MODULE 2

1. **Process Management**
    - Process Scheduling
    - Basic Concepts
    - Scheduling Criteria
    - Scheduling Algorithms
2. **Process Synchronization**
    - The Critical Section Problem
    - Peterson's Solution
    - Synchronization Hardware
    - Semaphores
    - Classic Problems of Synchronization
    - Monitors

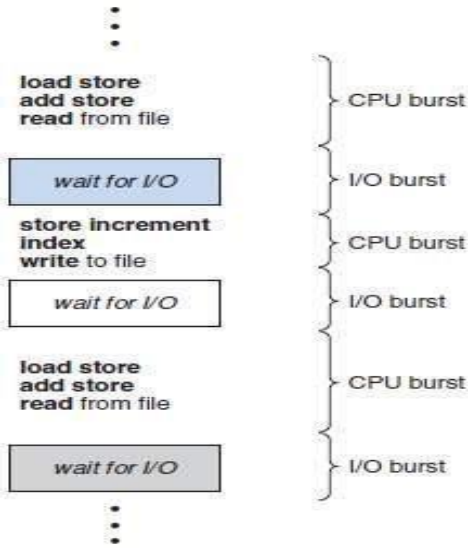
PROCESS MANAGEMENT

Process Scheduling

Basic Concepts

- In a single-processor system,
  - Only one process may run at a time.
  - Other processes must wait until the CPU is rescheduled.
- Objective of multiprogramming:
  - To have some process running at all times, in order to maximize CPU utilization.

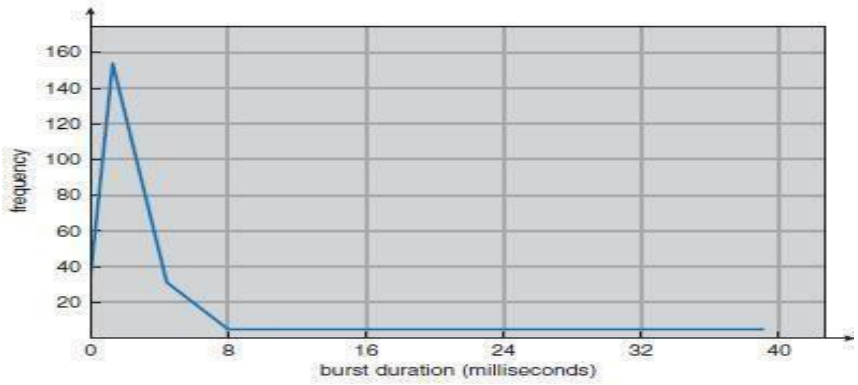
CPU-I/0 Burst Cycle



Alternating sequence of CPU and I/O bursts

- Process execution consists of a cycle of
  - CPU execution and
  - I/O wait
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…

- Finally, a CPU burst ends with a request to terminate execution.
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.



Histogram of CPU-burst durations

CPU Scheduler

- This scheduler
  - selects a waiting-process from the ready-queue and
  - allocates CPU to the waiting-process.
- The ready-queue could be a FIFO, priority queue, tree and list.
- The records in the queues are generally process control blocks (PCBs) of the processes.

CPU Scheduling

Four situations under which CPU scheduling decisions take place:

1. When a process switches from the running state to the waiting state.
   For ex; I/O request.
2. When a process switches from the running state to the ready state.
   For ex: when an interrupt occurs.
3. When a process switches from the waiting state to the ready state.
   For ex: completion of I/O.
4. When a process terminates.

Scheduling under 1 and 4 is non- preemptive. Scheduling under 2 and 3 is preemptive.

Non Preemptive Scheduling

- Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either
  - by terminating or
  - by switching to the waiting state.

Preemptive Scheduling

- This is driven by the idea of prioritized computation.
- Processes that are runnable may be temporarily suspended

Disadvantages:

1. Incurs a cost associated with access to shared-data.
2. Affects the design of the OS kernel.

Dispatcher

- It gives control of the CPU to the process selected by the short-term scheduler.
- The function involves: Switching context
  1. Switching to user mode &
  2. Jumping to the proper location in the user program to restart that program
- It should be as fast as possible, since it is invoked during every process switch.
- ***Dispatch latency*** means the time taken by the dispatcher to
  - stop one process and
  - start another running.

Scheduling Criteria

In choosing which algorithm to use in a particular situation, depends upon the properties of the various algorithms. Many criteria have been suggested for comparing CPU- scheduling algorithms. The criteria include the following:

1. **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
2. **Throughput:** If the CPU is busy executing processes, then work is being done. One

measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

3. **Turnaround time:** This is the important criterion which tells how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0**.**
4. **Waiting time:** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0, it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
5. **Response time:** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

SCHEDULING ALGORITHMS

- CPU scheduling deals with the problem of deciding which of the processes in the ready- queue is to be allocated the CPU.
- Following are some scheduling algorithms:
    1. FCFS scheduling (First Come First Served)
    2. Round Robin scheduling
    3. SJF scheduling (Shortest Job First)
    4. SRT scheduling
    5. Priority scheduling
    6. Multilevel Queue scheduling and
    7. Multilevel Feedback Queue scheduling

1) FCFS Scheduling

- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily done using a FIFO queue.
- Procedure:
  - When a process enters the ready-queue, its PCB is linked onto the tail of the queue.
  - When the CPU is free, the CPU is allocated to the process at the queue's head.
  - The running process is then removed from the queue.

Advantage:

- Code is simple to write & understand.

Disadvantages:

- Convoy effect: All other processes wait for one big process to get off the CPU.
- Non-preemptive (a process keeps the CPU until it releases it).
- Not good for time-sharing systems.
- The average waiting time is generally not minimal.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Example: Suppose that the processes arrive in the order **P1, P2, P3.**
- The **Gantt Chart** for the schedule is as follows:

- Waiting time for *P1* = 0; *P2* = 24; *P3* =27
- Average waiting time: (0 + 24 + 27)/3 = 17ms
- Suppose that the processes arrive in the order **P2, P3, P1.**
- The **Gantt chart** for the schedule is as follows:

- Waiting time for P1 = 6;P2 = 0; P3 =3
- Average waiting time: (6 + 0 + 3)/3 = 3ms

2) SJF Scheduling

- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
- For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the 'length'
- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst

Advantage:

- The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.

Disadvantage:

- Determining the length of the next CPU burst.

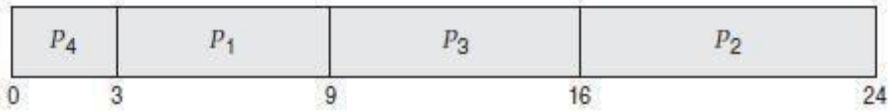
SJF algorithm may be either **1) non-preemptive or 2) preemptive.**

1. **Non preemptive SJF:** The current process is allowed to finish its CPU burst.
2. **Preemptive SJF:** If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted. It is also known as SRTF scheduling (Shortest-Remaining-Time-First).

Example **(for non-preemptive SJF):** Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- For non-preemptive SJF, the Gantt Chart is as follows:

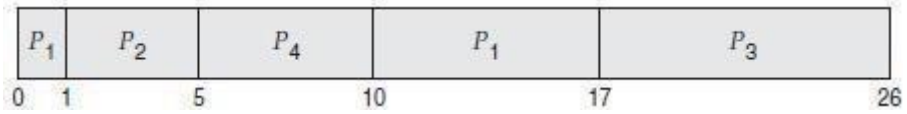


- Waiting time for P1 = 3; P2 = 16; P3 = 9; 4=0 Average waiting time: (3 + 16 + 9 + 0)/4= 7

Preemptive SJF/SRTF: Consider the following set of processes, with the length of the CPU-burst time given in milli seconds.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- For preemptive SJF, the Gantt Chart is as follows:



- The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 =6.5

3) Priority Scheduling

- A priority is associated with each process.
- The CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.

Internally-defined priorities.

- Use some measurable quantity to compute the priority of a process.
- For example: time limits, memory requirements, no. of open files

Externally-defined priorities.

- Set by criteria that are external to the OS For example:
- importance of the process, political factors

Priority scheduling can be either preemptive or non-preemptive.

Preemptive: The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.

Non Preemptive: The new process is put at the head of the ready-queue

Advantage:

- Higher priority processes can be executed first.

Disadvantage:

- Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU. Solution: *Aging* is a technique of increasing priority of processes that wait in system for a long time.

Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- The Gantt chart for the schedule is as follows:

- The average waiting time is 8.2milliseconds.

4) Round Robin Scheduling

- Designed especially for time sharing systems.
- It is similar to FCFS scheduling, but with preemption.

- A small unit of time is called a *time quantum(or timeslice).*
- Time quantum is ranges from 10 to 100ms.
- The ready-queue is treated as a circular queue.
- The CPU scheduler
  - goes around the ready-queue and
  - **allocates the CPU to each process for a time interval of up to 1 time quantum.**
- To implement: The ready-queue is kept as a FIFO queue of processes
- **CPU scheduler**
  1. Picks the first process from the ready-queue.
  2. Sets a timer to interrupt after 1 time quantum and
  3. Dispatches the process.
- One of two things will then happen.
  1. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.
  2. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS. The process will be put at the tail of the ready- queue.

Advantage:

- Higher average turnaround than SJF.

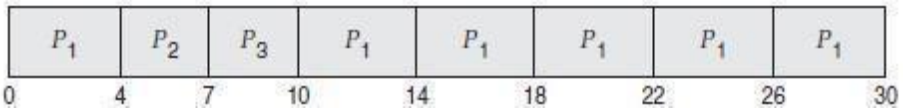
Disadvantage:

- Better response time than SJF.

Example: Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds.

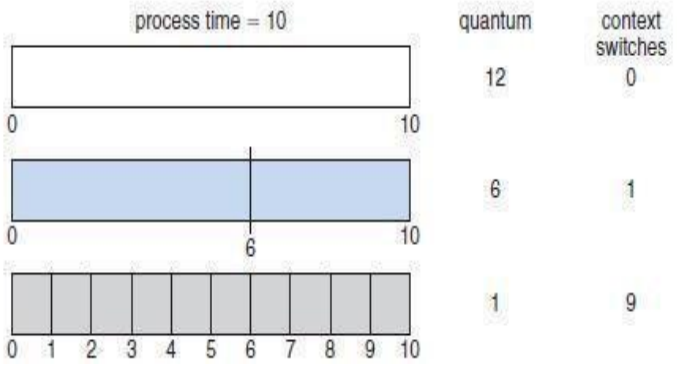
Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart for the schedule is as follows:

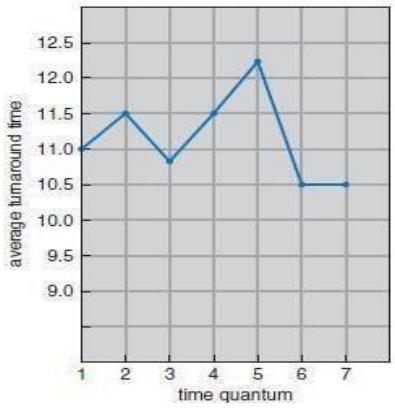


The average waiting time is 17/3 = 5.66 milliseconds.

- **The RR scheduling algorithm is preemptive.**
  No process is allocated the CPU for more than *1* time quantum in a row. If a process CPU burst exceeds *1* time quantum, that process is preempted and is put back in the ready-queue.
- The performance of algorithm depends heavily on the size of the time quantum.
  1. If time quantum=very large, RR policy is the same as the FCFS policy.
  2. If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
- In software, we need to consider the effect of context switching on the performance of RR scheduling
  1. Larger the time quantum for a specific process time, less time is spend on context switching.
  2. The smaller the time quantum, more overhead is added for the purpose of context-switching.



How a smaller time quantum increases context switches

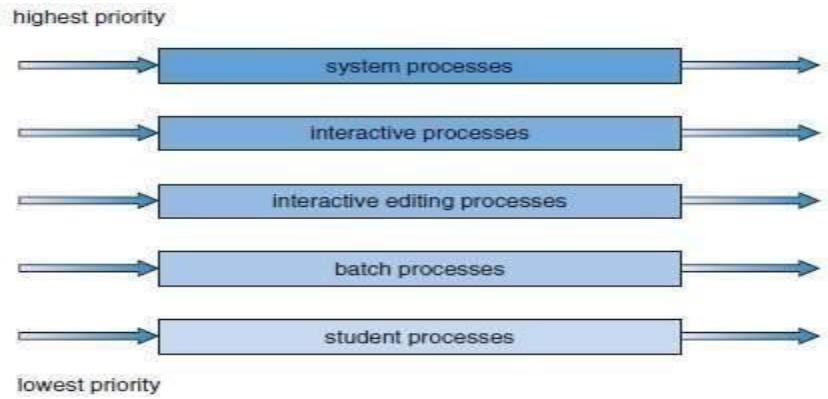


process	time
P_1	6
P_2	3
P_3	1
P_4	7

How turnaround time varies with the time quantum

5) Multilevel Queue Scheduling

- Useful for situations in which processes are easily classified into different groups.
- For example, a common division is made between
  - foreground (or interactive) processes and
  - background (or batch) processes.
- The ready-queue is partitioned into several separate queues.



- The processes are permanently assigned to one queue based on some property like
  - memory size
  - process priority or process type.
- Each queue has its own scheduling algorithm.

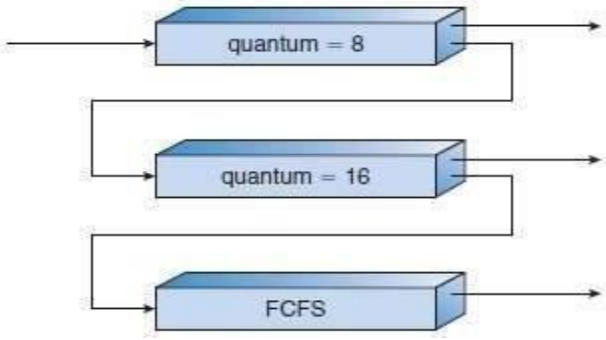
  For example, separate queues might be used for foreground and background processes.
- There must be scheduling among the queues, which is commonly implemented as fixed-

priority preemptive scheduling.

- For example, the foreground queue may have absolute priority over the background queue.
- Time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to background in FCFS.

6) Multilevel Feedback Queue Scheduling

- A process may move between queues
- The basic idea: Separate processes according to the features of their CPU bursts.
- For example
    1. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
    2. If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue. This form of aging prevents starvation.



In general, a multilevel feedback queue scheduler is defined by the following parameters:

1. The number of queues.
2. The scheduling algorithm for each queue.
3. The method used to determine when to upgrade a process to a higher priority queue.
4. The method used to determine when to demote a process to a lower priority queue.
5. The method used to determine which queue a process will enter when that process needs service.

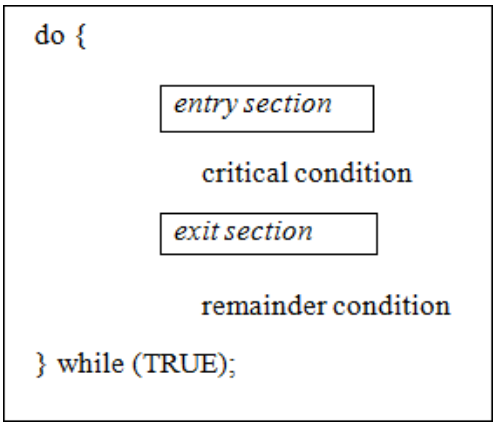
PROCESS SYNCHRONIZATION

The Critical Section Problems

- Consider a system consisting of n processes {Po, 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 soon
- The important feature of the system is that, 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.

The general structure of a typical process Pi is shown in below figure.

- 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 reminder section.



General structure of a typical process Pi

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

1. **Mutual exclusion:** If process Pi 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 deciding 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

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution 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.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_o and P_1 or P_i and P_j where j = 1-i Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

- **turn:** The variable turn indicates whose turn it is to enter its critical section. **Ex:** if turn == i, then process P_i is allowed to execute in its critical section
- **flag:** The flag array is used to indicate if a process is ready to enter its critical section. **Ex:** if flag [i] is true, this value indicates that P_i is ready to enter its critical section.

1. 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.
2. 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 over written immediately.

3. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

```
do {
        flag[i] = TRUE; turn =
        j;
        while (flag[j] && turn == j)
                ; // do nothing critical
         section
        flag[i] = FALSE;

                remainder section

} while (TRUE);
```

The structure of process P_i in Peterson's solution

To prove that solution is correct, then we need to show that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

1) To prove Mutual exclusion

- Each pi enters its critical section only if either flag [j] == false or turn ==i.
- If both processes can be executing in their critical sections at the same time, then flag[0]== flag [1]==true.
- These two observations imply that Pi and Pj could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes (Pj) must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn==j").
- However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section, as a result, mutual exclusion is preserved.

2) To prove Progress and Bounded-waiting

- 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] = 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] = 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 solution to the critical-section problem requires a simple tool-a **lock.**
- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section and it releases the lock when it exits the critical section

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

Solution to the critical-section problem using locks.

- The critical-section problem could be solved simply in a uniprocessor environment if interrupts are prevented from occurring while a shared variable was being modified. In this manner, 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.

- But this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

test and set( ) and swap( ) instructions

- Many modern computer systems provide special hardware instructions that allow to **test** and **modify** the content of a word or to **swap** the contents of two words **atomically**, that is, as one uninterruptible unit.
- Special instructions such as test_and_set () and swap() instructions are used to solve the critical-section problem.
- The test_and_set () instruction can be defined as shown in Figure. The important characteristic of this instruction is that it is executed atomically.

Definition:

```
boolean test_and_set(boolean  *target)
{

      boolean rv = *target;
      *target = true;
            return rv;
}
```

The definition of the test_and_set () instruction.

- Thus, if two TestAndSet () instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then implementation of mutual exclusion can be done by declaring a Boolean variable lock, initialized to false.

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

Mutual-exclusion implementation with test_and_Set ()

- The Swap() instruction, operates on the contents of two words, it is defined as shown below

Definition:

```
voidSwap (boolean *a, boolean *b)
    {

        boolean temp = *a;
        *a = *b;
        *b = temp:
    }
```

The definition of the Swap( ) instruction

- Swap() it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process Pi is shown in below.

```
do {
        key = TRUE;
        while ( key == TRUE)
        Swap (&lock, &key );

                //      critical section
         lock =FALSE;
                //       remainder section
    } while (TRUE);
```

Mutual-exclusion implementation with the Swap() instruction

- These algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded- waiting requirement.
- Below algorithm using the test_and_set () instruction that satisfies all the critical-section requirements. The common data structures are

boolean waiting[n];

boolean lock;

These data structures are initialized to false.

```
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);
```

Bounded-waiting mutual exclusion with test_and_set ()

1) To prove the mutual exclusion requirement

- Note that process Pi can enter its critical section only if either waiting [i] == false or key==false.
- The value of key can become false only if the test_and_set( ) is executed.
- The first process to execute the test_and_set( ) will find key== false; all others must wait.
- The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

2) To prove the progress requirement

Note that, the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3) To prove the bounded-waiting requirement

- Note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).
- It designates the first process in this ordering that is in the entry section (waiting[j]==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

SEMAPHORE

- A semaphore is a synchronization tool is used solve various synchronization problem and can be implemented efficiently.
- Semaphores do not require busy waiting.
- A semaphore S is an integer variable that is accessed only through two standard atomic operations: wait () and signal (). The wait () operation was originally termed P and signal() was called V.

Definition of wait ():

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

Definition of signal ():

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

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Binary semaphore

- The value of a binary semaphore can range only between 0 and 1.
- Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion. Binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to 1.

Each process Pi is organized as shown in below figure

```
do {
        wait (mutex);
                // Critical Section signal
          (mutex);
                // remainder section
} while (TRUE);
```

Mutual-exclusion implementation with semaphores

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore. When a process releases a resource, it performs a signal() operation.
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Implementation

- The main disadvantage of the semaphore definition 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 entry code.

- 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. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.

Semaphore implementation with no busy waiting

- The definition of the wait() and signal() semaphore operations is modified.
- 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.
- To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
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.
- **The wait()** semaphore operation can now be defined as:

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

- **The signal ()** semaphore operation can now 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.
- 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.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal( ) operation. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1.

Po	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
. .	
. .	
signal(S);	signal(Q);
signal(Q);	signal(S);

- Suppose that Po executes wait (S) and then P1 executes wait (Q). When Po executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until Po executes signal(S). Since these signal() operations cam1ot be executed, Po and P1 are deadlocked.
- Another problem related to deadlocks is indefinite blocking or starvation: A situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

1) 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.

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

The structure of the producer process

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

The structure of the consumer process

2) 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
    - Dataset
    - Semaphore **mutex** initialized to 1.
    - Semaphore **wrt** initialized to1.
    - Integer **readcount** initialized to 0.

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

The structure of a writer 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) ;
}
```

The structure of a reader process

3) Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

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. 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.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Solution: 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];

where all the elements of chopstick are initialized to 1. The **structure of philosopher** is shown

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

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 pickup her chopsticks only if both chopsticks are available.
- 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.

Problems with Semaphores

Correct use of semaphore operations:

- signal (mutex) …. wait (mutex) : Replace signal with wait and vice-versa
- wait (mutex) … wait(mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

Monitor

- An abstract data type—or ADT—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
- A **monitor type** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.
- The monitor construct ensures that only one process at a time is active within the monitor.

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

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .

   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Syntax of the monitor

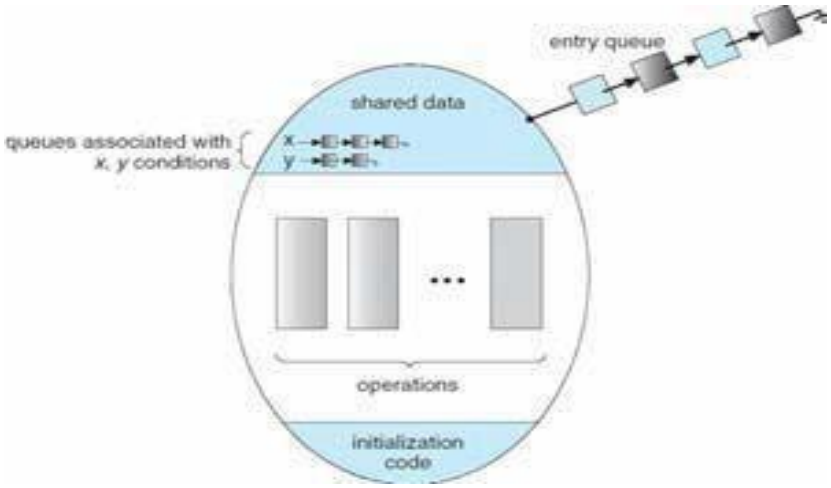
- To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition* Two operations on a condition variable:

Condition x, y

- The only operations that can be invoked on a condition variable are wait() and signal(). The operation

x.wait () – a process that invokes the operation is suspended.

x.signal () – resumes one of processes (if any) that invoked x.wait ()



Monitor with Condition Variables

Solution to Dining Philosophers

- Each philosopher I invokes the operations **pickup()** and **putdown()** in the following

```
monitor DP
{
        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);
}
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;
}
}
```

- For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

- For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

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

- The operation x.signal() can be implemented as

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

Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct

can be used. This construct has the form

x.wait(c);

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

- The Resource Allocator monitor shown in the above Figure, which controls the allocation of a single resource among competing processes.
- A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);

……

access the resource;

…..

R.release();

where, R is an instance of type Resource Allocator.

- The monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:
- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).