Chapter 5
Concurrency: Mutual Exclusion and Synchronization

Seventh Edition
By William Stallings
“Designing correct routines for controlling concurrent activities proved to be one of the most difficult aspects of systems programming. The ad hoc techniques used by programmers of early multiprogramming and real-time systems were always vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred. The errors are particularly difficult to locate, since the precise conditions under which they appear are very hard to reproduce.”

—THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY, MIT Press, 1980
Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
Concurrency Arises in Three Different Contexts:

- **Multiple Applications**: invented to allow processing time to be shared among active applications
- **Structured Applications**: extension of modular design and structured programming
- **Operating System Structure**: OS themselves implemented as a set of processes or threads
### Key Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic operation</td>
<td>A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.</td>
</tr>
<tr>
<td>critical section</td>
<td>A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.</td>
</tr>
<tr>
<td>deadlock</td>
<td>A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.</td>
</tr>
<tr>
<td>livelock</td>
<td>A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.</td>
</tr>
<tr>
<td>race condition</td>
<td>A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.</td>
</tr>
<tr>
<td>starvation</td>
<td>A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.</td>
</tr>
</tbody>
</table>
Interleaving and overlapping
- can be viewed as examples of concurrent processing
- both present the same problems

Uniprocessor – the relative speed of execution of processes cannot be predicted
- depends on activities of other processes
- the way the OS handles interrupts
- scheduling policies of the OS
Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible
Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - the “loser” of the race is the process that updates last and will determine the final value of the variable
Design and management issues raised by the existence of concurrency:

- The OS must:
  - be able to keep track of various processes
  - allocate and de-allocate resources for each active process
  - protect the data and physical resources of each process against interference by other processes
  - ensure that the processes and outputs are independent of the processing speed
## Process Interaction

<table>
<thead>
<tr>
<th>Degree of Awareness</th>
<th>Relationship</th>
<th>Influence that One Process Has on the Other</th>
<th>Potential Control Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes unaware of each other</td>
<td>Competition</td>
<td>• Results of one process independent of the action of others</td>
<td>• Mutual exclusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Timing of process may be affected</td>
<td>• Deadlock (renewable resource)</td>
</tr>
<tr>
<td>Processes indirectly aware of each other (e.g., shared object)</td>
<td>Cooperation by sharing</td>
<td>• Results of one process may depend on information obtained from others</td>
<td>• Starvation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Timing of process may be affected</td>
<td></td>
</tr>
<tr>
<td>Processes directly aware of each other (have communication primitives available to them)</td>
<td>Cooperation by communication</td>
<td>• Results of one process may depend on information obtained from others</td>
<td>• Deadlock (consumable resource)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Timing of process may be affected</td>
<td>• Starvation</td>
</tr>
</tbody>
</table>
Concurrent processes come into conflict when they are competing for use of the same resource:

- for example: I/O devices, memory, processor time, clock

In the case of competing processes, three control problems must be faced:

- the need for mutual exclusion
- deadlock
- starvation
**Mutual Exclusion**

```c
/* PROCESS 1 */
void P1
{
    while (true) {
/* preceding code */;
    entercritical (Ra);
/* critical section */;
    exitcritical (Ra);
/* following code */;
    }
}

/* PROCESS 2 */
void P2
{
    while (true) {
/* preceding code */;
    entercritical (Ra);
/* critical section */;
    exitcritical (Ra);
/* following code */;
    }
}

/* PROCESS n */
void Pn
{
    while (true) {
/* preceding code */;
    entercritical (Ra);
/* critical section */;
    exitcritical (Ra);
/* following code */;
    }
}
```

Figure 5.1 Illustration of Mutual Exclusion
Requirements for Mutual Exclusion

- Must be enforced
- A process that halts must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only
Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture
Special Machine Instructions
- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
  - a compare is made between a memory value and a test value
  - if the values are the same a swap occurs
  - carried out atomically
Figure 5.2 Hardware Support for Mutual Exclusion

```c
/* program mutual_exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap instruction
Figure 5.2  Hardware Support for Mutual Exclusion

```c
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}

(b) Exchange instruction
Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- Simple and easy to verify.
- It can be used to support multiple critical sections; each critical section can be defined by its own variable.
Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time.

Starvation is possible when a process leaves a critical section and more than one process is waiting.

Deadlock is possible.
### Common Concurrency Mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore</td>
<td>An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a <strong>counting semaphore</strong> or a <strong>general semaphore</strong>.</td>
</tr>
<tr>
<td>Binary Semaphore</td>
<td>A semaphore that takes on only the values 0 and 1.</td>
</tr>
<tr>
<td>Mutex</td>
<td>Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).</td>
</tr>
<tr>
<td>Condition Variable</td>
<td>A data type that is used to block a process or thread until a particular condition is true.</td>
</tr>
<tr>
<td>Monitor</td>
<td>A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively executing the monitor at any one time. The access procedures are <strong>critical sections</strong>. A monitor may have a queue of processes that are waiting to access it.</td>
</tr>
<tr>
<td>Event Flags</td>
<td>A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).</td>
</tr>
<tr>
<td>Mailboxes/Messages</td>
<td>A means for two processes to exchange information and that may be used for synchronization.</td>
</tr>
<tr>
<td>Spinlocks</td>
<td>Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.</td>
</tr>
</tbody>
</table>
Semaphore

A variable that has an integer value upon which only three operations are defined:

- May be initialized to a nonnegative integer value
- The semWait operation decrements the value
- The semSignal operation increments the value

There is no way to inspect or manipulate semaphores other than these three operations
Consequences

- There is no way to know before a process decrements a semaphore whether it will block or not.
- There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently.
- You don't know whether another process is waiting so the number of unblocked processes may be zero or one.
Semaphore Primitives

```c
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
A queue is used to hold processes waiting on the semaphore

**Strong Semaphores**

- the process that has been blocked the longest is released from the queue first (FIFO)

**Weak Semaphores**

- the order in which processes are removed from the queue is not specified
Example of Semaphore Mechanism

![Diagram of Semaphore Mechanism](image)

Figure 5.5 Example of Semaphore Mechanism
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */
        semSignal(s);
        /* remainder */
    }
}
void main()
{
    parbegin (P(1), P(2), ..., P(n));
}

Figure 5.6 Mutual Exclusion Using Semaphores
Shared Data Protected by a Semaphore

Figure 5.7  Processes Accessing Shared Data Protected by a Semaphore

Note that normal execution can proceed in parallel but that critical regions are serialized.
Producer/Consumer Problem

General Situation:
- one or more producers are generating data and placing these in a buffer
- a single consumer is taking items out of the buffer one at a time
- only one producer or consumer may access the buffer at any one time

The Problem:
- ensure that the producer can’t add data into full buffer and consumer can’t remove data from an empty buffer
Buffer Structure

Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem
/ * program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
<th>s</th>
<th>n</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>n++</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>if (n==1)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(semSignalB(delay))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>n--</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>n++</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>if (n==1)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(semSignalB(delay))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>if (n==0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(semWaitB(delay))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>n--</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>if (n==0)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(semWaitB(delay))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>semWaitB(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>n--</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>semSignalB(s)</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

*NOTE: White areas represent the critical section controlled by semaphore s*
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```c
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 5.11  A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores
Finite Circular Buffer

Figure 5.12  Finite Circular Buffer for the Producer/Consumer Problem
```c
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores
Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives.

- Can be implemented in hardware or firmware.
- Software schemes such as Dekker’s or Peterson’s algorithms can be used.
- Use one of the hardware-supported schemes for mutual exclusion.
Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data
Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure.

Only one process may be executing in the monitor at a time.

Process enters monitor by invoking one of its procedures.
Synchronization

Achieved by the use of condition variables that are contained within the monitor and accessible only within the monitor.

- Condition variables are operated on by two functions:
  - cwait(c): suspend execution of the calling process on condition c
  - csignal(c): resume execution of some process blocked after a cwait on the same condition
Figure 5.15  Structure of a Monitor
Problem Solution Using a Monitor

```c
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N]; /* space for N items */
int nextin, nextout; /* buffer pointers */
int count; /* number of items in buffer */
cond notfull, notempty; /* condition variables for synchronization */

void append (char x)
{   /* monitor body */
    if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty); /* resume any waiting consumer */
}

void take (char x)
{   /* monitor body */
    if (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* one fewer item in buffer */
    csignal(notfull); /* resume any waiting producer */
}

void producer()
{   char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{   char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void main()
{   parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor
When processes interact with one another two fundamental requirements must be satisfied:

- **synchronization**
  - to enforce mutual exclusion

- **communication**
  - to exchange information

Message Passing is one approach to providing both of these functions:

- works with distributed systems *and* shared memory multiprocessor and uniprocessor systems
The actual function is normally provided in the form of a pair of primitives:

send (destination, message)
receive (source, message)

A process sends information in the form of a message to another process designated by a destination

A process receives information by executing the receive primitive, indicating the source and the message
Table 5.5  Design Characteristics of Message Systems for Interprocess Communication and Synchronization

| Readers only in the system | wsem set  
| Writers only in the system | no queues  
| Both readers and writers with read first | wsem and rsem set  
| Both readers and writers with write first | writers queue on wsem  
|                                    | wsem set by reader  
|                                    | rsem set by writer  
|                                    | all writers queue on wsem  
|                                    | one reader queues on rsem  
|                                    | other readers queue on r
Synchronization

Communication of a message between two processes implies synchronization between the two.

The receiver cannot receive a message until it has been sent by another process.

When a receive primitive is executed in a process, there are two possibilities:

If there is no waiting message, the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive.

If a message has previously been sent, the message is received and execution continues.
Both sender and receiver are blocked until the message is delivered

Sometimes referred to as a *rendezvous*

Allows for tight synchronization between processes
Nonblocking Send

Nonblocking send, blocking receive

- sender continues on but receiver is blocked until the requested message arrives
- most useful combination
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait
Schemes for specifying processes in send and receive primitives fall into two categories:

- Direct addressing
- Indirect addressing
Send primitive includes a specific identifier of the destination process.

Receive primitive can be handled in one of two ways:

- require that the process explicitly designate a sending process
- effective for cooperating concurrent processes
- implicit addressing
- source parameter of the receive primitive possesses a value returned when the receive operation has been performed
Indirect Addressing

Messages are sent to a shared data structure consisting of queues that can temporarily hold messages.

Queues are referred to as *mailboxes*.

Allows for greater flexibility in the use of messages.

One process sends a message to the mailbox and the other process picks up the message from the mailbox.
Indirect Process

(a) One to one

(b) Many to one

(c) One to many

(d) Many to many
Figure 5.19  General Message Format
Mutual Exclusion

```c
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */
        send (box, msg);
        /* remainder */
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), ... , P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages
Message Passing Example

```c
const int
    capacity = /* buffering capacity */;
    null = /* empty message */;

int i;
void producer()
{
    message pmsg;
    while (true) {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}

void consumer()
{
    message cmsg;
    while (true) {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Figure 5.21  A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages
A data area is shared among many processes
- some processes only read the data area, (readers) and some only write to the data area (writers)

Conditions that must be satisfied:
1. any number of readers may simultaneously read the file
2. only one writer at a time may write to the file
3. if a writer is writing to the file, no reader may read it
Readers Have Priority

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority

```c
/* program readersandwriters */
int readcount;
s semaphore x = 1, wsem = 1;
void reader()
{
 while (true) {
   semWait (x);
   readcount++;
   if (readcount == 1) semWait (wsem);
   semSignal (x);
   READUNIT();
   semWait (x);
   readcount--;
   if (readcount == 0) semSignal (wsem);
   semSignal (x);
 }

 void writer()
{
  while (true) {
    semWait (wsem);
    WRITEUNIT();
    semSignal (wsem);
  }
}

 void main()
{
  readcount = 0;
  parbegin (reader, writer);
}
Solution: Writers Have Priority

```c
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}

void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority
# State of the Process Queues

<table>
<thead>
<tr>
<th>State of the System</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readers only in the system</td>
<td>- <code>wsem</code> set</td>
</tr>
<tr>
<td></td>
<td>- no queues</td>
</tr>
<tr>
<td>Writers only in the system</td>
<td>- <code>wsem</code> and <code>rsem</code> set</td>
</tr>
<tr>
<td></td>
<td>- writers queue on <code>wsem</code></td>
</tr>
<tr>
<td>Both readers and writers with read first</td>
<td>- <code>wsem</code> set by reader</td>
</tr>
<tr>
<td></td>
<td>- <code>rsem</code> set by writer</td>
</tr>
<tr>
<td></td>
<td>- all writers queue on <code>wsem</code></td>
</tr>
<tr>
<td></td>
<td>- one reader queues on <code>rsem</code></td>
</tr>
<tr>
<td></td>
<td>- other readers queue on <code>z</code></td>
</tr>
<tr>
<td>Both readers and writers with write first</td>
<td>- <code>wsem</code> set by writer</td>
</tr>
<tr>
<td></td>
<td>- <code>rsem</code> set by writer</td>
</tr>
<tr>
<td></td>
<td>- writers queue on <code>wsem</code></td>
</tr>
<tr>
<td></td>
<td>- one reader queues on <code>rsem</code></td>
</tr>
<tr>
<td></td>
<td>- other readers queue on <code>z</code></td>
</tr>
</tbody>
</table>
void reader(int i)
{
    message rmsg;
    while (true) {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[i], rmsg);
        READUNIT ();
        rmsg = i;
        send (finished, rmsg);
    }
}

void writer(int j)
{
    message rmsg;
    while (true) {
        rmsg = j;
        send (writerrequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}

void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writeRequest)) {
                receive (writeRequest, msg);
                writer_id = msg.id;
                count = count - 100;
            }
            else if (!empty (readRequest)) {
                receive (readRequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}

Figure 5.24 A Solution to the Readers/Writers Problem Using Message Passing
Summary

Messages

- Useful for the enforcement of mutual exclusion discipline

Operating system themes are:

- Multiprogramming, multiprocessing, distributed processing
- Fundamental to these themes is concurrency
  - issues of conflict resolution and cooperation arise

Mutual Exclusion

- Condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time
- One approach involves the use of special purpose machine instructions

Semaphores

- Used for signaling among processes and can be readily used to enforce a mutual exclusion discipline