TOPIC 8 : DEADLOCKS

System model:

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, I/O devices are examples of resource types. If a system has 2 CPUs, then the resource type CPU has 2 instances.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its task. The number of resources as it requires to carry out its task. The number of resources requested may not exceed the total number of resources available in the system. A process cannot request 3 printers if the system has only two.

A process may utilize a resource in the following sequence:

1. **REQUEST:** The process requests the resource. If the request cannot be granted immediately (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 .if the resource is a printer, the process can print on the printer.
3. **RELEASE:** The process release the resource.

For each use of a kernel managed by a process the operating system checks that the process has requested and has been allocated the resource. A system table records whether each resource is free (or) allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

To illustrate a deadlocked state, consider a system with 3 CDRW drives. Each of 3 processes holds one of these CDRW drives. If each process now requests another drive, the 3 processes will be in a deadlocked state. Each is waiting for the event "CDRW is released" which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type. Deadlocks may also involve different resource types. Consider a system with one printer and one DVD drive. The process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION:

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

Necessary Conditions:

A deadlock situation can arise if the following 4 conditions hold simultaneously in a system:

1. **MUTUAL EXCLUSION**: Only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **HOLD AND WAIT**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by otherprocesses.
3. **NO PREEMPTION**: Resources cannot be preempted. A resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **CIRCULAR WAIT:** A set {P_0,P_1,…..P_n} of waiting processes must exist such that P_0 is waiting for resource held by P_1, P_1 is waiting for a resource held by P_2,……,P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_0.

RESOURCE ALLOCATION GRAPH

Deadlocks 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 2 different types of nodes:

P = {P_1, P_2….P_n}, the set consisting of all the active processes in the system. R= {R_1, R_2….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 $P_i \to R_j$. It signifies that process P_i has 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 $R_j \to P_i$, it signifies that an instance of resource type R_j has been allocated to process P_i.

A directed edge $P_i \to R_j$ is called a requested edge. A directed edge $R_j \to P_i$is called an assignment edge.

We represent each process P_i as a circle, each resource type R_j as a rectangle. Since resource type R_j may have more than one instance. We represent each such instance as a dot within the rectangle. A request edge points to only the rectangle R_j. An assignment edge must also designate one of the dots in therectangle.

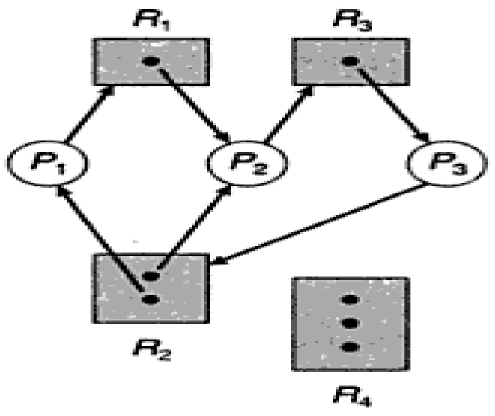
When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource, as a result, the assignment edge is deleted.

The sets P, R, E:

P= {P_1, P_2, P_3}

R= {R_1, R_2, R_3, R_4}

E= {$P_1 \to R_1$, $P_2 \to R_3$, $R_1 \to P_2$, $R_2 \to P_2$, $R_2 \to P_1$, $R_3 \to P_3$}



Resource-allocation graph with a deadlock.

One instance of resource type R1

Two instances of resource type R2

One instance of resource type R3

Three instances of resource type R4

Process States

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

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

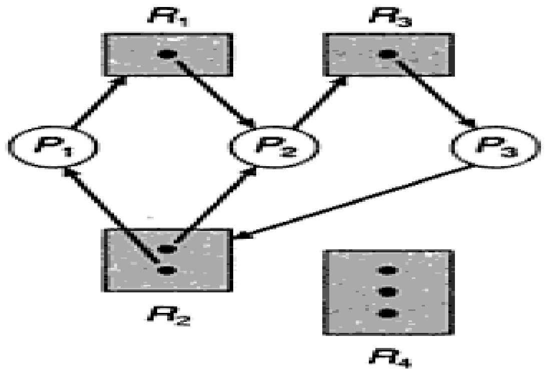
If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 ->R2 is added to the graph.

2 cycles:

P1 ->R1 ->P2 ->R3 ->P3 ->R2 ->P1

P2 ->R3 ->P3 ->R2 ->P2



Resource-allocation graph with a deadlock.

Processes P1, P2, P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3.process P3 is waiting for either process P1 (or) P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

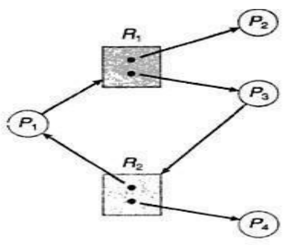


Figure Resource-allocation graph with a cycle but no deadlock.

We also have a cycle: P1 ->R1 ->P3 ->R2 ->P1. However there is no deadlock. Process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

DEADLOCK PREVENTION

For a deadlock to occur, each of the 4 necessary conditions must held. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
2. **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. Low resource utilization; starvation possible
3. **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

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 Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* 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

Avoidance algorithms

Single instance of a resource type

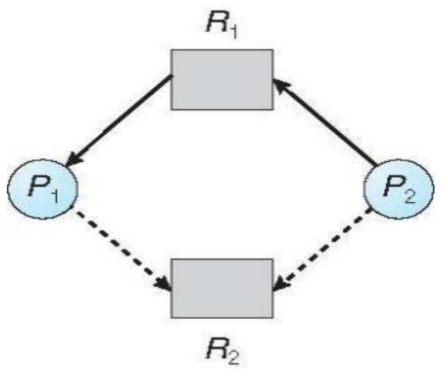
- Use a resource-allocation graph Multiple instances of a resource type
- Use the banker's algorithm

Resource-Allocation Graph Scheme

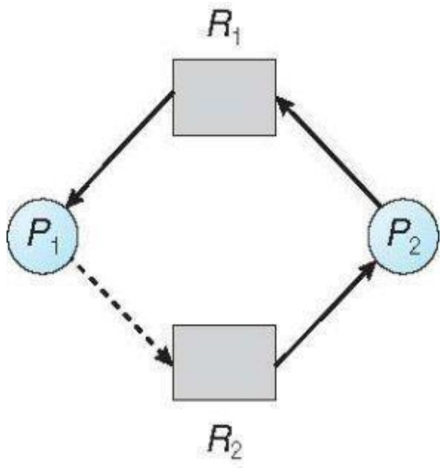
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the

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



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

Data Structures for the Banker's Algorithm

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 fori = 0, 1, …,n- 1
2. Find an isuch that both:

   (a) Finish [i] = false

   (b) Needi=Work

   If no such iexists, 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 *Pi*

Request = request vector for process *Pi*. If *Request i*[*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*

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 *Pi* must wait, since resources are not available
3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

   Available = Available – Request;

   $Allocation_i = Allocation_i + Request_i$;

   $Need_i = Need_i – Request_i$;

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 P_0 through P_4;

  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*	*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*

A B C

The system is in a safe state since the sequence <P1, P3, P4, P2, P0> 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
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

1. Allow system to enter deadlock state
2. Detection algorithm
3. Recovery scheme

Single Instance of Each Resource Type

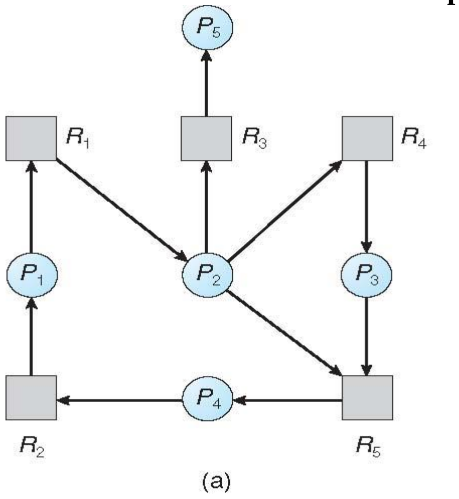
Maintain *wait-for* graph

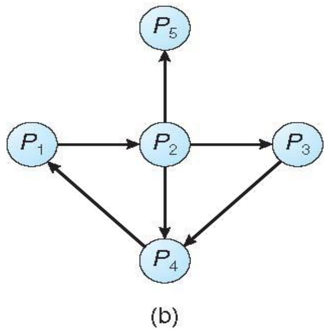
- Nodes are processes
- $P_i \rightarrow P_j$ if P_i is waiting for P_j

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:
   (a) *Work* = *Available*
   (b)For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then
   Finish[i] = false; otherwise, *Finish*[i] = *true*
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
   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*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x n^{2}) operations to detect whether the system is in deadlocked state

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 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 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:

1. **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?

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

Topic review questions

1. Define deadlock? Explain the necessary conditions for deadlock to occur.
2. List three examples of deadlocks that are not related to a computer-system environment.
3. Consider a computer system that runs 1000 jobs per month and has no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about ten jobs per deadlock. Each job is worth about two dollars (in CPU time), and the jobs terminated tend to be about half done when they are aborted. A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase of about 10 percent in the average execution time per job. Since the machine currently has 30 percent idle time, all 1000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.
    a. Discuss the arguments for installing the deadlock-avoidance algorithm into the system
    b. Discuss the arguments against installing the deadlock-avoidance Algorithm