Process Cooperation

• Why processes cooperate
  • Modularity
    • breaking up a system into several sub-systems
    • Example: an interrupt handler and the appropriate device driver which need to communicate
  • Convenience
    • users might want to have several processes share data
  • Speedup
    • a single program is run as several sub-programs sharing information
Communication Abstraction

- Developed to reason about communication
- Producers and Consumers
  - *Producer* produces a piece of information
  - *Consumer* uses it
- Typical of system “principles”
  - developed to deal with
    - general “phenomena”
    - ease of arguing correctness formally
A Solution for Bounded Buffers

N: integer size of the buffer
nextin = nextout = 1 initially; --start of buffer
buffer: array of size N
Producer:
  Repeat
    produce an item in tempin
    while (nextin +1) mod n = out do waitabit;
    buffer[nextin] := tempin;
    nextin := nextin + 1 mod n;

Consumer:
  Repeat
    while nextin = nextout do waitabit;
    tempout := buffer[nextout];
    nextout := (nextout + 1) mod n;
    consume the item in tempout
Bounded buffer (cont’d)

• Implementation given above
  • buffer: array of size N
  • buffer pointers nextin and nextout
  • empty buffer: nextin == nextout
  • capacity of buffer
    • N-1 items
• Question: how to store N items
Bounded buffer (cont’d)

Producer:
  Repeat
  --produce an item in tempin;
  while (nextin + 1) mod n = out do waitabit;
  buffer[nextin] := tempin;
  nextin := nextin + 1 mod n;

Consumer:
  Repeat
  while nextin = nextout do waitabit;
  tempout := buffer[nextout];
  nextout := (nextout + 1) mod n;
  --consume the item in tempout;
Using counters

Producer:
   Repeat
   --produce an item in tempin;
   while counter = n do waitabit;
   buffer[nextin] := tempin;
   nextin := nextin + 1 mod n;
   counter := counter +1;

Consumer:
   Repeat
   while counter = 0 do waitabit;
   tempout := buffer[nextout];
   nextout := (nextout + 1) mod n;
   counter := counter - 1;
   --consume the item in tempout;
What is wrong?

• Note the variable “counter” and the statements
  counter := counter +1;
  counter := counter -1;

• The producer and consumer can be executing asynchronously
  • due to multiprogramming, say
  • and can be interrupted while executing above code

• These are two independent code streams
  • that can, due to interrupts for example, become interleaved
Increment/decrement

• Each of increment and decrement are actually implemented as a series of machine instructions on the underlying hardware platform (processor)
  • PRODUCER INCREMENTS
    register1 := counter
    register1 := register1+1
    counter := register1
  • CONSUMER DECREMENTS
    register2 := counter
    register2 := register2-1
    counter := register2
An Interleaving

• Consider counter = 5; a producer is followed by a consumer
• Would expect counter = 5 at end
• However, with the interleaving
  • P1: register1 := counter
  • P2: register1 := register1+1
  • C1: register2 := counter
  • C2: register2 := register2-1
  • P3: counter := register1
  • C3: counter := register2
• counter has a value of 6 after P3 but a value of 4 after P2
The Problem

• The phenomenon is because
  • increment and decrement are not atomic or uninterruptable

• The code containing these operations
  • is sometimes referred to as creating a race condition
Atomic operations

- Two or more operations are executed \textit{atomically} if the result of their execution is equivalent to that of some serial order of execution.
- Operations which are always executed atomically are called \textit{atomic}
  - byte read; byte write;
  - word read; word write
The Solution

• At a high level
  • The producer and consumer processes
    • need to synchronize
      • so that they do not access shared variables at the same time
  • This is called mutual exclusion
    • the shared and critical variables
      • can be accessed one process at a time
  • Access must be serialized
    • even if the processes attempt concurrent access
      • as in the previous example
Critical-sections

• The general framework for achieving this independent of the particular context or need for synchronization is via critical-sections of code

• Critical sections are written as
  • entry-section
  • critical-section-code
  • exit-section
CS wrappers

- The *entry section* controls access to make sure no more than one process $P_i$ gets to access the critical section at any given time
  - acts as a *guard*
- The *exit section* does bookkeeping to make sure that other processes that are waiting know that $P_i$ has exited
- We will see two examples via *turn counters* and *flags* for realizing mutual exclusion on critical sections between a *pair* of processes
Using Turn Counters

- We use an integer variable “turn”
- For processes Pi and Pj, Pi executes:
  
  ```
  While (turn != i) waitabit;
  critical section;
  turn := j
  ```

  - and analogous code for Pj
- The while loop is the entry section
  - controls access to the critical section following it
  - process Pi waits till its turn occurs
- The single instruction
  ```
  turn := j
  ```

  - constitutes the exit section
Mutual Exclusion?

• Assume that load and store are atomic
• Convince yourselves that the above (very simple) scheme gives mutual exclusion
• A drawback
  • if Pj never wants to execute the critical section and is elsewhere, Pi cannot reenter;
  • access must alternate
  • Will return to this when discussing the progress condition
Mutual Exclusion via Flags

• The algorithm uses a boolean array Flag
  • Pi executes
    Repeat
      I: Flag[i] := true
      II: While Flag[j] waitabit;
      CRITICAL SECTION
      III: Flag[i] := false
           (Rest of code)
  • and analogous for Pj
Correctness of ME (sketch)

- Suppose *not*; then there exists a sequence such that both $P_i$ and $P_j$ are in the critical section
- When $P_i$ entered the critical section,
  - it found $\text{Flag}[j]$ to be false at time $t_1$, say, by executing instruction II
  - It had set $\text{Flag}[i]$ to true at time $t_0$, say, with $t_0 < t_1$
  - It executed instruction III at time $t_2$, say, with $t_2 > t_1$, resetting $\text{Flag}[i]$ to false
- If $P_j$ is in the critical section at the same time as $P_i$
  - it must have found $\text{Flag}[i]$ to be false at time $t_5$, say, with $t_5 < t_0$ (instruction II)
  - it must have set $\text{Flag}[j]$ to true at time $t_4$, say, with $t_4 < t_5$
  - so $P_i$ cannot find $\text{Flag}[j]$ false at time $t_1 > t_0 > t_5$
- A contradiction.
Is this good enough?

- No
  - Pi and Pj can be looping in their respective II instruction forever
    - Leads to deadlock, which will be discussed in detail later
  - some ME algorithms allow starvation
    - see, for example, ME using test-and-set (below)
Criteria for Correctness

• Three conditions
  • Mutual Exclusion
  • Progress
    • at least one process requesting entry to a critical section will be able to enter it if there is no other process in it
  • Bounded Waiting:
    • No process waits indefinitely to enter the critical section once it has requested entry
Turn Counter & Flags

- While providing mutual exclusion, neither of the previous two approaches guarantee the other conditions
- Turn counter
  - if one process terminates
  - violates Progress and Bounded Waiting condition
Petersen’s Algorithm

• combines the previous two ideas
• preserving all 3 conditions
  • the entry section
    • flag[i] := true
    • turn := j
    • while (flag[j] and (turn == j)) waitabit
  • the exit section
    • flag[i] := false
Proof of ME

• Suppose
  • i is in its critical section
  • j is wanting to enter
• This can happen only if either
  • i found flag[j] false, or
  • i found turn == i
• In the first case
  • j will set turn after i did, and find turn == i
• In the second case
  • j has already set turn = i
• In both cases
  • j will wait till flag[i] == false
Other Conditions

• To see that progress and bounded waiting are satisfied,
  • let Pi be in the critical section and Pj be waiting on instruction (2) above
  • if Pi exits and goes elsewhere, Pj will find flag[i] to be false
  • if not, Pi will attempt to reenter the critical section thereby rendering turn := j
  • in either case, Pj will find the condition for waiting in (2) to be false and will enter the critical section
• Note that entry to the critical section only depends on the processes executing the while condition
Other cases

• The proof for other cases is either/or
  • trivial
  • by similar argument
Synchronizing n Processes

• Some intuition:
  • The processes ask for a *ticket* from an *agent* and get an integer valued ticket
  • They then wait till all processes with smaller ticket values have finished going through the critical region
  • There can be ties in which case, PIDs are used to break them by letting the process with the smaller PID go first
  • Leads to a FCFS prioritizing strategy
  • The algorithm is akin to taking a ticket and waiting for a turn in a bakery and is called the *bakery algorithm*
Getting a ticket

• We will use the function max to get the next ticket
  • $1 + \text{max (other tickets)}$
• Breaking ties
  • Lexicographic ordering of pairs of integers
  • Given integers $a, b, c, d$, the pair $(a,b) < (c, d)$ if and only if either
    • $(a < c)$ or $(a = c$ and $b < d)$
    • or $c = 0$
The Bakery Algorithm

- We use two data structures, arrays of size n
  - choosing: a boolean array initialized to false
  - ticket: an array of integers initialized to zero
- Process Pi executes
  - get_ticket(i)
  - entry-section
  - critical section
  - exit-section
    - ticket[i] := 0
Getting a ticket

• Process Pi declares that it wants to choose a ticket by setting choosing[i] to be true
• It assigns ticket[i] a value that is one more than max of the tickets of all the processes
• Pi resets choosing[i] to false
Entry-section

• Pi checks and see of *any* Pj from the remaining n-1 processes are waiting for a ticket.
  • If yes, wait
    • This is because Pj might have requested a ticket concurrently and might get the same ticket value as Pi's; prepare for the worst case
  • If no proceed
• Check the remaining (n-1) processes for a Pj such that
  • ticket[j] is non-zero, and (ticket[j], j) < (ticket[i], i)
• Wait till this condition is false
Proof of ME

1 choosing[i] := true;
2 ticket[i] :=
   1 + max(for k in 1 to n)(ticket[k]);
3 choosing[i] := false;
4 l1: for k in 1 to n do
   if choosing[k] then goto l1;
5 l2: for k in 1 to n do
   if ticket[k] <> 0
      and (ticket[i],i) > (ticket[k],k)
   then goto l2;

• Consider
  • a process i in its critical section
  • a process j wanting to enter.
Proof

- There are 3 cases (process i executing step 2):
  - (a) Process i found choosing[j] true.
  - (b) Process i found choosing[j] false because process j had executed step 3.
  - (c) Process i found choosing[j] false because process j had not executed step 1.

- In cases (a) and (b) process i would have found ticket[j] non-zero, in which case the ticket-pair of process i must be less than that of j.

- In case (c), if process j accesses ticket[i] while process i is in its critical section, it will compute ticket[j] > ticket[i].

- In all cases process j will not get past step 5 before i leaves its critical section.
Progress & Bounded Waiting

• No processes fail (implicit assumption throughout)
• Ticketed processes will exit in FCFS order
• The set of processes is finite
  • implies that every process will get its turn in finite time
Hardware Support

• Primitives
  • atomic operations: hardware instructions; or software
• Criterion for choosing primitives:
  • universality, i.e., being able to build arbitrary functionality (e.g. mutual exclusion, etc.) from simpler units
  • minimising scope
    • don’t want to stop interrupts for whole critical sections
Classical Hardware Primitives

- Test&Set
- Swap
Semantics of Test&Set

• Given boolean variables X, Y, atomically
  • X := Y;  Y := true
• The following function is atomic
  • Function test_set(var target:boolean) boolean;
  • begin
  •   test_Set := target;
  •   target:= True;
  • end;
ME from Test&Set

- Lock : Boolean := false
- While test_set (lock) waitabit
- CRITICAL SECTION
  - lock := false
  - REMAINDER.....

- Mutual exclusion?
  - First process Pi entering critical section sets lock := true
  - Further processes that execute test_set don’t enter since test&set evaluates to true after (from atomicity) this
  - When Pi exits, lock is set to false, so the next process Pj to execute the instruction will find test&set = false and will enter the critical section
Unbounded Waiting?

- Possible since
  - depending on the timing of evaluating the test&set primitive, other processes can be the first to enter the critical section
- Does not guarantee fairness
The Swap Primitive

- Also atomic:
  - Exchange the values of given variables $x$ and $y$
- ME and Swapping
  - can emulate test&set by swap
    - function test_set(var v: boolean)
    - var t := true;
    - swap (v, t);
    - return t;
Bounded Waiting?

- Introduce a boolean array called waiting of size n and a single local boolean variable key; we are looking at process Pi
  - waiting[i] := true;  key := true;
- Process Pi is waiting till its key can open the lock which is initially false
  - while ( waiting[i] and key ) do  key := test&set (lock);
- The initialization in the previous step forces every process to execute the test&set at least once
  - waiting[i] := false
  - Execute CRITICAL SECTION
Exit section

- Find the next process $j$ that has $\text{waiting}[j] = 1$ by stepping through waiting
  - $\text{waiting}[j] := \text{false}$;
  - Process $P_j$ will immediately enter the critical section
- If no such process exists then
  - $\text{lock} := \text{false}$;
- This means the next process to come and wait will be the first process and can enter the critical section freely
Mutual Exclusion

- Every (interested) Pi executes that test&set at least once
- Pi enters the critical region provided:
  - key is false in which case there is no process in the critical region
- or
  - if it was waiting, because waiting[i] was reset to false by the unique process that was blocking it in the critical region
- Either of the above events can occur exactly once and hence mutual exclusion
Bounded Waiting

• Let us examine: Find the next process $j$ that has $\text{waiting}[j] = 1$ by stepping through waiting
  • $\text{waiting}[j] := \text{false}$;
  • Any concern about bounded waiting will mean that there must be a process $P_j$ waiting
  • This means that using the above approach for finding the "next" process $P_j$'s turn will arrive in no more than $(n-1)$ steps implying that there is a finite bound on the waiting

• Verify condition (2); easy
Solutions So Far

- Review
  - Mutual Exclusion
    - Bakery Algorithm
    - Hardware primitives

- Drawbacks
  - Does not solve more general synchronization problems
    - mutual exclusion only
  - Involves “busy-waiting”
    - waitabit is just a time-waster

- Looking for more general solutions
Synchronization Problems

- Classical problems
  - Sequencing
    - A must precede B
  - Readers/Writers
  - Dining Philosophers
Readers / Writers

• Accessing a shared data object X
  • Any number of readers can be accessing a shared data object at the same time
  • No writer can access it at the same time as a reader
  • No more than one writer can access it at any time
• This ensures consistency of updates to X
  • Mutual exclusion is too constraining
The Dining Philosophers

• The situation
  • There are 5 philosophers dining together
  • There are 5 chopsticks, placed between them
  • Each philosopher alternates
    • Thinking
    • Eating
  • ---using arbitrary timing

• The problem
  • What algorithm do they use
A bad solution

- Philosopher algorithm
  - Decide to eat
  - Wait till left fork free
  - Pick it up
  - Wait till right fork free
  - Pick it up
  - Eat
  - Put down left fork
  - Put down right fork
  - Think
Why it is bad

- Philosophers may deadlock (all starve)
- Other algorithms
  - as above, but don’t pick up left if right is in use
  - Ps take it in turns to not try and eat
  - variant of Bakery algorithm
- Issues
  - Deadlock, Starvation, Fairness
  - Efficiency
Semaphores

• A single integer variable S
• Accessed via two atomic operations
  • WAIT (sometimes denoted by P)
    • While S <= 0 do waitabit;
    • S := S-1;
  • SIGNAL (sometimes denoted by V)
    • S := S+1;
The atomic operations

• More precisely, the atomic operations are
  • a test-and-decrement operation
    • boolean function Ok;
    • if S <= 0 then return false;
    • S := S-1;
    • return true;
    • end;
  • an increment operation
    • S := S+1;
• Then P is
  • While not Ok do waitbit;
  • and the V operation is not locked out
Uses of semaphores 1

• Mutual exclusion
  • semaphore initialized to 1
    • P(S); critical section; V(S);

• Sequencing
  • semaphore intialized to 0
  • process 1
    • P(S); A();
  • process 2
    • B(); V(S);
Uses of semaphores 2

• Counting
  • semaphore numitems initialized to N
    • P(numitems); remove item;
    • insert item; V(numitems);

• Example -- bounded buffer
R/W using Semaphores 1

• Uses two semaphore variables
  • mutex (for mutual exclusion)
  • writer
    • Readers synchronize with writers via the writer variable
    • And among themselves to update readcount via mutex
  • Writers
    • P(writer);
    • WRITE.......
R/W using Semaphores 2

- Readers
  - P(mutex)
  - readcount := readcount + 1;
  - If readcount = 1 then P(writer);
  - --- Writer might be in the critical section;
  - V(mutex);
  - READ VARIABLE
  - P(mutex)
  - readcount := readcount -1;
  - if readcount = 0, then V(writer);
  - V(mutex);
Bounded buffers

- using semaphores
  - two semaphores
    - nfull
      - initialized to 0, counts full slots
    - nempty
      - initialized to N, counts empty slots
- complete on your own
Binary Semaphores

• Semaphores as defined above
  • sometimes called *counting* semaphores since their values range over an unrestricted domain

• A binary semaphore
  • takes on values 0 or 1 only
  • typically easier to implement
  • is *universal* in that counting semaphores can be built out of them
  • In a sense, a *minimal* structure from which the rest can be realized *without* additional requirements on atomicity
Showing Universality 1

- Implement operations on a (counting) semaphore mutex
  - use binary semaphores \( S_1 = S_2 = 1, S_3 = 0 \)
    - \( V : \)
    - \( P(S_1); \)
    - \( \text{mutex} := \text{mutex} + 1; \)
    - if \( \text{mutex} \leq 0 \) then
      - \( V(S_3); \)
      - \( V(S_1); \)
Showing Universality 2

- \( P : \)
- \( P(S2); \)
- \( P(S1); \)
- \( \text{mutex} := \text{mutex}-1 \)
- \( \text{If mutex} < 0 \text{ then} \)
- \( \quad \text{begin V(S1); P(S3); end;} \)
- \( \quad \text{else V(S1);} \)
- \( \quad \text{V(S2);} \)
Showing Universality 3

- Initially, $S1 = S3 = 1$, $S2 = 0$ and mutex is initialized to the appropriate counting value.
- Note that till $mutex < 0$ all the processes exit from the P segment;
- After this, all subsequent processes get *blocked* on the binary semaphore $S3$ till a $V$ releases them;
- Exercise: verify equivalence with original definition of P and V.
Deadlocks w/Semaphores

• Deadlock example
  • binary semaphores mutex1 and mutex2 initialized to 1:
    • Process P1: P(mutex1); P(mutex2)
    • Process P2: P(mutex2); P(mutex1);
  • the sequence of execution interleaves the two processes yielding P1, P2, P1, P2
  • trivial to see that P1 is waiting on mutex2 and vice-versa
    • deadlock and infinite starvation for both
• Deadlocks will be discussed in the next lecture
Starvation w/Semaphores

- Starvation example
  - in the reader-writer case
    - an infinite stream of readers may never permit the writer in
    - and vice versa (but less “expected”)
  - Semaphores do not guarantee progress by themselves
A Performance Drawback

• Revisit code for P
  • While $S \leq 0$ do waitabit;
  • A process spins on this loop till it gets a chance to enter critical section
• Can waste *substantial* amount of CPU cycles idling
  • Even if waitabit is implemented as
  • give up CPU (i.e. put at end of ready queue)
  • since there are still context switches
• Not a very useful utilization of valuable cycles
Efficient Semaphores

- Implement P and V differently
  - A semaphore is represented as a *record* data-type
    - type mutex = record value: integer; L: list of processes; end;
  - The value field is incremented and decremented as before but rather than *spinning*, the process is added to the end of a list L which is a *wait queue*
  - When the need for spinning is over, the process is removed from this queue and is added to the *ready queue*
- Leverages off the structure of a scheduler
Implementation

• P
  • mutex.value := mutex.value - 1;
  • if mutex.value < 0 then begin
    • add this process to mutex.L
    • BLOCK  <------ suspend the process
  • end;

• V
  • mutex.value := mutex.value + 1;
  • if mutex.value <= 0 then begin  <----TRICKY!!
    • remove a process p from mutex.L
    • WAKEUP(p)  <------ a system call
  • end;
Queue operations

• Note the importance of which process is removed
• Typically FIFO to ensure bounded waiting
• Reduces the indefinite waiting problem to one of correct scheduling
Bugs Using semaphores

- Semaphores are prone to constructing buggy programs
- Examples
  - Leaving out P and/or V
    - loss of mutual exclusion
  - Exchanging the order of P and V
    - deadlock
  - P followed by P
Language Support

- Useful to help with synchronization
  - more convenient
  - more secure
  - less buggy
- Based on
  - Critical Regions
  - communication
Concurrent languages

• Examples
  • Communicating Sequential Procedures (Hoare)
  • Concurrent Pascal
  • Modula 2
  • Ada83, Ada95
  • Newer object-oriented languages
    • Concurrent C, Java
Critical Regions 1

• A high-level language declaration
  • Informally, it can be used to specify that while a statement $S$ is being executed, no more than one process can access a distinguished variable $v$

• Notation
  • $\text{var } v: \text{shared } t;$
  • $\text{region } v \text{ when } B \text{ do } S;$
  • $v$ is shared and of type $t$
  • $B$ is a Boolean expression
  • $S$ is a statement

• Sometimes called Conditional Critical Regions
Critical Regions 2

- Example: bounded buffer producer/consumer
  - var buffer : shared record ... end;
  - region buffer when count < n do insert(item);
  - region buffer when count > 0 do item := extract();
Critical Regions 3

• The OS developer does not have to program the semaphores or alternate synchronization explicitly.
• The compiler "automatically" plugs in the synchronization code using predefined libraries.
• Once done carefully, no chance of mistakes or bugs in designing the delicate synchronization code.
Critical Regions 4

• Implementation using semaphores
  • var mutex, firstdelay, seconddelay: semaphore;
  • var firstcount, secondcount: integer;

  • P(mutex);
  • while not B do begin
    try-and-enter
    end;
  • S;
  • leave-critical-region;
Critical Regions 5

- two phases
  - there are firstcount processes waiting in firstdelay
  - there are secondcount processes waiting in seconddelay
- leave-critical-region
  - if firstcount > 0 then V(firstdelay)
  - else if secondcount > 0 then V(seconddelay) else V(mutex);
Critical Regions 6

- try-and-enter
  - firstcount++;
  - if secondcount > 0 then V(seconddelay) else V(mutex);
  - P(firstdelay);
  - firstcount--;
  - secondcount++;
  - if firstcount > 0 then V(firstdelay) else V(seconddelay);
  - P(seconddelay);
  - secondcount--;
Monitors

- A collection of
  - private data
  - public procedures
  - condition variables (queues of processes)
- Each procedure may have
  - Local variables
  - Formal parameters
Monitors 2

- Execution of these procedures is mutually exclusive
- Operations on condition variables
  - block on a condition variable
    - wait(cv);
  - unblock a process waiting on a condition variable
    - signal(cv);
  - unblock is a no-op if no process is waiting
Example: DP 1

- **Objective:** use a monitor to solve DP without deadlocks
- **Informal idea**
  - algorithm for Philosopher i
    - `dp.pickup(i); eat; dp.putdown(i);`
  - use array to describe Philosopher state
    - `var state: array [0..4] of (thinking, hungry, eating);`
  - use array to block on
    - `var self: array [0..4] of condition;`
Example: DP 2

- **Objective:** use a monitor to solve DP without deadlocks
- **Informal idea**
  - pickup(i)
    - changes state to hungry
    - checks if neighbors are eating
    - if not, grabs forks, and changes state to eating
    - otherwise, waits on self(i)
  - putdown(i)
    - checks both neighbors
    - if either is hungry and can proceed, releases him/her
Example: DP 3

- `type dining_philosophers = monitor
  var state: array [0..4] of (thinking, hungry, eating);
  var self: array [0..4] of condition;

  procedure entry pickup ...
  procedure entry putdown ...
  procedure entry test ...

  begin
    for i := 0 to 4 do state[i] := thinking;
  end;
Example: DP 4

- procedure entry pickup(i: 0..4);
  state[i] := hungry;
  if state(ln(i)) <> eating and state(rn(i)) <> eating then
    state[i] := eating;
  else
    self[i].wait;
Example: DP 5

- procedure entry putdown(i: 0..4);
  state[i] := thinking;
  test (ln(i));
  test (rn(i));
Example: DP 6

- procedure entry test(i: 0..4);
  state[i] := hungry;
  if state[i] = hungry and
    state(ln(i)) <> eating and
    state(rn(i)) <> eating then begin
    state[i] := eating;
    self[i].signal;
Waiting in the Monitor

- Note that the semantics of executing a wait in the monitor is that several processes can be waiting “inside” the monitor at any given time but only one is executing.
- Wait queues are internal to the monitor in that the “wait” and “signal” commands that manipulate the queue are executed from within.
- There can be multiple waiting queues due to conditionals of the above sort within a single monitor.
- Who executes after a “signal” operation?
Sequencing Policies

• P “signals” Q
  • example P finishes eating and signals P+1=Q
• Three choices
  • Signallee Q continues
    • Logically natural since the condition that enabled Q might no longer be true if when Q eventually executes
      • the other neighbor of Q started eating for example
  • Signaller continues until leaving or waiting
• Require that the signal be the last statement in the procedure
What is Missing?

- Philosophers cannot deadlock but can starve
  - For example, we can construct timing relationships such that a waiting philosopher will be stuck in the “self” queue forever
- Monitors have to be enhanced with a fair scheduling policy to avoid starvation
  - both at the level of accessing the monitor
  - as well as to regulate “waking-up” those that are waiting inside
Synchronization in real OSs

- Unix - single CPU OS
  - Interrupts never “force” a context switch
  - They just set flags, or wake up processes
  - Primitives
    - sleep (address, priority);
    - wake_up (address); -- wakes up all processes sleeping on address
  - Typical code
    - while (locked) sleep(bufaddr);
    - locked = false; wake_up (bufaddr);
Synchronization in real OSs 2

- Solaris 2 - multi-CPU OS
  - for brief accesses only
    - Adaptive Mutexes
    - standard spinlock semaphore initially
      - if lock held by running thread, spins
      - otherwise blocks
  - for long-held locks
    - Condition variables
      - wait and signal
    - Reader-writer locks
      - for frequent mostly read-only accesses
Synchronization & Communication

- Synchronization primitives
  - Assuming shared memory
    - locks
    - semaphores
    - monitors
- Providing communication
  - message-passing
  - remote procedure call
Message-passing

- **Primitives**
  - send (to, msg)
  - receive (from, msg)
- **Semantics - various**
  - synchronous or asynchronous
  - blocking or nonblocking
  - message copying / buffering or not
  - atomicity
Synchronous & blocking

• send
  • blocks until “to” does receive
  • message is then transmitted
  • resumes execution
• receive
  • blocks until “from” does send
  • message is then transmitted
  • resumes execution
• buffering not necessary in same name space
• usually atomic
Asynchronous & non-blocking

- messages are buffered
- send
  - blocks only if no buffer available
- receive
  - blocks only until message available
- usually atomic
Other Issues

• In addition to above, need to consider
  • specification of sender/receiver
    • processes
      • to & from are processIDs, or Any (for receive)
    • mail-boxes
      • allows anonymous computation
      • allows multi-process communication
  • replies / acknowledgements
  • error handling
Examples

• Synchronous & blocking
  • Ada83, Ada95
    • anonymous receive with reply
  • Tanenbaum’s Minix OS
    • processID or Any

• Asynchronous & non-blocking
  • Unix pipes & sockets
    • mailboxes, non-atomic

• Both
  • NYU’s Griffin
Equivalence

• Example
  • synchronous message-passing
  • asynchronous message-passing
• Example
  • message-passing
    • asynchronous & non-blocking
  • semaphores
Synch & Asynch MP 1

• asynch >= synch
  • simulate synch send by
    • asend (processto, msg); areceive (processto, ack);
  • simulate synch receive by
    • areceive (processfrom, msg); asend (processfrom, ack);
• synch >= asynch
  • use an intermediate agent process, which
    • processes (synchronous) message-pairs
      • send (agent, “send”); send (agent, msg);
      • send (agent, “receive”); receive (agent, msg);
    • executes
      • loop forever
        receive (anyone, msg);
        process msg;
    • does receive for send requests, & buffers messages
    • does send for receive requests (if msg available)
Message-passing & semaphores

• using semaphores (& shared memory)
  • message-passing is generalized bounded buffer problem
• using message-passing
  • use agent
  • (Exercise 2)