Chapter 6:

Process Synchronization

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Objectives/Outline

Objectives

- Introduce the critical-section problem whose solutions can be used to ensure the consistency of shared data
- Present both software and hardware solutions
- Introduce the concept of atomic transaction
- Describe mechanisms to ensure atomicity

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Background

- Cooperating processes are dependent processes that can affect or be affected by each other.
- Reasons for cooperating processes:
  - Information sharing, modularity, computation speed-up, convenience
- Concurrent access to shared data may result in data inconsistency (race condition).
  - Maintaining data consistency requires synchronization mechanisms to ensure the orderly execution of cooperating processes
  - Synchronization requires some form of communication
- In order to cooperate, processes must be able to:
  - Communicate with one another - Passing information between two or more processes
  - Synchronize their actions - Coordinating access to shared resources
    - Hardware (e.g., printers, drives), Software (e.g., shared code), Files (e.g., data), Variables (e.g., shared memory locations)

Background (cont.)

- Just like shuffling cards, the instructions of two processes are interleaved arbitrarily
- For cooperating processes, the order of some instructions is irrelevant. However, certain instruction combinations must be prevented
- For example:

Process A	Process B	concurrent access
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

Background (cont.): A Concurrency example

time	**Person A**	**Person B**
3:00	Look in fridge. *Out of milk*	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. *Out of milk*
3:15	Buy milk.	Leave for store.
3:20	Leave the store.	Arrive at store.
3:25	Arrive home, put milk away.	Buy milk.
3:30		Leave the store.
3:35		Arrive home. ***OH! OH!***

- Having too much milk isn't a big deal, but in terms of data access there is a problem
  - New milk/data may overwrite the old data
  - What about wasted resources? - too much milk

Background (cont.): Producer-Consumer w Bounded Buffer

- **Consumer process**

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

- **Producer process**

```
while (true) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Although the producer and consumer routines are correct separately, they may not function correctly when executed concurrently

For example: if count = 5 initially and both the producer and consumer are running concurrently, then count can be either 4, 5, or 6.

Producer-Consumer w Bounded Buffer (cont.)

- The statement "count++" may be implemented in machine language as:

  register1 = counter
  register1 = register1 + 1
  counter = register1

- The statement "count--" may be implemented as:

  register2 = counter
  register2 = register2 – 1
  counter = register2

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved
- Interleaving depends upon how the producer and consumer processes are scheduled

Producer-Consumer w Bounded Buffer (cont.)

- Assume counter is initially 5. One interleaving of statements is:

  producer: register1 = counter (register1 = 5)
  producer: register1 = register1 + 1 (register1 = 6)
  consumer: register2 = counter (register2 = 5)
  consumer: register2 = register2 – 1 (register2 = 4)
  producer: counter = register1 (counter = 6)
  consumer: counter = register2 (counter = 4)

- The value of count may be either 4 or 6, where the correct result should be 5
- Hence, count++ and count-- must be performed atomically
  - Atomic operation means an operation that completes in its entirety without interruption

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data may be changed
  - E.g. changing common variables, writing a file, etc
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section
- There is no problem with processes concurrently being in critical sections for different shared resources!

The Critical-Section Problem (cont.)

- We want to execute critical sections atomically
- Treating these as atomic operations is done to ensure that cooperating processes execute correctly
  - Otherwise part of a critical section might be done then another process could do its critical section and then the first could finish
- In the example, we had two people/processes buying milk using the same technique
- It's also possible for processes with different critical sections to access the same resource
- Regardless, only one process can be in a critical section accessing a given resource at a time
  - A critical section exists because of a shared resource. As there may be many shared resources, a process can have different critical sections for various resources

The Critical-Section Problem (cont.)

- General structure of a process

```
do {
        entry section
                critical section (CS)
        exit section
                remainder section
} while (true);
```


The Critical-Section Problem (cont.)

- A solution to the critical-section problem MUST satisfy:
  - Mutual Exclusion: At any time, at most one process can be executing critical section (CS) code
  - Progress: If no process is in its CS and there are one or more processes that wish to enter their CS, it must be possible for those processes to negotiate who will proceed next into CS
    - No deadlock
    - no process in its remainder section can participate in this decision
  - Bounded Waiting: After a process P has made a request to enter its CS, there is a limit on the number of times that the other processes are allowed to enter their CS, before P's request is granted
    - Deterministic algorithm, otherwise the process could suffer from starvation

Two-Process Solution to the Critical-Section Problem --- Algorithm I

```
turn := 0;
Process P0:                  Process P1:
repeat                       repeat

while(turn!=0){};            while(turn!=1){};
      CS                           CS
  turn:=1;                     turn:=0;
      RS                           RS
forever                      forever
```

- Algorithm I uses a shared variable (turn), which is initially assigned 0. Algorithms I:
  - Satisfies mutual exclusion
  - But not progress (i.e., processes MUST strictly alternate turns)

Two-Process Solution to the Critical-Section Problem --- Algorithm II

```
flag[0]:=false;               flag[1]:=false;
Process P0:                   Process P1:
repeat                        repeat
  flag[0]:=true;                flag[1]:=true;
  while(flag[1]){};             while(flag[0]){};
     CS                            CS
  flag[0]:=false;               flag[1]:=false;
     RS                            RS
forever                       forever
```

- Algorithm II uses a shared variable (boolean flag[2]), which is initially assigned as follows: flag [0] = flag [1] = false. That is, flag [i] = true implies that Pi ready to enter its critical section. Algorithm II:
  - Satisfies mutual exclusion
  - But not progress (i.e., interleaving flag[1]:=true and flag[0]:=true means neither can enter CS)

Two-Process Solution to the Critical-Section Problem --- Peterson's Solution

```
flag[0],flag[1]:=false
turn := 0;
Process P0:                        Process P1:
repeat                             repeat
  flag[0]:=true;                     flag[1]:=true;
    // 0 wants in                      // 1 wants in
  turn:= 1;                          turn:=0;
   // 0 gives a chance to 1            // 1 gives a chance to 0
  while(flag[1]&turn=1){};         while(flag[0]&turn=0){};
     CS                                 CS
  flag[0]:=false;                    flag[1]:=false;
   // 0 is done                        // 1 is done
     RS                                 RS
forever                            forever
```

- Algorithm III proved to be correct. Turn can only be 0 or 1 even if both flags are set to true

Activity

- Prove that Peterson's Solution is correct, i.e.
  - Mutual exclusion is preserved
  - The progress requirement is satisfied
  - The bounded-waiting requirement is met

Solution to the CS Problem using Locks

```
do{
    acquire lock
        CS
    release lock
        RS
}while(true);
```

Critical section is protected by locks

Synchronization Hardware

- Peterson's algorithm works only for a pair of processes. How about a mutual execution among three or more threads?
- It would be more efficient to *block* processes that are waiting (just as if they had requested I/O)
  - This suggests implementing the permission/waiting function in the OS
- Hardware solution makes programming task easier
- Hardware solutions:
  - Disable interrupts
  - Special hardware instructions
    - ((h/w implementation of these instructions is beyond the scope of this course and can be found in books on computer architecture))

Hardware Solution 1: Disable Interrupts

```
Process Pi :
repeat
   disable interrupts
      critical section
   enable interrupts
   remainder section
forever
```

- Disable interrupts even time interrupts (while a shared variable is being modified), thus not allowing preemption.
  - Malicious user program may hog CPU forever.
  - Generally, not a practical solution for user programs. But could be used inside an OS
- On a uniprocessor, mutual exclusion is preserved: while in CS, nothing else can run
- On a multiprocessor: mutual exclusion is not achieved
  - Interrupts are "per-CPU"; it is time consuming to disable interrupts on all processors

Hardware Solution 2: Special Hardware Instructions

- Many CPUs today provide hardware instructions to read, modify, and write a word atomically. Some common instructions with this capability include:
    - **TAS**—Test-And-Set (Motorola 68000)
    - **CAS**—Compare-And-Swap (IBM 370 and M68K)
    - **XCHG**—eXCHanGe or simply Swap (x86)
- The idea is to be able to read the contents of a variable (memory location), test it, and set it to something else. This is done all in one execution cycle
  - Hence not interruptible (i.e., atomic operation)

Hardware Solution 2: Special Hardware Instructions (cont.)

- Normally, the memory system restricts access to any particular memory word to one CPU at a time
- Useful extension:
  - machine instructions that perform actions atomically on the same memory location (ex: testing and writing)
- The execution of such an instruction is mutually exclusive on that location (even with multiple CPUs)
- These instructions can be used to provide mutual exclusion
  - but need more complex algorithms for satisfying the requirements of progress and bounded waiting

The Test-and-Set Instruction

- Test-and-Set expressed in "C":

```
boolean TestAndSet(boolean *target)
{
  boolean rv = *target;
  *target = true;
  return rv;
}
```

- Non Interruptible (atomic)!
- One instruction reads then writes the same memory location

Test-and-Set Instruction (cont.)

- An algorithm that uses TestAndSet for Mutual Exclusion:

```
do{
   while(TestAndSet(&lock))
           ;   // do nothing
   CS
   lock=false;
   RS
}while(true)
```

- Process Pi initializes the shared variable lock to false
- Only the first Pi that sets lock enters CS

Test-and-Set Instruction (cont.)

- Mutual exclusion is assured: if P_i enters CS, the other processes are busy waiting
- Satisfies progress requirement
- When P_i exits CS, the selection of the next P_j to enter CS is arbitrary
- Does not satisfy bounded waiting (it is a race!!!)

Swap Instruction

- Some processors (ex: Pentium) provide an atomic Swap(a,b) instruction that swaps the content of a and b
- Executed atomically

```
void Swap(boolean *a, boolean *b)
{
  boolean tmp = *a;
  *a = *b;
  *b=tmp;
}
```


Using Swap for Mutual Exclusion

- Shared variable *lock* is initialized to false
- Each Pi has a local variable *key*
- The only Pi that can enter CS is the one which finds lock=false
- This Pi excludes all other Pj by setting lock to true
- Same as test-and-set

```
do{
  key=true;
  while(key == true)
     Swap(&lock,&key);
  CS
  lock=false;
  RS
}while(true);
```


Semaphores

- Solutions based on machine instructions such as test and set are complicated for application programmers to use
  - E.g, SetAndTest algorithm does not satisfy all the requirements to solve the critical-section problem
    - Starvation is possible.
    - See Fig 6.8 in the textbook for a (complicated) solution
- To overcome this problem, some operating systems provide a synchronization tool called semaphores

Semaphores (cont.)

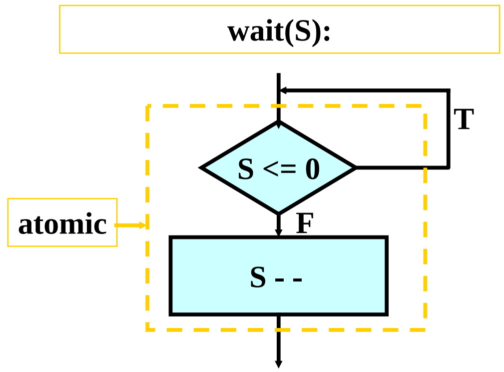
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
  - wait(S)
  - signal(S)
- Types of semaphores
  - Counting semaphore – ranges over unrestricted domain
  - Binary semaphore (also called mutex locks) – ranges only between 0 and 1
- Require disciplined use by programmers

```
wait(S) {
   while (S<=0)
     ;  //no-op
   S--;
}
```

```
signal(S){
   S++;
}
```

Atomicity in Semaphores

- The *test-and-decrement* sequence in *wait* must be atomic, but not the loop
- *Signal* is atomic
- No two processes can be allowed to execute atomic sections simultaneously



Semaphore usage

- Using binary semaphores for CS problem for multiple processes
  - For n processes sharing a semaphore mutex initialized to 1
  - Then only one process is allowed into CS (mutual exclusion)

[Process Pi]

```
do{
  wait(mutex);
   CS
  signal(mutex);
   RS
}while(true);
```


Binary Semaphores in Action

Initialize mutex to 1

[Process Pi]

```
do{
   wait(mutex);
   CS
   signal(mutex);
   RS
}while(true);
```

[Process Pj]

```
do{
  wait(mutex);
  CS
  signal(mutex);
  RS
}while(true);
```


Semaphore usage (cont.)

- Using a counting semaphore for resource allocation
  - to control access to a resource consisting of a finite number of instances
  - semaphore S is initialized to the number of resources available
  - to use a resource instance: perform wait()
    - when the count goes to zero, all resource instances are in use and the process has to wait
  - to release a resource instance: perform signal()

Semaphore usage (cont.)

- Assume P1 and P2 are running concurrently
- Using a binary semaphore to ensure that statement S1 in process P1 is executed before S2 in P2
  - By sharing a semaphore S initialized to 0 between P1 and P2

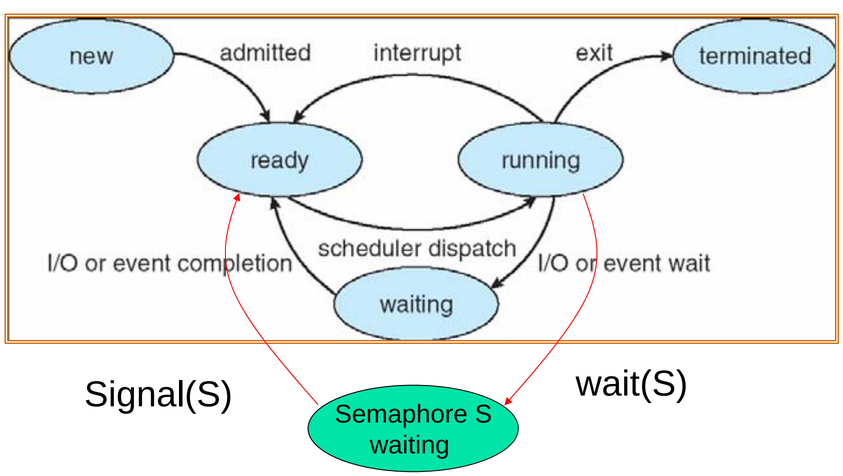
```
Process P1:
S1;
signal(S);
```

```
Process P2:
wait(S);
S2;
```


Semaphore Implementation

- Spinlock semaphores
  - previous semaphore definitions require a process to "spin" while waiting for the lock (busy waiting)
  - is preferred when locks are expected to be held for short times to avoid context switch overhead
  - continual looping is a problem in a real multiprogramming system
- Solution
  - modify the definition of the wait() and signal() semaphore operations
  - Uses a waiting queue for each semaphore
    - Rather than engaging in a busy waiting, the process can block itself and enters the semaphore waiting queue
    - A blocked process should be returned to the ready queue when another process executes the signal()

Semaphore Implementation (cont.)



Semaphore Implementation (cont.)

```
typedef struct {
  int value;
  struct process *list;
}semaphore;
```

```
wait(semaphore *S){
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block(); //suspend the process
   }
}
```

Semaphore Implementation (cont.)

```
signal(semaphore *S){
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P); // resume execution of P
   }
}
```


Deadlock and Starvation

- An implementation of a semaphore with a waiting queue may result in:
  - Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
    - Let *S* and *Q* be two semaphores initialized to 1

```
P0:
wait(S);
wait(Q);
…
signal(S);
signal(Q);
```

```
P1:
wait(Q);
wait(S);
…
signal(Q);
signal(S);
```

  - Starvation: indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
    - If we add or remove processes from the list associated with a semaphore in LIFO manner

Examples of Classic Synchronization Problems

- Classic synchronization problems
  - Bounded-Buffer Problem
  - Readers-Writers Problem
  - Dining-Philosophers Problem
- Commonly used to test and illustrate the power of a newly proposed synchronization (concurrency control) scheme

Bounded-Buffer Problem

- We have n buffers. Each buffer is capable of holding ONE item
- Shared data: **semaphore full, empty, mutex;**
- Initially: **full = 0, empty = n, mutex = 1**

[consumer]

```
do {
   wait(full)
   wait(mutex);
   ...
   remove an item from buffer to nextc
   ...
   signal(mutex);
   signal(empty);
   ...
   consume the item in nextc
   ...
} while (1);
```

[producer]

```
do {
   ...
   produce an item in nextp
   ...
   wait(empty);
   wait(mutex);
   ...
   add nextp to buffer
   ...
   signal(mutex);
   signal(full);
} while (1);
```

Readers-Writers Problem

- A database is shared among several concurrent processes
  - Readers: processes that want to read the database
  - Writers: processes that want to update the database
- If two readers access it simultaneously, no adverse effect would result
- But, a writer should have exclusive access to avoid difficulties that may arise if a writer and another process access the database simultaneously
- Several variations exist of this problem (all have priorities):
  - The first readers-writers problem: no reader should wait unless a writer has already obtained permission to use the shared object
  - The first readers-writers problem: if a writer is waiting, no new readers may start reading
- May result in starvation

Readers-Writers Problem (cont.)

- Solution to the first readers-writers problem:
  - Shared data: semaphore mutex, wrt; int readcount;
  - Initialization: mutex = 1, wrt = 1, readcount = 0

[Writer]

```
do{
   wait(wrt);
   ...
   //writing is performed
   ...
   signal(wrt);
}whie(true)
```

[Reader]

```
do{
  wait(mutex);
  readcount++;
  if (readcount == 1) // first reader
     wait(wrt);
  signal(mutex);
    …
    //reading is performed
    …
  wait(mutex);
  readcount--;
  if (readcount == 0) //last reader
      signal(wrt);
  signal(mutex);
}while(true)
```


Dining-Philosophers Problem

- Five philosophers are sitting around a circular table
- Each one is either thinking, eating or waiting
- There is a single chopstick between each pair of philosophers
- If a philosopher gets hungry, he tries to pick up the chopsticks on either side of him

- The philosopher picks up only one chopstick in a single operation
- Analog: need to allocate several resources among several processes in deadlock-free and starvation-free manner

Dining-Philosophers Problem (cont.)

- Shared data: **semaphore chopstick[5];**
- Initially all values are 1

- Simple solution
- Guarantees that no neighbors eat simultaneously
- Might cause deadlock
- Might cause starvation

[Philosopher i]

```
do {
   wait(chopstick[i])
   wait(chopstick[(i+1) % 5])
   ...
   eat
   ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
   ...
   think
   ...
} while (1);
```

Dining-Philosophers Problem (cont.)

- Deadlock-free solutions
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pickup chopsticks only if both are available (and to be performed atomically as a critical section)
  - Use an asymmetric solution:
    - Odd philosopher picks up left chopstick and then right chopstick
    - Even philosopher picks up right chopstick and then left chopstick
- A deadlock-free does not necessarily eliminate starvation

Incorrect Use of Semaphores

- Can result in timing errors that are difficult to detect
- Example
  - a process interchanges the order of wait and signal operations

```
signal(mutex);
  CS
wait(mutex);
  RS
```

  - several processes may be executing in their critical section simultaneously, violating the mutual exclusion requirement
- Solution
  - Monitors

Monitors

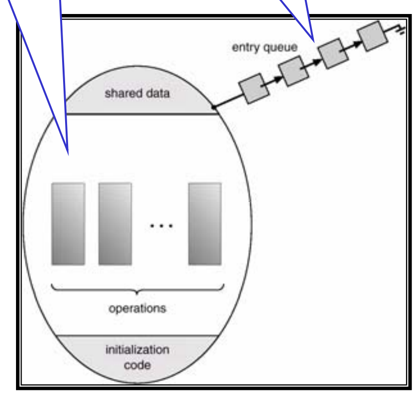
- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is "in the monitor")
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - So the monitor has to have a wait queue…
  - If a thread within a monitor blocks, another one can enter

Monitors

public operations can be called from outside the monitor. Only one process can be active at any moment.

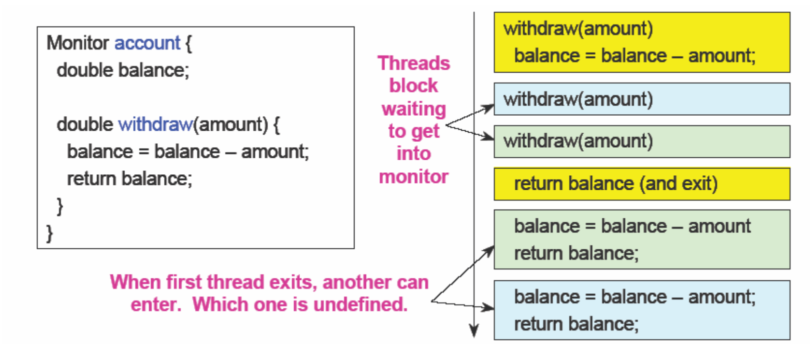
List of processes waiting to enter the monitor

```
monitor monitor-name {
  // shared variable declarations
   procedure P1 (…) {
   . . .
  }
   procedure P2 (…) {
   . . .
   }
   procedure Pn (…) {
    . . .
   }
   initialization code (….) {
    . . .
   }
}
```



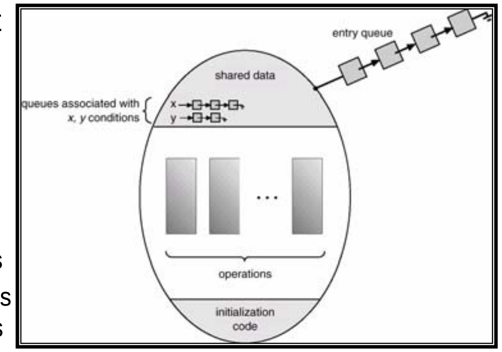
Ensure mutual exclusion

Example



Monitors (cont.)

- Condition variables (CVs) -- allow a process to wait inside the monitor, e.g.
  condition x, y;
- The only operations allowed on a CV is wait and signal
  - x.wait() suspend a process
  - x.signal() resume a process waiting on x; If no process is suspended, signal operation has no effect



Deadlock-Free Solution for Dining-Philosophers Problem Using Monitors

- Impose a restriction that a philosopher picks up chopsticks only if both are available

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i);
    void putdown(int i);
    void test(int i);
    void init() {
       for (int i = 0; i < 5; i++)
          state[i] = thinking;
    }
}
```

[Philosopher i]

```
do {
    dp.pickup(i);
    ...
    eat
    ...
   dp.putdown (i);
    ...
    think
    ...
} while (1);
```


Deadlock-Free Solution for Dining-Philosophers Problem Using Monitors (cont.)

```
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
          self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
}
```

```
void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
          (state[i] == hungry) &&
          (state[(i + 1) % 5] != eating)) {
               state[i] = eating;
               self[i].signal();
    }
}
```

Monitor Implementation Using Semaphores

- Variables
```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```
- Each external procedure ***F*** will be replaced by
```
wait(mutex);
   …
 body of F;
   …
if (next-count > 0)
   signal(next)
else
   signal(mutex);
```
- Mutual exclusion within a monitor is ensured
- Next, we consider how condition variables are implemented…

Monitor Implementation

- For each condition variable ***x***, we have:
```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```
- The operation **x.wait** can be implemented as:
```
x-count++;
if (next-count > 0)
   signal(next);
else
   signal(mutex);
wait(x-sem);
x-count--;
```
- The operation **x.signal** can be implemented as:
```
if (x-count > 0) {
  next-count++;
  signal(x-sem);
  wait(next);
  next-count--;
}
```


Monitor Implementation: Process-Resumption Order

- Conditional-wait construct: x.wait(c);
  - **c** – integer expression evaluated when the wait operation is executed
  - value of **c** (a priority number) stored with the name of the process that is suspended
  - when x.signal() is executed, process with smallest associated priority number is resumed next

```
monitor ResourceAllocator{
   boolean busy;
   condition x;
   void acquire(int time) {
       if (busy)
          x.wait(time);
       busy=true;
    }
   void release() {
       busy=false;
       x.signal();
   }
   initialization_code(){
       busy=false;
    }
}
```


Problems with Monitors

- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols

Solaris 2 Synchronization

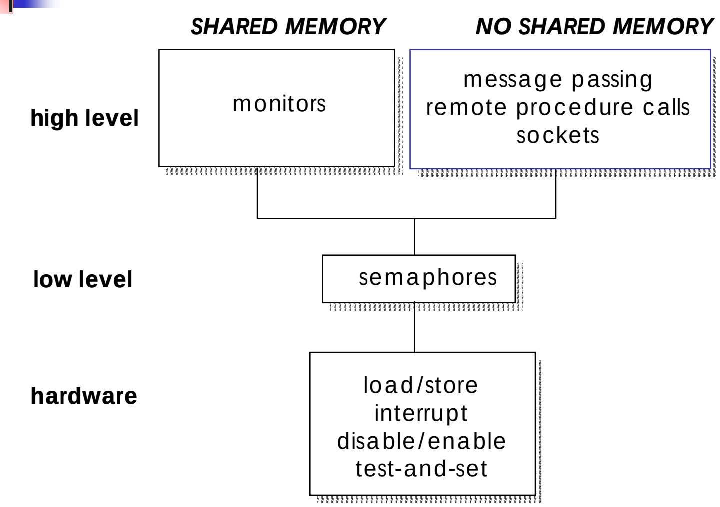
- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data
- Uses *turnstiles* (a queue structure containing threads blocked on a lock) to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows 2000 Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses *spinlocks* on multiprocessor systems

- Also provides *dispatcher objects* which may be used as mutexes and semaphores
- Dispatcher objects may also provide *events*
- An event acts much like a condition variable

Synchronization Primitives — Summary



End of Chapter 6

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005