Unit III: Process Coordination

Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

directly share a logical address space (that is, both code and data)

or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency!

In this chapter,
we discuss various mechanisms to ensure –
The orderly execution of cooperating processes that share a logical address space,

Background

- Concurrent processes executing in the OS may be either independent/ cooperating processes
- Process is independent if it cannot affect/ be affected by other processes
- A processes that does not share any data (temporary/ persistent) is independent
- Process is cooperating if it can affect/ be affected by other processes
- Process that shares data is cooperating
- Concurrent access to shared data may result in data inconsistency

Producer Consumer Problem

- Print program produces characters that are consumed by the printer driver
- Compiler produces assembly code which is consumed by assembler
- Assembler produces object modules, which are consumed by the loader
- To allow producer-consumer to run concurrently, we must have available buffer of items that are filled by the producer and emptied by the consumer
- Producer can produce 1 item while the consumer is consuming another item
- Both should be synchronized so that consumer does not try to consume an item that is not yet produced

- Unbounded buffer: no limit on the size of the buffer.
- Bounded buffer: assumes a fixed buffer size.
  - Consumer must wait if buffer is empty
  - Producer must wait if buffer is full
- 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 and is decremented by the consumer after it consumes a buffer.

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2

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

  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}

"counter++" may be implemented in machine language (on a typical machine) as:

$$
\begin{aligned}
register_1 &= counter \\
register_1 &= register_1 + 1 \\
counter &= register_1
\end{aligned}
$$

"counter--" may be implemented in machine language (on a typical machine) as:

$$
\begin{aligned}
register_2 &= counter \\
register_2 &= register_2 - 1 \\
counter &= register_2
\end{aligned}
$$

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1 + 1$	{ $register_1$ = [illegible] }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2 - 1$	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1 + 1$	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2 - 1$	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization.

Race Condition

- A situation like this where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which access takes place, is called race condition

- To guard against race condition, only 1 process should manipulate the data. Hence synchronization is required

Critical Section

- System has n processes {P0, P1…..Pn-1}
- Each process has a critical section in which process may change common variables, update table, write file etc
- When 1 process is executing the critical section no other process is to be allowed to execute in its critical section
- No 2 processes are executing in their critical sections at the same time.
- Execution of critical sections by the processes is mutually exclusive in time.
- Critical section problem is to design a protocol that processes can use to cooperate

The Critical-Section Problem

Consider a system consisting of n processes $\{ P_0, P_1, \ldots, P_n \}$.

Each process has a segment of code, called a

critical section

in which the process may be changing common variables, updating a table, writing a file, and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections at the same time.

The critical-section problem is to design a protocol that the processes can use to cooperate.

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.

```
do  {

      entry section

            critical section

      exit section

            remainder section

    } while (TRUE);
```

Figure: General structure of a typical process.

A solution to the 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting:

There exists a bound, or limit, 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.

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

Peterson's solution requires two data items to be shared between the two processes:

int turn

→ Indicates whose turn it is to enter its critical section.

boolean flag [2]

→ Used to indicate if a process is ready to enter its critical section.

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

```
do {
    flag [i] = true ;
    turn = j ;
    while ( flag [ j ] && turn == [ j ] ) ;

        critical section

    flag [i] = false ;

        remainder section

} while (TRUE) ;
```

int turn

→ Indicates whose turn it is to enter its critical section.

boolean flag [2]

→ Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

```
do {
      flag [i] = true ;
      turn = j ;
      while ( flag [ j ] && turn == [ j ] ) ;

        critical section

      flag [i] = false ;

        remainder section

  } while (TRUE) ;
```

Structure of process P_j in Peterson's solution

```
do {
      flag [ j ] = true ;
      turn = i ;
      while ( flag [ i ] && turn == [ i ] ) ;

        critical section

      flag [ j ] = false ;

        remainder section

  } while (TRUE) ;
```

- Each process must request permission to enter its critical section (**entry section**)
- Entry section is followed by **critical section**. Critical section may be followed by an **exit section**
- Remainder code is remainder section

do

{

Entry section

Critical section

Exit section

Remainder section

}while (1);

Solution to Critical-Section Problem

- Solution to critical section must satisfy the following requirements:

- Mutual Exclusion - If process $\mathbf{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sections.

- Progress - If no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which 1 will enter it CS next, and this cannot be postponed indefinitely.

- Bounded Waiting - there exists a bound, or limit, 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.

- Solution to critical section problem that satisfy these 3 requirements: If 2 instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order.

Two process solution

- Algorithm 1:
  - Let processes share a common integer variable turn initialized to 0 (or 1)
  - If turn==i, process Pi is allowed to execute in its critical section
  - This solution ensures that only 1 process at a time can be in its critical section
  - It does not satisfy progress requirement (if turn==0 and P1 is ready to enter its critical section, P1 cannot do so, even though P0 may be in its remainder section)

```
do
{
While(turn!=i);
Critical section
Turn=j;
Remainder section
}while (1);
```

P0

```
do
{
While(turn!=0);
Critical section
Turn=1;
Remainder section
}while (1);
```

P1

```
do
{
While(turn!=1);
Critical section
Turn=0;
Remainder section
}while (1);
```

- Algorithm 2:
  - Algo1 does not maintain sufficient information about state of each process.
  - Remembers only which process is allowed to enter CS
  - The remedy is replace turn with Boolean flag[2]
  - Array is initialized to false
  - If flag[i]=true, Pi is ready to enter the CS

P0

do

{

Flag[0]=true;---CS

While(flag[1]); T---P0

Critical section

Flag[0]=false

Remainder section

}while (1);

P1

do

{

Flag[1]=true;--CS

While(flag[0]); T----P1

Critical section

Flag[1]=false

Remainder section

}while (1);

do

{

Flag[i]=true;

While(flag[j]);

Critical section

Flag[i]=false

Remainder section

}while (1);

- Pi sets flag[i]=true, indicating Pi is ready to enter the CS
- Pi checks to verify that Pj is not also ready to enter CS. If Pj was ready then Pi should wait until Pj indicates it no longer needs the CS
- On exit Pi sets flag[i] to be false allowing other processes to enter its CS
- Here, mutual exclusion requirement is satisfied but progress requirement is not

- T0: P0 sets flag[0]=true
- T1: P1 sets flag[1]=true
- P0 and P1 are looping in their respective while loops forever
- Switching the order of flag[i], and testing the value of flag[j], may encounter a situation that violates mutual exclusion requirement

- **Algorithm 3 (PETERSON SOLUTION)**
- By combining Algo1 and Algo 2, we obtain a correct soln to CS, that satisfies all 3 requirements
- The two processes share two variables:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process P_i is ready!

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

- Pi enters the CS only if either flag[j]==false or turn==i

P0

```
do
{
  flag[0] = TRUE;
  turn = 1;
  while (flag[1] && turn == 1);
  critical section
  flag[0] = FALSE;
  remainder section
} while (TRUE);
```

P1

```
do
{
  flag[1] = TRUE;
  turn = 0;
  while (flag[0] && turn == 0);
  critical section
  flag[1] = FALSE;
  remainder section
} while (TRUE);
```

Multiple process solutions

- Bakery Algorithm

Synchronization Hardware

- Many systems provide hardware instructions which can be used to solve critical section problem
- Uniprocessors – could disable interrupts while shared variable is modified
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems. Disabling interrupts may be time consuming as message is passed to all processors
- Modern machines provide special atomic hardware instructions that can be used to solve the CS problem in a relatively simpler manner

TestAndSet instruction

- Instruction is executed atomically-one uninterruptable unit
- If 2 testandset instructions are executed simultaneously (each on different CPU), they will be executed sequentially in some arbitrary order

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

Solution using TestAndSet

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

- Shared Boolean variable lock, initialized to false.
- Solution:

```
do {
      while ( TestAndSet (&lock ));   // do nothing
              //   critical section
      lock = FALSE;
              //     remainder section
} while (TRUE);
```

Semaphores

- Solutions to CS problem discussed so far are not easy to generalize to more complex problems
- Synchronization tool-semaphore
  - **Counting** semaphore – integer value can range over an unrestricted domain
  - **Binary** semaphore (**mutex locks)** – integer value can range only between 0 and 1; can be simpler to implement
- Semaphore *S* – integer variable: apart from initialization can be accessed by 2 atomic operations wait() and signal()
- Less complicated

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

- Modifications to the integer value of S in wait and signal must be executed indivisibly (2 processes cannot modify simultaneously)
- In case of wait(s)
  - S<=0 and S– must be executed without interruption

- Semaphores can be used to deal with n-process CS problem.
- Semaphores can be used to solve various synchronization problems
- Consider 2 running processes P1 with statement S1 and P2 with statement S2
- S2 (P2) should be executed after S1 (P1)
- Let P1 and P2 share a common semaphore synch=0. insert the foll. statements in P1

  S1;

  Signal(synch);
- Foll. Statements in P2

  Wait(synch)

  S2
- Since synch =0, P2 will execute S2 only after P1 has invoked signal (synch), which is after S1

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

Busy Waiting

- The definitions of the `wait()` and `signal()` semaphore operations just described present the busy waiting.
- To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` operations as follows:

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

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

Structure of Pi

```
do
{
    Wait(mutex);
        Critical section
    Signal(mutex);
        Remainder section
} while(1);
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event (resource acquisition and release) that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
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
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Bounded-Buffer Problem

- Readers and Writers Problem

- Dining-Philosophers Problem

Bounded-Buffer Problem

- Bounded buffer: assumes a fixed buffer size.
  - Consumer must wait if buffer is empty
  - Producer must wait if buffer is full

Solution:

- *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.

Solution of Bounded Buffer Problem

- The structure of the producer process

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

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

- The structure of the consumer process

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

Readers-Writers Problem

- A data object (file) 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 –writer and some other process (reader/ writer) access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1 (controls access to readcount)
  - Semaphore wrt initialized to 1 (writer access)
  - Integer readcount initialized to 0 (how many processes are reading object)

Solution to reader-writer problem

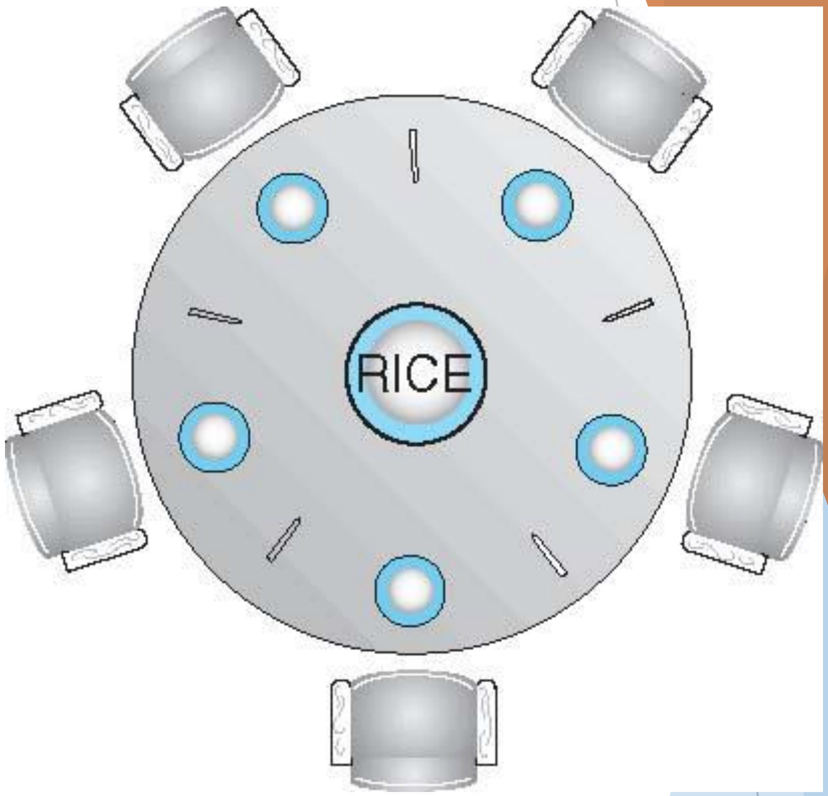
- The structure of a writer process

```
do {
        wait (wrt) ;
            //    writing is performed
        signal (wrt) ;
 } while (TRUE);
```

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
          wait (wrt) ;
        signal (mutex)
              // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
         signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Dining-Philosophers Problem



- 5 philosophers, 5 chairs, 5 single chopsticks, bowl of rice
- Considered as a classic synchronization problem because it is an example of large class of concurrency control problems
- It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free manner

- The 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);
```

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
  - A philosopher grabs the chopstick by executing a wait operation on that semaphore
  - Releases by executing a signal

Errors in semaphores

- Let mutex=1
- Each process must execute wait (mutex) before entering CS and signal (mutex) afterward.
- If this sequence is not observed, 2 processes may be in their CS simultaneously
- Suppose process interchange the order in which wait() and signal() execute

  Signal (mutex)

  CS

  Wait (mutex)

  - Several processes will be in CS simultaneously
- Suppose that a process replaces signal (mutex) with wait (mutex)

  Wait (mutex)

  CS

  Wait (mutex)

  - In this case deadlock will occur

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

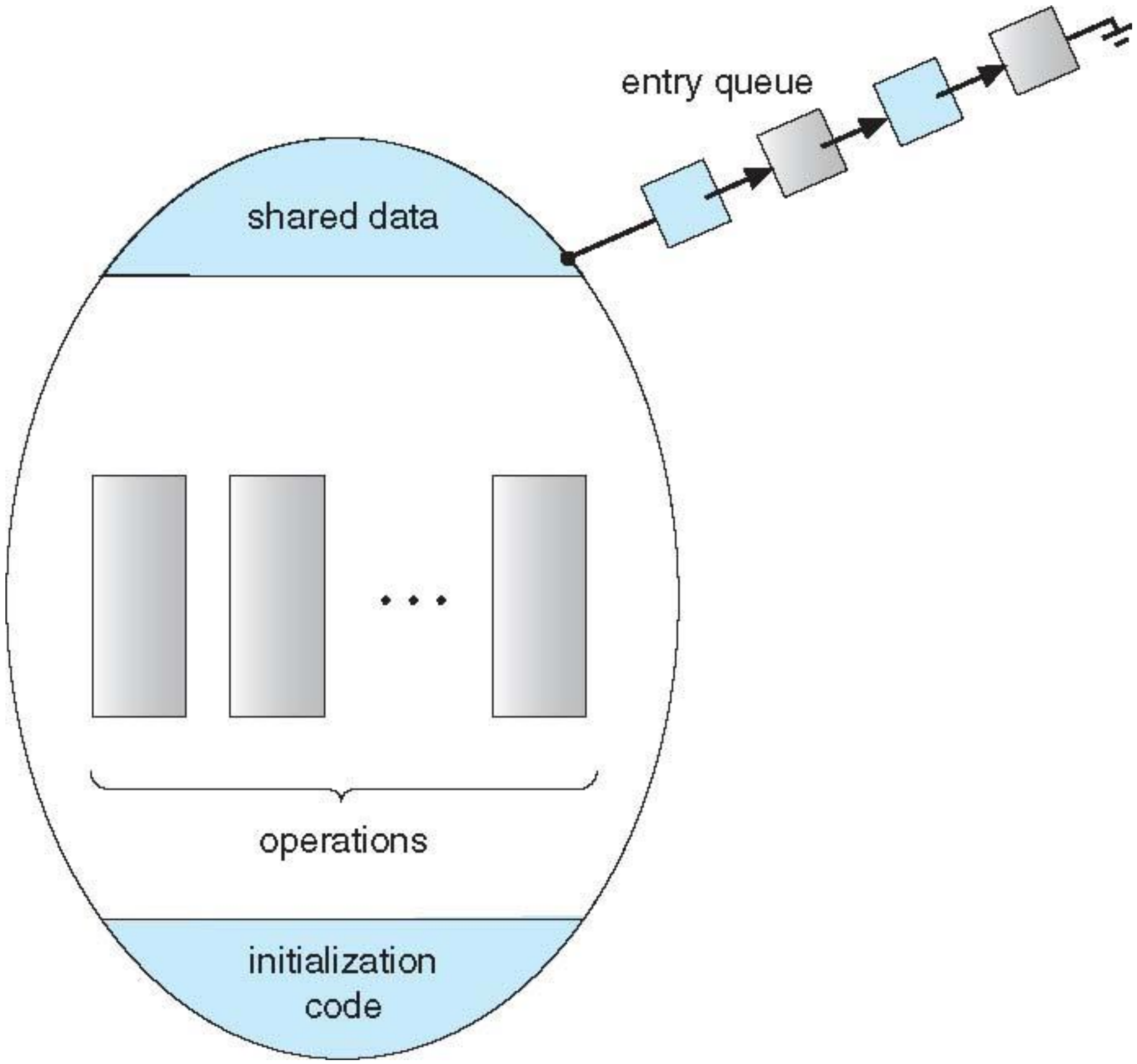
- Only one process may be active within the monitor at a time

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

- Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.

- Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

Schematic view of a Monitor



Condition Variables

- Monitor constructor defined so far is not powerful for modeling some synchronization schemes. Additional synchronization constructs (condition construct) are required
- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until another process invokes signal.
  - x.signal () – resumes exactly 1 suspended process that invoked x.wait ()
- Suppose that x.signal() is invoked by P, there exists a suspended process Q associated with x. 2 possibilities exists:
  - Signal and wait: P either waits until Q leaves the monitor or waits for another condition
  - Signal and continue: Q either waits until P leaves the monitor or waits for another condition

Monitor with Condition Variables