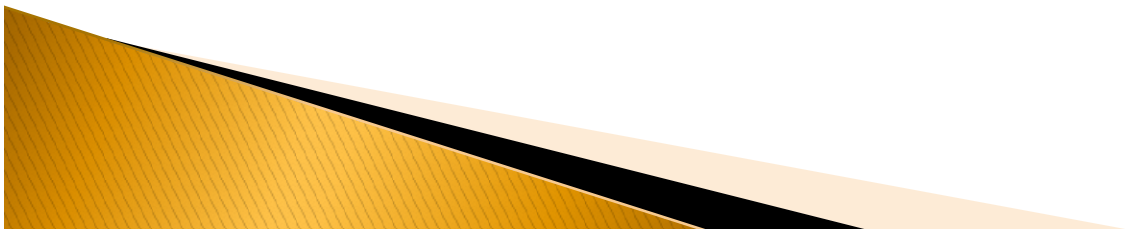


Deadlocks

Ms.Hitha Paulson
Assistant Professor, Dept of Computer Science
Little Flower College, Guruvayoor

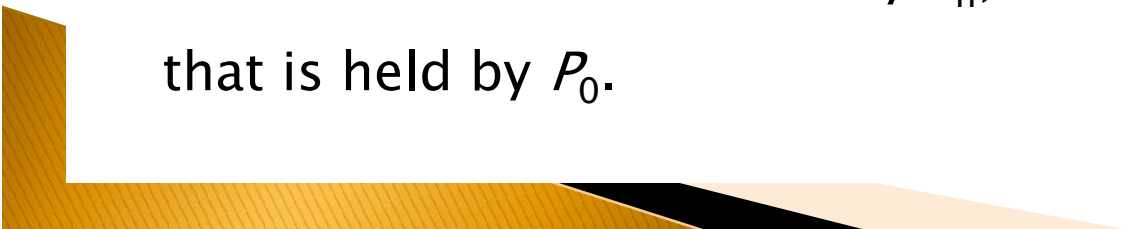
System Model

- ▶ System consists of resources
- ▶ Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- ▶ Each resource type R_i has W_i instances.
- ▶ Each process utilizes a resource as follows:
 - request
 - use
 - release



Deadlock Characterization

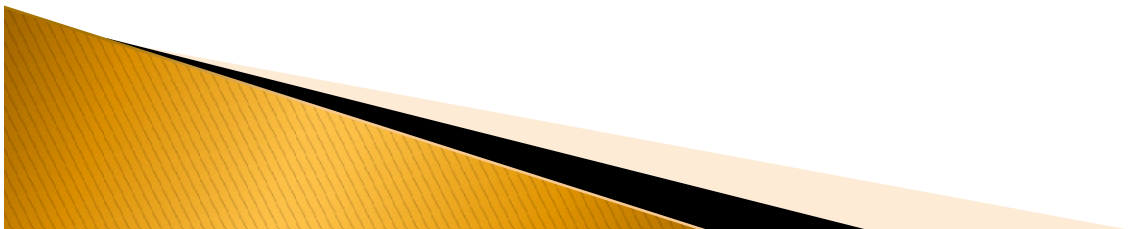
Deadlock can arise if four conditions hold simultaneously. Necessary Conditions

- ▶ **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 $\{P_0, P_1, \dots, P_n\}$ 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_n is waiting for a resource that is held by P_0 .
- 

Resource–Allocation Graph

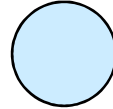
A set of vertices V and a set of edges E .

- ▶ V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- ▶ **request edge** – directed edge $P_i \rightarrow R_j$
- ▶ **assignment edge** – directed edge $R_j \rightarrow P_i$

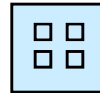


Resource-Allocation Graph

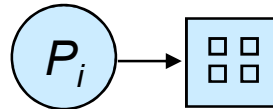
- ▶ Process



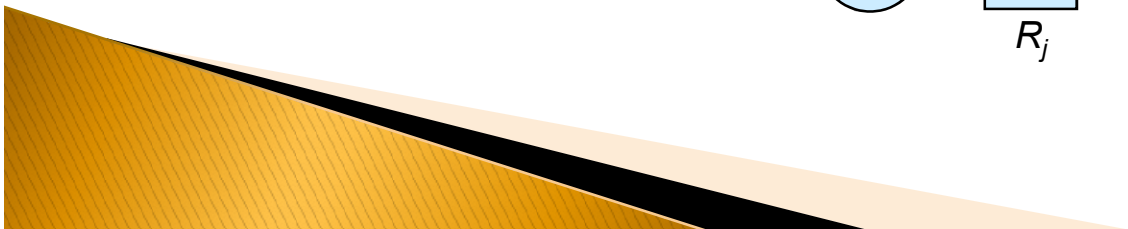
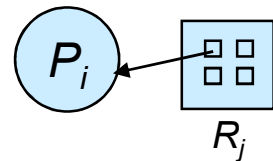
- ▶ Resource Type with 4 instances



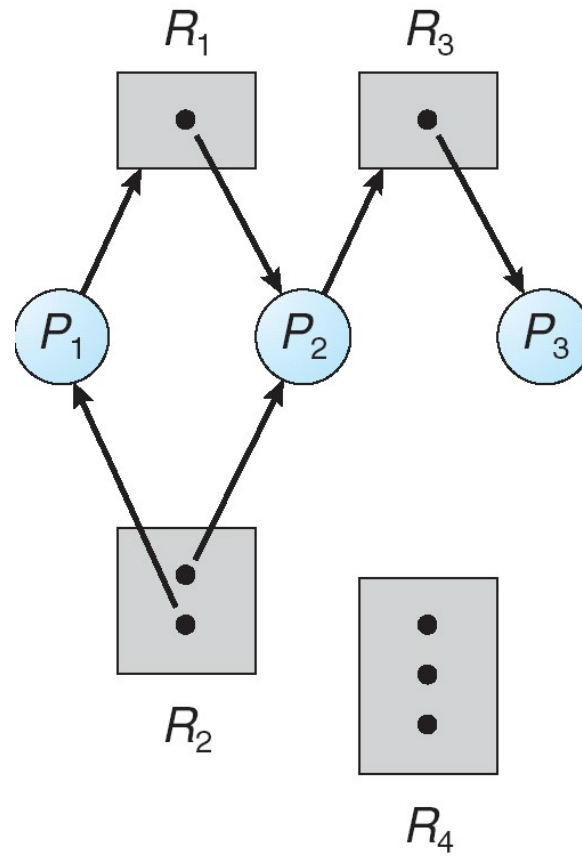
- ▶ P_i requests instance of R_j



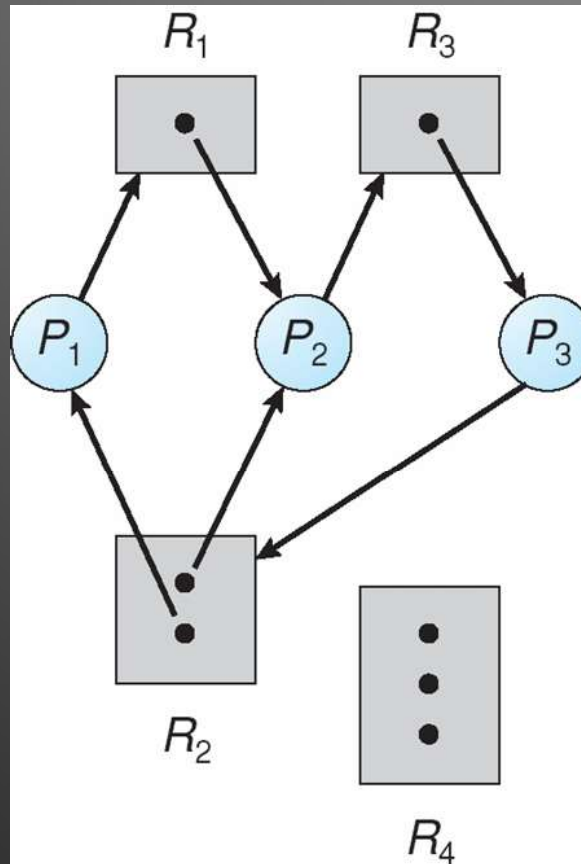
- ▶ P_i is holding an instance of R_j



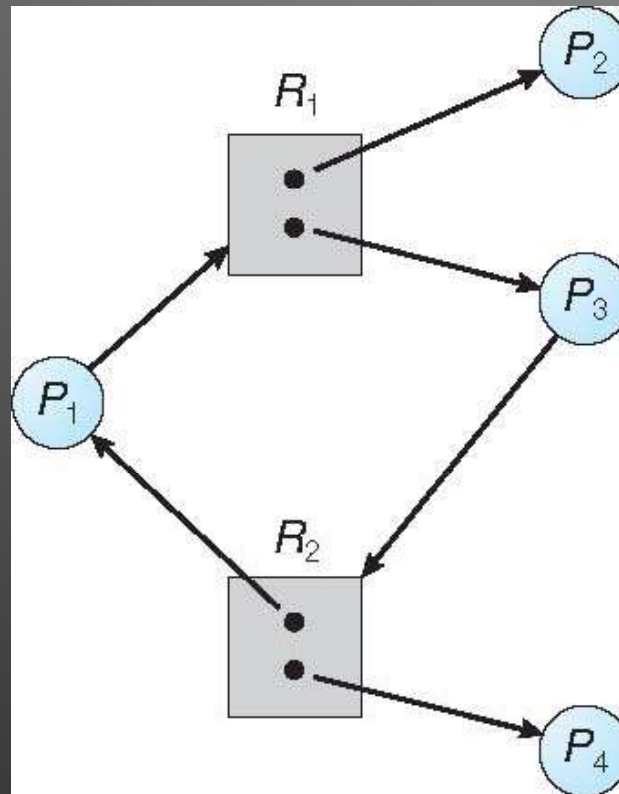
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock

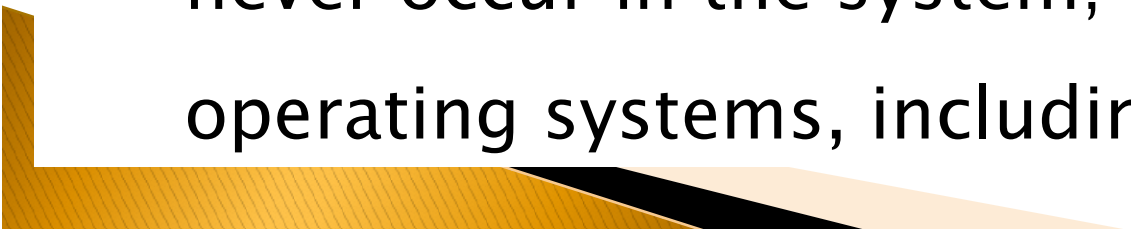


Basic Facts

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



Methods for Handling Deadlocks

1. Ensure that the system will *never* enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidance
 2. Allow the system to enter a deadlock state and then recover
 3. 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



Hold and Wait

These protocols have two main disadvantages. First, **resource utilization** may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

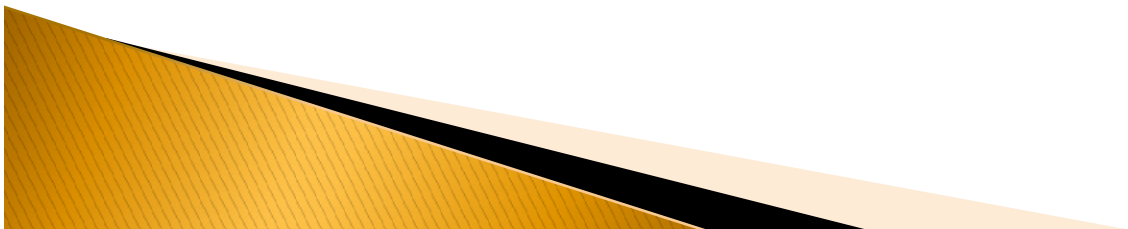
Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.




Deadlock Prevention (Cont.)

▶ 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



Deadlock Prevention (Cont.)

- ▶ No Preemption –
 - ▶ Another method is...request resource
 - ▶ Allocate if available
 - ▶ If not check whether they are allocated to other processes waiting for additional resource
 - ▶ If so preempt from waiting process and allocate to requesting process
 - ▶ If no such waiting process is there requesting process must wait and its resources are preempted if other process request them
- 

Deadlock Prevention (Cont.)

- ▶ **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



Deadlock Prevention (Cont.)

▶ Circular Wait

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:

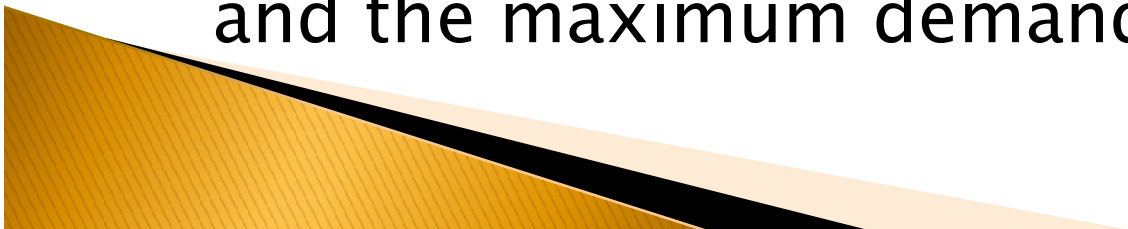
$$\begin{aligned}F(\text{tape drive}) &= 1, \\F(\text{disk drive}) &= 5, \\F(\text{printer}) &= 12.\end{aligned}$$

- ▶ Each process can request resource only in an increasing order of enumeration.
- ▶ A process having R_i can request R_j only if $F(R_j) > F(R_i)$
- ▶ Either it won't receive the resource or it releases R_i to get R_j .
- ▶ Since $F(R_0)$ is always less than $F(R_n)$ circular wait condition won't occur.

Deadlock Avoidance

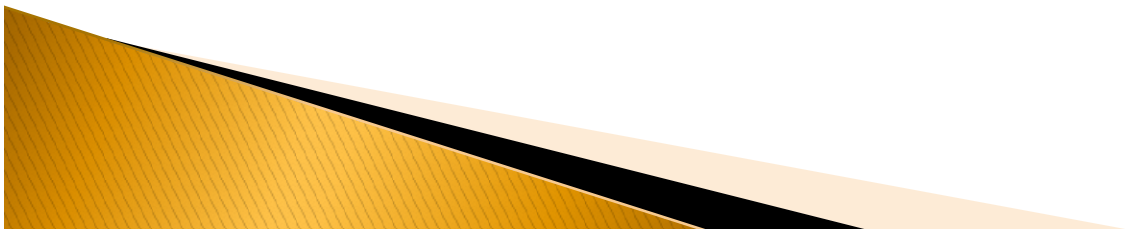
Requires that the system has some additional *a priori* information available

- ▶ 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 $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in 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 $j < i$

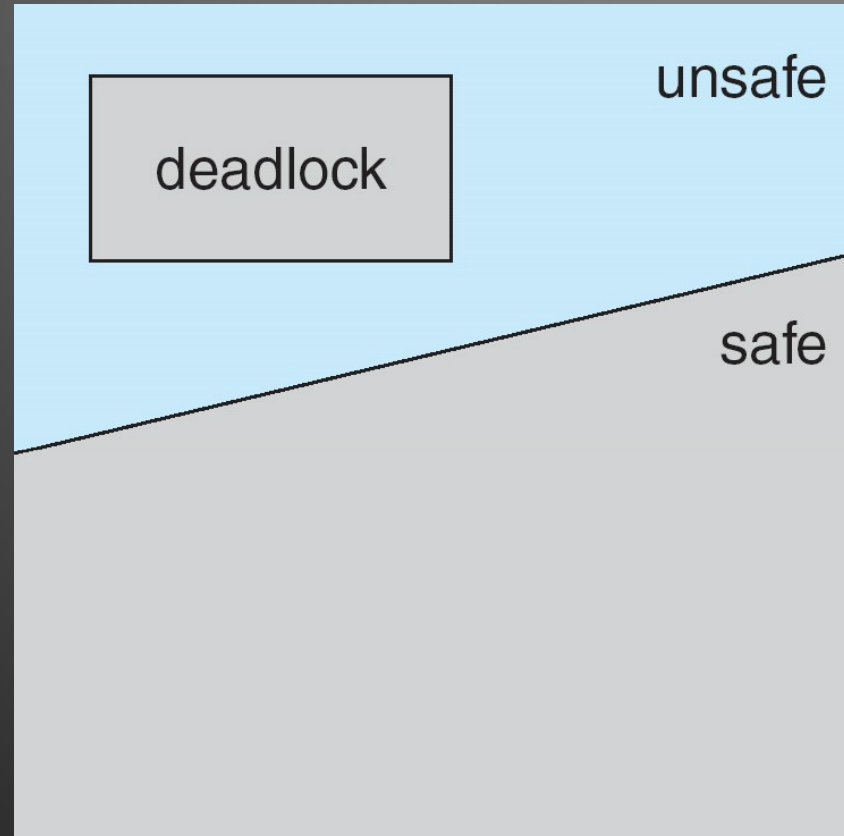


Basic Facts

- ▶ If a system is in safe state \Rightarrow no deadlocks
- ▶ If a system is in unsafe state \Rightarrow possibility of deadlock
- ▶ Avoidance \Rightarrow ensure that a system will never enter an unsafe state.



Safe, Unsafe, Deadlock State



Eg:

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: 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 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

| | <u>Maximum Needs</u> | <u>Current Needs</u> |
|-------|----------------------|----------------------|
| P_0 | 10 | 5 |
| P_1 | 4 | 2 |
| P_2 | 9 | 2 |

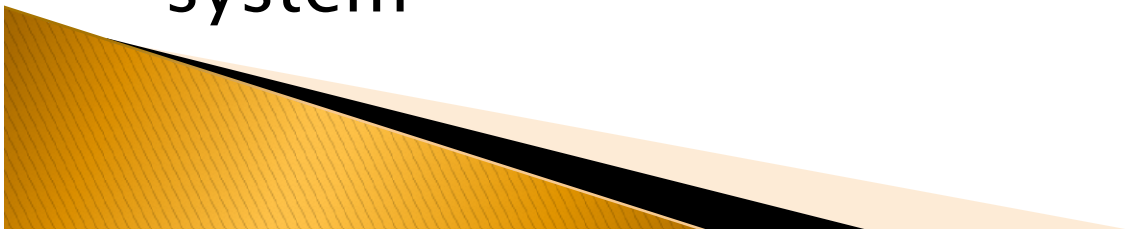
Avoidance Algorithms

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

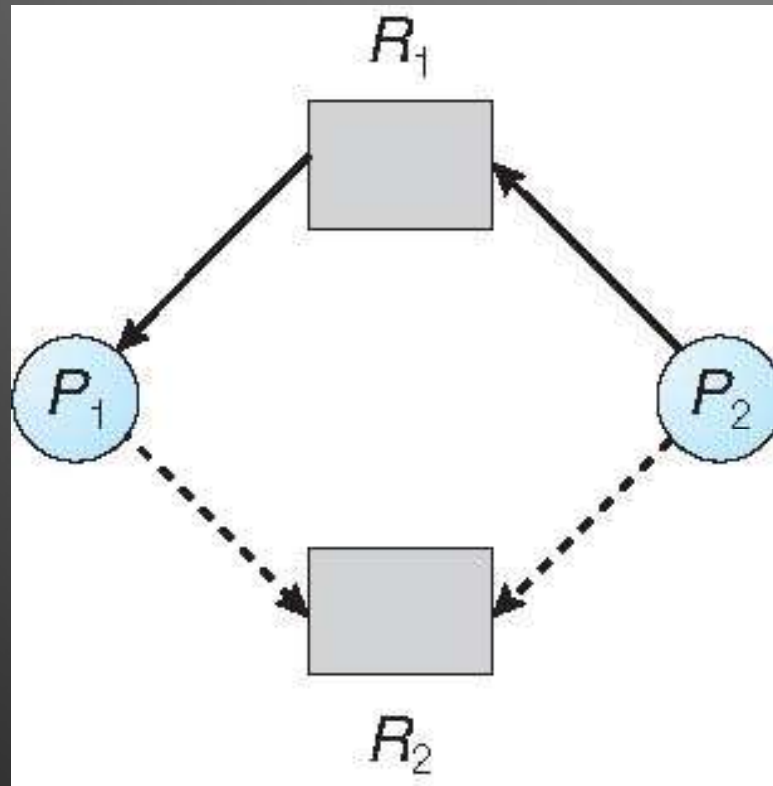


Resource–Allocation Graph Scheme

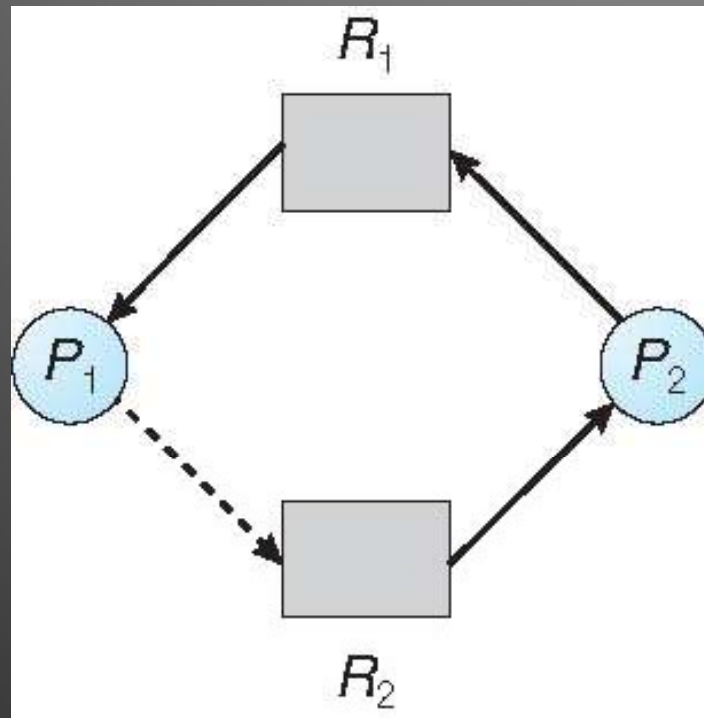
- ▶ **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line
- ▶ Claim edge converts to request edge when a process requests a resource
- ▶ Request edge converted to an assignment edge when the resource is allocated to the process
- ▶ When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



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



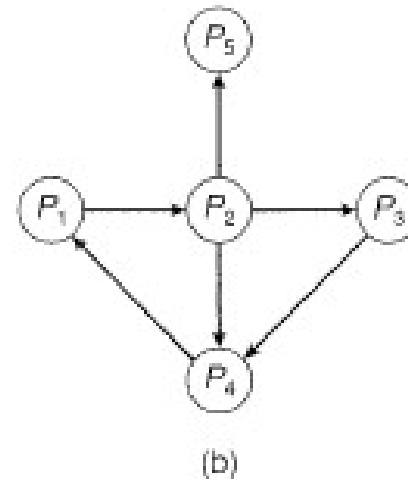
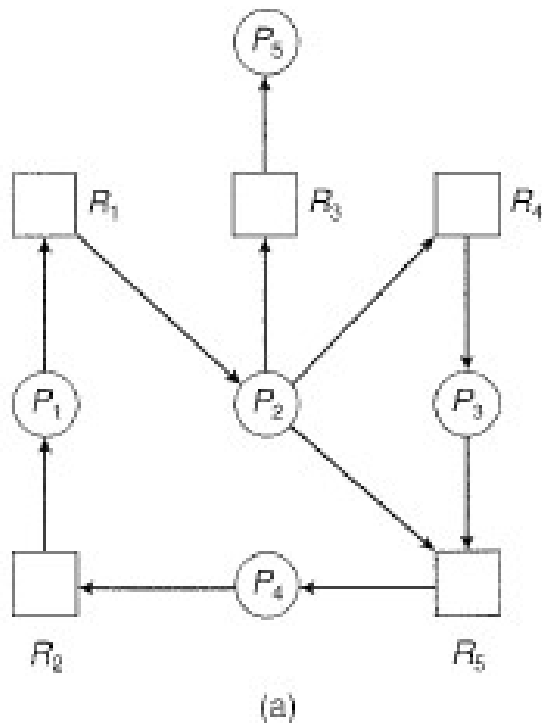
Deadlock Detection

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



Wait for graph

- ▶ Single instance resources – wait for graph method for detection

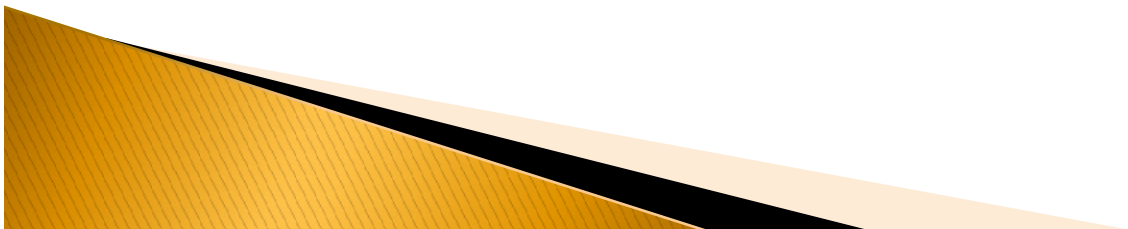


Cycle in
Wait for graph indicates
deadlock

Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Detection – Several instances

- **Available.** A vector of length m indicates the number of available resources of each type.
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request.** An $n \times m$ matrix indicates the current request of each process. If $Request[i][j]$ equals k , then process P_i is requesting k more instances of resource type R_j .

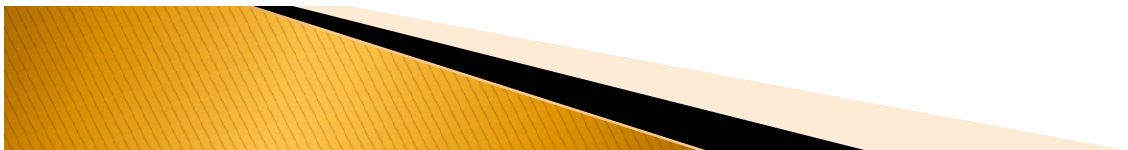


Detection Algorithm – Several instances


1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize $Work = Available$. For $i = 0, 1, \dots, n-1$, 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 $i, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.



When would we invoke this algorithm?

- ▶ we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
 - ▶ Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined interval
- 

Recovery from deadlock


- ▶ There are two options for breaking a deadlock
- ▶ One is simply to abort one or more processes to break the circular wait.
- ▶ The other is to preempt some resources from one or more of the deadlocked processes.



Process Termination

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

Many factors may affect 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 types of resources the process has used (for example, whether the resources are simple to preempt)
 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

- ▶ Issues
 - Selecting the victim
 - Rollback
 - Starvation
- ▶ No of rollbacks of the process need to be recorded

