophy and Instruction Handling,'' *IBM J. Res. Dev.*, **11**, pp. 8-24, 1967.

Arnold, J. S., D. P. Casey, and R. H. McKinstry, ''Design of Tightly-Coupled Multiprocessing Programming,'' *IBM Systems J.*, **13**, 1, pp. 60-92, 1974.

Bach, M. J., The Design of the Unix Operating System, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Baer, J.-L., ''Computer Architecture,'' *Computer*, **17**, pp. 77-87, October 1984.

Bailey, M. J., M. P. Barnett, and P. B. Burleson, ''Symbol Manipulation in Fortran - SASP I Subroutines,'' *CACM*, **7**, pp. 339-346, June 1964.

Ballhaus, W. F., Jr., ''Computational Aerodynamics and Supercomputers,'' *Digest of Papers: COMPCON Spring 84*, pp. 3-14, 1984.

Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, ''The Illiac IV Computer,'' *IEEE Trans. on Computers*, **C-17**, pp. 746-767, August 1968.

Batcher, K., ''Design of a Massively Parallel Processor,'' *IEEE Trans. on Computers*, **C-29**, pp. 836-840, September 1980.

BBN Laboratories, *Butterfly Parallel Processor Overview*, (Tech. Report), BBN Computer Co., Cambridge, MA, June 1985.

Beetem, J., M. Denneau and D. Weingarten, ''The GF-11 Supercomputer,'' *12th Annual Intl. Symp. on Computer Architecture*, (*Sigarch Newsletter*, **13**, 3), pp. 108-115, June 1985.

Benes, V., Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, New York, 1965.

Berke, W., private communication, 1985.

Berztiss, A. T., ''A Note on Storage of Strings'', *CACM*, **8**, pp. 512-513, August 1965.

Bigler, B. M., S. J. Allan, and R. R. Oldehoeft, ''Parallel Dynamic Storage Allocation,'' *Intl. Conf. on Parallel Processing*, pp. 272-275, 1985.

Birrell, A. D. and B. J. Nelson, ''Implementing Remote Procedure Calls,'' *ACM Trans. on Computer Systems*, **2**, pp. 39-59, February 1984.

Black, A. P., ''Supporting Distributed Applications: Experience with Eden,'' *Proc. 10th Symp. Operating Systems Principles*, pp. 181-193, 1985.

Borodin, A. and J. E. Hopcroft, ''Routing, Merging, and Sorting on Parallel Models of Computation,'' *Proc. 14th Ann. ACM Symp. Theory of Computing*, pp. 338-344, 1982.

Brantley, W. C., K. P. McAuliffe, and J. Weiss, ''RP3 Processor-Memory Element,'' *Intl. Conf. on Parallel Processing*, pp. 782-789, 1985.

Brinch Hansen, P., ''The Nucleus of a Multiprogramming System,'' *CACM*, **13**, pp. 238-241, April 1970.

Brinch Hansen, P., ''Concurrent Programming Concepts,'' *Computing Surveys*, **5**, pp. 221-245, December 1973a.

Brinch Hansen, P., Operating System Principles, Prentice-Hall, Englewood Cliffs, NJ, 1973b.

Brownbridge, D. R., L. F. Marshall, and B. Randell, ''The Newcastle connection–Or UNIXES of the world unite!,'' *Softw. Pract. Exper.*, **12**, pp. 1147-1162, December 1982.

Burns, J. E., ''Symmetry in Systems of Asynchronous Processes,'' *22nd Ann. Symp. on Foundations of Computer Science*, pp. 169-174, 1981.

Burns, J. E. and N. A. Lynch, ''Mutual Exclusion Using Indivisible Reads and Writes,'' *Proc. 18th Ann. Allerton Conf. on Communications, Control, and Computing*, pp. 833-842, October 1980.

Burroughs Corp., *Numerical Aerodynamic Simulation Facility Feasibility Study*, NAS2-9897, March 1979.

Cezzar, R. and D. Klappholz, ''Process Management Overhead in a Speedup-Oriented MIMD System,'' *Intl. Conf. on Parallel Processing*, pp. 395-403, 1983.

Cheriton, D. R., ''The V Kernel: A Software Base for Distributed Systems,'' *IEEE Software*, **1**, pp. 19-43, April 1984.

Cheriton, D. R. and T. P. Mann, ''Uniform Access to Distributed Name Interpretation in the V-System,'' *Proc. 4th Intl. Conf. Distr. Computing Systems*, pp. 290-297, 1984.

Cheriton, D. R. and W. Zwaenepoel, ''The Distributed V Kernel and Its Performance for Diskless Workstations,'' *Proc. 9th Symp. on Operating Systems Principles*, pp. 129-140, 1983.

Collier, W. W., ''Principles of Architecture for Systems of Parallel Processors,'', Tech. Report TR00.3100, IBM Poughkeepsie, N.Y., March 1981.

Comfort, W. T., ''Multiword List Items,'' *CACM*, **7**, pp. 357-362, June 1964.

Conradi, R., ''Some Comments on 'Concurrent Readers and Writers','' *Acta Informatica*, **8**, pp. 335-340, 1977.

Courtois, P. J., F. Heymans, and D. J. Parnas, ''Concurrent Control with 'Readers' and 'Writers','' *CACM*, **14**, pp. 667-668, October 1971.

Cytron, C., ''Limited Processor Scheduling of Doacross Loops,'' *Intl. Conf. on Parallel Processing*, pp. 226-234, 1987.

Davies, J., ‘‘Parallel Loop Constructs for Multiprocessors,’’ Tech Report UIUCDCS-R-81-1070, Univeristy of Illinois, Urbana, Illinois, May 1981.

DeBruijn, N. G., ‘‘Additonal Comments on a Problem in Concurrent Programming Control,’’ *CACM*, **10**, p. 137, March 1967.

DEC, ‘‘Multiprocessing Functional Specifications, ’’ *DECsystem 10 Software Notebook*, Digital Equipment Corp., Maynard, MA, 1971.

DEC, *DECsystem 10 Hardware Reference Manual*, DEC-10-XSRMA-A-D, Digital Equipment Corp., Maynard, MA, 1976a.

DEC, *Introduction to KL-10 based systems, Technical Description*, EK-KL-TD-002, Digital Equipment Corp., Maynard, MA, 1976b.

DEC, *DECsystem 10 Technical Summary*, Digital Equipment Corp., Maynard, MA, 1977.

Denning, D. J., ‘‘Virtual Memory,’’ *Computing Surveys*, **2**, pp. 153-190, September 1970.

Denning, D. J., ‘‘Parallel Computing and Its Evolution,’’ *CACM*, **29**, pp. 1163-1167, December 1986.

Dennis, J., ‘‘Data Flow Supercomputers,’’ *Computer*, **13**, pp. 48-56, November 1980.

Dennis, J. and E. C. Van Horn, ‘‘Programming Semantics for Multiprogrammed Computations,’’ *CACM*, **9**, pp. 143-155, March 1966.

Dijkstra, E. W., ‘‘Solution of a Problem in Concurrent Programming Control,’’ *CACM*, **8**, p. 569, September 1965.

Dijkstra, E. W., ‘‘Co-operating Sequential Processes’’ in Programming Languages, F. Genuys (ed.), Academic Press, NY, pp. 43-110, 1968a.

Dijkstra, E. W., ‘‘The Structure of the ‘THE’ Multiprogramming System,’’ *CACM*, **11**, pp. 341-346, May 1968b.

Dijkstra, E. W., ‘‘Hierarchical Ordering of Sequential Processes’’ in Operating Sys- tems Techniques, C. A. R. Hoare and R. H. Perrot (eds.), Academic Press, NY, pp. 72-93, 1972.

Dolev, D. and H. R. Strong, ‘‘Authenticated Algorithms for Byzantine Agreement,’’ *SIAM J. Comput.*, **12**, pp. 656-666, November 1983.

Dolev, D., N. A. Lynch, S. S. Pinter, and E. W. Stark, ‘‘Reaching Approximate Agreement in the Presence of Faults,’’ *JACM*, **33**, pp. 499-516, July 1986.

Dolev, D., C. Dwork, and L. Stockmeyer, ‘‘On the Minimal Synchronism Needed for Distributed Consensus,’’ *JACM*, **34**, pp. 77-97, January 1987.

Doran, R. W. and L. K. Thomas, ‘‘Variants of the Software Solution to Mutual Exclusion,’’ *Inf. Proc. Lett.*, **10**, pp. 206-208, July 1980.

Droughon, E., R. Grishman, J. Schwartz, and A. Stein, ‘‘Programming Considerations for Parallel Computers,’’ *IMM 362* Courant Institute, New York University, November 1967.

Edler, J., private communication, 1984.

Edler, J., A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. M. Wilson, ‘‘Issues Related to MIMD, Shared-Memory Computers: The NYU Ultracomputer

Approach,'' *Proc. 12th Annual Comp. Arch. Conf.*, pp. 126-135, 1985.

Edler, J., J. Lipkis, and E. Schonberg, ``Memory Management in Symunix II: A Design for Large-Scale Shared-Memory Multiprocessors,'' Ultracomputer Note #135, NYU, April 1988a.

Edler, J., J. Lipkis, and E. Schonberg, ``Process Management for Highly Parallel UNIX Systems,'' Ultracomputer Note #136, NYU, April 1988b.

Eisenberg, M. A. and M. R. McGuire, ``Further Comments on Dijkstra's Concurrent Programming Control Problem,'' *CACM*, **15**, p. 999, May 1972.

Ellis, C. S. and T. J. Olson, ``Parallel First Fit Memory Allocation,'' *Intl. Conf. on Parallel Processing*, pp. 502-511, 1987.

Enslow, P. H., Jr., ``Multiprocessor Organization–A Survey,'' *Computing Surveys*, **9**, pp. 103-129, March 1977.

Fang, Z., P.-C. Yew, P. Tang, and C.-Q. Zhu, ``Dynamic Processor Self-Scheduling for General Purpose Parallel Nested Loops,'' *Intl. Conf. on Parallel Processing*, pp. 1-10, 1987.

Feng, T., ``A Survey of Interconnection Networks,'' *Computer*, **14**, pp. 12-27, December 1981.

Finkel, R. A., An Operating Systems Vade Mecum, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

Fisher, J. A, ``The VLIW Machine: A Multiprocessor for Compiling Scientific Code,'' *Computer*, **17**, pp. 45-53, July 1984.

Floyd, R. W., ``Assigning Meanings to Programs,'' *Proc. Symp. in Applied Mathematics*, **19**, pp. 19-32, 1967.

Flynn, M. J., ``Very High-Speed Computing Systems,'' *Proc. IEEE*, **54**, pp. 1901-1909, December 1966.

Flynn, M. J., ``Some Computer Organizations and Their Effectiveness,'' *IEEE Trans. on Computers*, **C-21**, pp. 948-960, September 1972.

Fortune, S. and J. Wyllie, ``Parallelism in Random Access Machines,'' *Proc. 10th Ann. ACM Symp. on Theory of Computing*, pp. 114-118, 1978.

Freudenthal, E., private communication, 1987.

Gajski, D. and J. Peir, ``Comparison of Five Multiprocessor Systems,'' *Parallel Computing*, **2**, pp. 265-282, 1985.

Goke, L. R. and G. J. Lipovski, ``Banyan Neworks for Partitioning Multiprocessor Sytems,'' *Proc. 1st Ann. Computer Arch. Conf.*, pp. 21-28, December 1973.

Gosden, J. A., ``Explicit Parallel Processing Description and Control in Programs for Multi and Uniprocessor Computers,'' *Proc. AFIPS 1966 Fall Joint Computer Conf.*, Spartan Books, New York, pp. 651-660, 1966.

Gottlieb, A., ``An Overview of the NYU Ultracomputer Project'' in Experimental Parallel Computing Architectures, J. J. Dongarra (ed.), North-Holland, Amsterdam, NY, pp. 25-95, 1987.

Gottlieb, A. and C. P. Kruskal, ``Coordinating Parallel Processors: A Partial Unification,'' *Computer Architecture News*, pp. 16-24, October 1981.

Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, ''The NYU Ultracomputer — Designing an MIMD Shared-Memory Parallel Computer,'' *IEEE Trans. on Computers*, **C-32**, pp. 175-189, February 1983a. Also *9th Ann. Intl. Conf. on Computer Architecture*, pp. 27-42, 1982.

Gottlieb, A., B. Lubachevsky, and L. Rudolph, ''Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors,'' *ACM TOPLAS*, **5**, pp. 164-189, April 1983b. Also, Ultracomputer Note #16, Courant Institute, NYU, 1980.

Greenstadt, J. L., ''The IBM 709 Computer,'' *New Computers, Rept. from the Manufacturers ACM Conf.*, pp. 92-98, 1957.

Gregory, J. and R. McReynolds, ''The Solomon Computer,'' *IEEE Trans. on Electronic Computers*, **EC-12**, pp. 774-781, 1963.

Gurd, J. R., C. C. Kirkham, and I. Watson, ''The Manchester Prototype Dataflow Computer,'' *CACM*, **28**, pp. 34-52, January 1985.

Harrison, M. C., ''The Add-and-Lambda Operation: An Extension of F&A,'' Ultracomputer Note #104, Courant Institute, NYU, 1986.

Haynes, L. S., R. L. Lau, D. P. Siewiorek, and D. W Mizell, ''A Survey of Highly Parallel Computing,'' *Computer*, **15**, pp. 9-24, January 1982.

Hehner, E. C. R. and R. K. Shyamasundar, ''An Implementation of P and V,'' *Inf. Proc. Lett.*, **12**, pp. 196-198, August 1981.

Hillis, W. D., The Connection Machine, MIT Press, Cambridge, MA, 1985.

Hoare, C. A. R., ''An Axiomatic Basis for Computer Programming,'' *CACM*, **12**, pp. 576-583, October 1969.

Hoare, C. A. R., ''Towards a Theory of Parallel Programming'' in Operating Systems Techniques, C. A. R. Hoare and R. H. Perrot (eds.), Academic Press, NY, pp. 61-71, 1972.

Hoare, C. A. R., ''Monitors: An Operating System Structuring Concept,'' *CACM*, **17**, pp. 549-557, October 1974.

Hockney, R. W., ''MIMD Computing in the USA–1984,'' *Parallel Computing*, **2**, pp. 119-136, 1985.

Hockney, R. W. and C. R. Jesshope, Parallel Computers, Adam Hilger Ltd., Bristol, 1981.

Holland, J., ''A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously,'' *Proc. Eastern Joint Computer Conf. #16*, pp. 108-113, 1959.

Hwang, K. (ed.), Supercomputers: Design and Applications, IEEE Computer Society Press, Silver Springs, MD, August 1984.

Hwang, K. and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, NY, 1984.

IBM Corp., *IBM 7094 Principles of Operation*, A22-6703, 1959.

Jones, A. K., ''The Object Model: A Conceptual Tool for Structuring Software,'' *Operating Systems: An Advanced Course*, Springer-Verlag, pp. 7-16, 1978a.

Jones, A. K., ''Programming Issues Raised by a Multiprocessor,'' *Proc. IEEE*, **66**, pp. 229-237, February 1978b.

Jones, A. K., R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl, ‘‘ StarOS, a Multiprocessor Operating System for the Support Task Forces,’’ *Proc. 7th Symp. on Operating System Principles*, pp. 117-127, 1979.

Jones, A. K. and P. Schwarz, ‘‘Experience Using Multiprocessor Systems,’’ *Computing Surveys* **12**, pp. 121-165, June 1980.

Katseff, H., ‘‘A New Solution to the Critical Section Problem,’’ *Proc. of the Tenth Ann. Symp. on Theory of Computing*, pp. 86-88, 1978.

Keller, R. M., ‘‘Formal Verification of Parallel Programs,’’ *CACM*, **19**, pp. 371-384, July 1976.

Keller, R. M., F. C. H. Lin, and J. Tanaka, ‘‘Rediflow Multiprocessing,’’ *Digest of Papers: COMPCON Spring 84*, pp. 410-417, 1984.

Klappholz, D. and H. Park, ‘‘Parallelized Process Scheduling for a Tightly-Coupled MIMD Machine,’’ *Intl. Conf. on Parallel Processing*, pp. 315-321, 1984.

Knowlton, K., ‘‘A Fast Storage Allocator,’’ *CACM*, **8**, pp. 623-625, October 1965.

Knuth, D. E., ‘‘Additional Comments on a Problem in Concurrent Programming Control,’’ *CACM*, **9**, pp. 321-322, May 1966.

Knuth, D. E., The Art of Computer Programming, **1**, Addison-Wesley, Reading, MA, 1968.

Kozdrowicki, E. and D. Theis, ‘‘Second Generation of Vector Supercomputers,’’ *Computer*, **13**, pp. 71-83, November 1980.

Kruskal, C. P., L. Rudolph, and R. Cytron, ‘‘The Architecture of Parallel Computers’’ in Control Flow and Data Flow: Concepts of Distributed Programming, M. Broy (ed.), Springer-Verlag, pp. 279-344, 1985.

Kruskal, C. P. and M. Snir, ‘‘The Performance of Multistage Interconnection Networks for Multiprocessors,’’ *IEEE Trans. on Computers*, **C-32**, pp. 1091-1098, December 1983.

Kruskal, C. P. and A. Weiss, ‘‘Allocating Independent Subtasks on Parallel Processors,’’ *IEEE Trans. on Software Engineering*, **SE-11**, pp. 1001-1016, October 1985.

Kuck, D. J., ‘‘High Speed Machines and Their Compilers’’ in Parallel Processing Systems, D. J. Evans (ed.), Cambridge University Press, NY, pp. 193-214, 1982.

Kuck, D. J., E. S. Davidson, D. H. Lawrie, and A. H. Sameh, ‘‘Parallel Supercomputing Today and the Cedar Approach’’ in Experimental Parallel Computing Architectures, J. J. Dongarra (ed.), North-Holland, Amsterdam, NY, pp. 1-24, 1987.

Kuck, D. J. and R. Stokes, ‘‘The Burroughs Scientific Processor (BSP),’’ *IEEE Trans. on Computers*, **C-31**, pp. 363-376, May 1982.

Kuck, D. J., A. Sameh, R. Cytron, A. Viedenbaum, C. Polychronopoulos, G. Lee, T. McDaniel, B. Leasure, C. Beekman, J. Davies, and C. P. Kruskal, ‘‘The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance,’’ *Intl. Conf. on Parallel Processing*, pp. 129-138, 1984.

Kumar, M. and G. F. Pfister, ‘‘The Onset of Hot Spot Contention,’’ *Intl. Conf. on Parallel Processing*, pp. 28-34, 1986.

Kung, H. T., ‘‘Why Systolic Architectures,’’ *Computer*, **15**, pp. 37-46, January 1982.

Lamport, L., ‘‘A New Solution of Dijkstra’s Concurrent Programming Problem,’’ *CACM*, **17**, pp. 453-455, August 1974.

Lamport, L., ‘‘Proving the Correctness of Multiprocess Programs,’’ *IEEE Trans. on Software Engineering*, **SE-3**, pp. 125-143, March 1977.

Lamport, L., ‘‘A New Approach to Proving the Correctness of Multiprocess Programs,’’ *ACM TOPLAS*, **1**, pp. 84-97, July 1979.

Lamport, L., ‘‘The ‘Hoare Logic’ of Concurrent Programs,’’ *Acta Informatica*, **14**, pp. 21-37, 1980.

Lamport, L., R. Shostak, and M. Pease, ‘‘The Byzantine Generals Problems,’’ *ACM TOPLAS*, **4**, pp. 382-401, July 1982.

Lamport, L., ‘‘A Fast Mutual Exclusion Algorithm,’’ *ACM Trans. on Computer Systems*, **5**, pp. 1-11, February 1987.

Lanin, V., private communication, 1987.

Lawrie, D. H., ‘‘Access and Alignment of Data in an Array Processor,’’ *IEEE Trans. on Computers*, **C-24**, pp. 1145-1155, December 1975.

Lazou, C., Supercomputers and Their Use, Oxford: Clarendon Press, 1986.

Lee, G., C. P. Kruskal, and D. J. Kuck, ‘‘The Effectiveness of Combining in Shared-Memory Parallel Computers in the Presence of ‘Hot Spots’,’’ *Intl. Conf. on Parallel Processing*, pp. 35-41, 1986.

Levin, R., E. Cohen, W. Corwin, F. Pollack, and W. A. Wulf, ‘‘Policy/Mechanism Separation in Hydra,’’ *Proc. 5th Symp. on Operating Systems Principles*, pp. 132-140, 1975.

Levy, H. M., Capability-Based Computer Systems, Digital Press, 1984.

Lundstrom, S. F. and G. H. Barnes, ‘‘A Controllable MIMD Architecture,’’ *Intl. Conf. on Parallel Processing*, pp. 19-27, 1980.

MacKinnon, R. A., ‘‘Advanced Function Extended with Tightly-Coupled Multiprocessing,’’ *IBM Systems J.*, **13**, 1, pp. 32-59, 1974.

Madnick, S. E. and J. J. Donovan, Operating Systems, McGraw-Hill, NY, 1974.

Manna, Z. and A. Pneulli, ‘‘Verification of Concurrent Programs: The Temporal Framework’’ in The Correctness Problem in Computer Science, R. S. Boyers and J. Moore (eds.), Academic Press, 1981. pp. 35-41, 1981.

McAuliffe, K., ‘‘Analysis of Cache Memories in Highly Parallel Systems,’’ Ph.D. thesis, Courant Institute, New York University, 1986.

Ni, L. M. and C. E. Wu, ‘‘Design Trade-Offs for Process Scheduling in Tightly-Coupled Multiprocessor Systems,’’ *Intl. Conf. on Parallel Processing*, pp. 63-70, 1985.

Noyce, R. N., ‘‘Hardware Prospects and Limitations’’ in The Computer Age: A Twenty-Year Review, M. L. Dertouzos and J. Moses (eds.), MIT Press, Cambridge, MA, pp. 321-337, 1979.

Olson, R., ‘‘Parallel Processing in a Message-Based Operating System,’’ *IEEE Software*, **2**, pp.

39-49, July 1985.

Organick, E. I., Computer System Organization: The B5700/B6700 Series, Academic Press, NY, 1973.

Ousterhout, J. K., D. A. Scelza, and P. S. Sindhu, ''Medusa: An Experiment in Distributed Operating System Structures,'' *CACM*, **23**, pp. 92-105, February 1980.

Owicki, S., ''Axiomatic Proof Techniques for Parallel Programs,'' Ph.D. thesis, Cornell University, 1975.

Owicki, S. and D. Gries, ''An Axiomatic Proof Technique for Parallel Programs I,'' *Acta Informatica*, **6**, pp. 319-340, 1976.

Owicki, S. and L. Lamport, ''Proving Liveness Properties of Concurrent Programs,'' *ACM TOPLAS*, **4**, pp. 455-495, July 1982.

Parkinson, D. and H. M. Liddell, ''The Measurement of Performance on a Highly Parallel System,'' *IEEE Trans. on Computers*, **C-32**, pp. 32-37, January 1983.

Patel, J. H., ''Processor-Memory Interconnections for Multiprocessors,'' *Proc. Sixth Ann. Symp. on Computer Arch.*, pp. 168-177, April 1979.

Paul, G. (ed.), ''Special Issue on Supercomputers–Their Impact on Science and Technology,'' *Proc. IEEE*, **72**, pp. 1-144, January 1984.

Pease, M., R. Shostak, and L. Lamport, ''Reaching Agreement in the Presence of Faults,'' *JACM*, **27**, pp. 228-234, April 1980.

Perron, R. and C. Mundie, ''The Architecture of the Alliant FX/8 Computer,'' *Digest of Papers: COMPCON Spring 86*, pp. 390-393, 1986.

Peterson, G. L., ''Concurrency and Complexity,'' Tech. Report TR59, Computer Science Dept., Univ. of Rochester, NY, August 1979.

Peterson, G. L., ''Myths About the Mutual Exclusion Problem,'' *Inf. Proc. Lett.*, **12**, pp. 115-116, June 1981.

Peterson, G. L., ''A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables,'' *ACM TOPLAS*, **5**, pp. 56-65, January 1983.

Peterson, G. L. and M. Fisher, ''Economical Solutions for the Critical Section Problem in a Distributed System,'' *Proc. of the Ninth Ann. Symp. on Theory of Computing*, pp. 91-97, 1977.

Peterson, V. L., ''Impact of Computers on Aerodynamics Research and Development,'', *Proc. IEEE*, **72**, pp. 68-79, January 1984.

Pfister, G. F., W. C. Brantley, D. A, George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ''The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,'' *Intl. Conf. on Parallel Processing*, pp. 764-771, 1985.

Pfister, G. F. and V. A. Norton, '''Hot Spot' Contention and Combining in Multistage Interconnection Networks,'' *Intl. Conf. on Parallel Processing*, pp. 790-797, 1985.

Pneulli, A., ''The Temporal Logic of Programs,'' *18th Ann. Symp. on Foundations of Computer Science*, pp. 46-57, 1977.

Polychronopoulos, C. D. and D. J. Kuck, ‘‘Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,’’ *IEEE Trans. on Computers*, **C-36**, pp. 1425-1439, December 1987.

Polychronopoulos, C. D., D. J. Kuck, and D. A. Padua, ‘‘Execution of Parallel Loops on Parallel Processor Systems,’’ *Intl. Conf. on Parallel Processing*, pp. 519-527, 1986.

Ramamoorthy, C. V. and H. F. Li, ‘‘Pipeline Architecture,’’ *Computing Surveys*, **9**, pp. 61-102, March 1977.

Randell, B., ‘‘A Note on Storage Fragmentation and Program Segmentation,’’ *CACM*, **12**, pp. 365-372, July 1969.

Rashid, R. F., ‘‘Threads of a New System,’’ *Unix Review*, **4**, pp. 37-49, August 1986.

Rashid, R. F. and G. Robertson, ‘‘Accent: A Communication Oriented Network Operating System Kernel,’’ *Proc. 8th Symp. Operating Systems Principles*, pp. 164-185, 1981.

Rashid, R. F., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, ‘‘Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures,’’ *Proc. 2nd Intl. Conf. Arch. Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 31-39, October 1987.

Reed, D. P. and R. K. Kanodia, ‘‘Synchronization with Eventcounts and Sequencers,’’ *CACM*, **22**, pp. 115-123, February 1979.

Ritchie, D. M. and K. Thompson, ‘‘The Unix Time-Sharing System,’’ *CACM*, **17**, pp. 365-375, July 1974.

Riganati, J.P. and P. B. Schneck, ‘‘Supercomputing,’’ *Computer*, **17**, pp. 97-113, October 1984.

Rivest, R. L. and V. R. Pratt, ‘‘The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report,’’ *17th Ann. Symp. on Foundations of Computer Science*, pp. 1-8, 1976.

Ross, R. T., ‘‘The AED Free Storage Package,’’ *CACM*, **10**, pp. 481-492, August 1967.

Rudolph, L., ‘‘Software Structures for Ultraparallel Computing,’’ Ph.D. thesis, Courant Institute, New York University, February 1982.

Russel, R., ‘‘The Cray-1 Computer System,’’ *CACM*, **21**, pp. 63-72, January 1978.

Saltzer, J. H., ‘‘Traffic Control in a Multiplexed Computer System,’’ Sc.D. thesis, Electrical Engineering, MIT, Cambridge, MA, 1966.

Sanguinetti, J., ‘‘Performance of a Message-Based Multiprocessor,’’ *Computer*, **19**, pp. 47-55, September 1986.

Satyanarayanan, M., Multi-Processors: A Comparative Study, Prentice-Hall, NJ, 1980.

Schwartz, J., ‘‘Ultracomputers’’, *ACM TOPLAS*, **2**, pp. 484-521, October 1980.

Schwartz, J., ‘‘A Taxonomic Table of Parallel Computers, Based on 55 Designs,’’ Ultracomputer Note #69, Courant Institute, NYU, 1983.

Seitz, C. L., ‘‘The Cosmic Cube,’’ *CACM*, **28**, pp. 22-33, January 1985.

Sejnowski, M. C., E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, ‘‘An Overview of the Texas Reconfigurable Array Computer,’’ *Proceedings of AFIPS NCC Conf.*, pp.

631-641, 1980.

Sequent Computer Systems, *Balance Technical Summary*, MAN-0110-00, 1986.

Shaw, A. C., The Logical Design of Operating Systems, Prentice-Hall, Englewood Cliffs, NJ, 1974.

Shaw, D. E., ‘‘Architecture and applications of a Heterogeneous, Massively Parallel Machine’’ in Annual Review of Computer Science, J. F. Traub (ed.), Annual Reviews Inc., Palo Alto, CA, pp. 139-151, 1986.

Shore, J. E., ‘‘On the External Fragmentation Produced by First Fit and Best Fit Allocation Policies,’’ *CACM*, **18**, pp. 433-440, August 1975.

Siegel, H. J., Interconnection Networks for Large Scale Parallel Processing, Lexington Books, Lexington, MA, 1985.

Siegel, H. J., T. Schwederski, N. J. Davis IV, and J. T. Kuehn, ‘‘PASM: A Reconfigurable Parallel System For Image Processing’’ in Parallel Computing: Theory and Comparisons, G. J. Lipovski and M. Malek (eds.), John Wiley and Sons, NY, pp. 215-238, 1987.

Smith, B., ‘‘Architecture and Applications of the HEP Multiprocessor Computer System,’’ *Real Time Processing IV, Proc. SPIE*, pp. 241-248, 1981.

Smith, B., ‘‘The Architecture of HEP’’ in Parallel MIMD Computation: HEP Super- computer and Its Applications, J. S. Kowalik (ed.), MIT Press, Cambridge, MA, pp. 41-58, 1985.

Snir, M., ‘‘On Parallel Searching,’’ *Symp. on Principles of Distributed Computing*, pp. 242-253, 1982.

Snyder, L., ‘‘Introduction to the Configurable, Highly Parallel Computer,’’ *Computer*, **15**, pp. 47-56, January 1982.

Spector, A. Z., ‘‘Performing Remote Operations Efficiently on a Local Computer Network,’’ *CACM*, **25**, pp. 246-260, April 1982.

Stark, E. W., ‘‘Semaphore Primitives and Starvation-Free Mutual Exclusion,’’ *JACM*, **29**, pp. 1049-1072, October 1982.

Stolfo, S. J. and D. P. Miranker, ‘‘DADO: A Tree-Structured Architecture for Artificial Intelligence Computation’’ in Annual Review of Computer Science, J. F. Traub (ed.), Annual Reviews Inc., Palo Alto, CA, pp. 1-18, 1986.

Stone, H. S., ‘‘Parallel Processing With the Perfect Shuffle,’’ *IEEE Trans. on Computers*, **C-20**, pp. 153-161, February 1971.

Stone, H. S., ‘‘Parallel Computers’’ in Introduction to Computer Architecture, H. S. Stone (ed.), SRA, Chicago, IL, pp. 318-374, 1980.

Stone, H. S., ‘‘Parallel Memory Allocation using the Fetch-and-add Instruction,’’ Tech. Report RC9674, IBM Thomas J. Watson Research Center, November 1982.

Stone, H. S., High-Performance Computer Architecture, Addison-Wesley, Reading, MA, 1987.

Sullivan, H., T. R. Bashkow, and D. Klappholz, ‘‘The Node Kernel: Resource Management in a Self-Organizing Parallel Processor,’’ *Intl. Conf. on Parallel Processing*, pp. 157-162, 1977.

Swan, R. J., S. H. Fuller, and D. P. Siewiorek, ‘‘Cm^*–A Modular, Multi-microprocessor,’’ *Proc.*

AFIPS Conf., **46**, pp. 637-644, 1977.

Tanenbaum, A. S., S. J. Mullender, and R. Van Renesse, ‘‘Using Sparse Capabilities in a Distributed Operating System,’’ *Proc. 6th Intl. Conf. Distr. Computing Systems*, pp. 558-563, 1986.

Tanenbaum, A. S. and R. Van Renesse, ‘‘Distributed Operating Systems,’’ *Computing Surveys*, **17**, pp. 419-470, December 1985.

Tang, P. and P. Yew, ‘‘Processor Self-Scheduling for Multiple-Nested Parallel Loops,’’ *Intl. Conf. on Parallel Processing*, pp. 528-535, 1986.

Teller, P. J., ‘‘The Feasibility of Demand Paging in an MIMD Shared-Memory Parallel Processor,’’ Ph.D. thesis, Courant Institute, New York University, 1988, in preparation.

Theimer, M. T., K. A. Lentz, and D. R. Cheriton, ‘‘Preemptible Remote Execution Facilities for the V-System,’’ *Proc. 10th Symp. on Operating Systems Principles*, pp. 2-14, 1985.

Thornton, J. E., ‘‘Parallel Operations in the Control Data 6600,’’ *Proc. AFIPS Conf.*, **26(II)**, pp.33-40, 1964.

Vantiborgh, H. and A. van Lamsweerde, ‘‘On an Extension of Dijkstra’s Semaphore Primitives,’’ *Inf. Proc. Lett.*, **1**, pp. 181-186, October 1972.

Veen, A. H., ‘‘Dataflow Machine Architecture,’’ *Computing Surveys*, **18**, pp. 365-396, December 1986.

Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, ‘‘The LOCUS Distributed Operating System’’, *Proc. 9th Symp. on Operating Systems Principles*, pp. 49-70, 1983.

Weiser, M., S. Kogge, M. McElvany, R. Pierson, R. Post, and A. Thareja, ‘‘Status and Performance of the ZMOB Parallel Processing System,’’ *Digest of Papers: COMPCON Spring 85*, pp. 71-74, 1985.

Widdoes, L. C., Jr. and S. Correll, ‘‘The S-1 Project: Developing High-Performance Digital Computers,’’ *Digest of Papers: COMPCON Spring 80*, pp. 282-291, 1980.

Wijngaarden, A. van, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker (eds.), Revised Report on the Algorithmic Language Algol 68, Mathematisch Centrum, Amsterdam, 1976.

Wirth, N., ‘‘The Programming Language Pascal,’’ *Acta Informatica*, **1**, 1, pp. 35-63, 1971.

Wulf, W. A. and C. G. Bell, ‘‘C.mmp: A Multi-Mini-Processor,’’ *Proc. AFIPS Conf.*, **41(II)**, pp. 765-777, 1972.

Wulf, W. A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, ‘‘Hydra: The Kernel of a Multiprocessing Operating System,’’ *CACM*, **17**, pp. 337-345, June 1974.

Wulf, W. A. and S. P. Harbison, ‘‘Reflections in a Pool of Processors–An Experience Report on C.mmp/Hydra,’’ *Proc. AFIPS Conf.*, **47**, pp. 939-951, 1978.

Wulf, W. A., R. Levin, and C. Pierson, ‘‘Overview of the Hydra Operating System Development,’’ *Proc. 5th Symp. on Operating Systems Principles*, pp. 121-131, 1975.

Yew, P.-C., N.-F. Tzeng, and D. H. Lawrie, ‘‘Distributing Hot Spot Addressing in Large-Scale Multiprocessors,’’ *Intl. Conf. on Parallel Processing*, pp. 51-58, 1986.

Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, ‘‘The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System,’’ *Proc. 11th Symp. Operating Systems Principles*, pp. 63-76, 1987.

Zhu, C.-Q. and P.-C. Yew, ‘‘A Synchronization Scheme and Its Applications for Large Multiprocessor Systems,’’ *Proc. of 4th Intl. Conf. on Distributed Computing Systems*, pp. 486-493, 1985.

Operating System Data Structures for
Shared-Memory MIMD Machines with Fetch-and-Add

James M. Wilson

June 1988

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved: _______________

Allan Gottlieb

ABSTRACT

Title: Operating System Data Structures for Shared-Memory MIMD Machines
with Fetch-and-Add

Author: James M. Wilson

Advisor: Allan Gottlieb

Ideally, procedures and data structures on a shared-memory MIMD machine should be serialization-free and concurrently accessible to avoid (potential) performance-limiting bottlenecks. The fetch-and-add coordination primitive, in conjunction with ‘‘combining’’ interconnection networks, has been proposed as a means for achieving this goal. The first is essentially an indivisible add-to-memory and the second combines simultaneous requests to the same memory location.

In this thesis we address serialization-free memory and process management for a shared-memory MIMD machine with fetch-and-add and a combining network. To meet this goal we adopt a self-service paradigm for the operating system that permits each processing element (PE) to service its own requests (thereby avoiding central server bottlenecks). The success of this approach depends upon the use of concurrently accessible data structures to hold data shared among the PEs.

We begin by reviewing existing fetch-and-add based queue and multiqueue (a compressed queue) implementations that support concurrent queue insertion and deletion. We then extend these implementations to include new operations (e.g., the removal of an interior queue item) and new data structure representations (e.g., linked lists).

Parallel memory allocation algorithms, many based on the modified queue and multiqueue data structures, are then given. These algorithms include parallel analogs to a number of existing

serial algorithms such as Knuth's boundary tag method and the binary buddy system.

Next, we define a set of primitives that permit various task activities, such as creation and scheduling, to be done in parallel. Task-switching readers/writers and event primitives are given as well. In the readers/writers implementations, reader activity is fully parallel in the absence of writers. An important feature of both the readers/writers and event implementations is that tasks waiting for a resource can be resumed in parallel by multiple PEs.

We then demonstrate how high-level parallel programming constructs (e.g., parallel loops) may be implemented via the task primitives and the queue and multiqueue data structures.

Finally, we prove that one of the readers/writers implementations satisfies certain correctness criteria including freedom from deadlock and the mutual exclusion of readers and writers.

ACKNOWLEDGEMENTS

I thank my advisor Allan Gottlieb for his guidance and willingness to develop my research skills. I thank my fellow graduate student and friend Patricia Teller for our many discussions on the trials of the dissertation process, which served to remind me that my hopes and fears were not unique. Finally, I thank my parents Gertrude and John and my wife Caridad for their support, understanding, and love throughout these many years.

Table of Contents