

## CS8493 OPERATING SYSTEMS UNIT -II

### UNIT II - PROCESS MANAGEMENT

Processes-Process Concept, Process Scheduling, Operations on Processes, Interprocess Communication; Threads- Overview, Multicore Programming, Multithreading Models; Windows 7 - Thread and SMP Management. Process Synchronization - Critical Section Problem, Mutex Locks, Semaphores, Monitors; CPU Scheduling and Deadlocks.

#### PROCESS CONCEPT

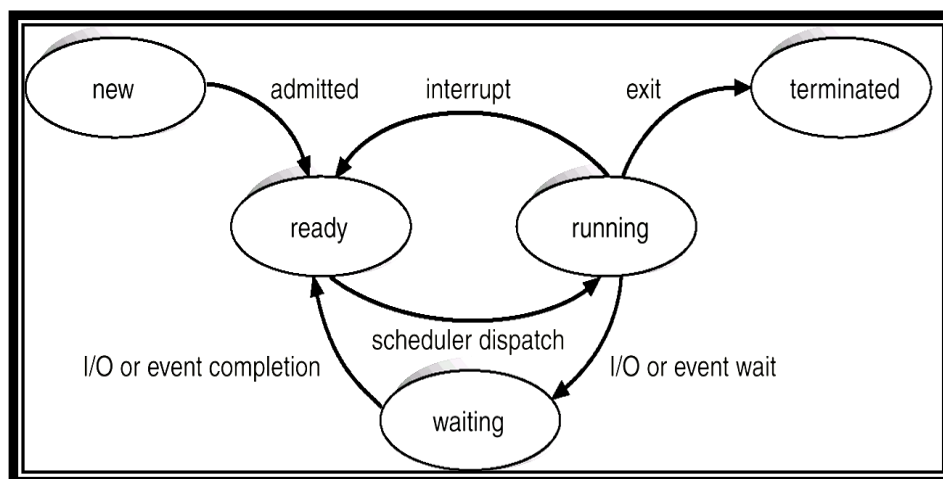
- A process can be thought of as a program in execution.
- A process is the unit of work in a modern time-sharing system.
- A process generally includes the process stack, which contains temporary data (such as method parameters, return addresses, and local variables), and a data section, which contains global variables.

#### Difference between program and process

- A **program** is a **passive** entity, such as the contents of a file stored on disk, whereas a **process** is an **active** entity, with a program counter specifying the next instruction to execute and a set of associated resources.

#### Process States:

- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:
  1. **New**: The process is being created.
  2. **Running**: Instructions are being executed.
  3. **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  4. **Ready**: The process is waiting to be assigned to a processor.
  5. **Terminated**: The process has finished execution.



**Process State Transition Diagram**

## CS8493 OPERATING SYSTEMS UNIT -II

### Process Control Block

- Each process is represented in the operating system by a process control block (PCB)- also called a task control block.
- A PCB defines a process to the operating system.
- It contains the entire information about a process.
- Some of the information a PCB contains are:
  1. **Process state:** The state may be new, ready, running, waiting or terminated.
  2. **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
  3. **CPU registers:** The registers vary in number and type, depending on the computer architecture.
  4. **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
  5. **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
  6. **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
  7. **Status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

**Process Control Block**

## CS8493 OPERATING SYSTEMS UNIT -II

### PROCESS SCHEDULING

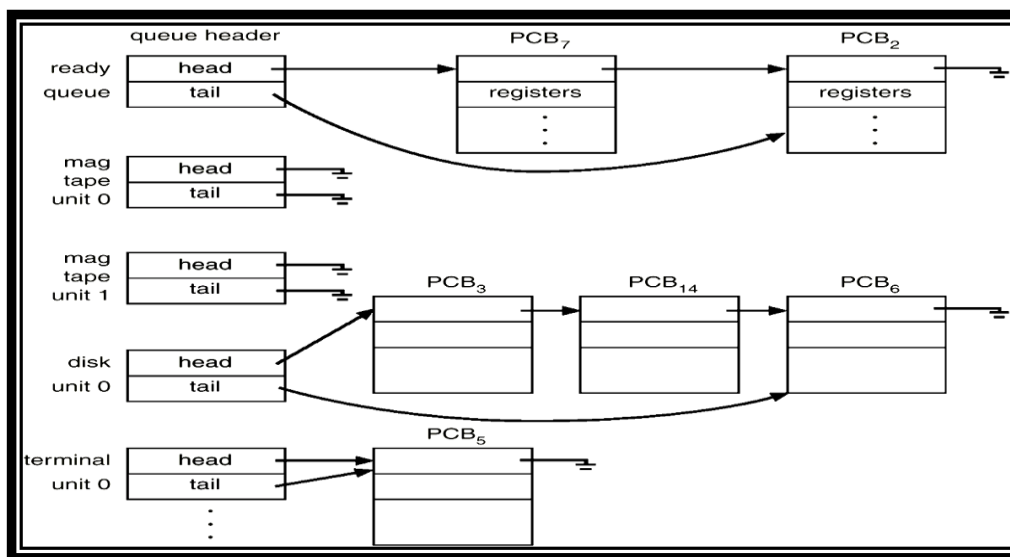
- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization.

#### Scheduling Queues

There are 3 types of scheduling queues .They are:

1. Job Queue
2. Ready Queue
3. Device Queue

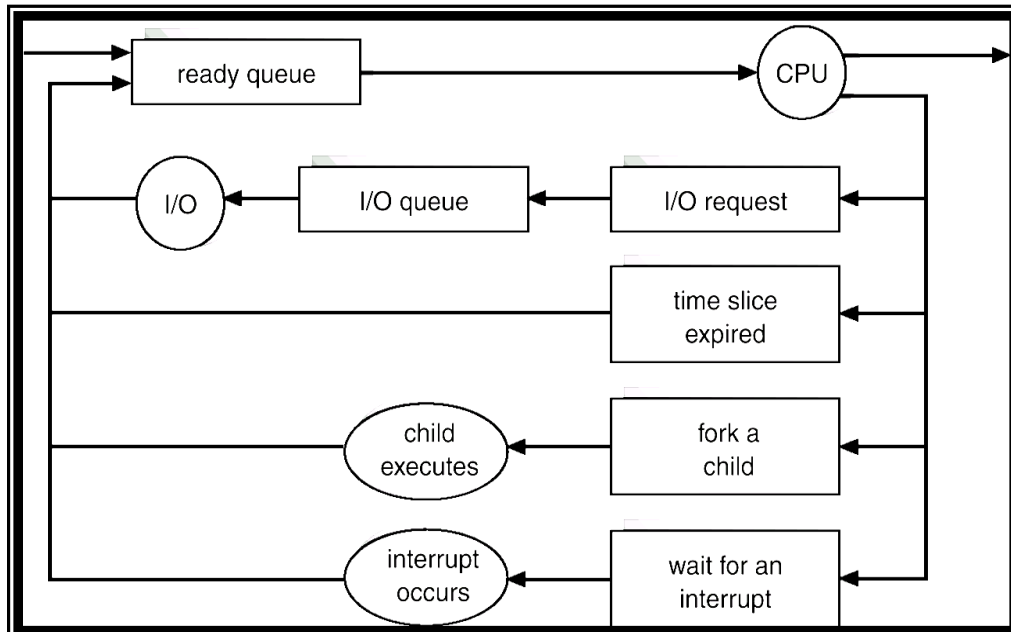
- As processes enter the system, they are put into a **job queue**.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- The list of processes waiting for an I/O device is kept in a **device queue** for that particular device.



**Various Scheduling Queues**

- A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or dispatched).
- Once the process is assigned to the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request, and then be placed in an I/O queue.
  - The process could create a new subprocess and wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready Queue.
- A common representation of process scheduling is a queuing diagram.

## CS8493 OPERATING SYSTEMS UNIT -II



**Queuing Diagram Representation of Process Scheduling**

### **Schedulers**

- A process migrates between the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.

There are three different types of schedulers. They are:

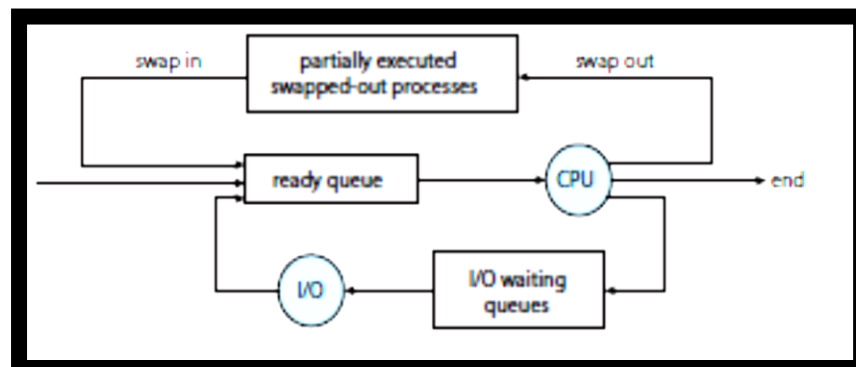
1. Long-term Scheduler or Job Scheduler
2. Short-term Scheduler or CPU Scheduler
3. Medium term Scheduler

- The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. It is invoked very infrequently. It controls the degree of multiprogramming.
- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute, and allocates the CPU to one of them. It is invoked very frequently.
- Processes can be described as either **I/O bound** or **CPU bound**.
- An **I/O-bound process** spends more of its time doing I/O than it spends doing computations.
- A **CPU-bound process**, on the other hand, generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses.
- The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

## CS8493 OPERATING SYSTEMS UNIT -II

### Medium term Scheduler

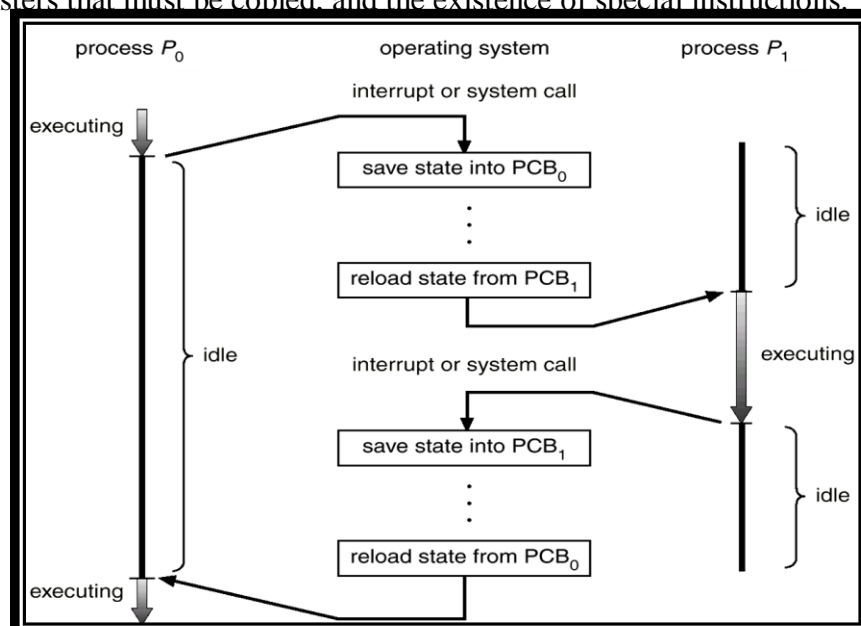
- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- The key idea is medium-term scheduler, removes processes from memory and thus reduces the degree of multiprogramming.
- At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called **swapping**.



**Addition of Medium term Scheduling to Queuing Diagram**

### Context Switch

- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.
- This task is known as a context switch.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.



## CS8493 OPERATING SYSTEMS UNIT -II

### OPERATIONS ON PROCESSES

#### 1. Process Creation

- A process may create several new processes, during the course of execution.
- The creating process is called a **parent** process, whereas the new processes are called the **children** of that process.
- When a process creates a new process, two possibilities exist **in terms of execution**:
  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of its children have terminated.
- There are also two possibilities **in terms of the address space** of the new process:
  1. The child process is a duplicate of the parent process.
  2. The child process has a program loaded into it.
- In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the **fork** system call.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

#### Creating a Process using fork() system call

#### 2. Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit** system call.
- At that point, the process may return data (output) to its parent process (via the **wait** system call).
- A process can cause the termination of another process via an appropriate system call.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

## CS8493 OPERATING SYSTEMS UNIT -II

1. The child has exceeded its usage of some of the resources that it has been allocated.
2. The task assigned to the child is no longer required.
3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

### Cooperating Processes

- The concurrent processes executing in the operating system may be either **independent** processes or **cooperating** processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.
- A process is cooperating if it can affect or be affected by the other processes executing in the system.
- **Benefits of Cooperating Processes**
  1. Information sharing
  2. Computation speedup
  3. Modularity
  4. Convenience

### Example

#### Producer – Consumer Problem

- A producer process produces information that is consumed by a consumer process.
- For example, a print program produces characters that are consumed by the printer driver. A compiler may produce assembly code, which is consumed by an assembler.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
  - **unbounded-buffer**: places no practical limit on the size of the buffer.
  - **bounded-buffer** : assumes that there is a fixed buffer size.

#### Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: **in** and **out**. The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer. The buffer is empty when **in == out** ; the buffer is full when **((in + 1) % BUFFERSIZE) == out**.

## CS8493 OPERATING SYSTEMS UNIT -II

### **Producer Process**

```
while (1)
{
    while (((in + 1) % BUFFER_SIZE) == out);
        /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

### **Consumer process**

```
while (1)
{
    while (in == out);
        /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

## **INTER-PROCESS COMMUNICATION**

- Operating systems provide the means for cooperating processes to communicate with each other via an inter process communication (IPC) facility.
- IPC provides a mechanism to allow processes to communicate and to synchronize their actions. IPC is best provided by a message passing system.

### **Basic Structure:**

- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.
- Physical implementation of the link is done through a hardware bus, network etc,
- There are several methods for logically implementing a link and the operations:
  1. **Direct or indirect communication**
  2. **Symmetric or asymmetric communication**
  3. **Automatic or explicit buffering**
  4. **Send by copy or send by reference**
  5. **Fixed-sized or variable-sized messages**

### **Naming**

- Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

#### 1. **Direct Communication**

- Each process that wants to communicate must explicitly name the recipient or sender of the communication.
- A communication link in this scheme has the following properties:
  - i. A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - ii. A link is associated with exactly two processes.



## CS8493 OPERATING SYSTEMS UNIT -II

iii. Exactly one link exists between each pair of processes.

- There are two ways of addressing namely
  - Symmetry in addressing
  - Asymmetry in addressing
- In symmetry in addressing, the send and receive primitives are defined as:  
send(P, message) → Send a message to process P  
receive(Q, message) → Receive a message from Q
- In asymmetry in addressing, the send & receive primitives are defined as:  
send(p, message) → send a message to process p  
receive(id, message) → receive message from any process, id is set to the name of the process with which communication has taken place

### 2. Indirect Communication

- With indirect communication, the messages are sent to and received from mailboxes, or ports.
- The send and receive primitives are defined as follows:  
send(A, message) → Send a message to mailbox A.  
receive(A, message) → Receive a message from mailbox A.  
A communication link has the following properties:
  - i. A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - ii. A link may be associated with more than two processes.
  - iii. A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox

### 3. Buffering

- A link has some capacity that determines the number of message that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link.
- There are three ways that such a queue can be implemented.
- **Zero capacity** : Queue length of maximum is 0. No message is waiting in a queue. The sender must wait until the recipient receives the message. (message system with no buffering)
- **Bounded capacity**: The queue has finite length n. Thus at most n messages can reside in it.
- **Unbounded capacity**: The queue has potentially infinite length. Thus any number of messages can wait in it. The sender is never delayed

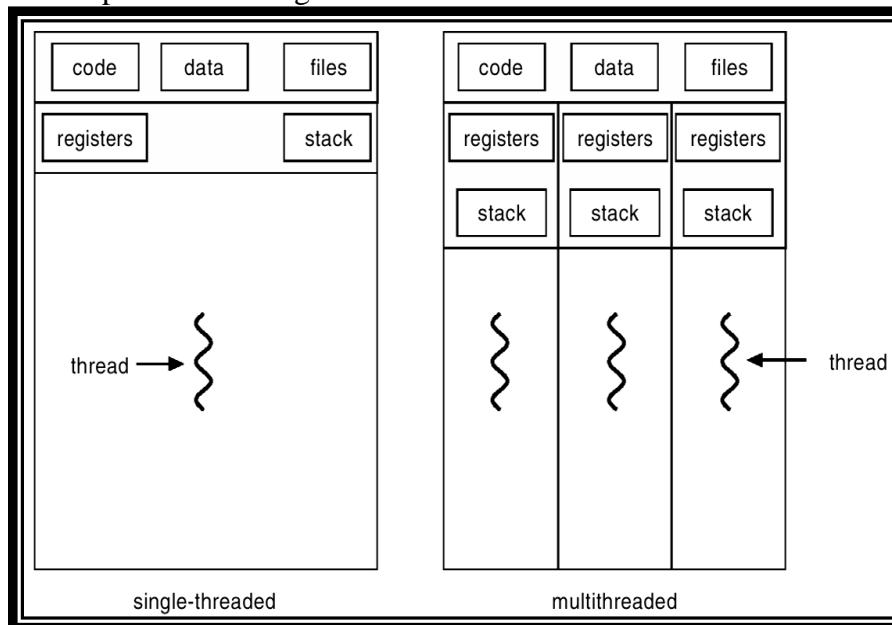
### 4. Synchronization

- Message passing may be either blocking or non-blocking.
  1. **Blocking Send** - The sender blocks itself till the message sent by it is received by the receiver.
  2. **Non-blocking Send** - The sender does not block itself after sending the message but continues with its normal operation.
  3. **Blocking Receive** - The receiver blocks itself until it receives the message.
  4. **Non-blocking Receive** – The receiver does not block itself.

## CS8493 OPERATING SYSTEMS UNIT -II

### THREADS - OVERVIEW

- A thread is the **basic unit of CPU utilization**.
- It is sometimes called as a **lightweight process**.
- It consists of a thread ID ,a program counter, a register set and a stack.
- It shares with other threads belonging to the same process its code section , data section, and resources such as open files and signals.



- A traditional or heavy weight process has a single thread of control.
- If the process has multiple threads of control, it can do more than one task at a time.

### Benefits of multithreaded programming

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

### User thread and Kernel threads

#### User threads

- Supported above the kernel and implemented by a thread library at the user level.
- Thread creation, management and scheduling are done in user space.
- Fast to create and manage
- When a user thread performs a blocking system call, it will cause the entire process to block even if other threads are available to run within the application.
- Example: POSIX Pthreads, Mach C-threads and Solaris 2 UI-threads.

#### Kernel threads

- Supported directly by the OS.
- Thread creation , management and scheduling are done in kernel space.
- Slow to create and manage

## CS8493 OPERATING SYSTEMS UNIT -II

- When a kernel thread performs a blocking system call ,the kernel schedules another thread in the application for execution.
- Example: Windows NT, Windows 2000 , Solaris 2,BeOS and Tru64 UNIX support kernel threads.

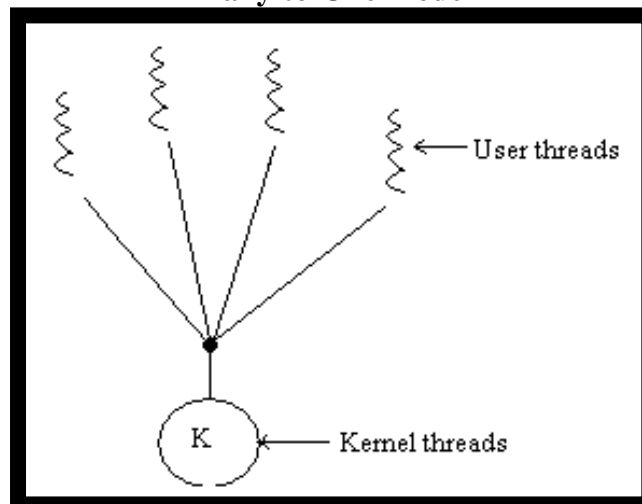
### MULTITHREADING MODELS

1. Many-to-One
2. One-to-One
3. Many-to-Many

#### 1. Many-to-One:

- Many user-level threads mapped to single kernel thread.
- Used on systems that does not support kernel threads.

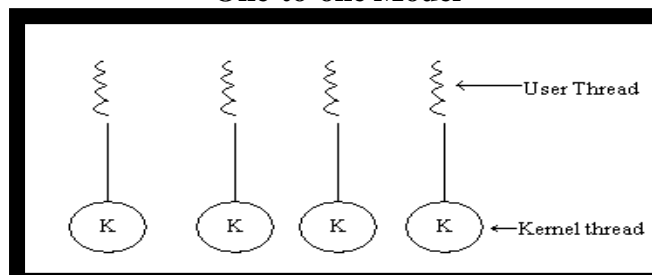
##### Many-to-One Model



#### 2. One-to-One:

- Each user-level thread maps to a kernel thread.
- Examples
  - Windows 95/98/NT/2000
  - OS/2

##### One-to-one Model

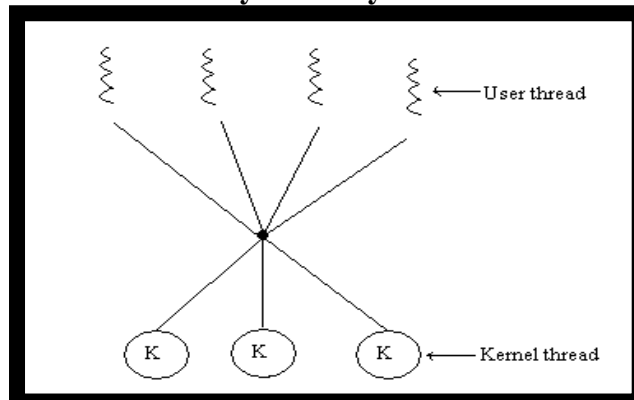


#### 3. Many-to-Many Model:

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Solaris 2
- Windows NT/2000

## CS8493 OPERATING SYSTEMS UNIT -II

### Many-to-Many Model



#### Threading Issues:

##### 1. fork() and exec() system calls.

A fork() system call may duplicate all threads or duplicate only the thread that invoked fork().

If a thread invoke exec() system call ,the program specified in the parameter to exec will replace the entire process.

##### 2. Thread cancellation.

It is the task of terminating a thread before it has completed .

A thread that is to be cancelled is called a target thread.

There are two types of cancellation namely

1. Asynchronous Cancellation – One thread immediately terminates the target thread.
2. Deferred Cancellation – The target thread can periodically check if it should terminate , and does so in an orderly fashion.

##### 3. Signal handling

1. A signal is a used to notify a process that a particular event has occurred.
2. A generated signal is delivered to the process.
  - a. Deliver the signal to the thread to which the signal applies.
  - b. Deliver the signal to every thread in the process.
  - c. Deliver the signal to certain threads in the process.
  - d. Assign a specific thread to receive all signals for the process.
3. Once delivered the signal must be handled.
  - a. Signal is handled by
    - i. A default signal handler
    - ii. A user defined signal handler

##### 4. Thread pools

Creation of unlimited threads exhausts system resources such as CPU time or memory. Hence we use a thread pool.

In a thread pool, a number of threads are created at process startup and placed in the pool.

When there is a need for a thread the process will pick a thread from the pool and assign it a task. After completion of the task, the thread is returned to the pool.

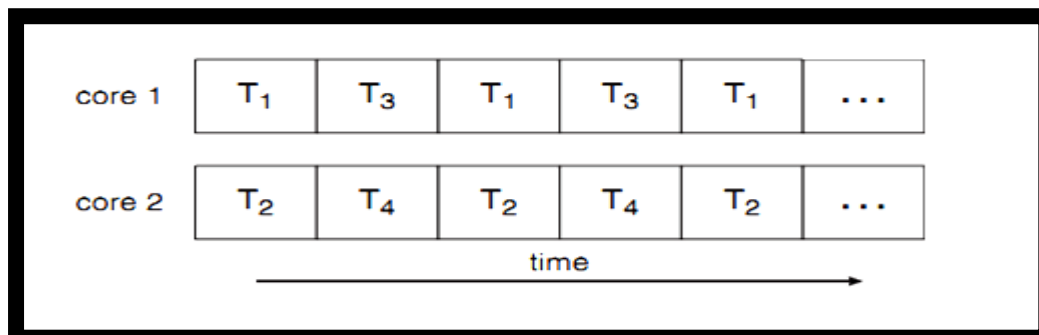
##### 5. Thread specific data

## CS8493 OPERATING SYSTEMS UNIT -II

Threads belonging to a process share the data of the process. However each thread might need its own copy of certain data known as thread-specific data.

### MULTI-CORE PROGRAMMING

- A concurrent system supports more than one task by allowing all the tasks to make progress.
- A system is parallel if it can perform more than one task simultaneously.
- Before the advent of SMP and multi-core architectures, most computer systems had only a single processor.
- CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes. Such processes were running concurrently but not in parallel.



#### Parallel execution on a multi-core system

- Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core.

#### Programming Challenges:

Multi-core architecture presents programming challenges to the system programmers and application programmers to make better use of the multiple computing cores. The challenges are

- 1. Identifying parallel tasks:** This involves examining applications to find areas that can be divided into separate, concurrent tasks.
- 2. Balance:** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
- 3. Data splitting:** The data accessed and manipulated by the tasks must be divided to run on separate cores.
- 4. Data dependency:** When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
- 5. Testing and debugging:** Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

#### Types of Parallelism

- In general, there are two types of parallelism: **Data parallelism** and **Task parallelism**.
- **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

## CS8493 OPERATING SYSTEMS UNIT -II

Example: In a dual core system, summing the contents of an array of size N .On a single-core system, one thread would simply sum the elements  $[0] \dots [N - 1]$ . On a dual-core system, however, thread A, running on core 0, could sum the elements  $[0] \dots [N/2 - 1]$  while thread B, running on core 1, could sum the elements  $[N/2] \dots [N - 1]$ . The two threads would be running in parallel on separate computing cores.

- **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

### WINDOWS 7 -THREAD AND SMP MANAGEMENT

#### Windows Threads:

- Windows implements the Windows API, which is the primary API for the family of Microsoft operating systems (Windows 98, NT, 2000, and XP, as well as Windows 7).
- A Windows application runs as a separate process, and each process may contain one or more threads.
- The general components of a thread include:
  1. A thread ID uniquely identifying the thread
  2. A register set representing the status of the processor
  3. A user stack, employed when the thread is running in user mode, and a
  4. A kernel stack, employed when the thread is running in kernel mode
  5. A private storage area used by various run-time libraries and dynamic link libraries (DLLs) The register set, stacks, and private storage area are known as the context of the thread.
- The primary data structures of a thread include:
  1. ETHREAD — Executive Thread Block
  2. KTHREAD — Kernel Thread Block
  3. TEB — Thread Environment Block
- The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.
- The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.
- The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-local storage.

### PROCESS SYNCHRONIZATION

## CS8493 OPERATING SYSTEMS UNIT -II

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem allows at most  $n - 1$  items in buffer at the same time. A solution, where all  $N$  buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and increment it each time a new item is added to the buffer
- **Race condition:** The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, **concurrent processes must be synchronized.**

### **Example:**

Consider the bounded buffer problem, where an integer variable **counter**, initialized to 0 is added. **counter** is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

The **code for the producer process** can be modified as follows:

```
while (true)
{
    /* produce an item in next produced */
    while (counter == BUFFER SIZE)
    ; /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
    counter++;
}
```

The **code for the consumer process** can be modified as follows:

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

Let the current value of counter be 5. If producer process and consumer process execute the statements counter++ and counter--concurrently then the value of counter may be 4,5 or 6 which is incorrect. To explain this further, counter ++ may be implemented in machine language as follows:

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

and counter - - may be implemented as follows:

## CS8493 OPERATING SYSTEMS UNIT -II

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

The concurrent execution of counter ++ and counter - - is equivalent to a sequential execution of the statement are interleaved in some arbitrary order. One such interleaving is given below:

```
T0: producer execute register1 = counter {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = counter {register2 = 5}
T3: consumer execute register2 = register2 - 1 {register2 = 4}
T4: producer execute counter = register1 {counter = 6}
T5: consumer execute counter = register2 {counter = 4}
```

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**.

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.

### **THE CRITICAL-SECTION PROBLEM**

- A critical section is the portion of a program that accesses shared data.
- Each process has a code segment, called critical section, in which the shared data is accessed.
- To ensure synchronization, no two processes should be allowed to execute in their critical sections at the same time.

#### **Requirements to be satisfied for a Solution to the Critical-Section Problem:**

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

#### **General structure of process $P_i$**

do {

entry section

critical section

exit section



## CS8493 OPERATING SYSTEMS UNIT -II

remainder section

```
} while (1);
```

- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **non-preemptive kernels**.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.
- Obviously, a non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.
- We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.
- Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

### **MUTEX LOCKS**

- Mutex (Mutual Exclusion) lock is a simple software tool that solves the critical section problem.
- The mutex lock is used to protect critical regions and thus prevent race conditions.
- 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.
- 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;;  
}
```

**Solution to the critical-section problem using mutex locks.**

```
do {  
    acquire lock  
    critical section
```

## CS8493 OPERATING SYSTEMS UNIT -II

```
    release lock
    remainder section
} while (true);
```

**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**.
- mutex lock is also called a **spinlock** because the process “spins” while waiting for the lock to become available.
- Advantage of Spinlocks is that no context switch is required when a process must wait on a lock.
- When locks are expected to be held for short times, spinlocks are useful.

### **SEMAPHORES**

- It is a synchronization tool that is used to generalize the solution to the critical section problem.
- A Semaphore S is an integer variable that can only be accessed via two indivisible (atomic) operations namely

1. wait or P operation ( to test )
2. signal or V operation ( to increment )

```
wait (s)
{
    while(s<=0);
    s--;
}
signal (s)
{
    s++;
}
```

### **Mutual Exclusion Implementation using semaphore**

```
do
{
    wait(mutex);
    critical section
    signal(mutex)
```

## CS8493 OPERATING SYSTEMS UNIT -II

remainder section

} while (1);

### **Semaphore Implementation**

- The semaphore discussed so far requires a busy waiting. That is if a process is in critical-section, the other process that tries to enter its critical-section must loop continuously in the entry code.
- To overcome the busy waiting problem, the definition of the semaphore operations wait and signal should be modified.
- When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block operation places the process into a waiting queue associated with the semaphore.
- A process that is blocked waiting on a semaphore should be restarted when some other process executes a signal operation. The blocked process should be restarted by a wakeup operation which put that process into ready queue.
- To implemented the semaphore, we define a semaphore as a record as:

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

- Assume two simple operations:
  1. **block** suspends the process that invokes it.
  2. **wakeup(P)** resumes the execution of a blocked process **P**.
- Semaphore operations now defined as

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

```
signal(S)
{
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
}
```

### **Deadlock & starvation:**

Example: Consider a system of two processes, P0 & P1 each accessing two semaphores, S & Q, set to the value 1.

<b>P0</b>	<b>P1</b>
Wait (S)	Wait (Q)

## CS8493 OPERATING SYSTEMS UNIT -II

Wait (Q)	Wait (S)
·	·
·	·
·	·
Signal(S)	Signal(Q)
Signal(Q)	Signal(S)

- Suppose that P0 executes wait(S), then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal operations cannot be executed, P0 & P1 are **deadlocked**.
- Another problem related to deadlock is **indefinite blocking or starvation**, a situation where a process wait indefinitely within the semaphore. Indefinite blocking may occur if we add or remove processes from the list associated with a semaphore in LIFO order.

### Types of Semaphores

- *Counting* semaphore – any positive integer value
- *Binary* semaphore – integer value can range only between 0 and 1

### MONITORS

- A monitor is a synchronization construct that supports mutual exclusion and the ability to wait /block until a certain condition becomes true.
- A monitor is an abstract datatype that encapsulates data with a set of functions to operate on the data.

### Characteristics of Monitor

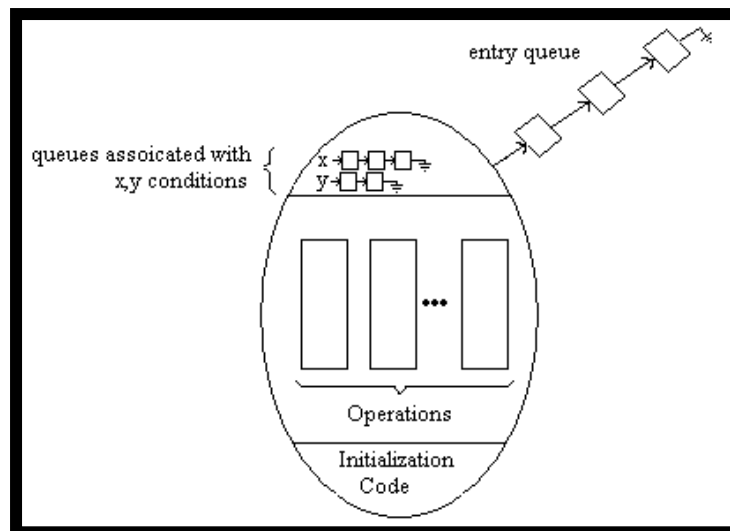
- The local variables of a monitor can be accessed only by the local functions.
- A function defined within a monitor can only access the local variables of a monitor and its formal parameter.
- Only one process may be active within the monitor at a time.

### Syntax of a Monitor

```
monitor monitor-name
{
// shared variable declarations
function P1 (...) { .... }
...
function Pn (...) {.....}
initialization code (...) {
}
}
```

## CS8493 OPERATING SYSTEMS UNIT -II

### Schematic view of a monitor



### Monitor with condition variables

- Instead of lock-based protection, monitors use a shared condition variable for synchronization and only two operations wait() and signal() can be applied on the condition variable.

```
condition x, y;  
x.wait (); // a process that invokes the operation is suspended.  
x.signal (); //resumes one of the suspended processes(if any)
```

### The 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. 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.

## CS8493 OPERATING SYSTEMS UNIT -II

### Solution to Dining Philosophers Problem

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

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((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;
    }
}
```

- Each philosopher, before starting to eat, must invoke the operation pickup() followed by eating and finally invoke putdown().
- This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. However, with this solution it is possible for a philosopher to starve to death.

#### **Implementing a Monitor using a semaphore**

- For each condition variable x, we introduce a semaphore x\_sem and an integer variable x\_count, both initialized to 0.
- The operation x.wait() is implemented as:

## CS8493 OPERATING SYSTEMS UNIT -II

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

- The operation x.signal() is 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, then on resuming we have to determine which process is to be resumed.
- One solution is to use FCFS ordering.
- For priority based scheme, the conditional wait construct is x.wait( c ) where c is the priority number

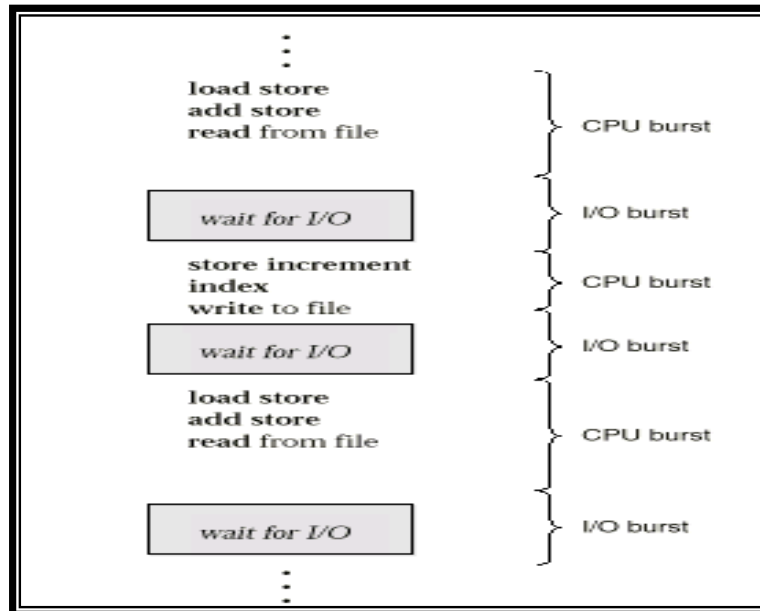
### CPU SCHEDULING

- CPU scheduling is the basis of multi-programmed operating systems.
- The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization.
- Scheduling is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.

### CPU-I/O Burst Cycle

- Process execution consists of a **cycle** of CPU execution and I/O wait.
- Processes alternate between these two states.
- Process execution begins with a **CPU burst**.
- That is followed by an **I/O burst**, then another CPU burst, then another I/O burst, and so on.
- Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.

## CS8493 OPERATING SYSTEMS UNIT -II



### CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **short-term scheduler** (or CPU scheduler).
- The ready queue is not necessarily a first-in, first-out (FIFO) queue. It may be a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

### Preemptive Scheduling

- CPU scheduling decisions may take place under the following four circumstances:
  1. When a process switches from the running state to the waiting state
  2. When a process switches from the running state to the ready state
  3. When a process switches from the waiting state to the ready state
  4. When a process terminates
- Under 1 & 4 scheduling scheme is non preemptive.
- Otherwise the scheduling scheme is preemptive.

### Non-preemptive Scheduling

- In non preemptive scheduling, once the CPU has been allocated a process, the process keeps the CPU until it releases the CPU either by termination or by switching to the waiting state.
- This scheduling method is used by the Microsoft windows environment.

### Dispatcher

- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.



## CS8493 OPERATING SYSTEMS UNIT -II

- This function involves:
  1. Switching context
  2. Switching to user mode
  3. Jumping to the proper location in the user program to restart that program

### **Scheduling Criteria**

1. **CPU utilization:** The CPU should be kept as busy as possible. CPU utilization may 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:** It is the number of processes completed per time unit. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.
3. **Turnaround time:** 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/O.
4. **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.
5. **Response time:** It is the amount of time it takes to start responding, but not the time that it takes to output that response.

### **CPU Scheduling Algorithms**

1. First-Come, First-Served Scheduling
2. Shortest Job First Scheduling
3. Priority Scheduling
4. Round Robin Scheduling

### **First-Come, First-Served Scheduling**

- The process that requests the CPU first is allocated the CPU first.
- It is a non-preemptive scheduling technique.
- The implementation of the FCFS policy is easily managed with a FIFO queue.

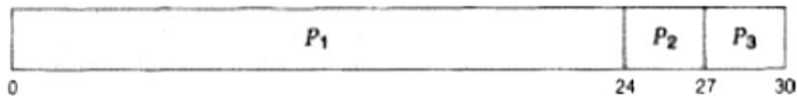
### **Example:**

Process	Burst Time
P1	24
P2	3
P3	3

## CS8493 OPERATING SYSTEMS UNIT -II

- If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:

Gantt Chart



Average waiting time =  $(0+24+27) / 3 = 17$  ms

Average Turnaround time =  $(24+27+30) / 3 = 27$  ms

- The FCFS algorithm is particularly troublesome for time – sharing systems, where it is important that each user get a share of the CPU at regular intervals.

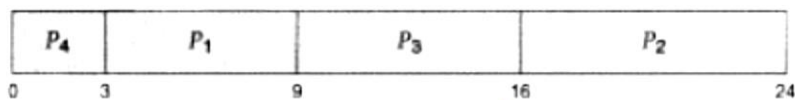
### Shortest Job First Scheduling

- It is a non-preemptive scheduling technique.
- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

**Example:**

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



Average waiting time is  $(3 + 16 + 9 + 0)/4 = 7$  ms

Average turnaround time =  $(3+9+16+24) / 4 = 13$  ms

### Shortest Remaining Time First Scheduling

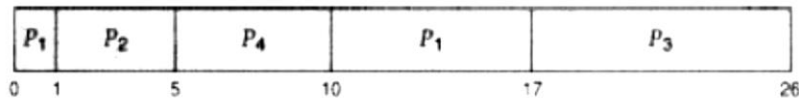
- It is a preemptive scheduling technique.
- Preemptive SJF is known as shortest remaining time first

**Example :**

Process	Arrival Time	Burst Time
P1	0	8

## CS8493 OPERATING SYSTEMS UNIT -II

P2	1	4
P3	2	9
P4	3	5



Waiting Time Calculation

$$P1 : 10 - 1 = 9$$

$$P2 : 1 - 1 = 0$$

$$P3 : 17 - 2 = 15$$

$$P4 : 5 - 3 = 2$$

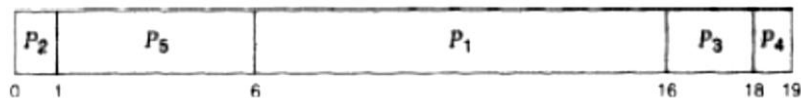
$$\text{Average waiting time} = (9+0+15+2) / 4 = 6.5 \text{ ms}$$

### Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.( smallest integer  $\equiv$  highest priority).

#### Example :

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



$$\text{Average waiting time} = 8.2 \text{ ms}$$

- Priority Scheduling can be preemptive or non-preemptive.
- **Drawback**  $\rightarrow$  Starvation – low priority processes may never execute.
- **Solution**  $\rightarrow$  Aging – It is a technique of gradually increasing the priority of processes that wait in the system for a long time.

### Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum (or time slice), is defined.
- The ready queue is treated as a circular queue.

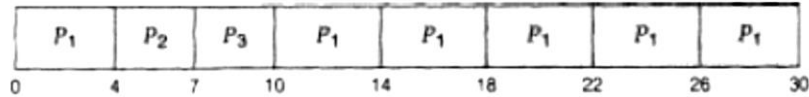
#### Example:

Process	Burst Time
P1	24
P2	3
P3	3

## CS8493 OPERATING SYSTEMS UNIT -II

Time Quantum = 4 ms.

Gantt chart



Waiting time

$$P_1 = 26 - 20 = 6$$

$$P_2 = 4$$

$$P_3 = 7 (6+4+7 / 3 = 5.66 \text{ ms})$$

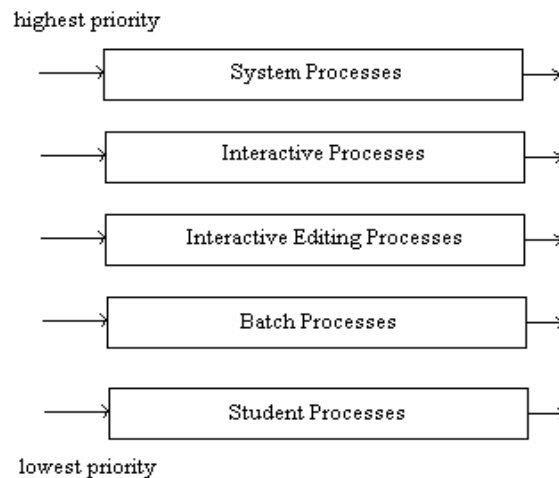
Average waiting time is  $17/3 = 5.66$  milliseconds.

- The performance of the RR algorithm depends heavily on the size of the time-quantum.
- If time-quantum is very large (infinite) then RR policy is same as FCFS policy.
- If time quantum is very small, RR approach is called processor sharing and appears to the users as though each of n process has its own processor running at 1/n the speed of real processor.

### **Multilevel Queue Scheduling**

- It partitions the ready queue into several separate queues .
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- There must be scheduling between the queues, which is commonly implemented as a fixed-priority preemptive scheduling.
- For example the foreground queue may have absolute priority over the background queue.
- Example of a multilevel queue scheduling algorithm with five queues
  1. System processes
  2. Interactive processes
  3. Interactive editing processes
  4. Batch processes
  5. Student processes
- Each queue has absolute priority over lower-priority queue.

## CS8493 OPERATING SYSTEMS UNIT -II

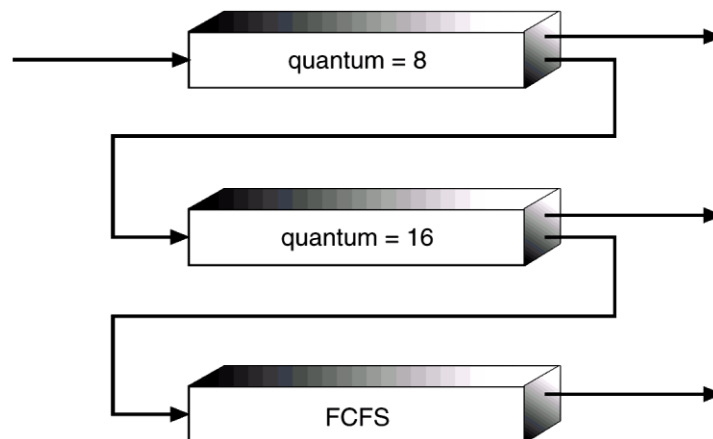


### **Multilevel Feedback Queue Scheduling**

- It allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- 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.
- Similarly, a process that waits too long in a lower priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.

### **Example:**

- Consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2.
- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process that arrives for queue 0 will, in turn, preempt a process in queue 1.



- A multilevel feedback queue scheduler is defined by the following parameters:
  1. The number of queues

## CS8493 OPERATING SYSTEMS UNIT -II

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

### DEADLOCKS

**Definition:** A process requests resources. If the resources are not available at that time, the process enters a wait state. Waiting processes may never change state again because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

A process must request a resource before using it, and must release resource after using it.

1. **Request:** If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource
3. **Release:** The process releases the resource.

### Deadlock Characterization

#### Four Necessary conditions for a deadlock

1. **Mutual exclusion:** At least one resource must be held in a non sharable mode. That is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted.
4. **Circular wait:**  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2 \dots P_{n-1}$ .

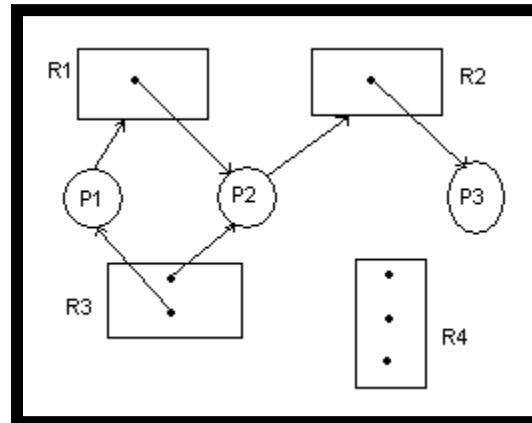
#### Resource-Allocation Graph

- It is a directed graph with a set of vertices  $V$  and a set of edges  $E$ .
  - $V$  is partitioned into two types:
    1. nodes  $P = \{p_1, p_2, \dots, p_n\}$
    2. Resource type  $R = \{R_1, R_2, \dots, R_m\}$
  - $P_i \rightarrow R_j$  - request  $\Rightarrow$  request edge
  - $R_j \rightarrow P_i$  - allocated  $\Rightarrow$  assignment edge.
  - $P_i$  is denoted as a circle and  $R_j$  as a square.
  - $R_j$  may have more than one instance represented as a dot within the square.
- Sets  $P, R$  and  $E$ .  
 $P = \{ P_1, P_2, P_3 \}$   
 $R = \{ R_1, R_2, R_3, R_4 \}$   
 $E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3 \}$

## CS8493 OPERATING SYSTEMS UNIT -II

- Resource instances

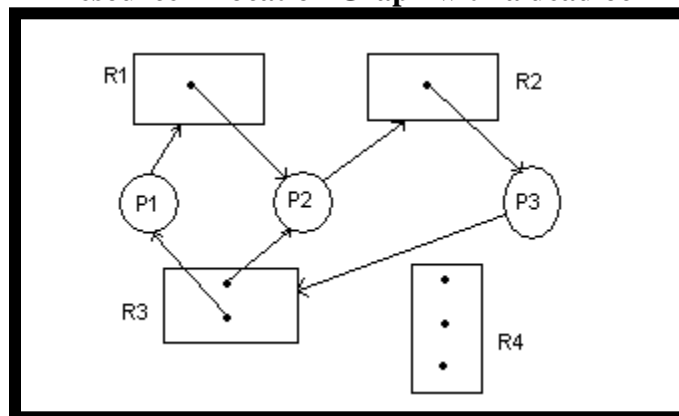
One instance of resource type R1, Two instance of resource type R2, One instance of resource type R3, Three instances of resource type R4.



### Process states

Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.

### **Resource Allocation Graph with a deadlock**



Process P2 is holding an instance of R1 and R2 and is waiting for an instance of resource type R3. Process P3 is holding an instance of R3.

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

### **Methods for handling Deadlocks**

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

### **Deadlock Prevention:**

- This ensures that the system never enters the deadlock state.
- Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold.
- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

## CS8493 OPERATING SYSTEMS UNIT -II

### 1. Denying Mutual exclusion

- Mutual exclusion condition must hold for non-sharable resources.
- Printer cannot be shared simultaneously shared by prevent processes.
- Sharable resource - example Read-only files.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.

### 2. Denying Hold and wait

- Whenever a process requests a resource, it does not hold any other resource.
- One technique that can be used requires each process to request and be allocated all its resources before it begins execution.
- Another technique is before it can request any additional resources, it must release all the resources that it is currently allocated.
- These techniques have two main **disadvantages** :
  - First, resource utilization may be low, since many of the resources may be allocated but unused for a long time.
  - We must request all resources at the beginning for both protocols. starvation is possible.

### 3. Denying No preemption

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it. (that is the process must wait), then all resources currently being held are preempted.(**ALLOW PREEMPTION**)
- These resources are implicitly released.
- The process will be restarted only when it can regain its old resources.

### 4. Denying Circular wait

- Impose a total ordering of all resource types and allow each process to request for resources in an increasing order of enumeration.
- Let  $R = \{R_1, R_2, \dots, R_m\}$  be the set of resource types.
- Assign to each resource type a unique integer number.
- If the set of resource types  $R$  includes tapedrives, disk drives and printers.
  - $F(\text{tapedrive})=1,$
  - $F(\text{diskdrive})=5,$
  - $F(\text{Printer})=12.$
- Each process can request resources only in an increasing order of enumeration.

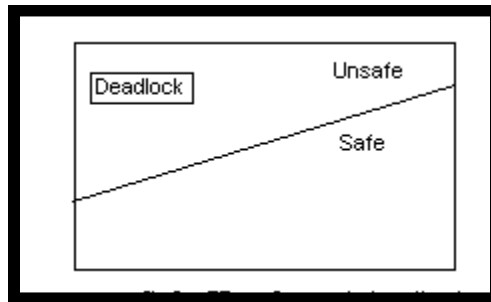
### Deadlock Avoidance:

- Deadlock avoidance request that the OS be given in advance additional information concerning which resources a process will request and use during its life time. With this information it can be decided for each request whether or not the process should wait.
- To decide whether the current request can be satisfied or must be delayed, a system must consider the resources currently available, the resources currently allocated to each process and future requests and releases of each process.
- **Safe State**  
A state is safe if the system can allocate resources to each process in some order and still



## CS8493 OPERATING SYSTEMS UNIT -II

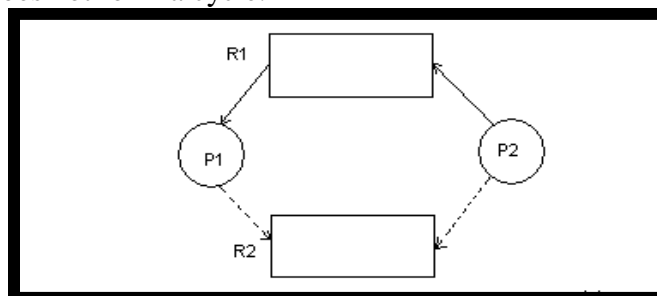
avoid a dead lock.



- A deadlock is an unsafe state.
- Not all unsafe states are dead locks
- An unsafe state may lead to a dead lock
- Two algorithms are used for deadlock avoidance namely;
  1. Resource Allocation Graph Algorithm - single instance of a resource type.
  2. Banker's Algorithm – several instances of a resource type.

### **Resource allocation graph algorithm**

- **Claim edge** - Claim edge  $P_i \dashrightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$  at some time, represented by a dashed directed edge.
- When process  $P_i$  request resource  $R_j$ , the claim edge  $P_i \dashrightarrow R_j$  is converted to a request edge.
- Similarly, when a resource  $R_j$  is released by  $P_i$  the assignment edge  $R_j \rightarrow P_i$  is reconverted to a claim edge  $P_i \dashrightarrow R_j$
- The request can be granted only if converting the request edge  $P_i \dashrightarrow R_j$  to an assignment edge  $R_j \rightarrow P_i$  does not form a cycle.



- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state.

### **Banker's algorithm**

- **Available:** indicates the number of available resources of each type.
- **Max:**  $\text{Max}[i, j]=k$  then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
- **Allocation :**  $\text{Allocation}[i, j]=k$ , then process  $P_i$  is currently allocated  $K$  instances of resource type  $R_j$
- **Need :** if  $\text{Need}[i, j]=k$  then process  $P_i$  may need  $K$  more instances of resource type  $R_j$

$$\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$$

### **Safety algorithm**

## CS8493 OPERATING SYSTEMS UNIT -II

1. Initialize work := available and Finish [i]:=false for i=1,2,3 .. n
2. Find an i such that both
  - a. Finish[i]=false
  - b. Need<sub>i</sub> ≤ Workif no such i exists, goto step 4
3. work :=work+ allocation;  
Finish[i]:=true  
goto step 2
4. If finish[i]=true for all i, then the system is in a safe state

### Resource Request Algorithm

Let Request<sub>i</sub> be the request from process P<sub>i</sub> for resources.

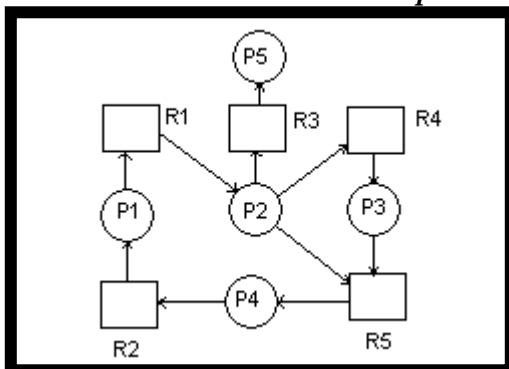
1. If Request<sub>i</sub> ≤ Need<sub>i</sub> goto step2, otherwise raise an error condition, since the process has exceeded its maximum claim.
2. If Request<sub>i</sub> ≤ Available, goto step3, otherwise P<sub>i</sub> must wait, since the resources are not available.
3. Available := Available-Request<sub>i</sub>;  
Allocation<sub>i</sub> := Allocation<sub>i</sub> + Request<sub>i</sub>;  
Need<sub>i</sub> := Need<sub>i</sub> - Request<sub>i</sub>;
  - Now apply the safety algorithm to check whether this new state is safe or not.
  - If it is safe then the request from process P<sub>i</sub> can be granted.

### **Deadlock detection**

#### **(i) Single instance of each resource type**

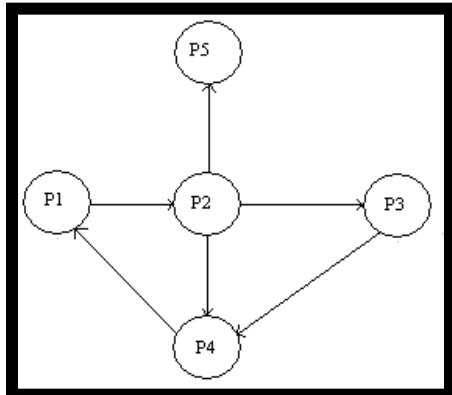
- If all resources have only a single instance, then we can define a deadlock detection algorithm that use a variant of resource-allocation graph called a wait for graph.

#### *Resource Allocation Graph*



#### *Wait for Graph*

## CS8493 OPERATING SYSTEMS UNIT -II



### (ii) Several Instances of a resource type

**Available** : Number of available resources of each type

**Allocation** : number of resources of each type currently allocated to each process

**Request** : Current request of each process

If Request  $[i,j]=k$ , then process  $P_i$  is requesting  $K$  more instances of resource type  $R_j$ .

1. Initialize work := available  
Finish[i]=false, otherwise finish [i]:=true
2. Find an index i such that both
  - a. Finish[i]=false
  - b. Request<sub>i</sub>≤workif no such i exists go to step4.
3. Work:=work+allocation<sub>i</sub>  
Finish[i]:=true  
goto step2
4. If finish[i]=false  
then process  $P_i$  is deadlocked

### Deadlock Recovery

#### 1. Process Termination

1. Abort all deadlocked processes.
2. Abort one deadlocked process at a time until the deadlock cycle is eliminated.

After each process is aborted , a deadlock detection algorithm must be invoked to determine where any process is still dead locked.

#### 2. Resource Preemption

Preemptive some resources from process and give these resources to other processes until the deadlock cycle is broken.

- i. **Selecting a victim**: which resources and which process are to be preempted.
- ii. **Rollback**: if we preempt a resource from a process it cannot continue with its normal execution. It is missing some needed resource. we must rollback the process to some safe state, and restart it from that state.
- iii. **Starvation** : How can we guarantee that resources will not always be preempted from the same process.