MODULE IV

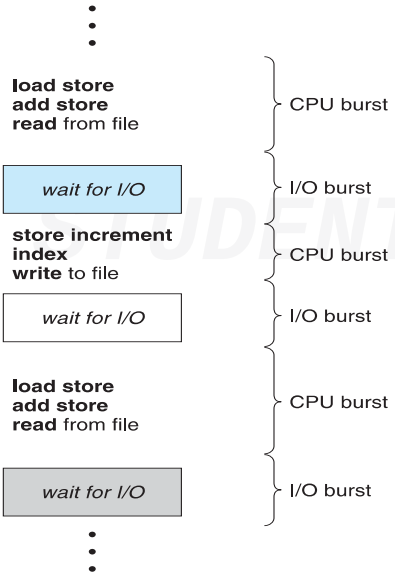
CPU SCHEDULING

Basic concepts

- Maximum CPU utilization obtained with multiprogramming

CPU-I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/0 wait. Processes alternate between these two states.
- Process execution begins with a CPU burst.
- Final CPU burst ends with a system request to terminate execution.



Alternating sequence of CPU and 1/0 bursts.

- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts.
- This distribution can be important in the selection of an appropriate CPU-scheduling algorithm

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 scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

- A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or sirnply an unordered linked list.
- The records in the queues are generally process control blocks (PCBs) of the processes.

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
- Scheduling under 1 and 4 is nonpreemptive.
- All other scheduling is preemptive
- Consider access to shared data
- Consider preemption while in kernel mode
- Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
    1. Switching context
    2. Switching to user mode
    3. Jumping to the proper location in the user program to restart that program
- The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

Scheduling Criteria

- Many criteria have been suggested for comparing CPU-scheduling algorithms.
- The criteria include the following:
    1. **CPU utilization**- Keep the CPU as busy as possible.
    2. **Throughput** – Number of processes that complete their execution per time unit
    3. **Turnaround Time**-The interval from the time of submission of a process to the time of completion. Turnaround Time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0 waiting time.
    4. **Waiting Time**-sum of the periods spent waiting in the ready queue.

5. **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)

SCHEDULING ALGORITHMS

First- Come, First-Served (FCFS) Scheduling

- 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.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

Process	Burst Time
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: P1 , P2 , P3
- The Gantt Chart for the schedule is:

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

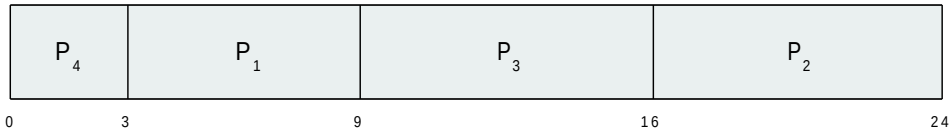
- Waiting time for P1 = 6; P2 = 0; P3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Convoy effect -All the processes wait for the one big process to get off the CPU.
- FCFS scheduling algorithm is Non-preemptive.

Shortest-Job-First Scheduling

- This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- More appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm.

Process	Burst Time
P1	6
P2	8
P3	7
P_4	3

- SJF scheduling chart:



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- SJF is optimal – Gives minimum average waiting time for a given set of processes.
- The difficulty is knowing the length of the next CPU request.
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

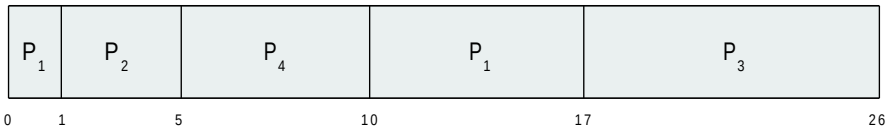
1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \le \alpha \le 1$
4. Define :

- Commonly, α set to ½
- Preemptive version called shortest-remaining-time-first
- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
- τn+1 = α tn+(1 - α)α tn -1 + … +(1 - α)j α tn -j + … +(1 - α)n +1 τ0
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor
- A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.

- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.
- Now we add the concepts of varying arrival times and preemption to the analysis

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

- Preemptive SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive

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

- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec
- Priorities can be defined either internally or externally.

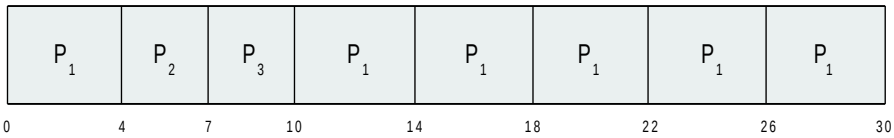
- A rnajor problem with priority scheduling algorithms is indefinite blocking, or starvation, leaving low priority processes waiting indefinitely.
- Solution for starvation is Aging, a technique of gradually increasing the priority of processes that wait in the system for a long time.

Round-Robin Scheduling

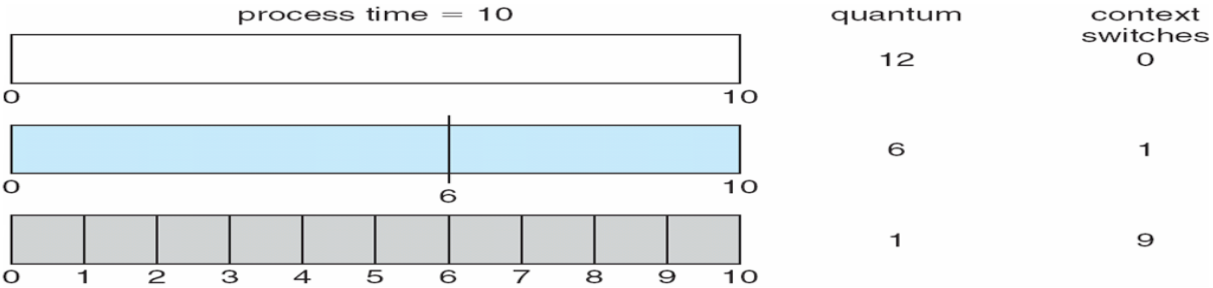
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - q small ⇒ 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
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 microsecond

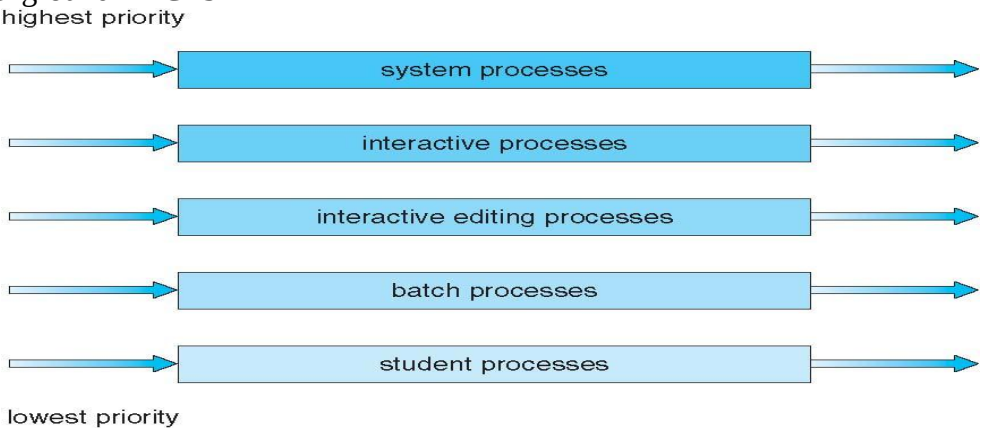


- Turnaround time also depends on the size of the time quantum.

- In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Multilevel Queue Scheduling

- In this scheduling processes are classified into different groups
- Ready queue is partitioned into separate queues,
  - **foreground** (interactive)
  - **background** (batch)
- These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.
- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS

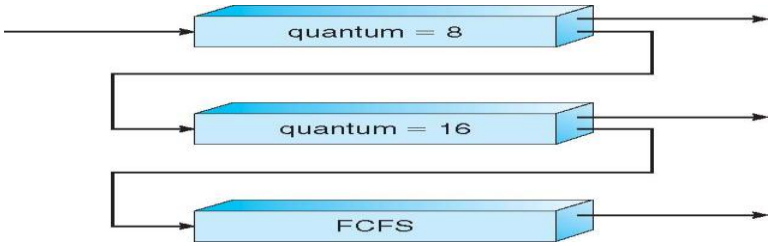


- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Feedback Queue

- The multilevel feedback queue scheduling algorithm, allows a process to move between 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



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - *Q1 – RR time quantum 16 milliseconds*
  - *Q2 – 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 Q1
  - 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

Deadlocks

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: The process requests the resource. If the request cannot be granted immediately (for example, if 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.

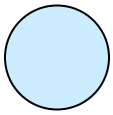
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

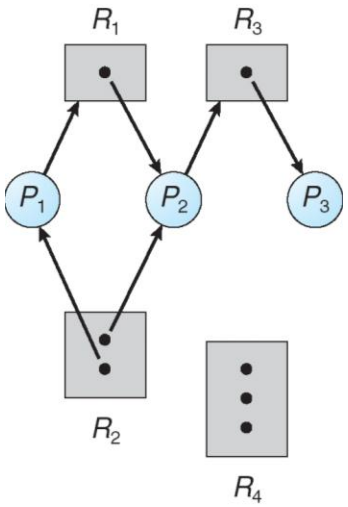
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

Resource-Allocation Graph

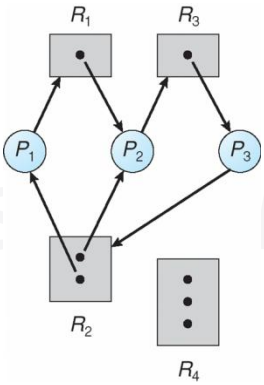
Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. A set of vertices V and a set of edges E.

- 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
- **request edge** – directed edge Pi → Rj
- **assignment edge** – directed edge Rj → Pi
- Process 
- Resource Type with 4 instances
- Pi requests instance of Rj
- Pi is holding an instance of Rj

Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Since no resource instance is currently available, we add a request edge $P3 \rightarrow R2$ to the above graph .At this point, two minimal cycles exist in the system:

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

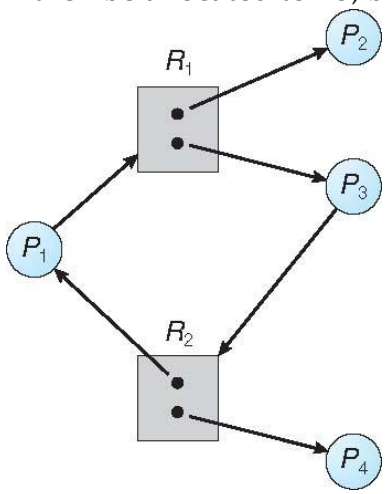
Processes *P*1, *P*2, and *P*3 are deadlocked. Process *P*2 is waiting for the resource *R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process *P*2 to release resource *R*1.

Graph With A Cycle But No Deadlock

Now consider the resource-allocation graph below. In this example, we also have a cycle:

$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

However, there is no deadlock. Observe that process $P4$ may release its instance of resource type $R2$. That resource can then be allocated to $P3$, breaking the cycle.



Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

- **Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
    - Low resource utilization; starvation possible
- **No Preemption –**
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
    - Preempted resources are added to the list of resources for which the process is waiting
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

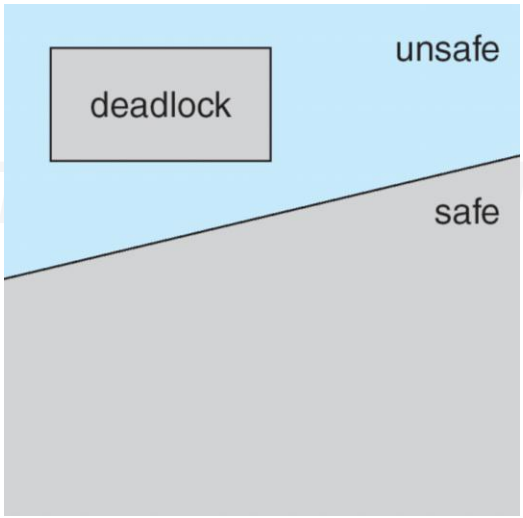
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state 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 <P1, P2, …, Pn> of ALL the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, 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 Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
    - When Pi terminates, Pi +1 can obtain its needed resources, and so on
- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State

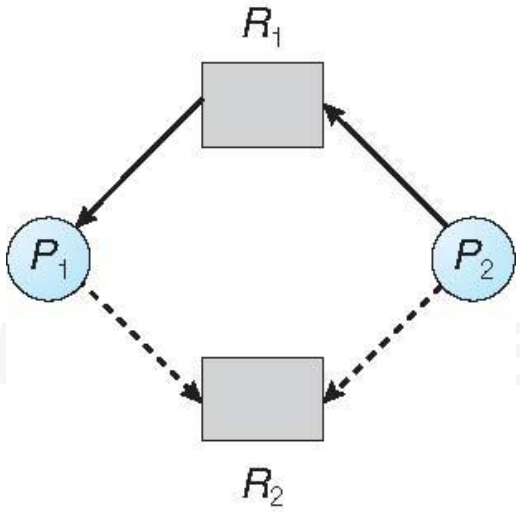


Avoidance Algorithms

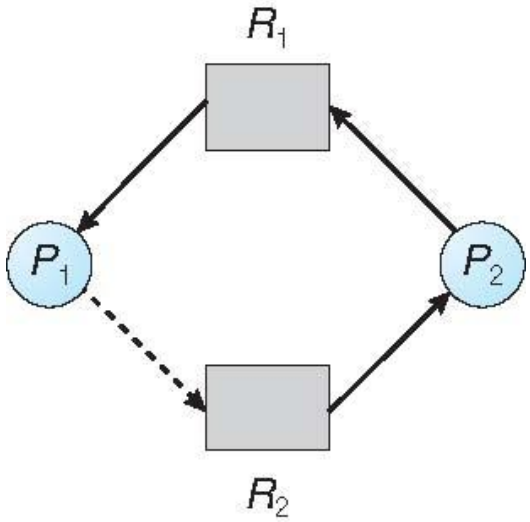
- Single instance of a resource type
    - Use a resource-allocation graph
- Multiple instances of a resource type
    - Use the banker's algorithm

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



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 Rj available
- Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj
- Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
- Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj 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 $i = 0, 1, \ldots, n-1$

2. Find an i such that both:

(a) Finish [i] = false

(b) Needi ≤ Work

If no such i exists, go to step 4

3. Work = Work + Allocationi
   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

$Request_i$ = 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 Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available

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

   Available = Available – Requesti;

   Allocationi = Allocationi + Requesti;

   Needi = Needi – Requesti;

   - If safe ⇒ the resources are allocated to Pi
   - If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P0 through P4;
- 3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time T0:

	Allocation	*Max*	*Available*
	A B C	*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 ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*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 2	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

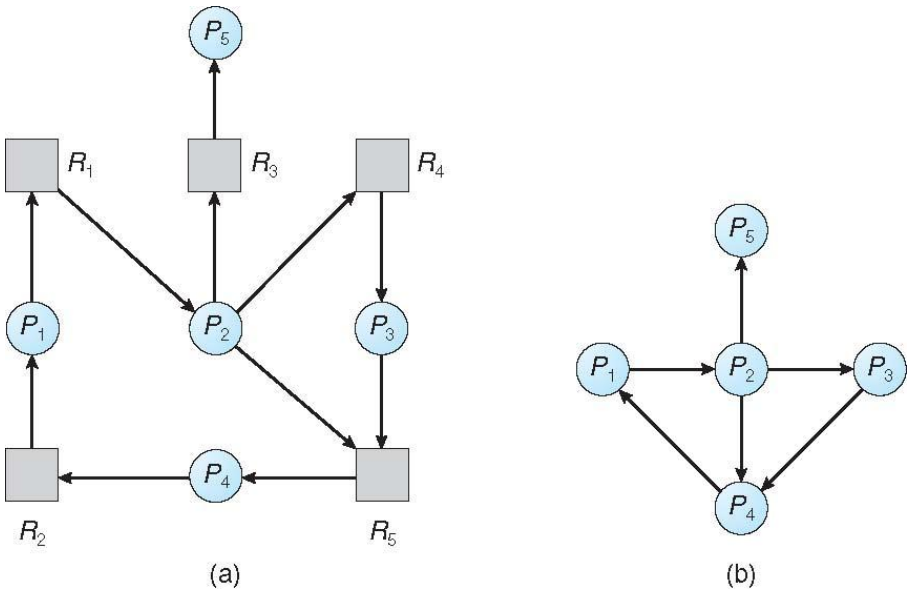
Deadlock Detection

- 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 n2 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 Pi is requesting k more instances of resource type Rj.

Detection Algorithm

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

(a) *Work = Available*

(b) *For i = 1,2, …, n, if Allocationi ≠ 0, then Finish[i] = false; otherwise, Finish[i] = true*

2. Find an index i such that both:

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

(b) *Requesti ≤ Work*

If no such i exists, go to step 4

3. Work = Work + Allocationi
Finish[i] = true
go to step 2

4. If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in 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 T0:

	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 <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i
- P2 requests an additional instance of type C

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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 P1, P2, P3, and P4\

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Break the deadlock cycle
- But at grant expense
- Abort one process at a time until the deadlock cycle is eliminated
- After each process is aborted deadlock detection algorithm must be invoked to determine whether any process still in dead lock.
- The order should we choose to abort
  1. Priority of the process
  2. How long process has computed, and how much longer to completion?

3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Which process and which resources are to be preempted.
- We must determine the order of preemption to 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

Prapared By: Shahad P., Jasheeda P

Assistant Professors,

Department of CSE,

MEA ENGINEERING COLLEGE,

PERINTHALMANNA