MODULE 3

CHAPTER 1 – PROCESSES SYNCHRONIZATION

CO – Students will be able to outline process synchronization mechanisms and deadlock handling methods.

Prepared By Mr. EBIN PM, AP

COOPERATING PROCESS

- The processes which execute concurrently and affect or get affected by other processes executing in the system are called cooperating processes.
- The processes can affect or get affected by other processes by sharing data or by sharing other resources in the system.
- Thus any process which shares data with other process is a cooperating process.
- The need of such processes is obvious. They will be required to speed up the computation, sharing of information, user convenience etc.

Prepared By Mr. EBIN PM, AP EDULINE 2

- Breaking up of one task into subtasks such that each of them can run in parallel and thus give faster computation results may be one of the requirements.
- In a multi programming environment, processes executing concurrently.
- Concurrent execution of cooperating process requires mechanisms that allow processes to communicate with one another and to synchronize their actions.
- Examples of cooperating processes are transaction processes in airline reservation system. They share a common database, update and read same data on a sharing basis.

RACE CONDITION

- Consider two cooperating processes, sharing variables A and B and having the following set of instructions in each of them:

Process 1	Process 2	Concurrent access
A=1	B=2	Does not matter
A= B+1 B=B+1	B=B*2	Important!

- Suppose our intention is to get **A as 3** and **B as 6** after the execution of both the processes. The interleaving of these instructions should be done in order to avoid race condition.
- If the order of execution is like:

A=1

B=2

A= B+1 A will contain 3 and B will contain 6, as desired.

B=B+1

B=B*2

whereas if the order of execution is like:

A=1

B=2

B=B*2 A will contain 5 and B will contain 5 which is not desired.

A= B+1

B=B+1

- Thus the output of the interleaved execution depends on the particular order in which the access takes place.
- If several processes access and manipulate the same data concurrently, the outcome of the execution depends on the Particular order in which the access takes place. This is called **Race condition.**
- To solve this problem, shared variables A and B should not be updated simultaneously by process 1 and 2.
- Only one process at a time should manipulate the shared variable.
- For that purpose we can use synchronization mechanism.

CRITICAL SECTION PROBLEM

- Consider a system consisting of n processes {P0, P1, ..., Pn−1}.
- 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.
- The important feature of the system is that, when one process is executing in its critical section, no other process is 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.
- Thus the critical section of a process should not be executed concurrently with the critical section of another process.
- This should be ensuring by the synchronization mechanism.

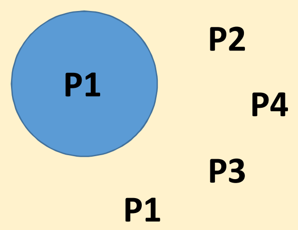
- The general structure of a typical process Pi is shown in Figure:

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


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

- **Mutual exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.(no two processes will simultaneously be inside their critical section)
- **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 deciding which will enter its critical section next, and this selection 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.



PETERSON'S SOLUTION

- It is a classic software-based solution to the critical-section problem.

Process Pi Code

```
do
 { flag[i]=TRUE;
   turn = j;
while( flag[ j] and turn = j )
do no-op;
CRITICAL SECTION
flag[ i] = FALSE;
REMAINDER SECTION
} while(1)
```

Process Pj Code

```
do
   { flag[j] = TRUE;
     turn= i;
    while( flag[ i] and turn = i )
        do no-op;
     CRITICAL SECTION
    flag[ j] = FALSE;
     REMAINDER SECTION
    } while(1)
```

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 − i.
- Peterson's solution requires the two processes to share two data items:

  int turn;

  boolean flag[2];

- The variable turn indicates whose turn it is to enter its critical section. That is, **if turn == i, then process Pi is allowed to execute in its critical section.**

- The flag array is used to indicate if a process is ready to enter (intention) its critical section. For example, if **flag[i] is true**, this value indicates that Pi is ready to enter its critical section.
- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The final value of turn determines which of the two processes is allowed to enter its critical section first.

➢Here,

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded-waiting requirement is met
- To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i.
- To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible.

- If Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section.
- If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section.
- However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. If Pj resets flag[j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

SYNCHRONIZATION HARDWARE

- The software-based solution to the critical-section Problem such as Peterson's are not guaranteed to work on modern computer architectures. The following solutions are based on the premise of locking —that is, protecting critical regions through the use of locks.

Disable interrupts

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified by a process.

- In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by non-preemptive kernels.
- Unfortunately, this solution is not as feasible in a multiprocessor environment

Mutex Locks

- Operating-systems designers build software tools to solve the critical-section problem

- The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.)
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire() function acquires the lock, and the release() function releases the lock.

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


- A mutex lock has a Boolean variable —**available** whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released

The definition of acquire() is as follows:

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

The definition of release() is as follows:

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

- Calls to either acquire() or release() must be performed atomically. The main disadvantage of the implementation given here is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- In fact, this type of mutex lock is also called a **spinlock** because the process spins while waiting for the lock to become available.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.

- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- Spinlocks do have an advantage that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when locks are expected to be held for short times, spinlocks are useful.
- They are often employed on multiprocessor systems where one thread can spin on one processor while another thread performs its critical section on another processor.

SEMAPHORES

- Semaphore is used to solve critical section problem. A semaphore **S** is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait()** and **signal()**
- The wait() operation was originally termed P , signal() was originally called V .

The definition of wait() is as follows:

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

The definition of signal() is as follows:

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

Prepared By Mr. EBIN PM, AP EDULINE 23

- The wait() operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.
- The signal() operation increments the semaphore value. If the value is not positive, then the process blocked by a wait() operation is unblocked.
- Semaphore may be initialized to a non-negative value.
- Semaphores are executed automatically.

Prepared By Mr. EBIN PM, AP EDULINE 24

❖Semaphore Implementation

- The implementation of mutex locks suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same problem.
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal()operations as follows:
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can block itself.

- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

❖To implement semaphores under this definition, we define a semaphore as follows:

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

- Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

wait() semaphore operation can be defined as

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

signal() semaphore operation can be defined as

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

- The block() operation suspends the process that invokes it. The wakeup(P)operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

- Note that in this implementation, semaphore values may be negative. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore

❖**Some properties of semaphore are:**

- Semaphores are machine independent
- Semaphores are simple to implement
- Correctness is easy to determine
- Can have many different critical sections with different semaphores
- Semaphore acquire many resources simultaneously

MONITORS

- The incorrect use of semaphores can result in timing errors.
- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
      ...
critical section
      ...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
    ...
critical section
    ...
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Prepared By Mr. EBIN PM, AP EDULINE 31

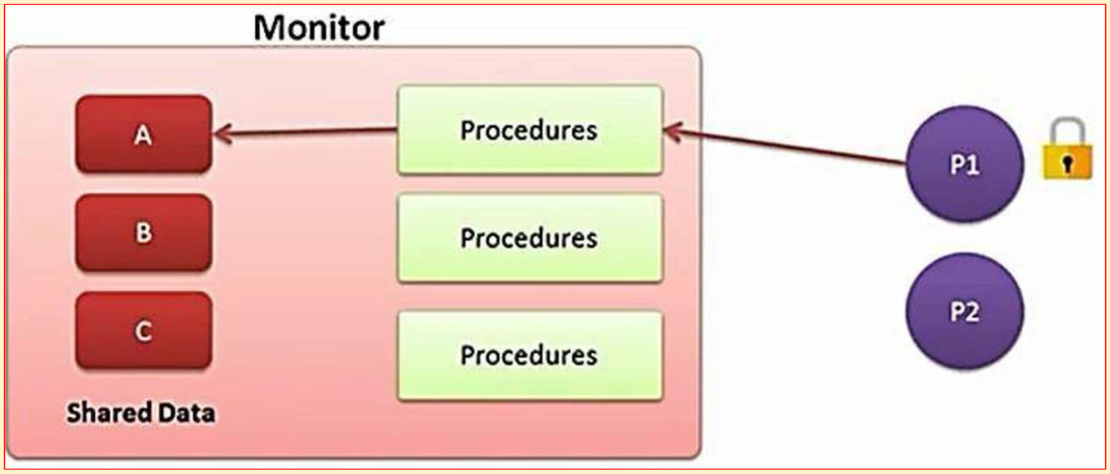
- To deal with such errors, researchers have developed high-level language constructs- the **MONITOR**
- Monitors are based on abstract data types.
- A monitor is a programming language construct that provides equivalent functionality to that of semaphores but is easier to control.
- Monitor provides high-level of synchronization.

➢ Monitor is a module that encapsulates

- Shared data structures
- Procedures that operates on shared data
- Synchronization between concurrent procedure invocations.

➢ Only one process is allowed to enter in Monitor at a time

Prepared By Mr. EBIN PM, AP EDULINE 32



Prepared By Mr. EBIN PM, AP EDULINE 33

Syntax of a monitor

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

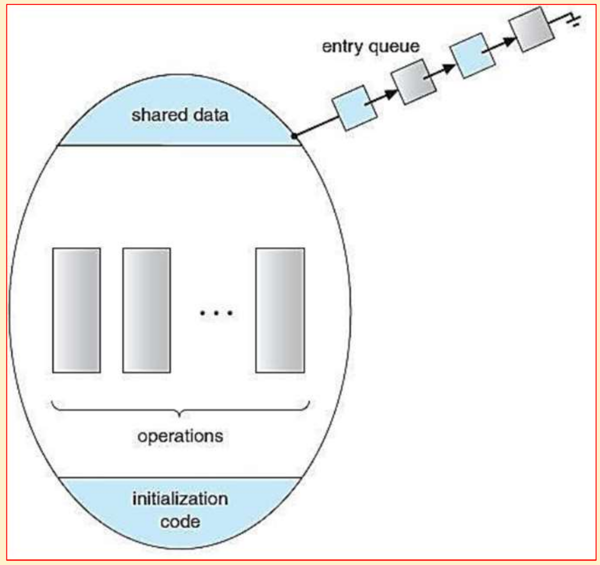
Prepared By Mr. EBIN PM, AP EDULINE 34

- The monitor construct ensures that only one process at a time is active within the monitor.
- Consequently, the programmer does not need to code this synchronization constraint explicitly.
- However, the monitor construct, is not sufficiently powerful for modeling some synchronization schemes.
- For this purpose, we need to define additional synchronization mechanisms.
- These mechanisms are provided by the condition construct.

condition x, y;

- The only operations that can be invoked on a condition variable are wait() and signal().
- The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal();
- The x.signal() operation resumes exactly one suspended process.
- If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed

Schematic view of a monitor



Prepared By Mr. EBIN PM, AP EDULINE 37

- Suppose that, when the x.signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

✓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.

Prepared By Mr. EBIN PM, AP EDULINE 38

READERS-WRITERS PROBLEM

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database **(called Readers)**
- Others may want to update (that is, to read and write) the database **(called Writers)**
- If two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may come.

- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem.

❖**Solution**

➢**Data structures used**

- semaphore **wrt = 1;**
- semaphore **mutex = 1;**
- int **readcount = 0;**

❖The semaphore wrt is common to both reader and writer processes.

❖The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.

The structure of a writer process

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

initially , wrt=1

Prepared By Mr. EBIN PM, AP EDULINE 41

The structure of a reader process

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

initial value of mutex=1

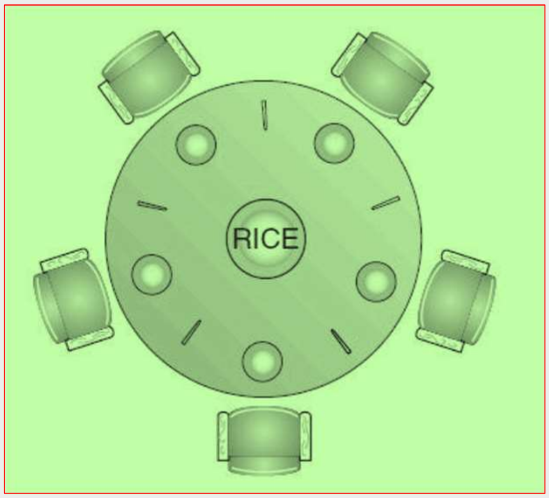
read count=0

wrt =1

Prepared By Mr. EBIN PM, AP EDULINE 42

DINING PHILOSOPHERS PROBLEM

❖The situation of the dining philosophers



- Table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

The structure of philosopher I

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

- where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure.
- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
- All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Prepared By Mr. EBIN PM, AP EDULINE 47

➢ Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick

Prepared By Mr. EBIN PM, AP EDULINE 48