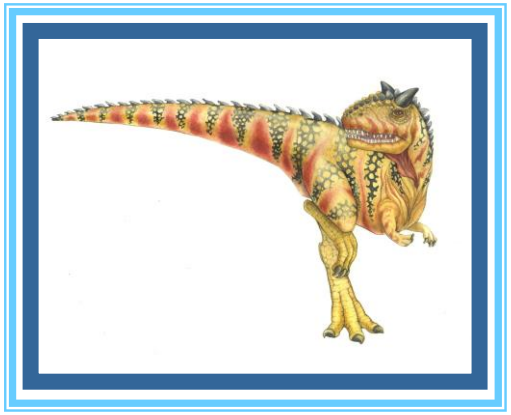
Chapter 7: Deadlocks



Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

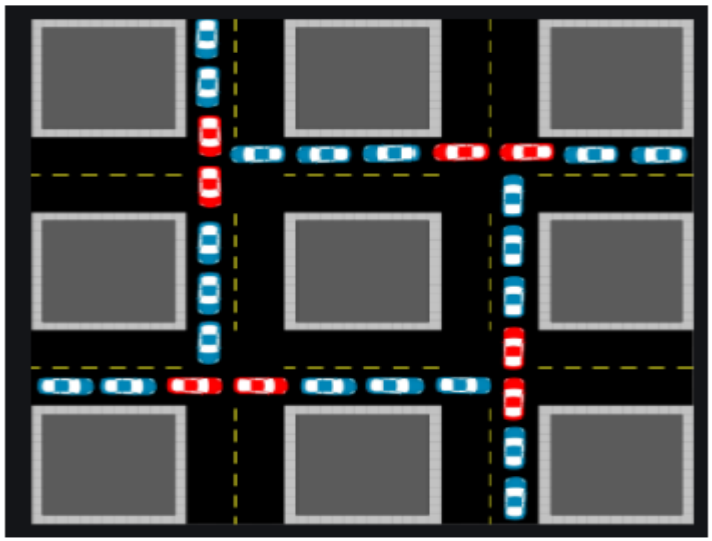
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

The Deadlock Problem

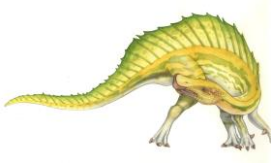
- In a multiprogramming environment, several processes may compete for a finite number of resources.

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**



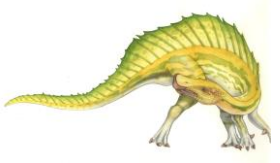
Traffic Gridlock



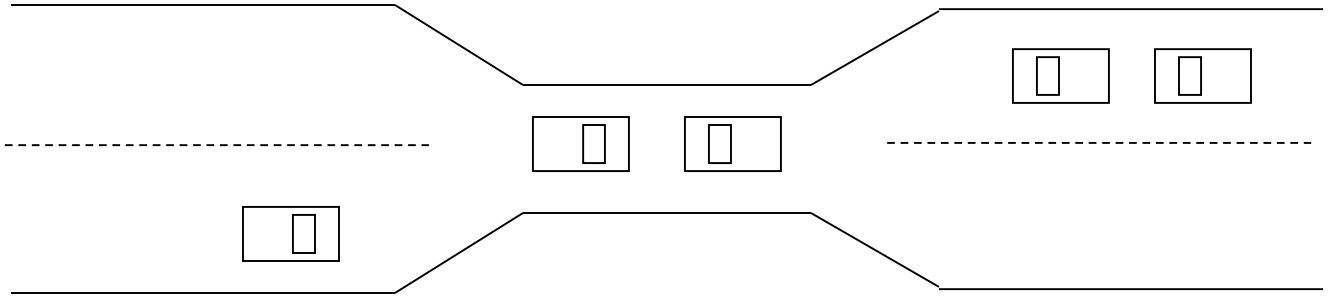
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait (B);
wait(B)	wait(A)



Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **Release**

How do you visualize the overall system from the perspective of the resources and deadlock

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

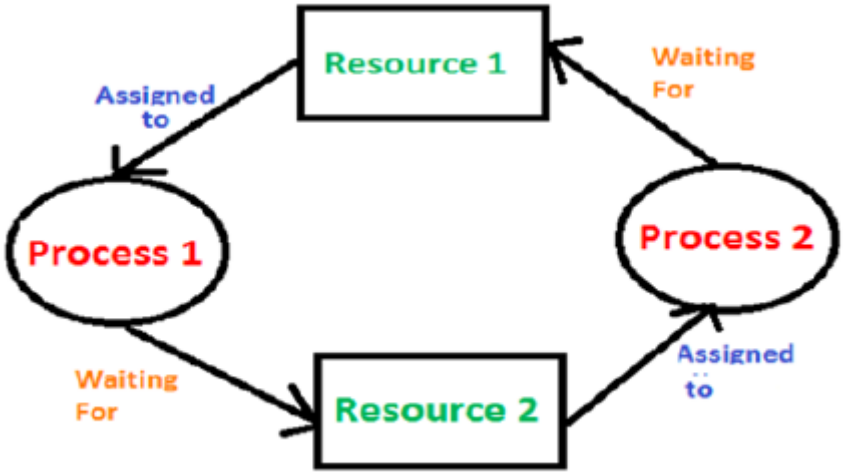
- **Mutual exclusion:** only one process at a time can use a resource
- **Non-sharable resources**
- **Eg. Printer**

- At least one resource must be held in a nonsharable 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.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

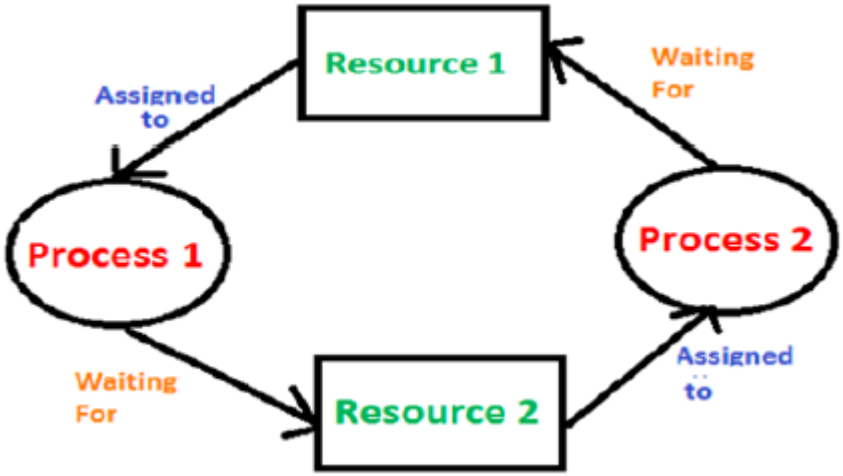
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

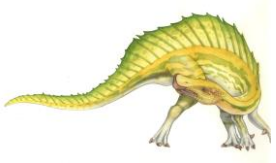


Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- If Resources are of type preemptive →No deadlock

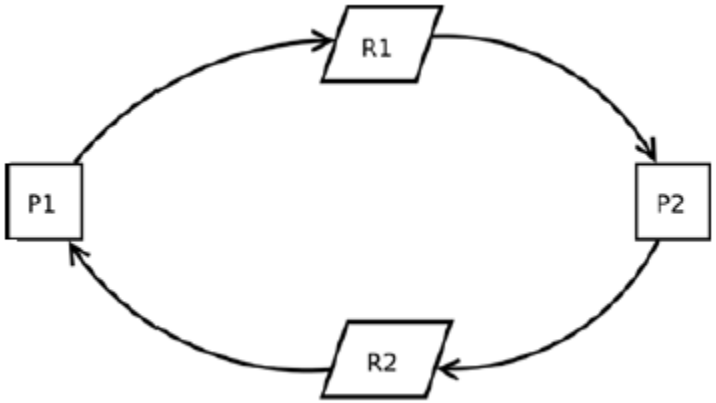




Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

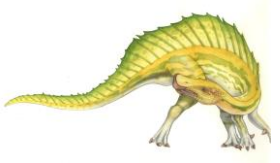
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



Resource-Allocation Graph

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

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, 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 $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

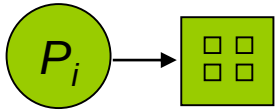


Resource-Allocation Graph (Cont.)

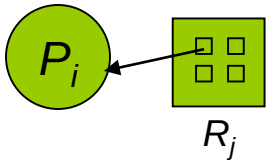
- Process

- Resource Type with 4 instances

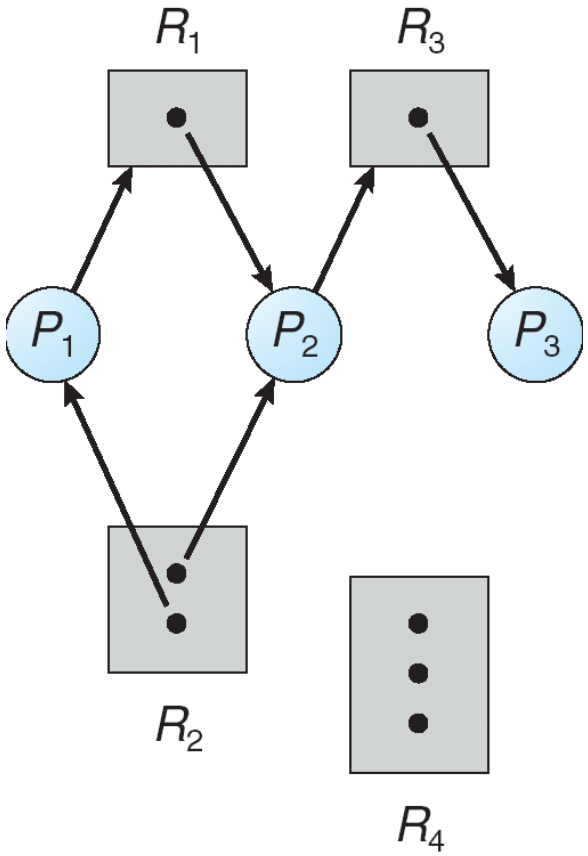
- P_i requests instance of R_j



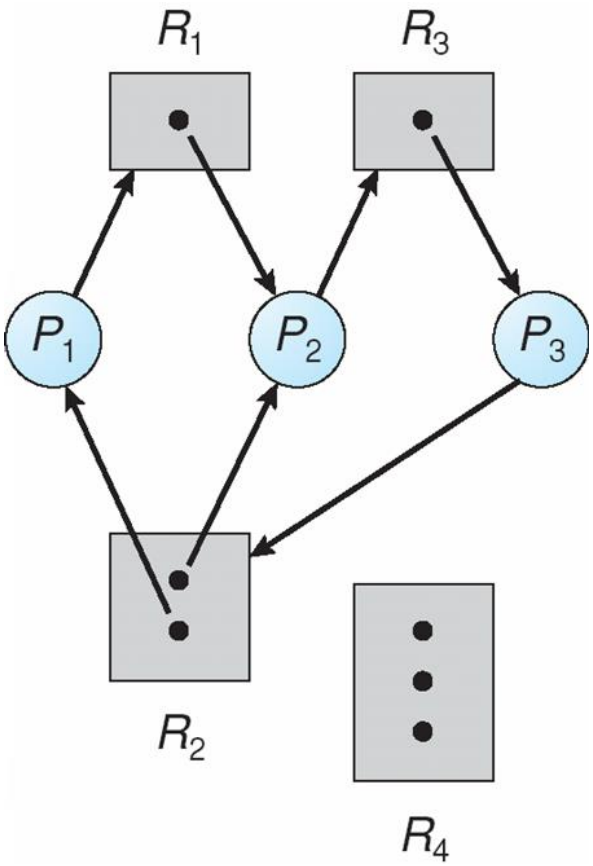
- P_i is holding an instance of R_j



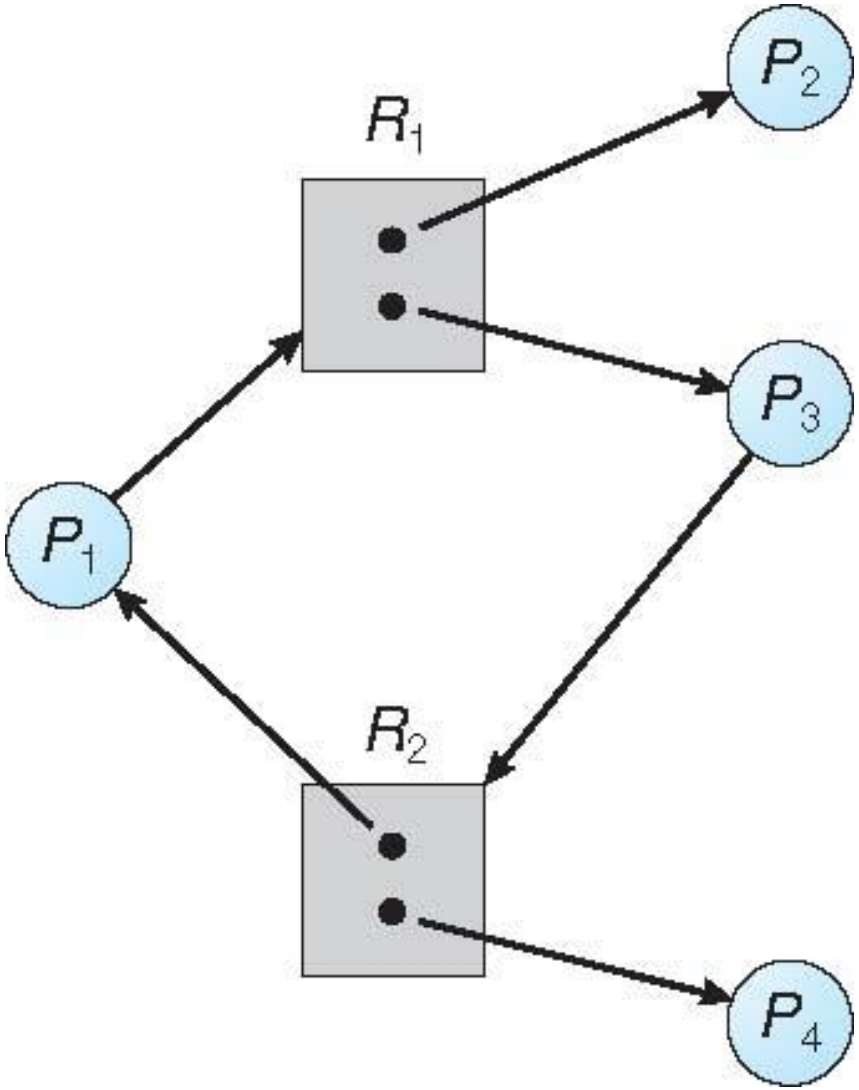
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock

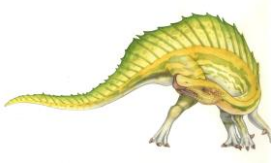


Graph With A Cycle But No Deadlock



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

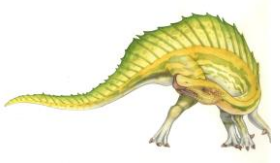
- Ensure that the system will ***never*** enter a deadlock state
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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
  - Low resource utilization; starvation possible



Deadlock Prevention (Cont.)

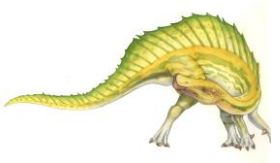
- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that can be used, requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.



Hold and Wait

- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD 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.

Hold and Wait

- Both these protocols have two main disadvantages

- First, resource utilization may be low, since resources may be allocated but unused for a long period.
- In the example given, for instance, we can release the DVD drive and disk file, and then again request the disk file and printeter only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the begin
- 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.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.

To ensure that this condition does not hold, we can use the following protocol.

If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then **all resources the process is currently holding are preempted**.

- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its **old resources, as well as the new ones that it is requesting**.

No Preemption

- Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.

  If they are not, we check whether they are allocated to some other process that is waiting for additional resources.

  If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

  If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them.

  A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

- This protocol is often applied to resources **whose state can be easily saved** and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

Circular Wait

- The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is **to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration**.

- To illustrate, we let *R* = { R1, R2, ... , Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, 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(R0)<F(R1)<F(R2)…..F(RN)<F(R0)

Circular Wait

- *F* (tape drive) = 1
- *F* (disk drive) = 5
- *F* (printer) = 12

- We can now consider
- 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;.* After that, the process can request instances of resource type *Rj* if and only if F(Rj) > F(R;). the following protocol to prevent deadlocks

- F(first_mutex)=1
- F(second_mutex)=5

Circular Wait

- For example, using the function defined previously,
- a process that wants to use the tape drive and printer at the same time must
- first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type *Rj* must have released any resources *R;* such that F(Ri) >= F(Rj). It must also be noted that if several inlstances of the same resource type are needed, a *single* request for all of them must be issued.

Deadlock Avoidance

Ensures that the system will never enter a deadlock state.

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

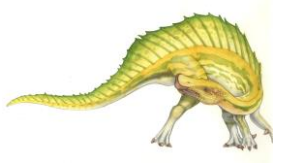
- Simplest and most useful model requires that each process ***declare the maximum number of resources of each type*** that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that **there can never be a circular-wait condition**
- Resource-allocation *state* is defined by
  - *the number of available and allocated resources,*
  - *and the maximum demands of the processes*

Safe State

- When a process requests an available resource, **system must decide if immediate allocation leaves the system in a safe state**

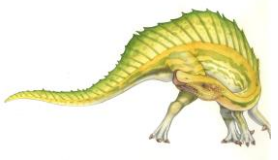
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$

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

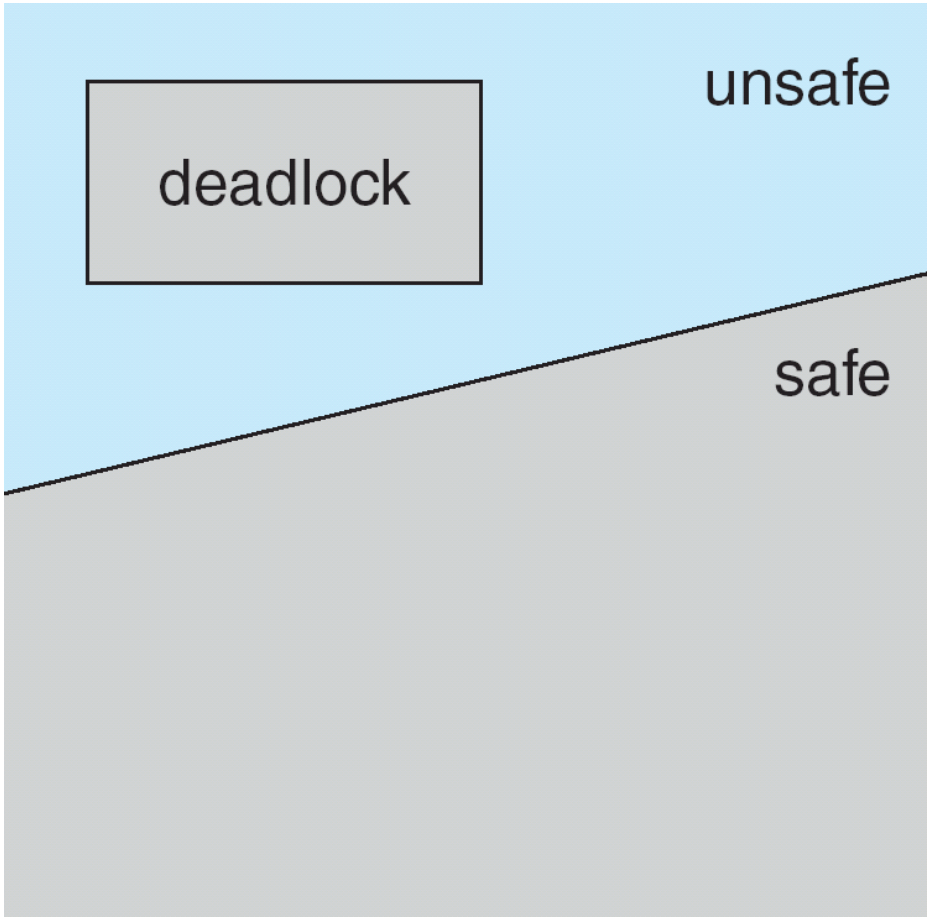


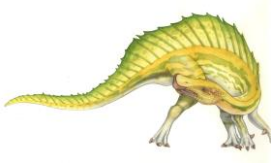
Basic Facts

- **A state is safe if the system can allocate resources(max) to each processes in some order and avoid deadlock.**
- **A system is in a safe state only if there exists a safe sequence.**
- 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





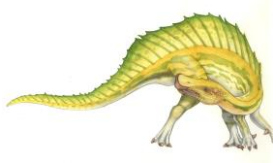
Avoidance algorithms

Avoidance ⇒ ensure that a system will never enter an unsafe state.

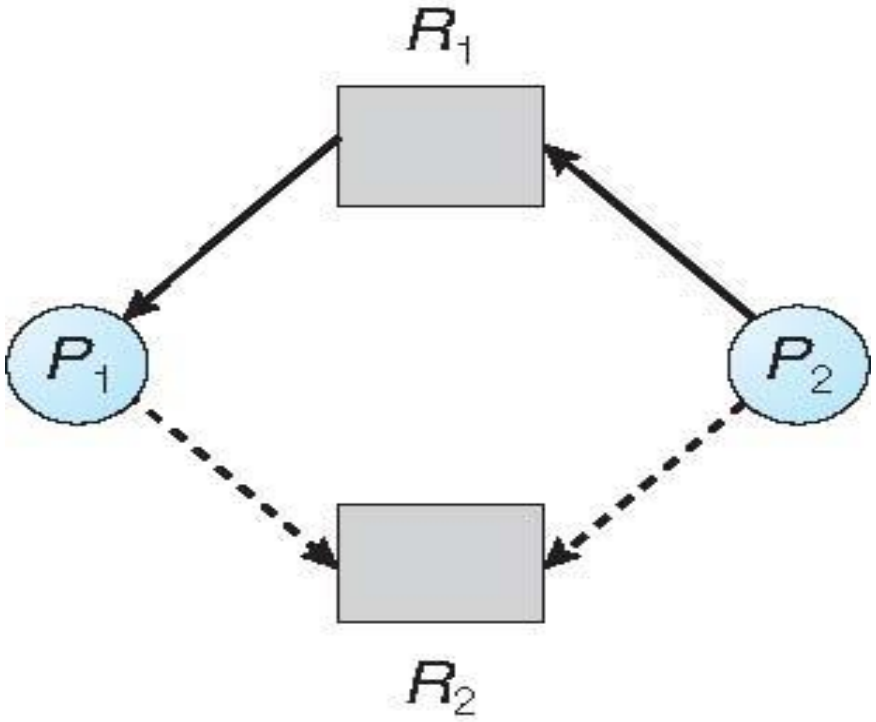
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

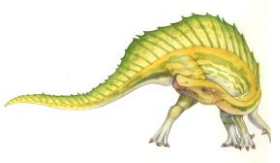
- **Single instance of a resource type**
- **Each process must a priori claim maximum resource use.**
- **Use a variant of the resource allocation graph with claim edges.**
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line
- **Claim 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
- ***A cycle in the graph implies that the system is in unsafe state*.**



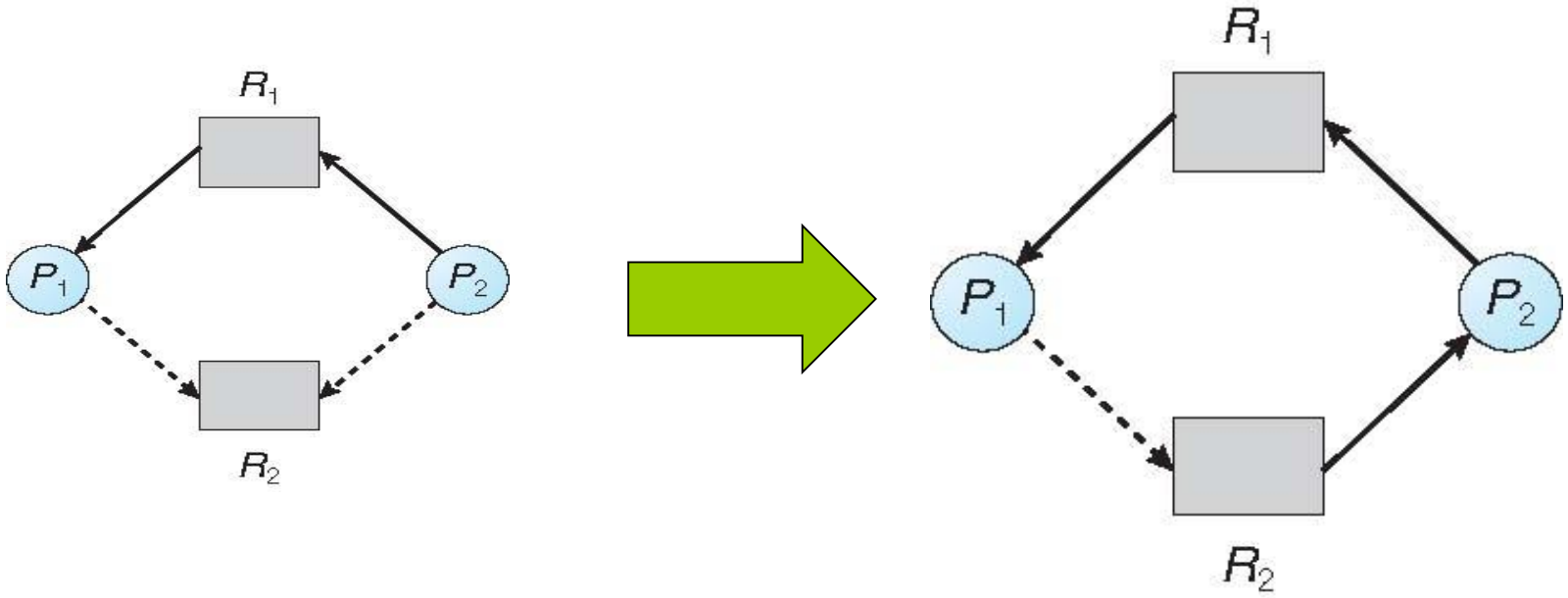
Resource-Allocation Graph



- **P1 is holding resource R1 and has acclaim on R2.**
- **P2 is requesting R1 and has a claim on R2.**
- **No cycle, so system is in safe state.**



Unsafe State In Resource-Allocation Graph



- The graph on the last slide with a **claim edge from P2 to R2 is changing to an assignment edge.**
- There is a cycle in the graph-→ unsafe state.
- Is there is a deadlock?

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
- Otherwise, the process must wait

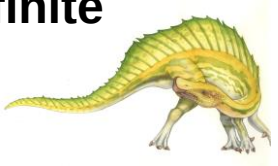
Deadlock Avoidance

- Is there an algorithm that can always avoid deadlocks by conservatively make the right choice
  - Ensures system never reaches an unsafe state
  - Safe State: A state is said to be safe, if there is some scheduling order in which every process can run to completion even if all of them suddenly requests their maximum number of resources immediately
  - An unsafe state does not have to lead to a deadlock**; it could lead to a deadlock**

Deadlock Avoidance:Banker's Algorithm

- Multiple instances of a resource type
- 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

- Eg: Interest free bank where:
- A customer establishes a line of credit.
- Borrows money in chunks that together never exceed the total line of credit.
- Once it reaches the maximum, the customer must pay back in finite amount of time,

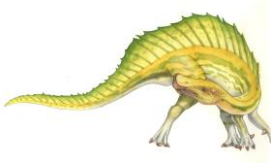


Data Structures for the Banker's Algorithm

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

- **Available:** Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max:** n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation:** n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need:** n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$



Deadlock Avoidance

Example with a Banker

- Consider a banker with 3 clients (A, B, C).
  - Each client has certain credit limits (totaling 20 units)
  - The banker knows that max credits will not be used at once, so he keeps only 10 units

	Has	Max
A	3	9
B	2	4
C	2	7

Total : 10 units

free : 3 units

  - Clients declare maximum credits in advance. The banker can allocate credits provided no unsafe state is reached.

Deadlock Avoidance

Safe State

	Has	Max
A	3	9
B	2	4
C	2	7

free : 3 units

Allocate 2 units to B

	Has	Max
A	3	9
B	4	4
C	2	7

free : 1 units

B completes

	Has	Max
A	3	9
B	0	-
C	2	7

free : 5 units

Allocate 5 to C

	Has	Max
A	3	9
B	0	-
C	7	7

free : 0 units

C completes

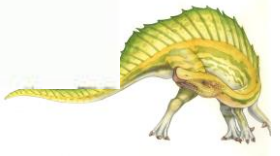
	Has	Max
A	3	9
B	0	-
C	0	-

free : 7 units

Allocate7 units to A

	Has	Max
A	9	9
B	0	-
C	0	-

free : 1 units



Deadlock Avoidance

Unsafe State

Allocate 2 units to B

B completes

	Has	Max
A	4	9
B	2	4
C	2	7

free : 2 units

	Has	Max
A	4	9
B	4	4
C	2	7

free : 0 units

	Has	Max
A	4	9
B	0	-
C	2	7

free : 4 units

This is an unsafe state because there exists NO scheduling order in which every process executes

Safety Algorithm

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

   Work = *Available*

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

2. Find an *i* such that both:

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

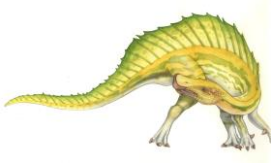
   (b) $Need_i \leq Work$

   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*] == true for all *i*, then the system is in a safe state

   Otherwise, system is in unsafe state.



Example of Banker's Algorithm

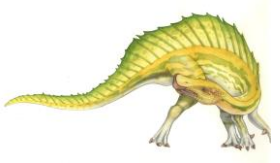
- 5 processes P_0 through P_4;

  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	

1. What is the content of the Need matrix?
2. Is the system in a safe state?



Example (Cont.)

- Is the system in a safe state?

	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	

	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

Finish

p0	p1	p2	p3	p4
F	F	F	F	F

Example (Cont.)

- 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

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 ***Request***$_i \leq$ ***Need***$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request***$_i \leq$ ***Available***, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

(Update the state table after performing resource request algorithm, check if new state is safe or not using Banker's algorithm)

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

Example: P_1 Request (1,0,2)

- 5 processes P_0 through P_4; 3 resource types: A (10 instances), B (5instances), and C (7 instances)
- **Check that Request ≤ Need (that is, (1,0,2) ≤ (1,2,2) ⇒ true**
- **Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true**

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- Snapshot at time T_0:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Example: P_1 Request (1,0,2)

	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

Example: P_4 Request (3,3,0)

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

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

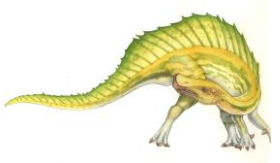
A request for (3,3,0) by P_4 cannot be granted, since the resources are not available.

Example: P_0 Request (0,2,0)

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

A request for (0,2,0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.



Example

Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

Example

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Example

a. The values of **Need** for processes P_0 through P_4 respectively are (0, 0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0), and (0, 6, 4, 2).

b. The system is in a safe state? Yes. With **Available** being equal to (1, 5, 2, 0), either process P_0 or P_3 could run. Once process P_3 runs, it releases its resources, which allow all other existing processes to run.

c. The request can be granted immediately? This results in the value of **Available** being (1, 1, 0, 0). One ordering of processes that can finish is P_0, P_2, P_3, P_1, and P_4.

Example

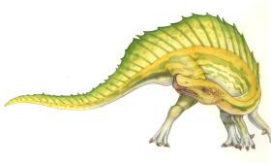
Consider the following snapshot of a system:

	Allocation	*Max*
	A B C D	*A B C D*
P_0	3 0 1 4	5 1 1 7
P_1	2 2 1 0	3 2 1 1
P_2	3 1 2 1	3 3 2 1
P_3	0 5 1 0	4 6 1 2
P_4	4 2 1 2	6 3 2 5

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe.

a. *Available* = (0, 3, 0, 1)

b. *Available* = (1, 0, 0, 2)



Example

Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	2 0 0 1	4 2 1 2	3 3 2 1
P_1	3 1 2 1	5 2 5 2	
P_2	2 1 0 3	2 3 1 6	
P_3	1 3 1 2	1 4 2 4	
P_4	1 4 3 2	3 6 6 5	

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.

b. If a request from process P_1 arrives for $(1, 1, 0, 0)$, can the request be granted immediately?

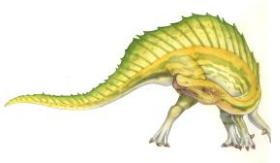
c. If a request from process P_4 arrives for $(0, 0, 2, 0)$, can the request be granted immediately?

Deadlock Detection

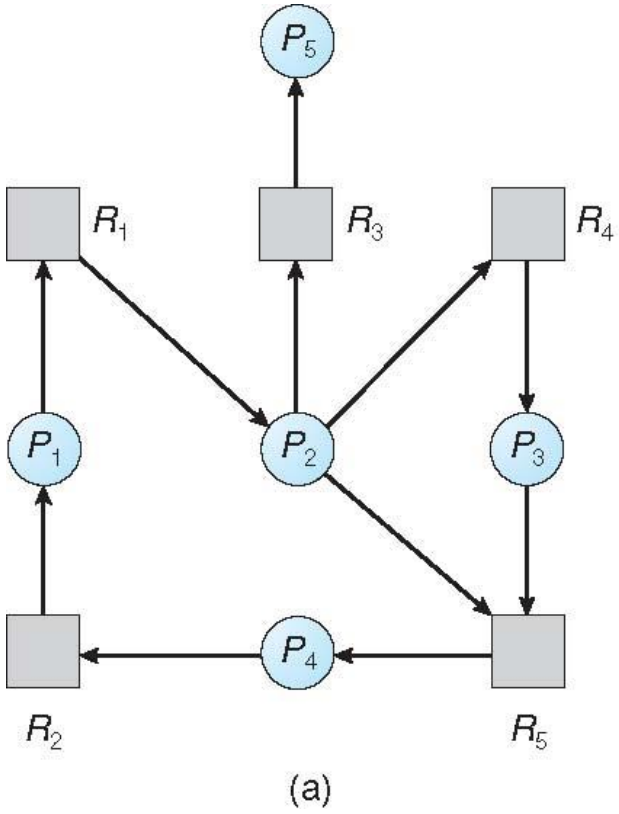
- Allow system to enter deadlock state
- Detection algorithm:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred
- Recovery scheme:
  - An algorithm to recover from the deadlock

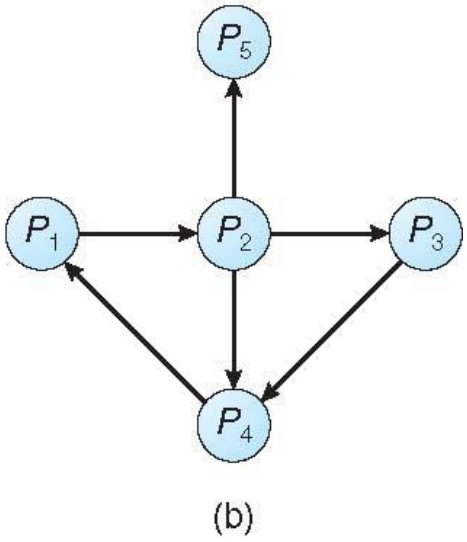
Single Instance of Each Resource Type

- 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 $i = 1,2, \ldots, n$, 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

Detection Algorithm (Cont.)

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

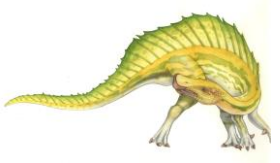
Example of Detection Algorithm

- 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*

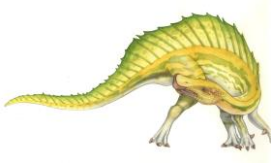


Example (Cont.)

- P_2 requests an additional instance of type *C*

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

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

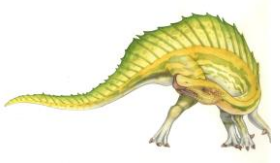


Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes**
- **Abort one process at a time until the deadlock cycle is eliminated**
- **In which order should we choose to abort?(Decision policy:- Terminate the process which incur less cost of operation)**
  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?**

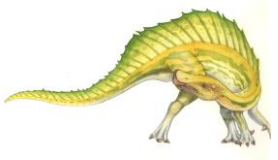


Recovery from Deadlock: Resource Preemption

- **Selecting a victim – 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**
  - Ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Summary

- A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- There are three principal methods for dealing with deadlocks:
  - Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
  - Allow the system to enter a deadlocked state, detect it, and then recover.
  - Ignore the problem altogether and pretend that deadlocks never occur in the system.
- The third solution is the one used by most operating systems, including Linux and Windows.



End of Chapter 7