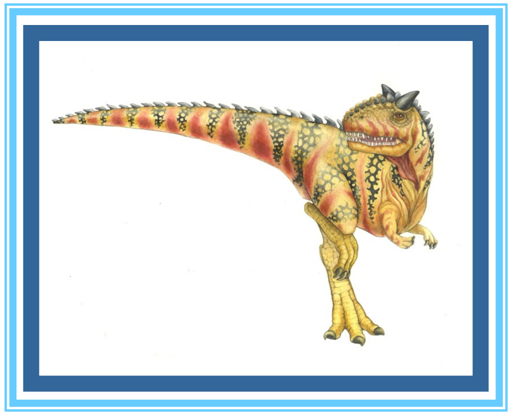
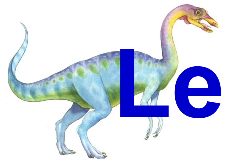
Lecture 6: Process Synchronization





Lecture 6: Process Synchronization

- Background
- Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

Objectives

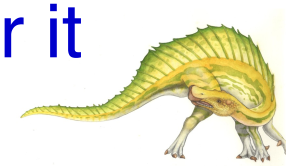
- To Present Concept of Process Synchronization
- To Introduce Critical-Section Problem
  - Whose solutions can be used to ensure consistency of shared data
- To Present both SW & HW Solutions of Critical-Section Problem
- To Examine Several Classical Process-Synchronization Problems

Background

- Processes can Execute Concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent Access to Shared Data may Result in Data Inconsistency
- Maintaining Data Consistency Requires Mechanisms to Ensure Orderly Execution of Cooperating Processes

Background (cont.)

- Problem Statement
  - Suppose that we want to provide a solution to consumer-producer problem that fills ***all*** buffers
- Solution
  - Having an integer **`counter`** that keeps track of number of full buffers
  - **`Counter`** = 0
  - **`Counter`** incremented by producer after it produces a new buffer
  - **`Counter`** decremented by consumer after it consumes a buffer



Producer

```
while (true) {
/* produce an item in next produced */

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

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
/* consume item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

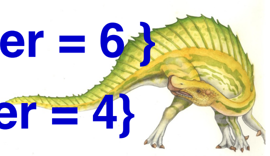
```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute	`register1 = counter`	{register1 = 5}
S1: producer execute	`register1 = register1 + 1`	{register1 = 6}
S2: consumer execute	`register2 = counter`	{register2 = 5}
S3: consumer execute	`register2 = register2 - 1`	{register2 = 4}
S4: producer execute	`counter = register1`	{counter = 6 }
S5: consumer execute	`counter = register2`	{counter = 4}



Critical Section Problem

- Consider System of ***n*** Processes {$p_0,\ldots,p_{n-1}$}
- Each Process has **Critical Section** Segment of Code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical Section Problem:*** Design a Protocol so that Processes can cooperate
  - Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

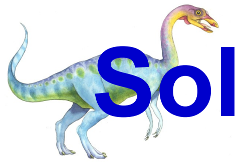
Critical Section

- General Structure of Process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

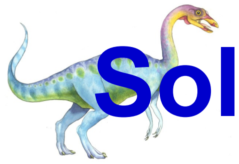
Algorithm for Process P_i

```
do {
    while (turn == j);
           critical section
    turn = j;
           remainder section
  } while (true);
```



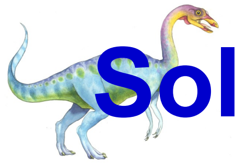
Solution to Critical-Section Problem

- Critical-Section Problem Must Satisfy Following Requirements
  1. Mutual exclusion
  2. Progress
  3. Bounded waiting



Solution to Critical-Section Problem

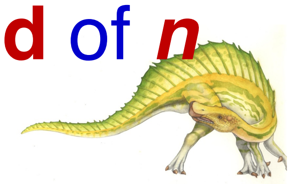
1. **Mutual Exclusion**
   - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress**
   - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then selection of processes that will enter critical section next cannot be postponed indefinitely



Solution to Critical-Section Problem

3. **Bounded Waiting**
   - A bound must exist on number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Other Assumptions
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of ***n*** processes



Race Conditions in OS Kernel

- Example 1
  - A kernel data structure maintaining a list of all open files
  - Two processes open files to simultaneously, separate update to the list ➔ RC
- Example 2:
  - Updating structures maintaining memory allocation
- Example 3:
  - Updating list of processes

Critical-Section Handling in OS

Two approaches for handling CS depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until it exits kernel mode, blocks, or voluntarily yields CPU
  - Does not allow a process running in kernel mode to be preempted
  - Essentially **free of race** conditions in kernel mode
  - Only one process is active in kernel at a time

Preemptive vs. Non-Preemptive Kernels

- Preemptive Kernels
  - More suitable for real-time programming ☺
  - More responsive ☺
  - Very difficult to design for SMP architectures ☹
    - Two kernel-mode processes run simultaneously on different cores
- Non-Preemptive Kernels
  - Can run for an arbitrary long period before relinquishing CPU ☹
  - No race condition in kernel mode ☺

Peterson's Solution

- Good Algorithmic Description of Solving Problem (a SW solution)
- Two-Process Solution
- Assume that `load` and `store` Machine-Language Instructions are Atomic
  - That is, cannot be interrupted
- Two Processes Share Two Variables:
  - `int turn;`
  - `Boolean flag[2]`

Peterson's Solution

- Variable `turn`
  - Indicates whose turn it is to enter critical section
  - `(turn= i or j)`
- `flag` Array
  - Used to indicate if a process is ready to enter critical section
  - `flag[i] = true` implies that process P_i is ready to enter its critical section

Algorithm for Process P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn = = j);
        critical section
   flag[i] = false;
        remainder section
 } while (true);
```

Peterson's Solution (cont.)

- Provable that three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied (no deadlock)
  3. Bounded-waiting requirement is met (no starvation)

Synchronization Hardware

- SW Solutions such as Peterson's Not Guaranteed to Work on Modern Architectures
- Many Systems Provide HW Support for Implementing Critical Section Code
- Any Solution to CS requires Locking
  - All solutions hereafter based on **locking** Idea
  - Protecting critical regions via locks
- How Locking Solves CS Problem?
  - Process must acquire a lock before entering a CS
  - Process releases lock when it exists CS

Synchronization Hardware (cont.)

- Uniprocessors – could Disable Interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Too time-consuming
- Modern Machines Provide Special Atomic HW Instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to CS Problem using Locks

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
   {
          boolean rv = *target;
          *target = TRUE;
          return rv:
   }
```

1. Executed atomically
2. Returns original value of passed parameter
3. Set new value of passed parameter to "TRUE"

Solution using test_and_set()

- Shared Boolean Variable Lock, Initialized to FALSE
- Solution:

```
do {
        while (test_and_set(&lock))
          ; /* do nothing */
                /* critical section */
      lock = false;
                /* remainder section */

   } while (true);
```

compare_and_swap Instruction

Definition:

```
 int compare_and_swap(int *value, int
expected, int new_value) {
        int temp = *value;
        if (*value == expected)
           *value = new_value;
     return temp;
    }
```

compare_and_swap Instruction

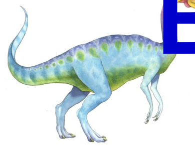
1. Executed Atomically
2. Returns Original Value of Passed Parameter "value"
3. Set Variable "value" the value of passed parameter "new_value" but only if "value" =="expected". That is, swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
 do {
   while (compare_and_swap(&lock,0,1)!= 0)
          ; /* do nothing */
           /* critical section */
     lock = 0;
           /* remainder section */
 } while (true);
```

```
int compare _and_swap(int *value, int expected, int new_value)
```

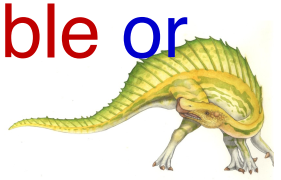


Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Mutex Locks

- HW Solutions Complicated and generally Inaccessible or Invisible to Application Programmers
- OS Designers build SW Tools to Solve Critical Section Problem
- Simplest is mutex lock
  - Abbreviate for mutual exclusion
- Protect a Critical Section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicates if lock is available or not



acquire() and release()

```
acquire() {
   while (!available)
     ; /* busy wait */
available = false;
}
```

```
release() {
     available = true;
}
```

```
do {
      acquire lock
         critical section
      release lock
        remainder section
   } while (true);
```

Mutex Locks (cont.)

- Calls to `acquire()` and `release()` must be Atomic
  - Usually implemented via HW atomic instructions
- But this Solution requires **Busy Waiting**
  - This lock therefore called a **spinlock**
  - Can degrades performance
  - When locks expected to be held for short times, spinlocks are useful (no context switch time)
    - More useful in multi-processors

Semaphore

- Synchronization Tool that Provides more Sophisticated Ways (than Mutex locks) for Process to Synchronize their Activities
- Semaphore ***S*** – Integer Variable
- Can only be Accessed via Two Indivisible (Atomic) Operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()`

Semaphore (cont.)

- Definition of **wait()** operation

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of **signal()** operation

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

Semaphore Usage

- **Counting Semaphore**
  - Integer value can range over an unrestricted domain
  - Used to control access to a given resource consists of a finite number of instances
    - Semaphore is initialized to # of available resources
- **Binary Semaphore** (Same as a **mutex lock**)
  - Can deal with CS problem for multiple processes
  - Integer value can range only between 0 and 1
    - Mutex is initialized to 1

Semaphore Usage (cont.)

- Semaphore can be Used for various Synch Problems
- Consider P_1 & P_2 that require S_1 to happen before S_2: Create a semaphore "**synch**" initialized to 0

```
P1:
    S₁;
    signal(synch);
```

```
P2:
    wait(synch);
    S₂;
```

Semaphore Implementation

- Must Guarantee that no Two Processes can Execute **`wait()`** and **`signal()`** on Same Semaphore at Same Time
- Implementation becomes Critical Section Problem
  - **`wait`** and **`signal`** code are placed in critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note: Applications may Spend lots of Time in Critical Sections and hence this is not a good solution

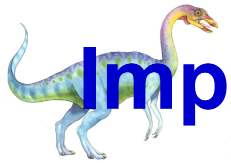
Semaphore Implementation with no Busy waiting

- With each Semaphore, there is an Associated Waiting Queue
- Each Entry in a Waiting Queue has two Data Items:
  - Value (of type integer)
  - Pointer to next record in list

Semaphore Implementation with no Busy waiting

- Two Operations:
  - **Block** – place process invoking operation on appropriate waiting queue
  - **Wakeup** – remove one of processes in waiting queue and place it in ready queue

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



Implementation with no Busy waiting (cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add Pi to S->list;
      block();
   }
}
```

```
wait(S) {
   while (S <= 0)
       ; //busy wait
   S--;
}
```

```
*****************************************************
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove Pi from S->list;
      wakeup(Pi);
   }
}
```

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

Deadlock and Starvation

- **Deadlock** – Two or More Processes are Waiting Indefinitely for an Event that can be Caused by only one of Waiting Processes
- Let *S* and *Q* be 2 Semaphores Initialized to 1

```
          P0                          P1

     wait(S);                    wait(Q);
    wait(Q);                    wait(S);
 ...                          ...
    signal(S);                  signal(Q);
     signal(Q);                 signal(S);
```

Deadlock and Starvation (cont.)

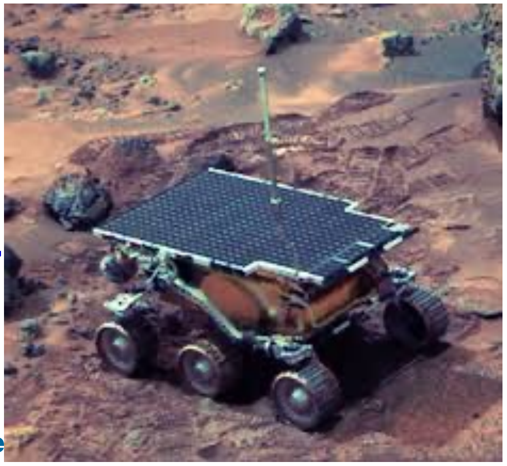
- **Starvation – Indefinite Blocking**
  - A process may never be removed from semaphore queue in which it is suspended
- Example
  - When we remove processes from the semaphore waiting listing in LIFO (Last-In First Out) order

Deadlock and Starvation (cont.)

- **Priority Inversion**
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Example: Consider L, M, and H
  - Priorities L < M < H
  - H requires resource R
  - R is being used by L
  - M comes to ready queue ➔ preempts process L
  - ➔ A lower priority (M) has affected how long H must wait for L to relinquish resource R

Deadlock and Starvation (cont.)

- Possible Solution to Priority Inversion
  - Only allow two levels of priorities
    - ➔ Insufficient for most general-purpose OSes
  - Priority-inheritance protocol
    - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with resource in question
  - Previous example: priorities L < M < H
    - Using priority-inheritance: L=H
  - A real issue in Mars Pathfinder
    - A NASA robot landed on Mars in 1997
      - Read more on Wiki



Classical Problems of Synchronization

- Classical Problems used to Test Synchronization Schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- **n** buffers, each can hold one item
- Semaphore mutex initialized to value 1
- Semaphore full initialized to value 0
- Semaphore empty initialized to value n

Bounded Buffer Problem (Cont.)

- Structure of Producer Process

```
     do {
            . . .
   /* produce an item in next_produced */
            . . .
          wait(empty);
          wait(mutex);
             . . .
    /* add next produced to the buffer */
             . . .
          signal(mutex);
          signal(full);
     } while (true);
```

Bounded Buffer Problem (Cont.)

- Structure of Consumer Process

```
    Do {
         wait(full);
         wait(mutex);
              . . .
/*remove an item from buffer to
next_consumed*/
              . . .
         signal(mutex);
         signal(empty);
              . . .
 /* consume the item in next consumed */
              . . .
       } while (true);
```

Readers-Writers Problem

- A Data Set is Shared among a Number of Concurrent Processes
  - Readers – only read data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – Allow Multiple Readers to Read at Same Time
  - Only one single writer can access shared data at the same time

Readers-Writers Problem

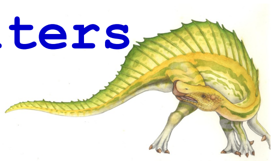
- Several Variations of How Readers and Writers are Considered
  - All involve some form of priorities to readers or writes
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
    - Common to both reader and writer
  - Semaphore **`mutex`** initialized to 1
    - Used to ensure mutual exclusion when read_count is updated
  - Integer **`read_count`** initialized to 0
    - How many processes are currently reading the object

Readers-Writers Problem (Cont.)

- Structure of a Writer Process

```
        do {
              wait(rw_mutex);
                   ...
              /* writing is performed */
                   ...
            signal(rw_mutex);
      } while (true);
```

- `Rw_mutex: Ensures mutual exclusion between writers`



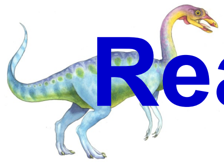
Readers-Writers Problem (Cont.)

- Structure of a Reader Process

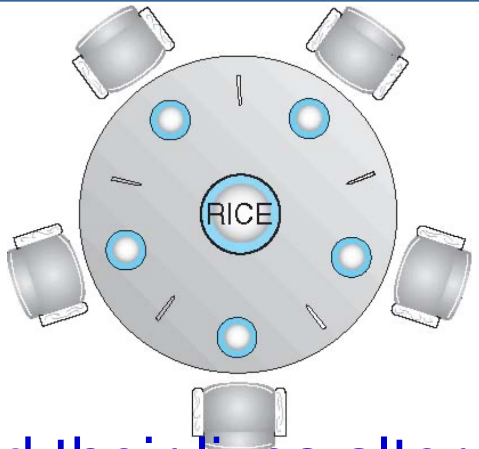
```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
                ...
        /* reading is performed */
                ...
        wait(mutex);
        read count--;
        if (read_count == 0)
                signal(rw_mutex);
    signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***One*** Variation – no reader kept waiting unless writer has permission to use shared object
- ***Another*** Variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks



Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking & eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- Structure of Philosopher *i*:

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

- What is problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock Handling
  - Allow **at most 4** philosophers to be sitting simultaneously at table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution
    - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
    - Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect Use of Semaphore Operations:
  - signal (mutex) …. wait (mutex)
    - ➔ several processes maybe executing in their critical sections simultaneously, violating mutual-exclusion requirement
  - wait (mutex) … wait (mutex)
    - ➔ Deadlock will occur
  - Omitting of wait (mutex) or signal (mutex) (or both)
    - ➔ Mutual exclusion violated or a deadlock will occur
- Main Reasons for Incorrect Use of Semaphores
  - Programming error
  - An uncooperative programmer

Monitors

- A High-Level Abstraction that Provides a Convenient and Less Error-Prone Mechanism for Process Synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
  - External procedures are not allowed to access variables (similar to object-oriented principles)
- A Process Enters Monitor by Invoking one of Its Procedures
- Only One Process may be Active within Monitor at a time (Typically handled by compilers)

Monitors (cont.)

```
monitor monitor-name
{
   Local Data
   Condition Variables
    procedure P1 (…) { …. }
    procedure Pn (…) {……}
   Initialization code (…) { … }
  }
}
```

But not powerful enough to model some synchronization schemes

Condition Variables

- `condition x, y;`
- Two Operations are Allowed on a Condition Variable:
  - `x.wait()` – a process that invokes operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - **If no `x.wait()` on the variable, then it has no effect on the variable**

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel
    - If Q is resumed, then P must wait

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   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);
}
```

Solution to Dining Philosophers (cont.)

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

        initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
        }
}
```

Solution to Dining Philosophers (cont.)

- Each philosopher *i* invokes operations `pickup()` and `putdown()` in following sequence:

```
DiningPhilosophers.pickup(i);

          EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Synchronization Examples

- Reading Assignment
  - More details on monitors
  - Synchronization in Solaris
  - Synchronization in Windows
  - Synchronization in Linux
  - Synchronization in Pthreads

Condition Variables Choices

- Options Include:
  - **Signal and wait** – P waits until Q either leaves monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves monitor or it waits for another condition
- Both have Pros and Cons
  - Language implementer can decide
- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves monitor, Q is resumed
- Implemented in other languages (Mesa, C#, Java)

Monitor Implementation Using Semaphores

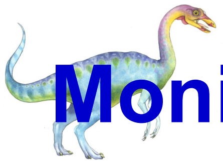
- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each 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



Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- Operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation (cont.)

- Operation **x.signal** can be Implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

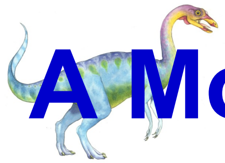
Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify maximum time a process plans to use resource

```
R.acquire(t);

...

access the resource;

...

R.release;
```

- Where R is an Instance of Type

```
ResourceAllocator
```



A Monitor to Allocate Single Resource

```
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;
  }
}
```

Solaris 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
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

Solaris Synchronization (cont.)

- Uses **Condition Variables**
- Uses **Readers-Writers** locks when longer sections of code need access to data
- Uses **Turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives running thread highest of priorities of threads in its turnstile

Windows Synchronization

- Uses Interrupt Masks to Protect Access to Global Resources on Uniprocessor Systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - Spinlocks
  - Reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It Provides:
  - Mutex locks
  - Condition variable
- Non-Portable Extensions include:
  - Read-write locks
  - Spinlocks

Alternative Approaches

- Reading Assignment
  - Transactional memory
  - OpenMP
  - Functional programming languages

Transactional Memory

- A **Memory Transaction**
  - A sequence of read-write operations to memory that are performed atomically

```
void update()
 {
      /* read/write memory */
 }
```

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
{
     #pragma omp critical
     {
          count += value
     }
}
```

Code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

End of Lecture 6