The Basics of Processes

- Processes are the *OS-provided abstraction* of multiple tasks (including user programs) executing concurrently.
- One instance of a program (which is only a passive set of bits) *executing* (implying an execution context – register state, memory resources, etc.)
- OS schedules processes to share CPU.

Why Use Processes?

- To capture naturally concurrent activities within the structure of the programmed system.
- To gain speedup by overlapping activities or exploiting parallel hardware.
  - From DMA to multiprocessors
Separation of Policy and Mechanism

- “Why and What” vs. “How”
- Objectives and strategies vs. data structures, hardware and software implementation issues.
- Process abstraction vs. Process machinery

Can you think of examples?

Process Abstraction

- Unit of scheduling
- One (or more*) sequential threads of control
  - program counter, register values, call stack
- Unit of resource allocation
  - address space (code and data), open files
  - sometimes called tasks or jobs
- Operations on processes: fork (clone-style creation), wait (parent on child), exit (self-termination), signal, kill.

Process-related System Calls in Unix.
Threads and Processes

• Decouple the resource allocation aspect from the control aspect
• Thread abstraction - defines a single sequential instruction stream (PC, stack, register values)
• Process - the resource context serving as a "container" for one or more threads (shared address space)
• Kernel threads - unit of scheduling (kernel-supported thread operations → still slow)
An Example

Doc formatting process

Address Space

Thread

Editing thread: Responding to your typing in your doc

Thread
doc

Autosave thread: periodically writes your doc file to disk

User-Level Threads

- To avoid the performance penalty of kernel-supported threads, implement at user level and manage by a run-time system
  - Contained “within” a single kernel entity (process)
  - Invisible to OS (OS schedules their container, not being aware of the threads themselves or their states). Poor scheduling decisions possible.
- User-level thread operations can be 100x faster than kernel thread operations, but need better integration / cooperation with OS.
Process Mechanisms

- PCB data structure in kernel memory represents a process (allocated on process creation, deallocated on termination).
- PCBs reside on various state queues (including a different queue for each “cause” of waiting) reflecting the process’s state.
- As a process executes, the OS moves its PCB from queue to queue (e.g. from the “waiting on I/O” queue to the “ready to run” queue).

Context Switching

- When a process is running, its program counter, register values, stack pointer, etc. are contained in the hardware registers of the CPU. The process has direct control of the CPU hardware for now.
- When a process is not the one currently running, its current register values are saved in a process descriptor data structure (PCB - process control block)
- Context switching involves moving state between CPU and various processes’ PCBs by the OS.
Process State Transitions

Create Process → Ready

Wakeup (due to interrupt) → Blocked

suspended while another process scheduled → Running

sleep (due to outstanding request of syscall) → Done

Interleaved Schedules

logical concept / multiprocessor implementation

Uni-processor implementation

context switch
The Trouble with Concurrency in Threads...

```
while(i<10)
    {x=x+1; i++;}
```
```
while(j<10)
    {x=x+1; j++;}
```

What is the value of x when both threads leave this while loop?
Nondeterminism

- What unit of work can be performed without interruption? **Indivisible** or **atomic** operations.
- **Interleavings** - possible execution sequences of operations drawn from all threads.
- **Race condition** - final results depend on ordering and may not be “correct”.

```plaintext
while (i<10) {x=x+1; i++;}
load value of x into reg
yield( )
add 1 to reg
yield ( )
store reg value at x
yield ( )
```

### Interleaving

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load x (value 0)</td>
<td>Load x (value 0)</td>
</tr>
<tr>
<td>Incr x (value 1 in reg)</td>
<td>Incr x (value 1 in reg)</td>
</tr>
<tr>
<td>Store x (value 1)</td>
<td>Store x (value 1)</td>
</tr>
<tr>
<td></td>
<td>... Store x for 9th iteration (value 9)</td>
</tr>
<tr>
<td>Store x (value 1)</td>
<td>Load x (value 1) for 10th iteration</td>
</tr>
<tr>
<td>Load x (value 1) for 2nd iteration</td>
<td>Incr x (value 2 in reg)</td>
</tr>
<tr>
<td>Incr x (value 2 in reg)</td>
<td>...</td>
</tr>
<tr>
<td>Store x for 10th iteration (value 10)</td>
<td>Store x (value 2) for 10th iteration</td>
</tr>
</tbody>
</table>
Reasoning about Interleavings

• On a uniprocessor, the possible execution sequences depend on when context switches can occur
  – Voluntary context switch - the process or thread explicitly yields the CPU (blocking on a system call it makes, invoking a Yield operation).
  – Interrupts or exceptions occurring - an asynchronous handler activated that disrupts the execution flow.
  – Preemptive scheduling - a timer interrupt may cause an involuntary context switch at any point in the code.
• On multiprocessors, the ordering of operations on shared memory locations is the important factor.

Critical Sections

• If a sequence of non-atomic operations must be executed as if it were atomic in order to be correct, then we need to provide a way to constrain the possible interleavings in this critical section of our code.
  – Critical sections are code sequences that contribute to “bad” race conditions.
  – Synchronization needed around such critical sections.
• Mutual Exclusion - goal is to ensure that critical sections execute atomically w.r.t. related critical sections in other threads or processes.
  – How?
The Critical Section Problem

Each process follows this template:

while (1)
{
    ...other stuff...  // processes in here shouldn't stop others
    enter_region( );
    critical section
    exit_region( );
}

The problem is to define enter_region and exit_region to ensure mutual exclusion with some degree of fairness.

Implementation Options for Mutual Exclusion

• Disable Interrupts
• Busywaiting solutions - spinlocks
  – execute a tight loop if critical section is busy
  – benefits from specialized atomic (read-mod-write) instructions
• Blocking synchronization
  – sleep (enqueued on wait queue) while C.S. is busy

Synchronization primitives (abstractions, such as locks) which are provided by a system may be implemented with some combination of these techniques.
The Critical Section Problem

```c
while (1)
{
    ...other stuff...

critical section
exit_region();
}
```

Boolean flag[2];
proc (int i) {
    while (TRUE){
        compute;
        flag[i] = TRUE ;
        while(flag[(i+1) mod 2]) ;
        critical section;
        flag[i] = FALSE;
    }
}

flag[0] = flag[1]= FALSE;
fork (proc, 1, 0);
fork (proc, 1,1);

Is it correct?
Assume they go lockstep.
Both set their own
flag to TRUE. Both busywait
forever on the other’s flag ->
deadlock.

Proposed Algorithm for 2 Process Mutual Exclusion
Proposed Algorithm for 2 Process Mutual Exclusion

• enter_region:
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) {no_op};

• exit_region:
  needin [me] = false;

Is it correct?

Interleaving of Execution of 2 Threads (blue and green)

enter_region:  enter_region:
  needin [me] = true;  needin [me] = true;
  turn = you;  turn = you;
  while (needin [you] &&  while (needin [you] &&
      turn == you) {no_op};  turn == you) {no_op};

Critical Section  Critical Section

exit_region: exit_region:
  needin [me] = false;  needin [me] = false;
needin [blue] = true;
needin [green] = true;
turn = green;
turn = blue;
while (needin [green] && turn == green)

*Critical Section*
while (needin [blue] && turn == blue){no_op};
while (needin [blue] && turn == blue){no_op};
needin [blue] = false;
while (needin [blue] && turn == blue)

*Critical Section*
needin [green] = false;

---

**Greedy Version (turn = me)**

needin [blue] = true;
needin [green] = true;
turn = blue;
while (needin [green] && turn == green)

*Critical Section*
turn = green;
while (needin [blue] && turn == blue)

*Critical Section*
Oooops!
Synchronization

• We illustrated the dangers of race conditions when multiple threads execute instructions that interfere with each other when interleaved.
• Goal in solving the critical section problem is to build synchronization so that the sequence of instructions that can cause a race condition are executed \textit{as if} they were indivisible (just appearances)
• “Other stuff” can be interleaved with critical section code as well as the enter\_region and exit\_region protocols, but it is deemed OK.

Peterson’s Algorithm for 2 Process Mutual Exclusion

• enter\_region:
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) {no\_op};
• exit\_region:
  needin [me] = false;

What about more than 2 processes?
Can we extend 2-process algorithm to work with n processes?

Idea: Tournament
Details: Bookkeeping (left to the reader)

Lamport’s Bakery Algorithm

• enter_region:
  choosing[me] = true;
  number[me] = max(number[0:n-1]) + 1;
  choosing[me] = false;
  for (j=0; n-1; j++) {
    { while (choosing[j] != 0) {skip}
      while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
    }
  }

• exit_region:
  number[me] = 0;
Interleaving / Execution Sequence with Bakery Algorithm

Thread 0
Choosing = False
Number [0] = 0

Thread 1
Choosing = False
Number [1] = 0

Thread 2
Choosing = False
Number [2] = 0

Thread 3
Choosing = False
Number [3] = 0

Thread 0
Choosing = True
Number [0] = 0

Thread 1
Choosing = True
Number [1] = 1

Thread 2
Choosing = False
Number [2] = 0

Thread 3
Choosing = True
Number [3] = 1
for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}
for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}

Thread 0
Choosing= False
Number [0]= 2

Thread 1
Choosing= False
Number [1]= 1

Thread 2
Choosing= True
Number [2]= 3

Thread 3
Choosing= False
Number [3]= 1

for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}

Thread 0
Choosing= False
Number [0]= 2

Thread 1
Choosing= False
Number [1]= 1

Thread 2
Choosing= True
Number [2]= 3

Thread 3
Choosing= False
Number [3]= 1

Thread 3 Stuck
for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me))) {skip}
}

Thread 0
    Choosing = False
    Number [0] = 2

Thread 1
    Choosing = False
    Number [1] = 0

Thread 2
    Choosing = True
    Number [2] = 3

Thread 3
    Choosing = False
    Number [3] = 1

for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me))) {skip}
}

Thread 0
    Choosing = False
    Number [0] = 2

Thread 1
    Choosing = False
    Number [1] = 0

Thread 2
    Choosing = False
    Number [2] = 3

Thread 3
    Choosing = False
    Number [3] = 1
Hardware Assistance

- Most modern architectures provide some support for building synchronization: atomic read-modify-write instructions.
- Example: **test-and-set (loc, reg)**
  [ sets bit to 1 in the new value of loc; returns old value of loc in reg ]
- Other examples: compare-and-swap, fetch-and-op

Busywaiting with Test-and-Set

- Declare a shared memory location to represent a busyflag on the critical section we are trying to protect.
- enter_region (or acquiring the “lock”):
  waitloop: tsl busyflag, R0  // R0 = busyflag; busyflag = 1
  bnz R0, waitloop // was it already set?
- exit region (or releasing the “lock”):
  busyflag = 0
Pros and Cons of Busywaiting

• Key characteristic - the “waiting” process is actively executing instructions in the CPU and using memory cycles.

• Appropriate when:
  – High likelihood of finding the critical section unoccupied (don’t take context switch just to find that out) or estimated wait time is very short

• Disadvantages:
  – Wastes resources (CPU, memory, bus bandwidth)

Blocking Synchronization

• OS implementation involving changing the state of the “waiting” process from running to blocked.

• Need some synchronization abstraction known to OS - provided by system calls.
  – mutex locks with operations acquire and release
  – semaphores with operations P and V (down, up)
  – condition variables with wait and signal
Template for Implementing Blocking Synchronization

- Associated with the lock is a memory location (busy) and a queue for waiting threads/processes.
- Acquire syscall:
  ```
  while (busy) {enqueue caller on lock's queue}
  /\ upon waking to nonbusy lock*/ busy = true;
  ```
- Release syscall:
  ```
  busy = false;
  /* wakeup */ move any waiting threads to Ready queue
  ```

Pros and Cons of Blocking

- Waiting processes/threads don’t consume CPU cycles
- Appropriate: when the cost of a system call is justified by expected waiting time
  - High likelihood of contention for lock
  - Long critical sections
- Disadvantage: OS involvement → overhead
Semaphores

• Well-known synchronization abstraction
• Defined as a non-negative integer with two atomic operations
  \[ P(s) - \text{[wait until } s > 0; s--] \]
  \[ V(s) - [s++] \]
• The atomicity and the waiting can be implemented by either busywaiting or blocking solutions.

Semaphore Usage

• Binary semaphores can provide mutual exclusion (solution of critical section problem)
• Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)
• Signaling events (persistent events that stay relevant even if nobody listening right now)
The Critical Section Problem

while (1)
{
  ...other stuff...

  P(mutex)

  critical section

  V(mutex)

}

Semaphore: mutex initially 1

When is a code section a critical section?

Thread 0       Thread 1

a = a + c;      a = a + c;

b = b + c;      b = b + c;
When is a code section a critical section?

Thread 0                  Thread 1
P(mutex)                  P(mutex)
\[ a = a + c; \]           \[ a = a + c; \]
V(mutex)                  V(mutex)
\[ b = b + c; \]           \[ b = b + c; \]

When is a code section a critical section?

Thread 0                  Thread 1
P(mutexa)                 P(mutexa)
\[ a = a + c; \]           \[ a = a + c; \]
V(mutexa)                 V(mutexa)
P(mutexb)                 P(mutexb)
\[ b = b + c; \]           \[ b = b + c; \]
V(mutexb)                 V(mutexb)
When is a code section a critical section?

Thread 0  Thread 1  Thread 2
\( \text{P}(\text{mutex0}) \)  \( \text{P}(\text{mutex1}) \)
\( a = a + c; \)  \( a = a + c; \)  \( c = a + b; \)
\( b = b + c; \)  \( b = b + c; \)
\( \text{V}(\text{mutex0}) \)  \( \text{V}(\text{mutex1}) \)
### When is a code section a critical section?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(\text{mutexa})$</td>
<td>$P(\text{mutexa})$</td>
<td>$P(\text{mutexa})$</td>
</tr>
<tr>
<td>$a = a + c;$</td>
<td>$a = a + c;$</td>
<td>$c = a + b;$</td>
</tr>
<tr>
<td>$V(\text{mutexa})$</td>
<td>$V(\text{mutexa})$</td>
<td>$V(\text{mutexa})$</td>
</tr>
<tr>
<td>$P(\text{mutexb})$</td>
<td>$P(\text{mutexb})$</td>
<td>$P(\text{mutexb})$</td>
</tr>
<tr>
<td>$b = b + c;$</td>
<td>$b = b + c;$</td>
<td>$c = a + b;$</td>
</tr>
<tr>
<td>$V(\text{mutexb})$</td>
<td>$V(\text{mutexb})$</td>
<td>$V(\text{mutexb})$</td>
</tr>
</tbody>
</table>
When is a code section a critical section?

Thread 0
P(mutex);
a = a + c;
b = b + c;
V(mutex);

Thread 1
P(mutex);
a = a + c;
b = b + c;
V(mutex);

Thread 2
P(mutex);
c = a + b;
V(mutex);

Classic Problems

There are a number of “classic” problems that represent a class of synchronization situations

• Critical Section problem
• Producer/Consumer problem
• Reader/Writer problem
• 5 Dining Philosophers
Producer / Consumer

Producer:
while(whatever)
{
    locally generate item
    fill empty buffer with item
}

Consumer:
while(whatever)
{
    get item from full buffer
    use item
}

Producer / Consumer
(with Counting Semaphores)

Producer:
while(whatever)
{
    locally generate item
    P(emptybuf);
    fill empty buffer with item
    V(fullbuf);
}

Consumer:
while(whatever)
{
    P(fullbuf);
    get item from full buffer
    V(emptybuf);
    use item
}

Semaphores: emptybuf initially N; fullbuf initially 0;
What does it mean that Semaphores have *persistence*?
Tweedledum and Tweedledee Problem

- Separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation.
- The `Sleep()` and `Wakeup()` routines operate as follows:
  - Sleep blocks the calling thread,
  - Wakeup unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable.

The code shown above exhibits a well-known synchronization flaw. Outline a scenario in which this code would fail, and the outcome of that scenario:

```c
void Tweedledum()
{
  while(1) {
    Sleep();
    x = Quarrel(x);
    Wakeup(Tweedledee);
  }
}

void Tweedledee()
{
  while(1) {
    x = Quarrel(x);
    Wakeup(Tweedledum);
    Sleep();
  }
}
```

Lost Wakeup:
If dee goes first to sleep, the wakeup is lost (since dum isn’t sleeping yet). Both sleep forever.
Show how to fix the problem by replacing the Sleep and Wakeup calls with semaphore P (down) and V (up) operations.

```c
void Tweedledum()
{
    while(1) {
        P(dum);
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee);
        V(dee);
    }
}

void Tweedledee()
{
    while(1) {
        x = Quarrel(x);
        Wakeup(Tweedledum);
        Sleep();
        V(dum);
        P(dee);
    }
}
```

Semaphore dee = 0;
Semaphore dum = 0;

Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated lock).
- Associated Condition Variables with operations of Wait and Signal.
Condition Variables

• We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.

• **Wait on condition**: releases lock held by caller, caller goes to sleep on condition’s queue. When awakened, it must reacquire lock.

• **Signal condition**: wakes up one waiting thread.

• **Broadcast**: wakes up all threads waiting on this condition.

Monitor Abstraction

EnQ:{acquire (lock);
   if (head == null)
      {head = item;
       signal (lock, notEmpty);}
   else tail->next = item;
   tail = item;
   release(lock);}

deQ:{acquire (lock);
   if (head == null)
      wait (lock, notEmpty);
   item = head;
   if (tail == head) tail = null;
   head=item->next;
   release(lock);}
Monitor Abstraction

EnQ: {acquire (lock);
   if (head == null)
      {head = item;
       signal (lock, notEmpty);}
   else tail->next = item;
   tail = item;
   release(lock);} 

deQ: {acquire (lock);
   if (head == null)
      wait (lock, notEmpty);
   item = head;
   if (tail == head) tail = null;
   head=item->next;
   release(lock);}
Monitor Abstraction

EnQ: {acquire (lock);
  if (head == null)
    {head = item;
     signal (lock, notEmpty);}
  else tail->next = item;
  tail = item;
  release(lock);}

deQ: {acquire (lock);
  if (head == null)
    wait (lock, notEmpty);
  item = head;
  if (tail == head) tail = null;
  head=item->next;
  release(lock);}
Monitor Abstraction

EnQ:
{acquire (lock);
 if (head == null)
   {head = item;
    signal (lock, notEmpty);}
 else tail->next = item;
 tail = item;
 release (lock);}

deQ:
{acquire (lock);
 while (head == null)
   wait (lock, notEmpty);
 item = head;
 if (tail == head) tail = null;
 head = item->next;
 release (lock);}

The Critical Section Problem

while (1)
{
   other stuff ...

   acquire (mutex);

   // conceptually "inside" monitor

   critical section

   release (mutex);

}

conditions
P&V using Locks & CV (Monitor)

P: {acquire (lock);
    while (Sval == 0)
        wait (lock, nonZero);
    Sval = Sval – 1;
    release(lock);}

V: {acquire (lock);
    Sval = Sval + 1;
    signal (lock, nonZero);
    release(lock);}

Nachos-style Synchronization

synch.h, cc

- Semaphores
  
  Semaphore::P
  Semaphore::V

- Locks and condition variables
  
  Lock::Acquire
  Lock::Release
  Condition::Wait (conditionLock)
  Condition::Signal (conditionLock)
  Condition::Broadcast (conditionLock)
Design Decisions / Issues

- Locking overhead (granularity)
- Broadcast vs. Signal
- Nested lock/condition variable problem

```plaintext
LOCK a DO
LOCK b DO
while (not_ready) wait (b, c) //releases b not a
END
END
```

- My advice – correctness first!

Lock Granularity – how much should one lock protect?

Diagram showing a linked list with nodes labeled 2, 4, 6, 8, 3, and 10. The list is directed from head to tail, with arrows indicating connections between nodes. Nodes 2 to 8 and 3 to 10 are highlighted to illustrate the granularity of lock protection.
Lock Granularity – how much should one lock protect?

Concurrency vs. overhead
Complexity threatens correctness

Using Condition Variables

while (! required_conditions) wait (m, c);

- Why we use “while” not “if” – invariant not guaranteed
- Why use broadcast vs. signal – can arise if we are using one condition queue for many reasons. Waking threads have to sort it out (spurious wakeups). Possibly better to separate into multiple conditions (but more complexity to code).
5 Dining Philosophers

Template for Philosopher

```java
while (food available)
{
    /*pick up forks*/
    eat;
    /*put down forks*/
    think awhile;
}
```
Naive Solution

while (food available)
{
    /*pick up forks*/
    P(fork[left(me)]);
    P(fork[right(me)]);
    eat;
    /*put down forks*/
    V(fork[left(me)]);
    V(fork[right(me)]);
    think awhile;
}

Does this work?

Simplest Example of Deadlock

Thread 0          Interleaving          Thread 1

P(R1) →          P(R1)               P(R2)
P(R2)             P(R2)               P(R1)
V(R1)             P(R1) waits         V(R2)
V(R2)             P(R2) waits         V(R1)

R1 and R2 initially 1 (binary semaphore)
Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.

Philosophy 101
(or why 5DP is interesting)

- How to eat with your Fellows without causing Deadlock.
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why Starvation exists and what we can do about it.
Dealing with Deadlock

It can be prevented by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- circular waiting
  - Example: Define an ordering on resources and acquire them in order
- hold and wait
- no pre-emption

Circular Wait Condition

```c
while (food available)
{
  if (me == 0) {P(fork[left(me)]); P(fork[right(me)]);}
  else {(P(fork[right(me)]); P(fork[left(me)]); }
  eat;
  V(fork[left(me)]); V(fork[right(me)]);
  think awhile;
}
```
Hold and Wait Condition

while (food available)
{
    P(mutex);
    while (forks [me] != 2)
    {
        blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);
    }
    forks [leftneighbor(me)] --; forks [rightneighbor(me)]--;
    V(mutex);
    eat;
    P(mutex); forks [leftneighbor(me)] ++; forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)]) {blocking [leftneighbor(me)] = false;
                 V(sleepy[leftneighbor(me)]); }
    if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false;
                 V(sleepy[rightneighbor(me)]); }    V(mutex);
    think awhile;
}

Starvation

The difference between deadlock and starvation is subtle:

– Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.

– In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.

– Rollback and Retry solutions are prone to starvation.

– Continuous arrival of higher priority processes is another common starvation situation.
5DP - Monitor Style

Boolean eating [5];
Lock forkMutex;
Condition forksAvail;

void PickupForks (int i) {
    forkMutex.Acquire( );
    while ( eating[(i-1)%5] || eating[(i+1)%5] )
    forksAvail.Wait(&forkMutex);
    eating[i] = true;
    forkMutex.Release( );
}

void PutdownForks (int i) {
    forkMutex.Acquire( );
    eating[i] = false;
    forksAvail.Broadcast(&forkMutex);
    forkMutex.Release( );
}

What about this?

while (food available)
{
    forkMutex.Acquire( );
    while (forks [me] != 2) {blocking[me]=true;
        forkMutex.Release( ); sleep( ); forkMutex.Acquire( );}
    forks [leftneighbor(me)]--; forks [rightneighbor(me)]--;
    eat;
    forkMutex.Acquire( );
    forks[leftneighbor(me)] ++; forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)] || blocking[rightneighbor(me)])
        wakeup ( ); forkMutex.Release( );
    think awhile;
}
Readers/Writers Problem

Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.

Template for Readers/Writers

Reader()
{while (true) {
  read
  /*request r access*/
  /*release r access*/
}
}

Writer()
{while (true) {
  write
  /*request w access*/
  /*release w access*/
}
}
Template for Readers/Writers

Reader()
{while (true)
{
    fd = open(foo, 0);
    read
    close(fd);
}
}

Writer()
{while (true)
{
    fd = open(foo, 1);
    write
    close(fd);
}
}

startRead();
read
endRead();

startWrite();
write
endWrite();
R/W - Monitor Style

Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire( );
    while ( busy )
        OKtoRead.Wait(&filesMutex);
    numReaders++;
    filesMutex.Release( );
}

void endRead () {
    filesMutex.Acquire( );
    numReaders--;
    if (numReaders == 0)
        OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );
}

void startWrite() {
    filesMutex.Acquire( );
    while (busy || numReaders != 0)
        OKtoWrite.Wait(&filesMutex);
    busy = true;
    filesMutex.Release( );
}

void endWrite() {
    filesMutex.Acquire( );
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );
}

Semaphore Solution with Writer Priority

int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1;
semaphore writePending = 1;
semaphore writeBlock = 1;
Assume the writePending semaphore was omitted. What would happen?
Assume the `writePending` semaphore was omitted. What would happen?
Assume the writePending semaphore was omitted in the solution just given. What would happen?

This is *supposed* to give writers priority. However, consider the following sequence:
Reader 1 arrives, executes thro’ `P(readBlock)`;
Reader 1 executes `P(mutex1)`;
Writer 1 arrives, waits at `P(readBlock)`;
Reader 2 arrives, waits at `P(readBlock)`;
Reader 1 executes `V(mutex1)`; then `V(readBlock)`;
Reader 2 may now proceed…*wrong*