OPERATING SYSTEM

UNIT II

CPU Scheduling

Basic concepts(Introduction)

- All computer resources are scheduled before use, one among them is CPU.

CPU-I/O Burst Cycle

- Process execution consists of a cycle of CPU burst and I/O burst.
- An I/O bound program would have short CPU bursts.
- A CPU bound program have long CPU bursts.
- It is important in the selection of CPU scheduling algorithm.

CPU Scheduler

✓ Short term scheduler or CPU scheduler is used to select one of the processes from ready Queue to execute.

Dispatcher

It gives control of the CPU to the process selected by scheduler.

Dispatch Latency

Time taken for the dispatcher to stop one process and start another process

Scheduling Criteria

- ✓ **CPU Utilization**
  - ▪ To keep the CPU busy
- ✓ **Throughput**

  The number of processes that are completed per time unit is called Throughput (high).
- ✓ **Turnaround time**

  The time between job submission and job completion (low)
- ✓ **Waiting time**

  Time that a process spends waiting in the ready queue (low)
- ✓ **Response time**

  The amount of time it takes to start responding.(low)

  It is desirable to maximize CPU utilization and throughput and to minimize turn around time,waiting time and response time.

Pre-emptive & Non Pre-emptive Scheduling

Non-Pre-emptive

- ❖ In non Pre-emptive scheduling once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU.
- ❖ Process moves from running state to waiting state.
- ❖ Process terminates.

Pre-emptive

In pre-emptive scheduling the CPU allocated to a process can be taken away in the middle and allocated to another process.

- ❖ Process switches from running state to ready state.
- ❖ Process switches from waiting state to ready state.

****

Scheduling Algorithm

* CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

* There are many different CPU scheduling algorithms. In this section, we describe several of these algorithms.

First-come, First-Serve Scheduling

* By far the simplest CPU scheduling algorithm is the first-come, first-served scheduling (FCFS) algorithm .

* With this scheme, the process that requests the CPU first is allocated the CPU first.

* The implementation of the FCFS policy is easily managed with a FIFO queue.

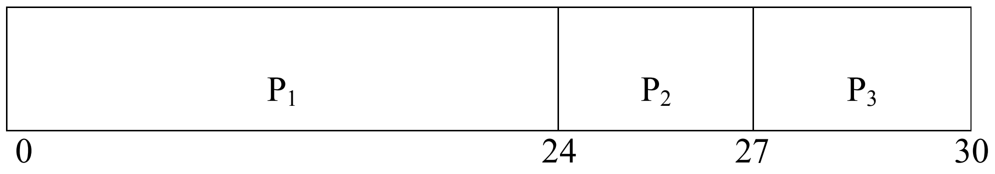
* The code for FCFS scheduling is simple to write and understand.

* The average waiting time under the FCFS policy, however, is often quite long.

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

Process	Burst Time
P1	24
P2	3
P3	3

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



The waiting time is 0 milliseconds for process P_1,

24 milliseconds for process P_2, and

27 milliseconds for process P_3.

Thus, the average waiting time is (0+24+27)/3=17 milliseconds.

The average waiting time under a FCFS policy is generally not minimal, and may vary substantially if the process CPU burst times vary greatly.

- The FCFS scheduling algorithm is **non-preemptive**. Once the CPU has been allocated to a process, that process keeps the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is particularly troublesome for time-sharing systems.

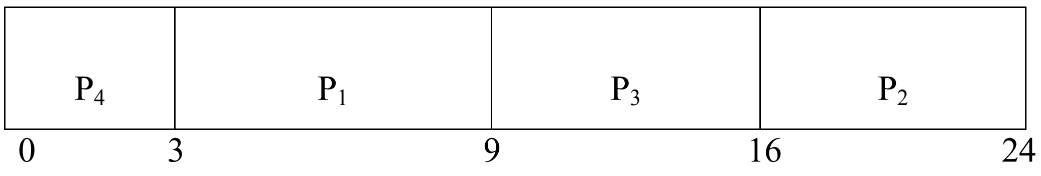
Shortest-Job First Scheduling

A different approach to CPU scheduling is the shortest-job-first(SJF) algorithm. When the CPU is available, it is assigned to the process that has the **smallest burst**. The scheduling is done by examining the length of the next CPU burst of a process, rather than its total length.

As an example, consider the following set of process, with the length of the CPU burst time given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling we would schedule these processes according to the following Gantt Chart:



The waiting time is 3 milliseconds for process P_1,

16 milliseconds for process P_2,

9 milliseconds for process P_3, and

0 milliseconds for P_4.

Thus, the average waiting time is (3+16+9+0)/4=**7** milliseconds.

If we were using the FCFS scheduling scheme, then the average waiting time would be **10.25** milliseconds.

- The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.(advantage)
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request. (disadvantage)
- SJF scheduling is used frequently in long term scheduling.
- The SJF algorithm may be either **preemptive or non-preemptive**.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process, to finish its CPU burst.

Shortest-remaining-time-first scheduling

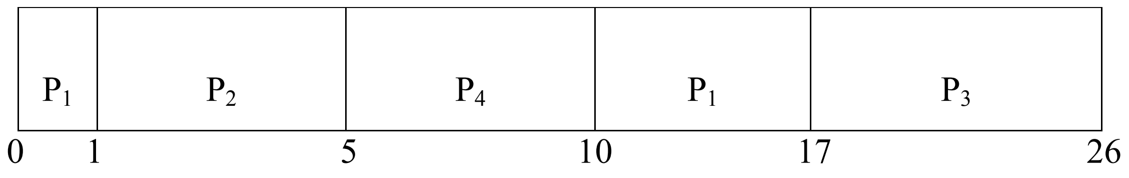
Shortest-remaining-time-first scheduling

- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling.**

As an example, consider the following four processes, with the length of the CPU-burst time given in milliseconds:

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

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting SJF schedule is as depicted in the following Gantt chart:



- Process P_1 is started at time 0, since it is the only process in the queue.
- Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2(4milliseconds), so Process P_1 is preempted, and process P_2 is scheduled.
- The average waiting time for this example is((10-1) + (1-1) + (17-2) + (5-3)) /4=26/4=6.5 milliseconds**.**
  (allocated time – arrival time).

A non-preemptive SJF scheduling would result in average waiting time of 7.75 milliseconds.

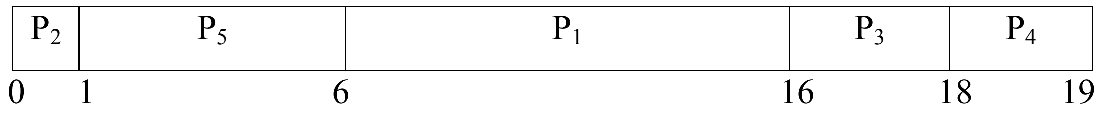
PRIORITY SCHEDULING

- The SJF algorithm is a special case of the general priority scheduling algorithm.
- A priority is associated with each process , and the CPU is allocated to the process with the highest priority.
- Equal priority processes are scheduled in FCFS order.
- Priorities are generally some fixed range of numbers,such as 0 to 7,or 0 to 4095.However there is no general agreement on whether 0 is the highest or lowest priority.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order p_1,……p_5, with the length of the CPU burst time given in milliseconds:

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

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds

- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time

limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.

- External priorities are set by criteria that are external to the operating system, such as the importance of the process, the type and amount of fund being paid for computer use, the department sponsoring the work, and other, often political, factors.
- Priority scheduling can be either **preemptive or non-preemptive.**

A major problem with priority scheduling algorithm is **indefinite blocking or starvation.**

- A process that is ready to run but lacking the CPU can be considered blocked, waiting for the CPU.
- A priority scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU.
- In a heavily loaded computer system, a stream of higher-processes can prevent a low-priority process from ever getting the CPU.
- A solution to the problem of indefinite blockage of low-priority processes is **aging**.
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 0 (low) to 127 (high), we could increment the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 0 would have the highest priority in the system and would be executed.

Round-Robin scheduling

The **round-robin(RR)** scheduling algorithm is designed especially for time-sharing 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.
- A time quantum is generally from 10 to 100 milliseconds.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.

- The CPU scheduler picks the first process from the read queue, sets a time to interrupt after 1 time quantum, add dispatches the process.

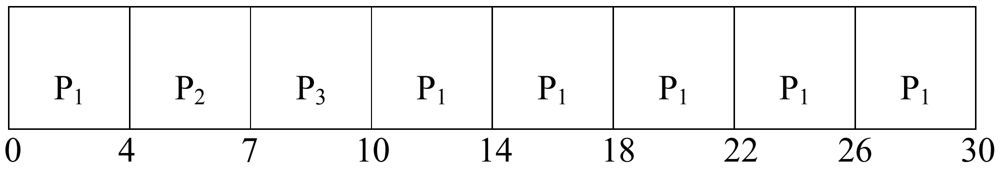
One of two things will then happen.

- The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy, however, is often quite long.Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it pre-empted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires.
- The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



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

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row.

- If a process CPU burst exceeds 1 time quantum, that process is **preempted** and is put back in the ready queue. The RR scheduling algorithm is preemptive.

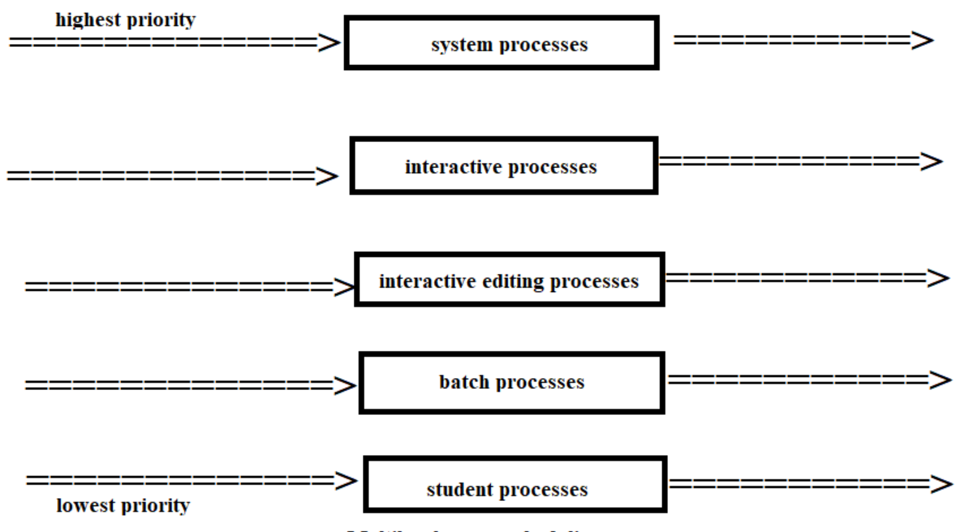
The performance of the RR algorithm depends heavily on the size of the time quantum. If the time quantum is very small (say 1 microsecond), the RR approach is called **processor sharing.**

Multilevel Queue Scheduling

A **multilevel queue-scheduling algorithm** 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.
- Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes.
- The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- In addition, 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.

Look at an example of a multilevel queue scheduling algorithm with five queues:



Multilevel queue scheduling

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. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
    - If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time slice between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue.

DEADLOCK

A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

➢ **System Model**

A process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release:** The process releases the resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by only another process in the set

To illustrate are three processes, each holding one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state.

****

Deadlock Characterization

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever starting.

❖ **Necessary Conditions**

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

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:** There must exist a process that is with atleast one resource and is waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** There must exist a set $\{P_0,P_1,\ldots..,P_n\}$ of waiting processes such that P_0 is waiting for a resources that is held by P_1,P_1 is waiting for a resource that is held by $P_2,\ldots P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

❖ **Resource-Allocation Graph**

Deadlock can be described more precisely in terms of a directed graph called a **system resource-allocation graph.** This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into different types of nodes $P=\{P_1,P_2,\ldots..,P_n\}$,the set consisting of all the active processes in the system, and $R=\{R_1,R_2,\ldots..,R_m\}$, the set consisting of all resource types in the system.

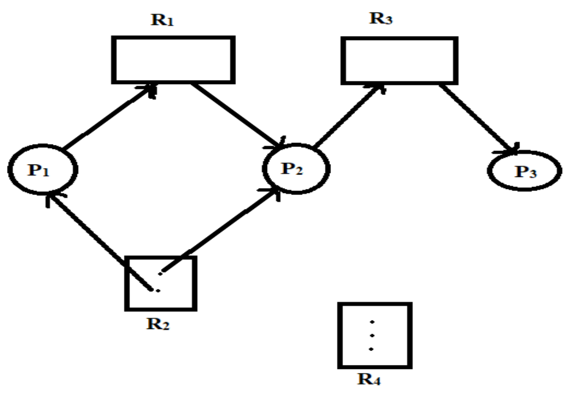
A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type $R_j$has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a **request edge;** a directed edge $R_j \rightarrow P_i$ is called an **assignment edge.**

Pictorially, we represent each process P_i as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the square.

When process P_i request an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is **instantaneously transformed** to an assignment edge.

The resource-allocation graph show in depicts the following situation.

- ✓ The sets P,R, and E:
  - o $P=\{P_1,P_2,P_3\}$
  - o $R=\{R_1,R_2,R_3,R_4\}$
  - o $E=\{P_1\rightarrow R_1, P_2\rightarrow R_3, R_1\rightarrow P_2, R_2\rightarrow P_2, R_2\rightarrow P_1, R_3\rightarrow P_3\}$
- ✓ Resource instances:
  - o One instance of resource type R_1
  - o Two instances of resource type R_2
  - o One instance of resource type R_3
  - o Three instances of resource type R_4



✓ Process states:

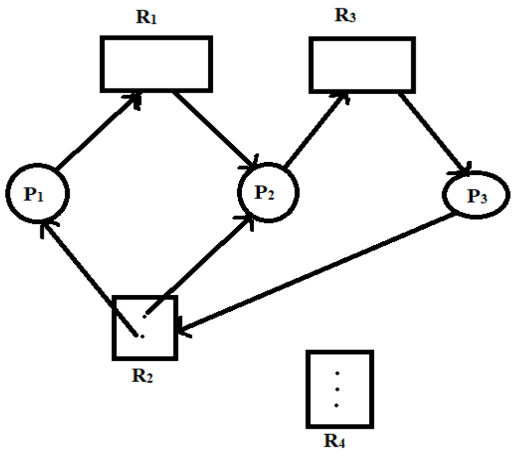
- ♦ Process P_1 is holding an instance type R_2, and is waiting for an instance of resource type R_1.
- ♦ Process P_2 is holding an instance of R_1 and R_2, and is waiting for an instance of resource type R_3.
- ♦ Process P_3 is holding an instance of R_3.

If the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains a cycle, then a deadlock may exist.

To illustrate condition, let us return to the resource-allocation graph depicted in suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

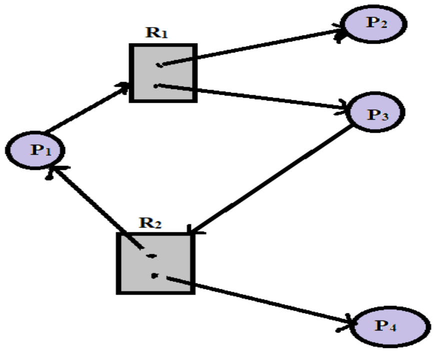
$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



Methods for Handling Deadlocks

There are three different methods for dealing with the deadlock problem:

- ❖ We can use a protocol to ensure that the system will never enter a deadlock state.
- ❖ We can allow the system to enter a deadlock state and then recover.



- ❖ We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

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 requests for resources can be made.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime.

Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

❖ **Mutual Exclusion**

- The mutual-exclusion condition must hold for non sharable resources. For example, a printer cannot be simultaneously shared by several processes.
- Read-only files are a good example of a sharable resource.

❖ **Hold and Wait**

- To ensure that the hold-and-wait condition never occurs in the system.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when the process has none.
- Eg: consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process, then the process must initially request the tape drive, disk file, and the printer.
- The second method allowed the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.
- There are two main disadvantages to these protocols. First, resource utilization may be low.
- Second, starvation is possible.

❖ **No preemption**

- The third necessary condition is that there be no preemption of resources.
- To ensure that this condition does not hold, we can use the following protocol. If a process that is holding some resources requests another resource that cannot be immediately to it (that is, the process must wait),then all resources currently being held are preempted.
- If a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we pre-empt the desired resources from the waiting process and allocate them to the requesting process.

❖ **Circular Wait**

- One way to ensure that the circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.
- Let R={R_1,R_2,……,R_m} be the set of resource types. We assign to each resource type a unique integer number.
- We define a one-to-one function F:R→N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

  F(tape drive)=1,

  F(disk drive)=5,

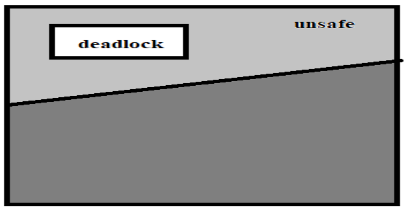
  F(Printer) =12.
- We can now consider the following protocol to prevent deadlock: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say R_i. After that, the process can request instances of the same resource type R_j if and only if F(R_j) > F(R_j).

- We can require that, whenever a process requests an instance of resource type R_j, it has released any resources R_i such that $F(R_i) \geq F(R_j)$.
- If these two protocols are used, than the circular-wait condition cannot hold.

Deadlock Avoidance

An alternative method for avoiding deadlock is to require additional information about how resources are to be requested.

- **Safe State**
  - A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of processes $<P_1,P_2,\ldots,P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j< i$. In this situation, if the resources the process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be **unsafe.**



- An unsafe state may lead to a deadlock.
- To illustrate, we consider a system with 12 magnetic tape drives and 3 processes process: P_0,P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0, process P_0 holding 5 tape drives, process P_1 is holding 2, and process P_2 id holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	**Current Needs**	**Allocated**
P_0	10	5	5
P_1	4	2	2
P_2	9	7	2

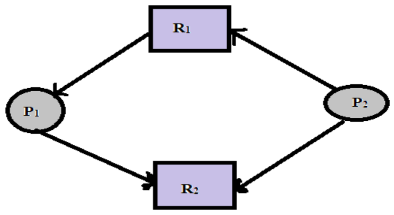
At time t_0, the system is in a safe state. The sequence $<P_1,P_0,P_2>$ satisfies the safety condition.

❖ **Resource-Allocation Graph Algorithm**

- If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in can be used for deadlock avoidance.
- In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge.** A claim edge $P_i \rightarrow R_j$ indicates that process

P_i may request resource R_j at some time in the future. This edge resembles a requests edge in direction, but is represented by a dashed line.

- Note that we check for safety by using a **cycle-detection algorithm.**
- 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.



- To illustrate this algorithm, we consider the resource-allocation graph. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

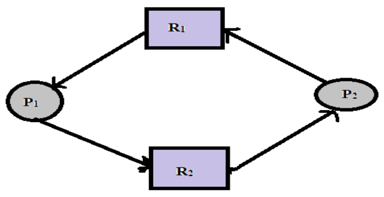
❖ Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm.**

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

We need the following data structures:

- **Available:** A vector of length m indicates the number of available resources of each type. If **available[j]=k,** there are k instances of resource type R_j available.



- **Max:** An n x m matrix defines the maximum demand of each process. If Max[i,j]=k, then P_i may request at most k instances of resource type R_j.
- **Allocation:** An n x m matrix defines the number of resources of each; type currently allocated to each process. If Allocation[i,j]=k, then process P_i is currently allocated k instances of resource type R_j.

- **Need:** An n x m matrix indicates the remaining resource need of each process. If Need[i,j]=k, then P_i may need k more instances of resource type R_j to complete its task. Note that Need[i,j]=Max[i,j]-Allocation[i,j].

- **Safety Algorithm**

  The algorithm for finding out whether or not a system is in a safe state can be described as follows:

  1. Let work and finish be vectors of length m and n, respectively. Initialize work:=Available and Finish[i]:=false for i=1,2,…,n.
  2. Find an i such that both
     a. Finish[i]=false
     b. $Need_i \leq$ work

     If no such i exists, go to step
  3. Work:=work+Allocation,

     Finsh[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**

  Let $Request_i$ be the request vector for process P_i. If $Request_i$[j]=k, then process P_i wants k instances of resources type R_j. When a request for resources is made by process P_i the following actions are taken:

  1. If $Request_i \leq Need_i$, go to step2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
  2. If $Request_i \leq$ Available, go to step 3. Otherwise, P_i must wait, since the resources are not available.
  3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

     Available:=Available-$Request_i$;

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

$Need_i := Need_i - Request_i;$

If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$ and the old resource-allocation state is restored.

****

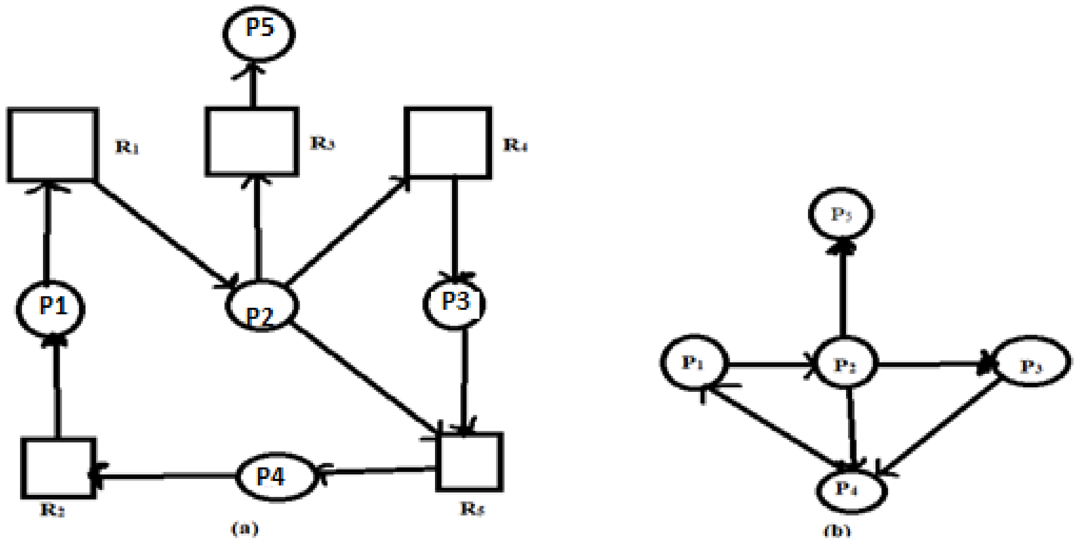
Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur.

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource- allocation graph, called a wait for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.
- To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.



Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.
- **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 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.
- Initialize work := Available,if allocation#0 then Finish [i] := false; otherwise, Finish[i] := true.
- Find an index i such that both

  a. Finish[i]=false.

  b. $\text{Request}_i \leq$ work.

  If no such i exists, go to setp4.

- Work := work + allocation$_i$. Finish[i] := true, go to step2.
- If Finish[i]=false, for some i,$1 \leq i \leq n$, then the system is in a deadlock state.
- This algorithm requires an order of m x n^2 operations to detect whether the system is in a deadlock state.

❖ **Detection-Algorithm Usage**

- Detection algorithm is invoked based depends on two factors:
  1. How often is a deadlock likely to occur?
  2. How many processes will be affected by deadlock when it happens?
- If deadlock occur frequently, then the detection algorithm should be invoked frequently.
- A less expensive alternative is simple to invoke the algorithm at less frequent intervals.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists,, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock.

❖ **Process Termination**

To eliminate deadlocks by aborting a process, we use one of two methods.

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed later.

- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- Many factors may determine which process is chosen, including:
  1. What the priority of the process is?
  2. How long the process has computed, and how much longer the process will compute before completing its designated task?
  3. How many and what type of resources the process has used (for example, whether the resources are simple to pre-empt)?
  4. How many more resources the process needs in order to complete ?
  5. How many processes will need to be terminated?
  6. Whether the process is interactive or batch?

❖ **Resource Preemption**

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim:** We should determine which resources and which processes are to be pre-empted.
2. **Rollback:** If we pre-empt a resource from a process, Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.
3. **Starvation:** we should ensure that resources will not always be preempted from the same process.

****