Operating Systems: CSE 3204

Chapter Two
Process Management

Lecture 5: Process Synchronization

ASTU
Department of CSE

Content

Background to Process Synchronization

- Concurrent Processes can be
  - Independent processes
    - cannot affect or be affected by the execution of another process.
  - Cooperating processes
    - can affect or be affected by the execution of another process.
- Cooperating processes can either share directly similar address space or share data through files and messages.
- Concurrent access to shared data may lead to data inconsistency.
- Concurrent execution requires
    - process communication and process synchronization

➢ **Process Synchronization**

- …mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Background to Process synchronization

- Concurrent processes may have access to shared data and resources
- Concurrent access to shared data may result in data inconsistency
- If there is no controlled access to shared data, some processes will obtain an inconsistent view of the shared data
  - Consider two processes P1 and P2, accessing shared data. while P1 is updating data, it is preempted (because of timeout, for example) so that P2 can run. Then P2 try to read the data, which are partly modified, which will result in ***data inconsistency***
- In such cases, the outcome of the action performed by concurrent processes will then depend on the order in which their execution is interleaved
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Synchronization in Producer-Consumer Problem

- producer process produces information that is consumed by a consumer process.
- The previous solution considered the use of a **bounded buffer**
  - Producer produce items and puts it into the buffer
  - Consumer consumes items from the buffer
- Producer and Consumer must **synchronize.**
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **count** that keeps track of the number of full buffers.
- Initially, **count** is set to 0.
- It is **incremented** by the producer after it produces a new buffer
- It is **decremented** by the consumer after it consumes a buffer

Producer - Consumer Problem

Producer

```
while (true)
{
   /*  produce an item and put in
     nextProduced  */
     while (count == BUFFER_SIZE)
             ; // do nothing
      buffer [in] = nextProduced;
 in = (in + 1) % BUFFER_SIZE;
                    count++;
}
```

Consumer

```
while (true)
{
 while (count == 0)
                    ; // do nothing
nextConsumed =  buffer[out];
out = (out + 1) % BUFFER_SIZE;
                             count--;
/*  consume the item innextConsumed

 }
```

- Although both producer and consumer routines are correct when executed separately, they may not properly work when executed concurrently.
- To illustrate this, let’s assume the current value of the count variable is 5

Producer Consumer Problem (cont'd)

- count++ could be implemented as (in a typical machine language):

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as (in a typical machine language):

```
register2 = count
register2 = register2 - 1
count = register2
```

- The concurrent execution of count ++ and count -- is equivalent with the execution of the statements of each operations where arbitrary interleaving of the statements occur. A simple arbitrary interleaving of the operations can be as follows:

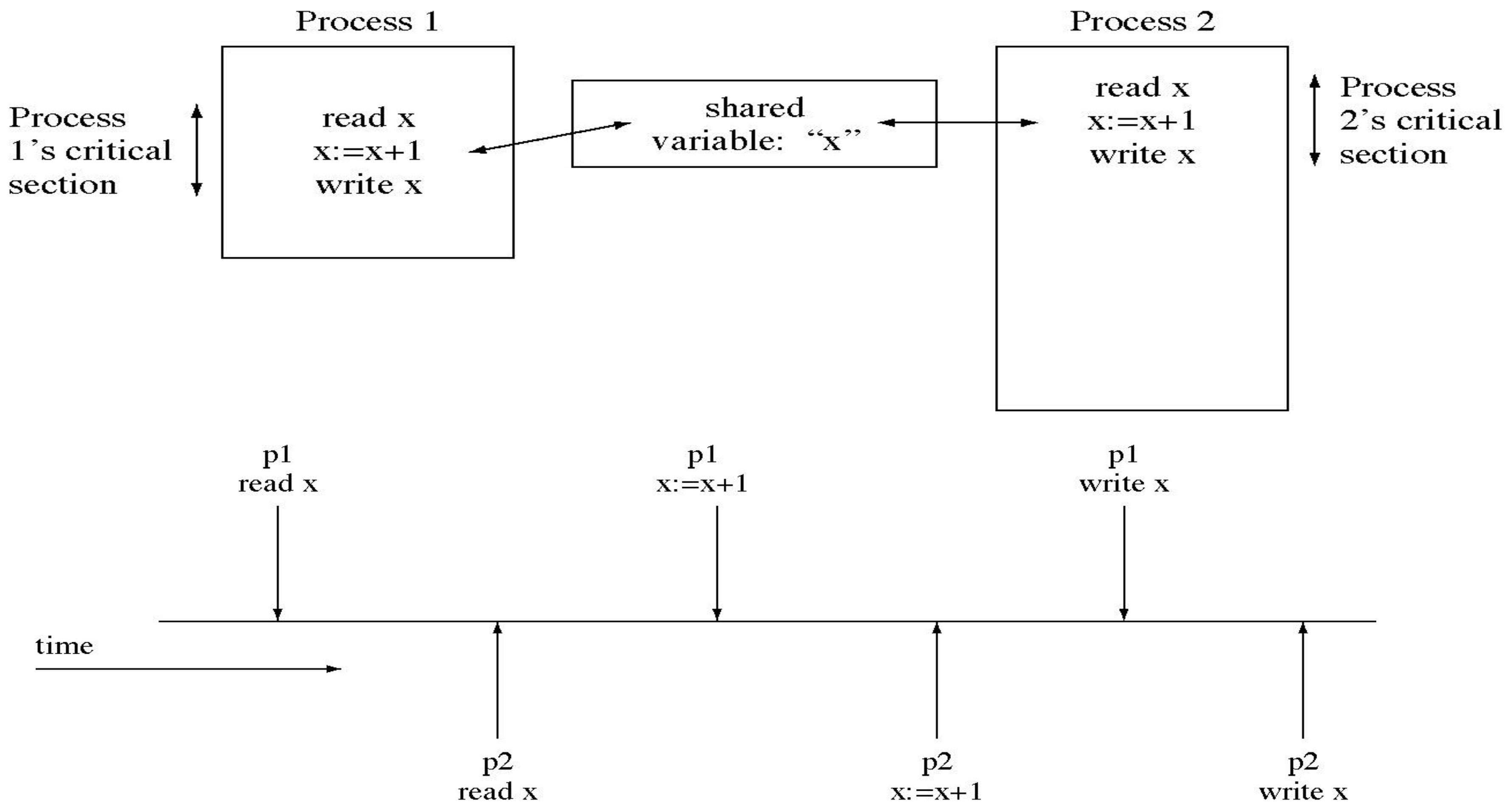
S0: producer execute **register1 = count** **{register1 = 5}**
S1: producer execute **register1 = register1 + 1** **{register1 = 6}**
S2: consumer execute **register2 = count** **{register2 = 5}**
S3: consumer execute **register2 = register2 - 1** **{register2 = 4}**
S4: producer execute **count = register1**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

- At S5, we have arrived at the value of **counter=4**, if we reversed S4 and S5, we would have arrived at count=6
- We would arrive at the incorrect value of the count variable because we allowed both processes to manipulate the variable concurrently

Race condition

- **Race condition** is the situation where several processes access and manipulate shared data concurrently
  - The final value of the shared data depends upon which process finishes last.
  - The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time .
  - To prevent race conditions, concurrent processes must coordinate or be **synchronized**.

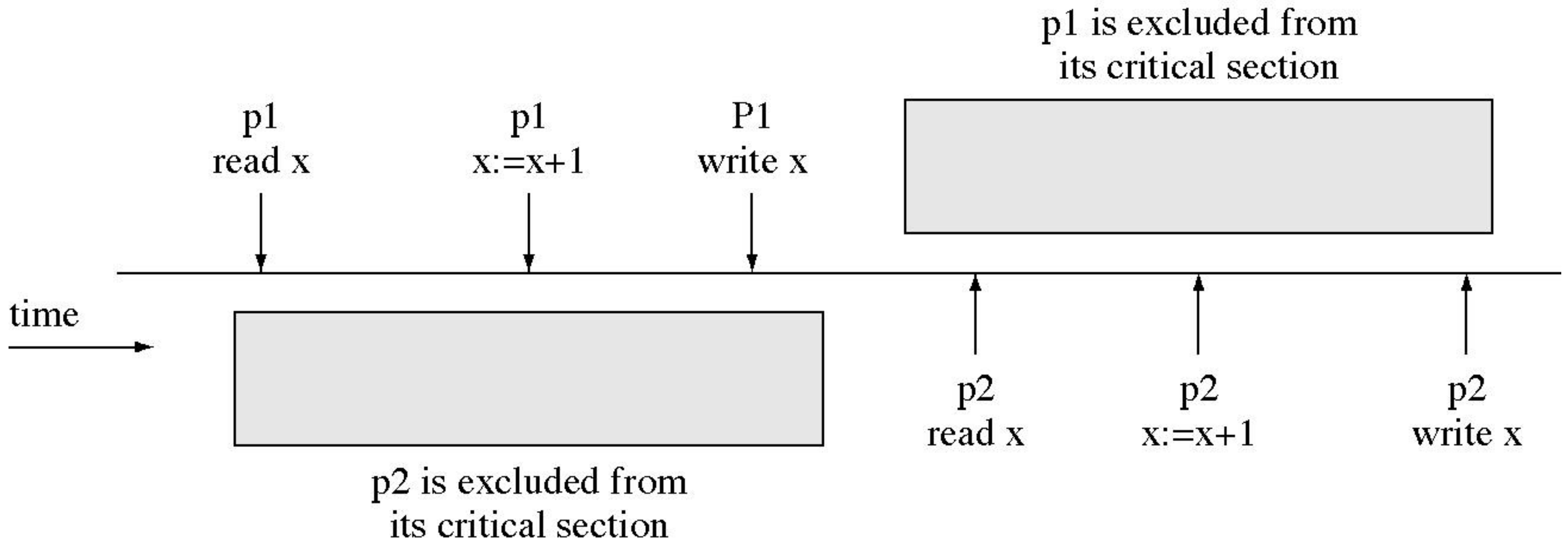
Example: Race condition updating a variable



The Critical-Section Problem

- n processes competing to use some shared data
- Each process has a code segment, called **Critical Section** (CS), in which the shared data is accessed
- A critical section is a piece of code in which a process or thread accesses a common resource
  - So each process must first request permission to enter its critical section. The section of code implementing this request is called the Entry Section (ES)
  - The critical section (CS) might be followed by a Leave/Exit Section (LS)
  - The remaining code is the Remainder Section (RS)
- **Problem** – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS
- When a process executes code that manipulates shared data (or resource), we say that the process is in it’s Critical Section (for that shared data)
- The execution of critical sections must be **mutually exclusive**: at any time, only one process is allowed to execute in its critical section (even with multiple processors)

Critical section to prevent a race condition



- Multiprogramming allows logical parallelism, uses devices efficiently but we lose correctness when there is a race condition.
- So we forbid logical parallelism inside critical section so we lose some parallelism but we regain correctness.

The Critical-Section Problem(cont…)

- So each process must first request permission to enter its critical section.
- The section of code implementing this request is called the **Entry Section (ES).**
- The critical section (CS) might be followed by a **Leave/Exit Section (LS).**
- The remaining code is the **Remainder Section (RS).**
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

- **General structure of process P_i (other process P_j)**

```
do {
    entry section
        critical section
    leave/exit section
        reminder section
} while (1);
```

Solution to critical section problem

- A solution to a critical section problem must satisfy the following **three requirements:**

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 the selection of the processes that will enter the critical section next cannot be postponed indefinitely. Then only those processes that are not executed in their **remainder sections** can participate in the decision on which will enter its critical section next.
3. **Bounded Waiting** - A bound must exist on the 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
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the N processes

Solution to CS problem: Mutual Exclusion

Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections

Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock
- If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the **processor(CPU)** to wait for the critical region

Semaphore

- A Special variable, **S** called a semaphore if it is used for signaling
- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - Wait operation decrements the semaphore value
  - Signal operation increments semaphore value
- Two standard operations modify S: wait ( ) and signal ( )
- Wait and signal operations cannot be interrupted
- Queue is used to hold processes waiting on the semaphore
- If a process is waiting for a signal, it is suspended until that signal is sent

```
wait (S) {
      while S <= 0

; // no-op
              S--;
      }
signal (S) {
        S++;
     }
```

• Less complicated than the hardware based solutions (using TESTandSET() and swap ( ) instructions

Semaphores as General Synchronization Tool

✓ **Two Types of Semaphores**

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as ***mutex*** locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
- Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.

```
Semaphore S;
   //  initialized to 1
wait (S);
   Critical Section
    signal (S);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    - But implementation code of the critical section is short
    - Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore 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 the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of the processes in the waiting queue and place it in the ready queue.

Implementation of wait:

```
wait (S){
    value--;
        if (value < 0) {
add this process to waiting queue
block();  }
  }
```

Implementation of signal:

```
Signal (S){
    value++;
        if (value <= 0) {
remove a process P from the waiting queue

wakeup(P);  }
      }
```

Problems with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes, But wait (S) and signal (S) are scattered among several processes. Hence, difficult to understand their effects
- Usage must be correct in all the processes (correct order, correct variables, no omissions)
- One bad (or malicious) process can fail the entire collection of processes
- **Deadlock and Starvation**
  - **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          P1
wait(S);    wait(Q);
wait(Q);    wait(S);
 ⋮           ⋮
signal(S);        signal(Q);
signal(Q)         signal(S);
```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- The Sleeping Barber Problem

(Read Chapter 6.6 of the text book)

Bounded-Buffer Problem

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

✓ **The structure of the producer process**

```
while (true)  {
                //   produce an item
    wait (empty);
    wait (mutex);
        //  add the item to the  buffer
     signal (mutex);
     signal (full);
}
```

✓ **The structure of the consumer process**

```
while (true) {
    wait (full);
    wait (mutex);
        //  remove an item from  buffer
    signal (mutex);
    signal (empty);
            //  consume the removed
        item
        }
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
    Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- ✓ **The structure of a writer process**

```
while (true) {
    wait (wrt) ;
//    writing is performed

    signal (wrt) ;
}
```

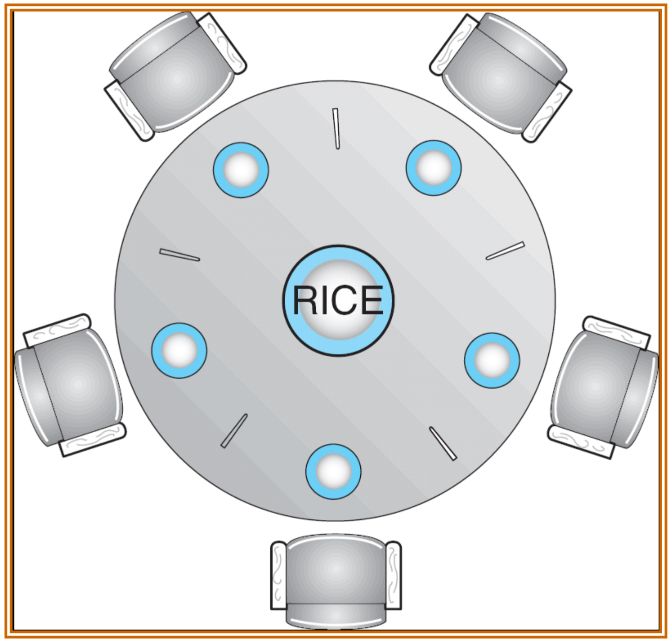
- ✓ **The structure of a reader process**

```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)  wait (wrt) ;
    signal (mutex)

        // reading is performed

    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)  signal (wrt) ;
    signal (mutex) ;
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true)  {
      wait ( chopstick[i] );
     wait ( chopStick[ (i + 1) % 5] );

          //  eat

     signal ( chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );
          //  think
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

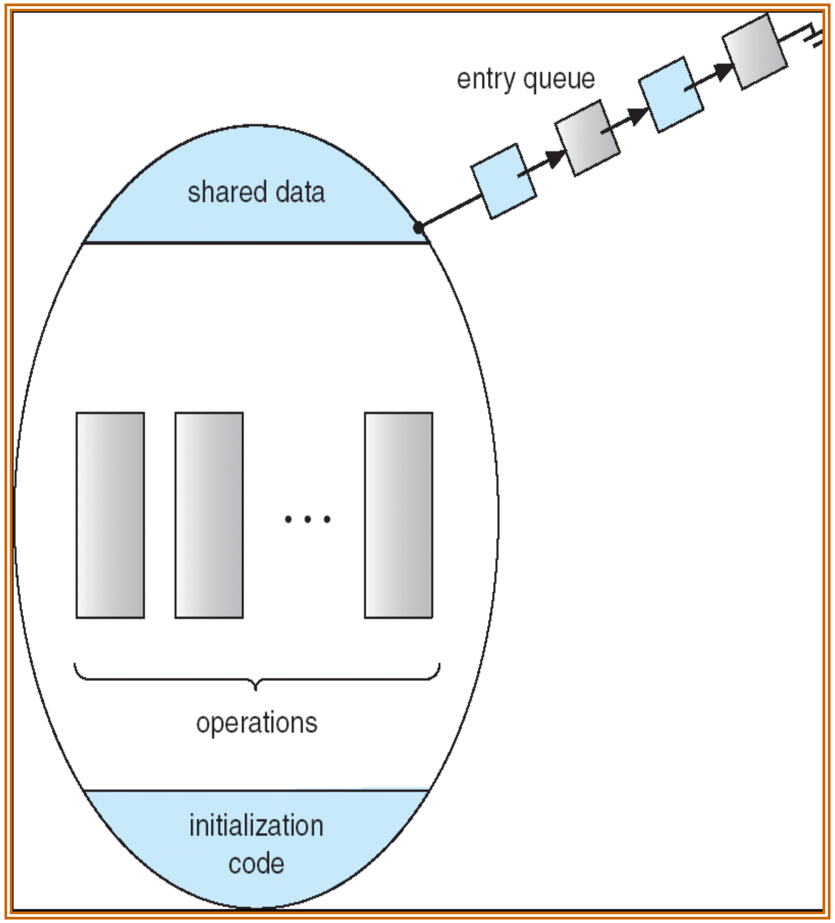
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor is a software module consisting of sets of procedures and local data variables
- **The main characteristics of monitors are**:
  - ✓ Local variables of a monitor are only accessible by the monitor's procedures (no external procedure can access variables of the monitor)
  - ✓ Only one process may be active within the monitor at a time
  - ✓ A process enters the monitor by invoking one of its procedures
  - ✓ By only allowing one process to execute at a time, the monitor provides a mutual exclusion facility
  - ✓ Thus, a shared data structure can be included as part of a monitor and will be protected
  - ✓ If the data in the monitor represents a shared resource, then the monitor provides a mutual exclusion facility on the resource

Monitor (contd.)

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

    Initialization code ( ….) { … }
                …
        }
}
```



Monitors: Condition Variables

- A monitor supports synchronization by the use of **condition variables that are** contained within the monitor and accessible only within the monitor
- Condition variables are a special data type in monitors,
  - Conditon x, y;
- The only operations that can be invoked on condition variables are wait() and signal()
- The operation x.wait () means a process that invokes the operation is suspended until another process invokes the x.signal() operation
  - x.signal () – resumes one of the suspended processes. If there are no suspended operations, then the operation will have no effect.
- **Note that:** The wait() and signal() operations in monitor and semaphore are different. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost. However, in semaphores, the signal operation will always have effect on the state/value of the semaphore variable.

Reading Assignments

- Software Solutions to Synchronization problems
  - Peterson's solution
- Hardware based solutions
  - Swap
  - Test and Set
  - Lock