Syllabus

KCS-401: Operating System

B.TECH. (COMPUTER SCIENCE AND ENGINEERING)

FOURTH SEMESTER (DETAILED SYLLABUS)

Course Outcome (CO) Bloom's Knowledge Level (KL)

At the end of course, the student will be able to understand

CO 1: Understand the structure and functions of OS	K1, K2
CO 2: Learn about Processes, Threads and Scheduling algorithms.	K1, K2
CO 3: Understand the principles of concurrency and Deadlocks	K2
CO 4: Learn various memory management scheme	K2
CO 5: Study I/O management and File systems.	K2,K4

Unit-I

Introduction : Operating system and functions, Classification of Operating systems- Batch, Interactive, Time sharing, Real Time System, Multiprocessor Systems, Multiuser Systems, Multiprocess Systems, Multithreaded Systems, Operating System Structure- Layered structure, System Components, Operating System services, Reentrant Kernels, Monolithic and Microkernel Systems.

Unit-II

Concurrent Processes: Process Concept, Principle of Concurrency, Producer / Consumer Problem, Mutual Exclusion, Critical Section Problem, Dekker's solution, Peterson's solution, Semaphores, Test and Set operation; Classical Problem in Concurrency- Dining Philosopher Problem, Sleeping Barber Problem; Inter Process Communication models and Schemes, Process generation.

Unit-III

CPU Scheduling: Scheduling Concepts, Performance Criteria, Process States, Process Transition Diagram, Schedulers, Process Control Block (PCB), Process address space, Process identification information, Threads and their management, Scheduling Algorithms, Multiprocessor Scheduling. Deadlock: System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock.

Unit-IV

Memory Management: Basic bare machine, Resident monitor, Multiprogramming with fixed partitions, Multiprogramming with variable partitions, Protection schemes, Paging, Segmentation, Paged segmentation, Virtual memory concepts, Demand paging, Performance of demand paging, Page replacement algorithms, Thrashing, Cache memory organization, Locality of reference.

Unit-V

I/O Management and Disk Scheduling: I/O devices, and I/O subsystems, I/O buffering, Disk storage and disk scheduling, RAID. File System: File concept, File organization and access mechanism, File directories, and File sharing, File system implementation issues, File system protection and security.

Text Book:

T1: Silberschatz, Galvin and Gagne, **"Operating Systems Concepts"**, Wiley

References:

R1: Andrew S. Tanenbaum **"Modern Operating System "**Pearson Education

R2: D M Dhamdhere, "**Operating Systems : A Concept based Approach**", 2nd Edition,TMH

R3: William Stallings, "**Operating Systems: Internals and Design Principles**", 6th Edition, Pearson Education

R4: Harvey M Dietel, " **An Introduction to Operating System**", Pearson Education

R5: Sibsankar Halder and Alex A Aravind, "**Operating Systems**", Pearson Education

Unit – III

CPU Scheduling: Scheduling Concepts, Performance Criteria, Process States, Process Transition Diagram, Schedulers, Process Control Block (PCB), Process address space, Process identification information, Threads and their management, Scheduling Algorithms, Multiprocessor Scheduling. **Deadlock:** System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock.

Process Concept

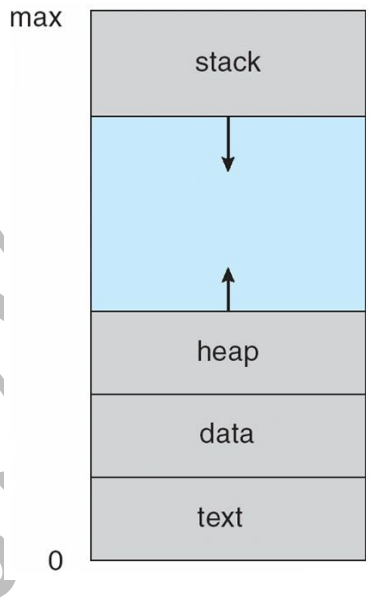
- An operating system executes a variety of programs:
- Batch system – jobs
- Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

A process includes:

- program counter
- stack
- data section

Process in Memory

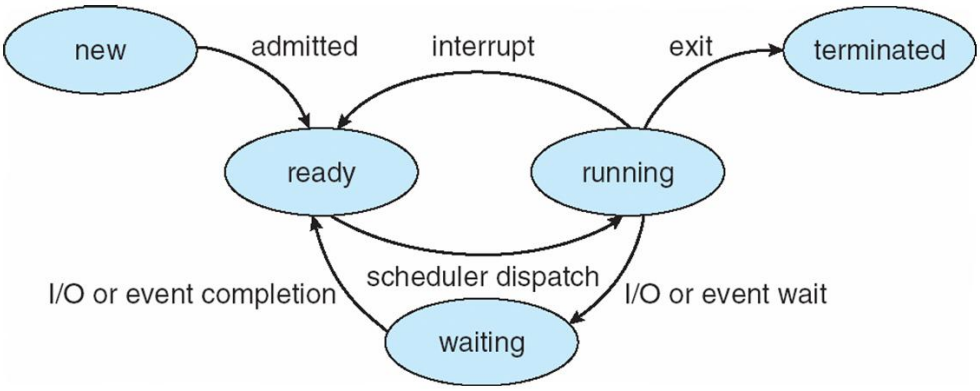


Process State

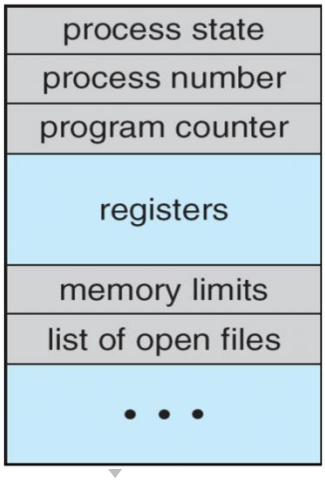
As a process executes, it changes *stat*e

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Diagram of Process State



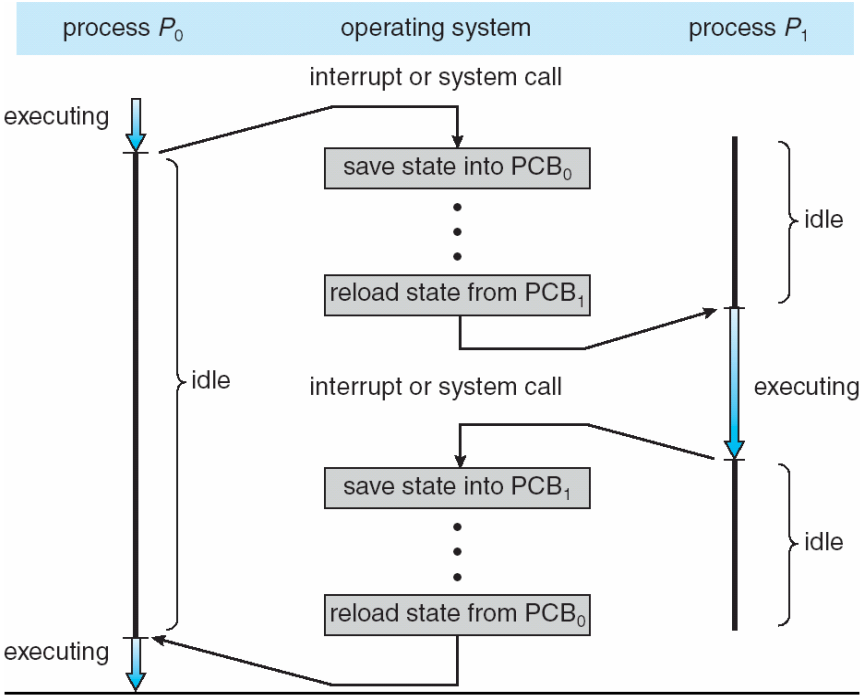
Process Control Block (PCB)



Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

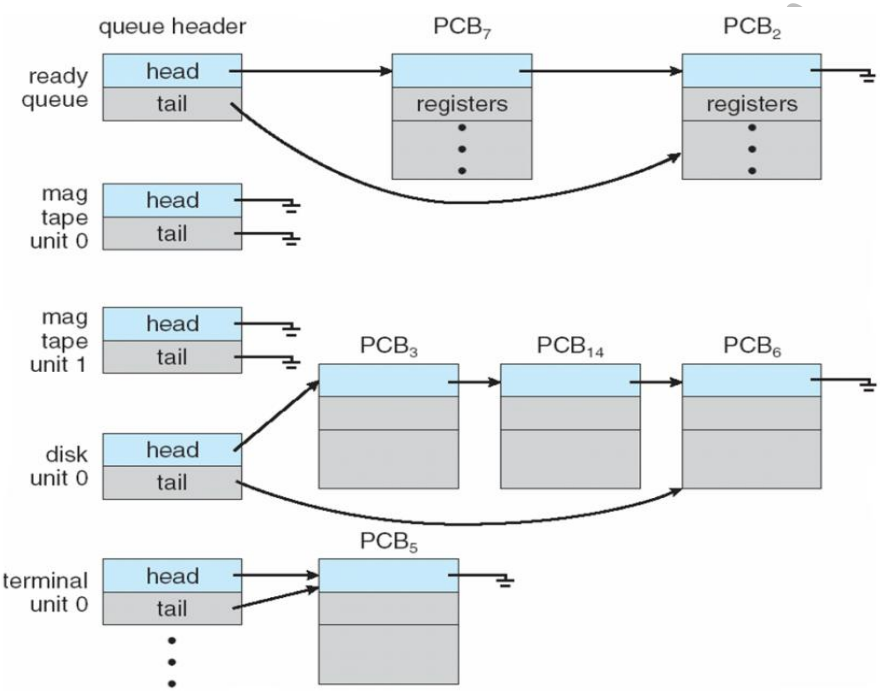
CPU Switch from Process to Process



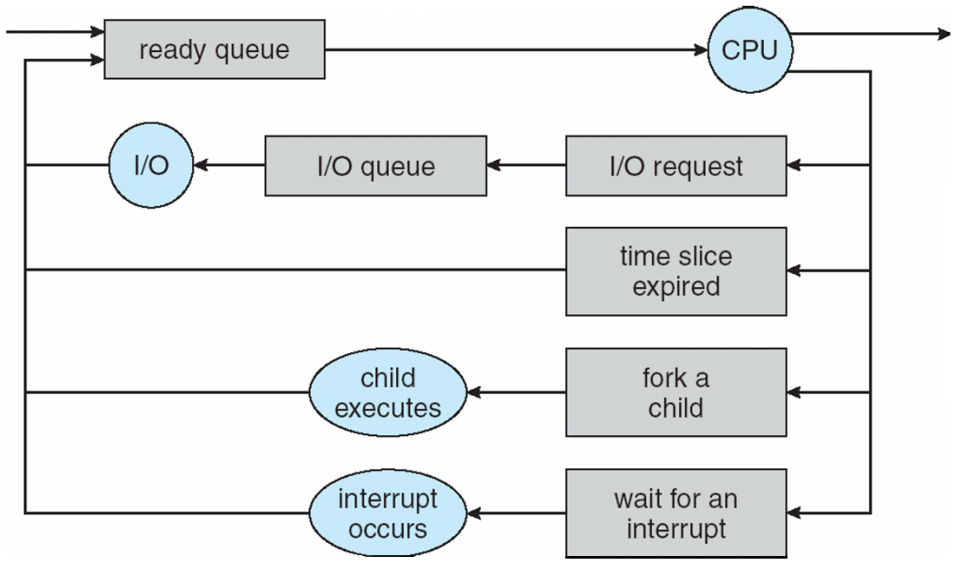
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



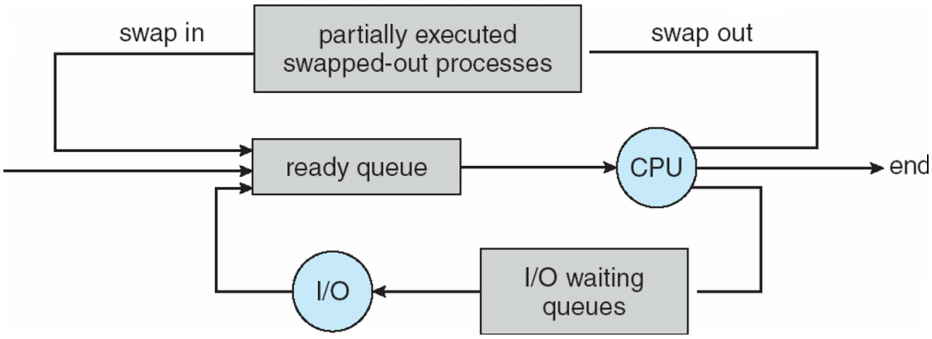
Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling



- Short-term scheduler is invoked very frequently (milliseconds) Þ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) Þ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Context Switch

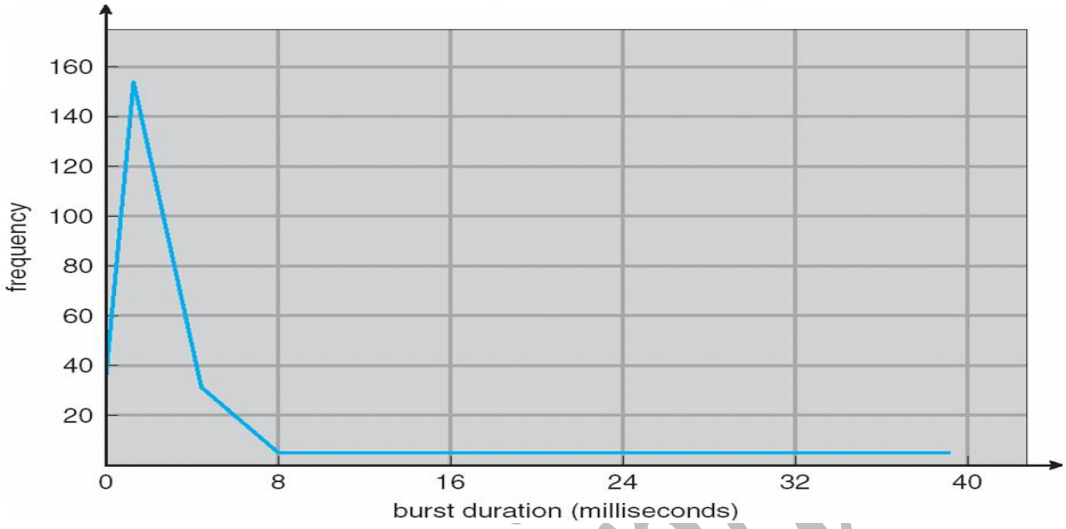
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support
- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

CPU Scheduling

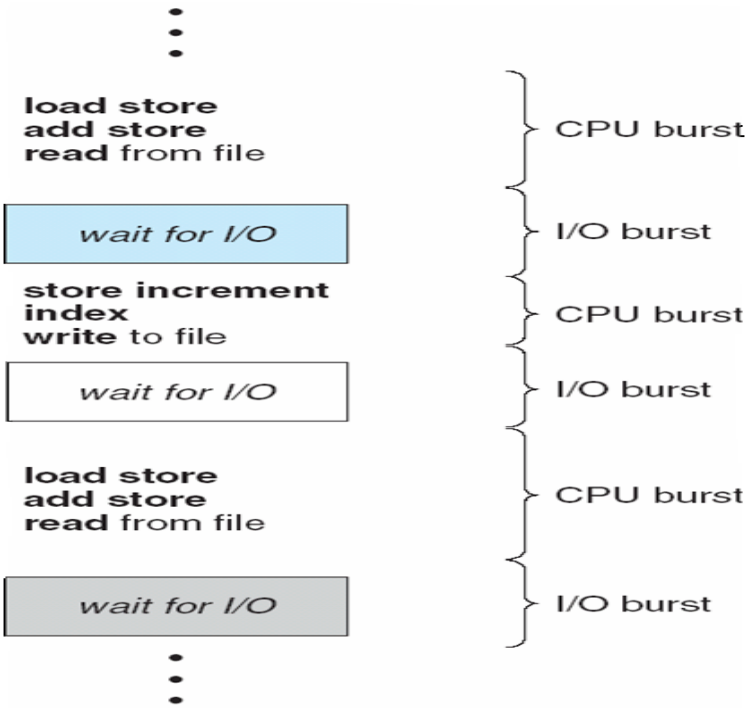
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Scheduling under 1 and 4 is **nonpreemptive**

All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

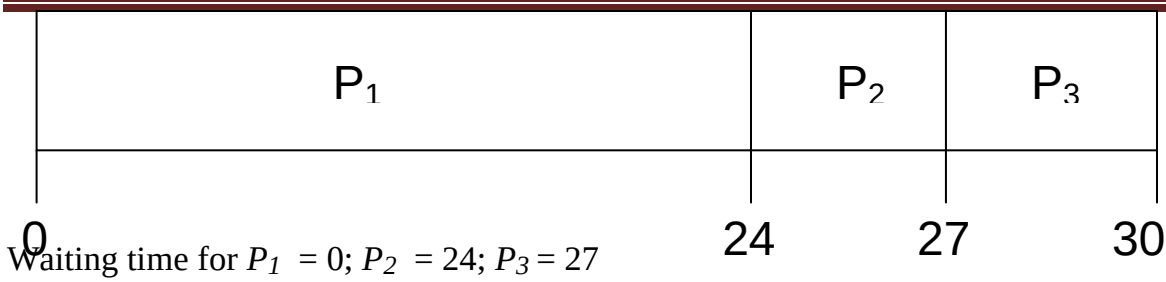
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3

The Gantt Chart for the schedule is:



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

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

Suppose that the processes arrive in the order P_2 , P_3 , P_1

The Gantt chart for the schedule is:

Waiting time for P_1 = 6; P_2 = 0; P_3 = 3

Average waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case

Convoy effect short process behind long process

Shortest-Job-First (SJF) Scheduling

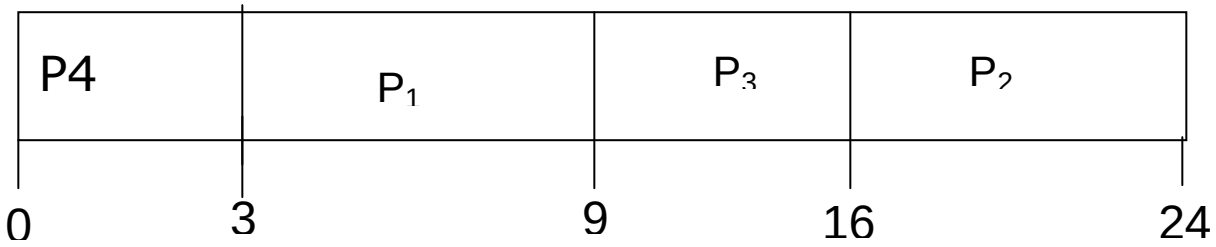
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

The difficulty is knowing

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

SJF scheduling chart

average waiting time = (3 + 16 + 9 + 0) / 4 = 7the length of the next CPU request

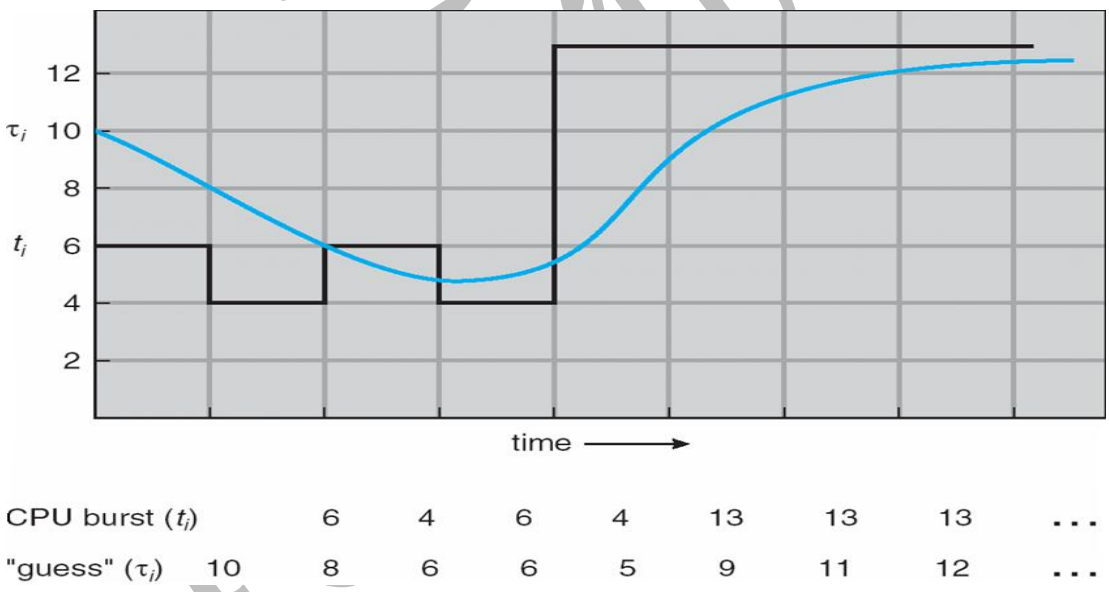


1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

Prediction of the Length of the Next CPU Burst



Examples of Exponential Averaging

a =0

$t_{n+1} = t_n$

Recent history does not count

a =1

$t_{n+1} = a\ t_n$

Only the actual last CPU burst counts

If we expand the formula, we get:

$t_{n+1} = a\, t_n + (1 - a)a\, t_n - 1 + \ldots$

$+(1 - a)^j a\, t_{n-j} + \ldots \qquad +(1 - a)^{n+1} t_0$

Since both a and (1 - a) are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer º highest priority)
- Preemptive
- nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem º **Starvation** – low priority processes may never execute
- Solution º **Aging** – as time progresses increase the priority of the process

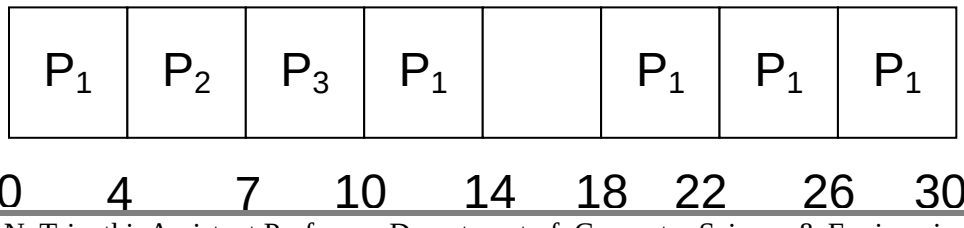
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
- *q* large Þ FIFO
- *q* small Þ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

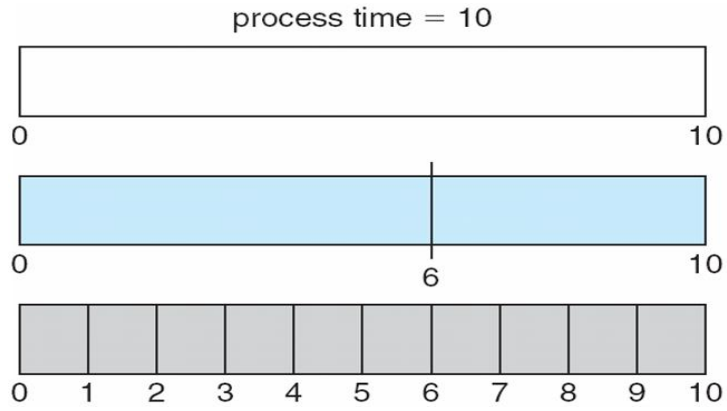
Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:

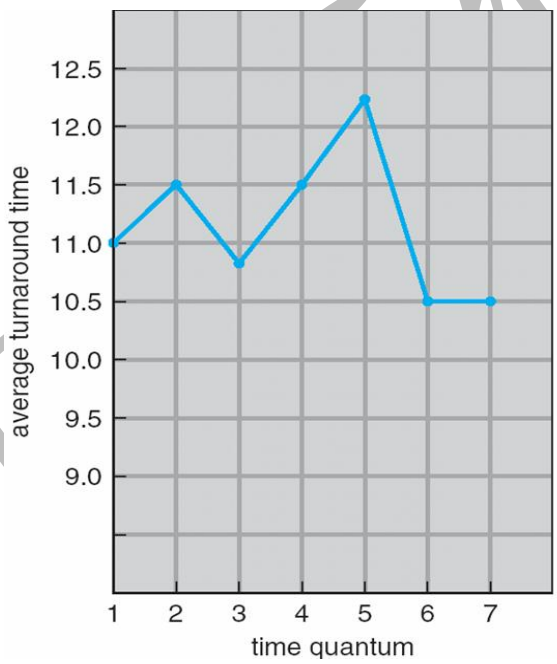


Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

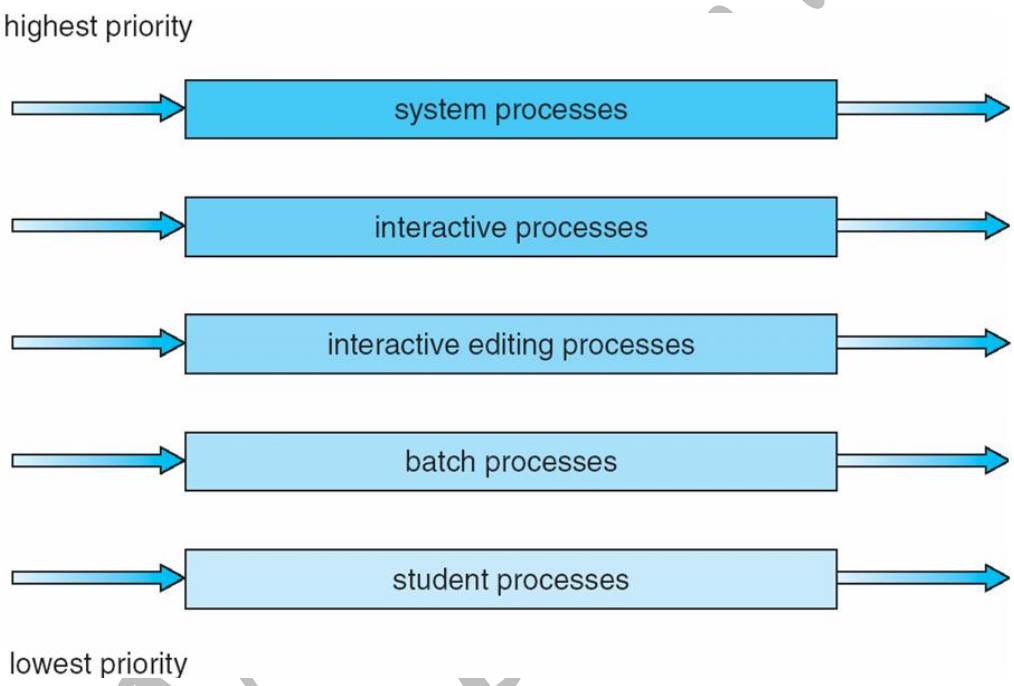
Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
- foreground – RR

- background – FCFS
- Scheduling must be done between the queues
- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- 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

Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process

method used to determine which queue a process will enter when that process needs service

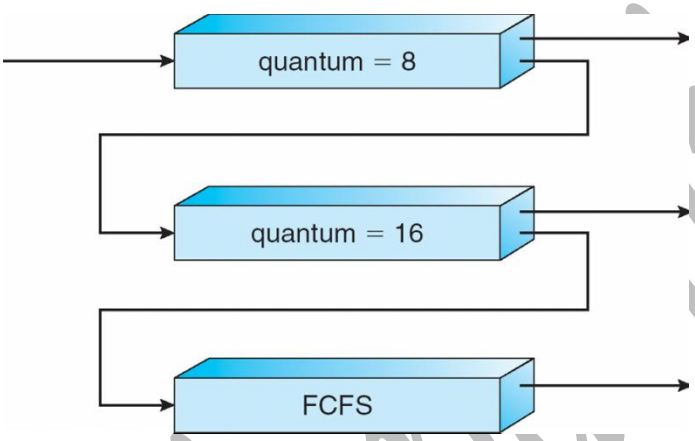
Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds

- Q_2 – FCFS
- Scheduling
- A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
- At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
- Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
- PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
- PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

Pthread Scheduling API

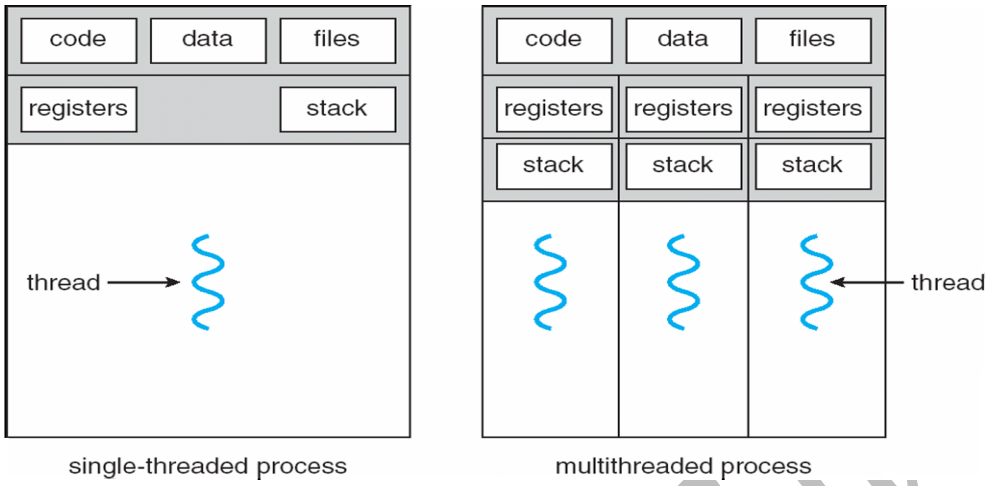
```
#include <pthread.h>

#include <stdio.h>

#define NUM THREADS 5

int main(int argc, char *argv[])

{

        int i;    pthread t tid[NUM THREADS];
```

```
        pthread attr t attr;
        /* get the default attributes */
        pthread attr init(&attr);
        /* set the scheduling algorithm to PROCESS or SYSTEM */
        pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
        /* set the scheduling policy - FIFO, RT, or OTHER */
        pthread attr setschedpolicy(&attr, SCHED OTHER);
        /* create the threads */
        for (i = 0; i < NUM THREADS; i++)
                pthread create(&tid[i],&attr,runner,NULL);
/* now join on each thread */
        for (i = 0; i < NUM THREADS; i++)
                pthread join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{
        printf("I am a thread\n");
        pthread exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

Single and Multithreaded Processes



Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

User Threads

- Thread management done by user-level threads librarynThree primary thread libraries:
- POSIX Pthreadsl Win32 threads
- Java threads

Kernel Threads

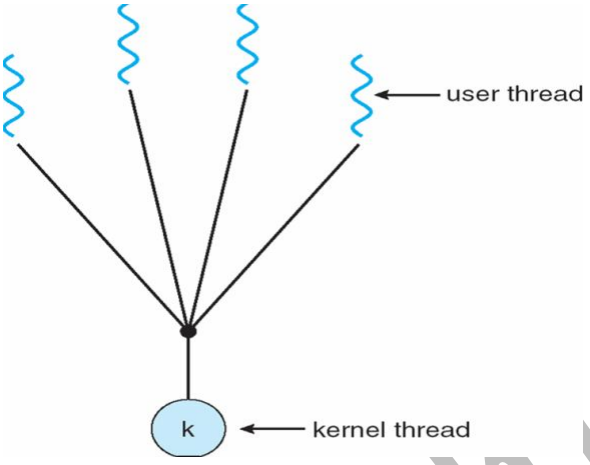
Supported by the Kernel

Examples

- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many



Many-to-One

Many user-level threads mapped to single kernel thread

Examples:

- Solaris Green Threads
- GNU Portable Threads

One-to-One

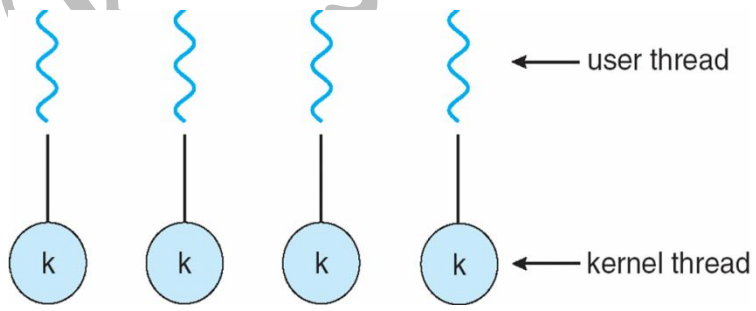
Each user-level thread maps to kernel thread

Examples

Windows NT/XP/2000

Linux

Solaris 9 and later

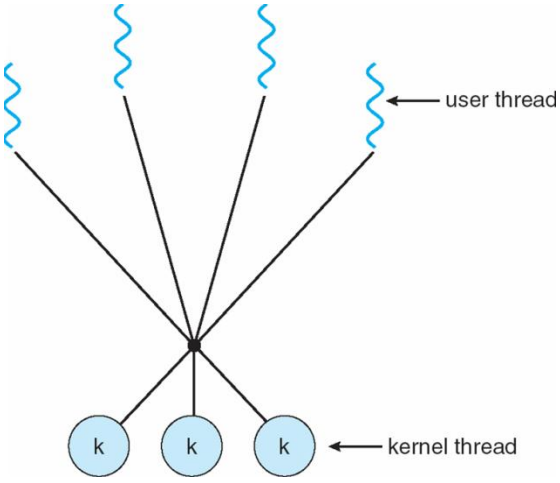


Many-to-Many Model

- 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 prior to version 9

Windows NT/2000 with the *ThreadFiber* package

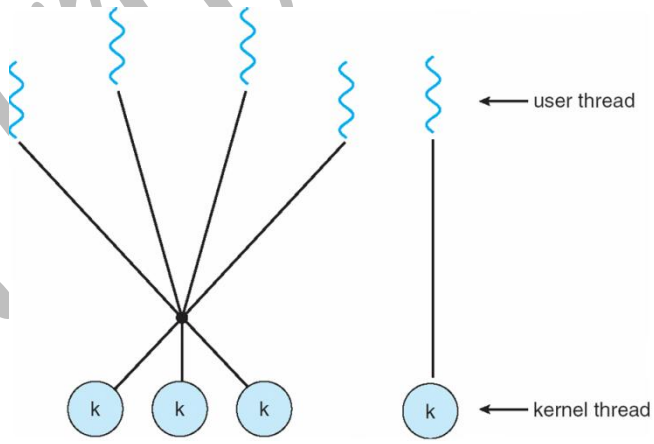


Two-level Model

Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

Examples

- IRIX
- HP-UX
- Tru64 UNIX
- Solaris 8 and earlier



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
- Library entirely in user space
- Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:lExtending Thread class
- Implementing the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
- Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
- **Asynchronous cancellation** terminates the target thread immediately
- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
- 1.Signal is generated by particular event
- 2.Signal is delivered to a process
- 3.Signal is handled
- Options:
- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific threa to receive all signals for the process

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
- Usually slightly faster to service a request with an existing thread than create a new thread

- Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Implements the one-to-one mapping, kernel-level

- Each thread contains
- A thread id
- Register set
- Separate user and kernel stacks
- Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
- ETHREAD (executive thread block)
- KTHREAD (kernel thread block)
- TEB (thread environment block)

Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

Example 1:

System has 2 disk drives

P_1 and P_2 each hold one disk drive and each needs another one

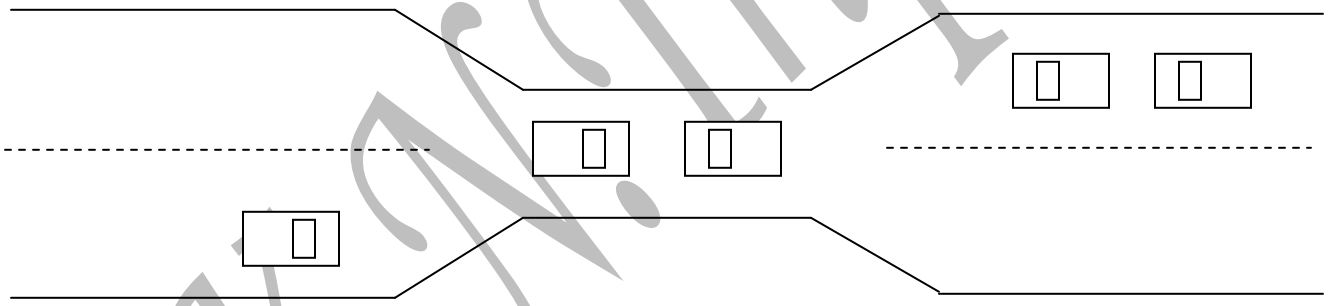
Example 2:

semaphores *A* and *B*, initialized to 1

P_0	*P1*
wait (A);	wait(B)
wait (B);	wait(A)

Example 3:

Bridge Crossing Example



Traffic moves only in one direction. Each section of a bridge can be viewed as a resource

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback) Several cars may have to be backed up if a deadlock occurs Starvation is possible.

Note – Most OS do not prevent or deal with deadlocks

System Model

Resource types $R_1, R_2, \ldots, R_m$.Like *CPU cycles, memory space, I/O devices e*ach resource type R_i has W_i instances.

Each process utilizes a resource in following steps:

1. **Request**
2. **Use**
3. **Release**

Deadlock Characterization:

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion: Only one process at a time can use a resource.

Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.

No preemption: A resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait: there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that 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, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource-Allocation Graph:

A set of vertices V and a set of edges E

V is partitioned into two types:

$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

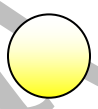
$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

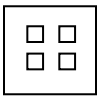
Edges are of two types:

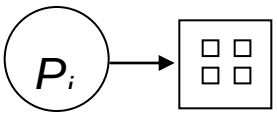
Request edge – directed edge $P_1 \rightarrow R_j$

Assignment edge – directed edge $R_j \rightarrow P_i$

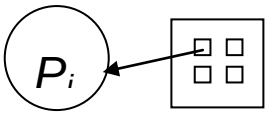
Graphical representation of graph G=(V,E):

Process 

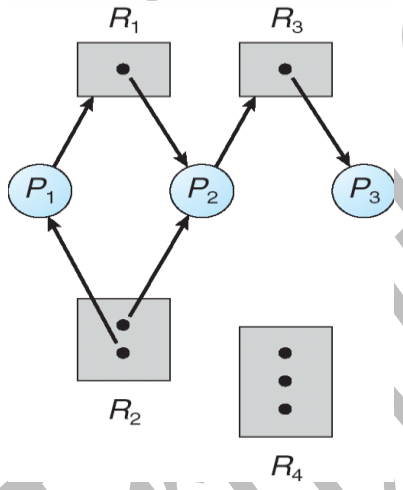
Resource Type with 4 instances 

P_i requests an instance of R_j 

P_i is holding an instance of R_j



Example of a Resource Allocation Graph:



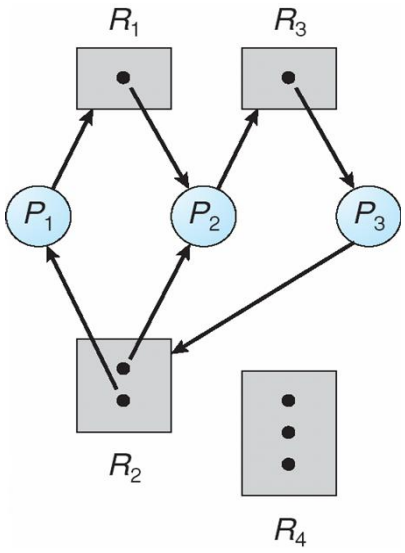
Resource Allocation Graph With a Deadlock:

Reason: There are two cycles in the graph.

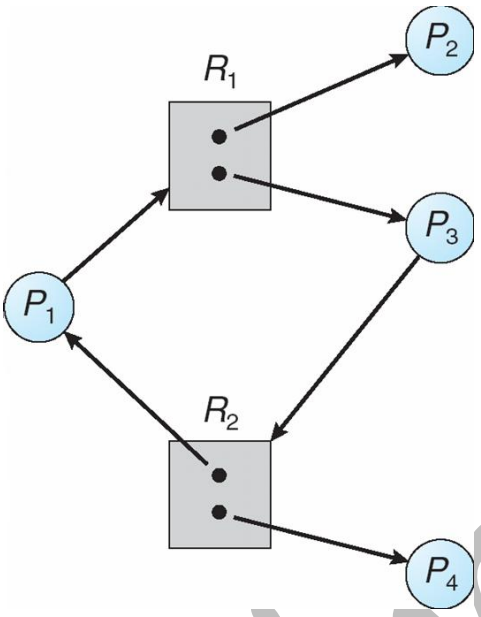
{P1 → R1, R1 → P2, P2 → R3, R3 → P3, P3 → R2, R2 → P1} and

{P2 → R3, R3 → P3, P3 → R2, R2 → P2}

No process will release an already acquired resource and the three processes will remain in the deadlock state.



Graph with a Cycle but No Deadlock



Conclusion from Resource Allocation Graph

1. If graph contains no cycles no deadlock.
2. And If graph contains a cycle & if only one instance per resource type, then deadlock .
3. If several instances per resource type, possibility of deadlock.

Deadlock Handling

We can deal with deadlocks in a number of ways:

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover from deadlock.
- Ignore the problem and pretend that deadlocks never occur in the system.

These three ways result in the following general methods of handling deadlocks:

1. Deadlock prevention: is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how processes can request for resources.

2. Deadlock Avoidance: This method of handling deadlocks requires that processes give advance additional information concerning which resources they will request and use during their lifetimes. With this information, it may be decided whether a process should wait or not.

3. Allowing Deadlocks and Recovering: One method is to allow the system to enter a deadlocked state, detect it, and recover.

Deadlock Prevention:

By ensuring that one of the four necessary conditions for a deadlock does not occur, we may prevent a deadlock.

Mutual exclusion

The mutual exclusion condition must hold for non-sharable resources, e.g., printer. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock, e.g., read-only files. Also, resources whose states can be saved and restored can be shared, such as a CPU. In general, we cannot prevent deadlocks by denying the mutual exclusion condition, as some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold and wait condition does not occur in a system, we must guarantee that whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol requires a process to request resources only when it has none. A process may request some resources and use them. But it must release these before requesting more resources.

The two main disadvantages of these protocols are: firstly, resource utilization may be low, since many resources may be allocated but unused for a long time. Secondly, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No preemption

To ensure that this condition does not hold we may use the protocol: if a process is holding some resources and requests another that cannot be allocated immediately to it, then all resources currently being held by the process are preempted. These resources are implicitly released, and added to the list of resources for which the process is waiting. The process will be restarted when it gets all its old, as well as the newly requested resources.

Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing ordering of enumeration. Let R={ R1, R2, R3 }be resource types. We assign to each a unique integer, which allows us to compare two resources and to determine whether one precedes another in our ordering. For example, if the set of resource types R includes tape drivers, disk drives, and printers then the function F: R→N might be used to assign positive integers to these resources as follows:

F(tape drive) =1

F(disk drive) =5

F(printer)=12

Each process can request resources in an increasing order of enumeration. For

example, a process wanting to use the tape and the disk drive must first request the tape drive and then the disk drive.

We can prove that if processes use this protocol then circular wait can never occur. We will prove this by contradiction. Let's assume that there is a cycle involving process P0 through Pk and that Pi is holding an instance of Ri, as shown below. The proof follows.

P0 → P1 → P2 → … → Pk → P0

R0 R1 R2 Rk R0

- ⇒ F(R0) < F(R1) < … < F(Rk) < F(R0)
- ⇒ F(R0) < F(R0), which is impossible
- ⇒ There can be no circular wait.

Deadlock Avoidance:

Requires that the system has some additional *a priori* information available. Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state System is in safe state if there exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

That is:

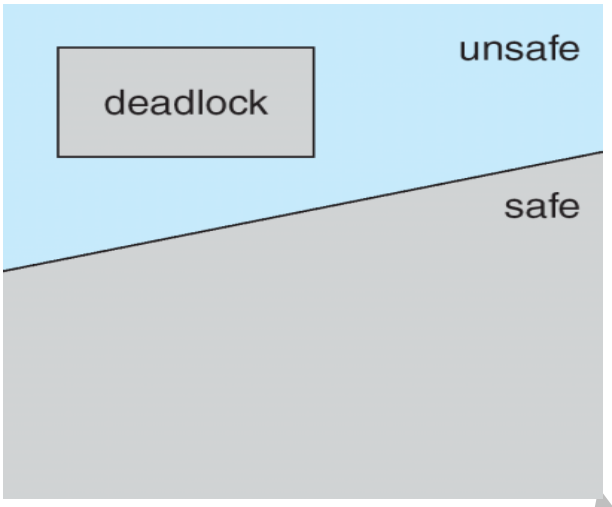
If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished. When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate .When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

If a system is in safe state no deadlock

If a system is in unsafe state possibility of deadlock and Avoidance ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



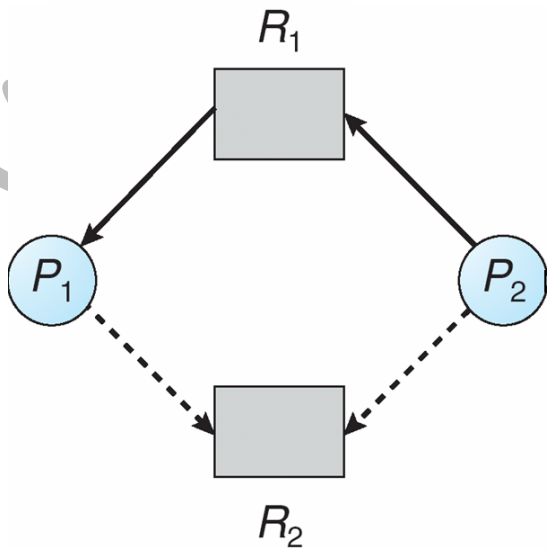
Deadlock Avoidance algorithms

1. Single instance of a resource type→Use a resource-allocation graph
2. Multiple instances of a resource type→Use the banker's algorithm

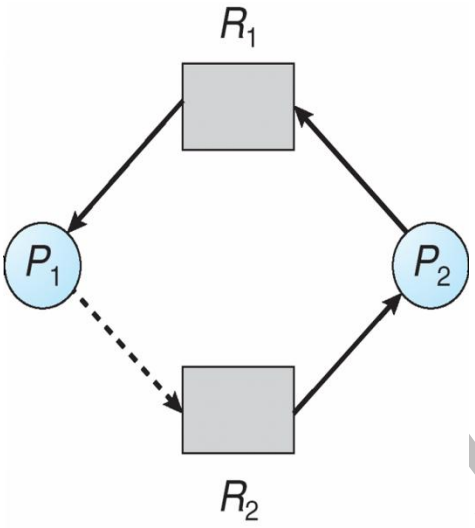
Resource-Allocation Graph Scheme

Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line Claim edge converts to request edge when a process requests a resource Request edge converted to an assignment edge when the resource is allocated to the process .When a resource is released by a process, assignment edge reconverts to a claim edge Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

Suppose that process P_i requests a resource R_j.The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

Multiple instances Each process must a priori claim maximum use When a process requests a resource it may have to wait When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm :

Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.

Max: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j.

Allocation: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j

Need: n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task

$Need$ [i,j] = Max[i,j] – $Allocation$ [i,j]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively.

Initialize:

Work = *Available*

Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find and *i* such that both:

(a) *Finish* [*i*] = *false*

(b) $Need_i$ < *Work*

If no such *i* exists, go to step 4

3. *Work* = *Work* + $Allocation_i$
Finish[*i*] = *true*
go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

If $Request_i$ [*j*] = *k* then process P_i wants *k* instances of resource type R_j

1. If $Request_i$ < $Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i$ < *Available*, go to step 3. Otherwise P_i must wait, since resources are not available

3.Pretend to allocate requested resources to P_i by modifying the state as follows:

Available = *Available* – *Request;*

$Allocation_i$ = $Allocation_i$ + $Request_i$;

$Need_i$ = $Need_i$ – $Request_i$;

Invoke the Safety algorithm. If the resulting resource allocation graph is safe, the transaction is completed. Else, the old resource allocation state is restored and process Pi must wait for $Request_i$.

Example of Banker's Algorithm

Let their re 5 processes P_0 through P_4,and 3 resource types:*A* (10 instances), *B* (5instances), and *C* (7 instances) Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be = *Max – Allocation*

Need	*A B C*
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

Check that

Request < Need (that is, (1,0,2) <(1,2,2) true

Request < Available (that is, (1,0,2) <(3,3,2) true

	Allocation	*Need*	*Available*
	A B C	*A B C*	
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

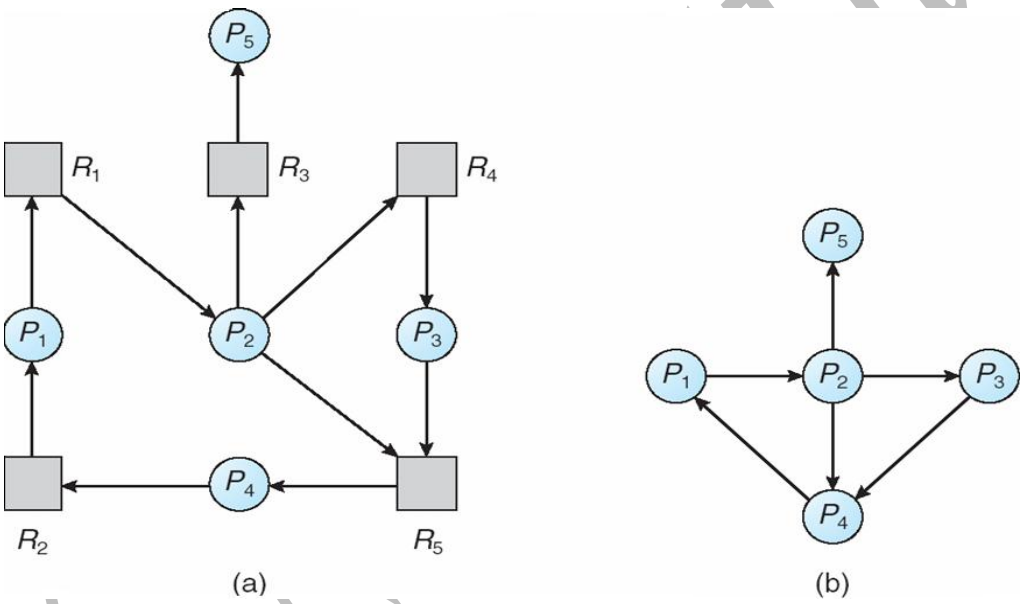
Deadlock Detection

Allow system to enter deadlock state Detection algorithm Recovery scheme

Single Instance of Each Resource Type

Maintain *wait-for* graph. Nodes are processes, $P_i \rightarrow P_j$ if P_i is waiting for P_j Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock .An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph corresponding wait-for graph

Several Instances of a Resource Type

Available*:* A vector of length *m* indicates the number of available resources of each type.

Allocation*:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request*:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1) Let Work and Finish be vectors of length m and n respectively. Initialize

Work=Available. For i=1, 2,… , n if Allocation[i] ≠ 0 the Finish[i]=false;

otherwise Finish[i]=true

2) Find an index i such that both

a. Finish[i] = = false

b. Requesti ≤ Work

c. If no such i exists go to step 4.

3) Work=Work + Allocationi

a. Finish[i]=true

b. Go to step 2.

4) If Finish[i] = = false, for some i, 1≤ i ≤n, then the system is in a deadlock state.

Moreover, if Finish[i] = = false, then Pi is deadlocked

Example of Detection Algorithm

Five processes P_0 through P_4; three resource types
A (7 instances), *B* (2 instances), and *C* (6 instances)

Snapshot at time T_0:

	Allocation	***Request***	***Available***
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

P_2 requests an additional instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system?

Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

When, and how often, to invoke depends on:

How often a deadlock is likely to occur?

How many processes will need to be rolled back?

One for each disjoint cycle. If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock

Recovery from Deadlock: Process Termination

Abort all deadlocked processes. Abort one process at a time until the deadlock cycle is eliminated In which order should we choose to abort?

- Priority of the process
- How long process has computed, and how much longer to completion
- Resources the process has used
- Resources process needs to complete
- How many processes will need to be terminated
- Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor