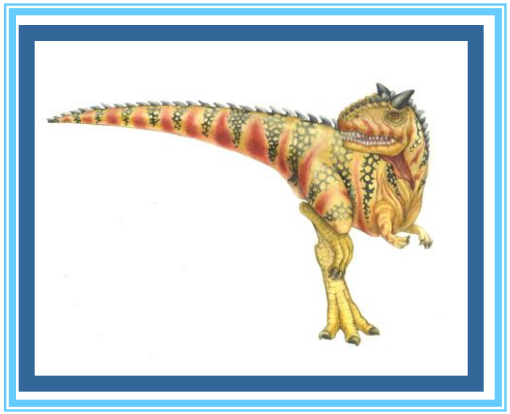
Chapter 7: Deadlocks



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

Chapter 7: Deadlocks

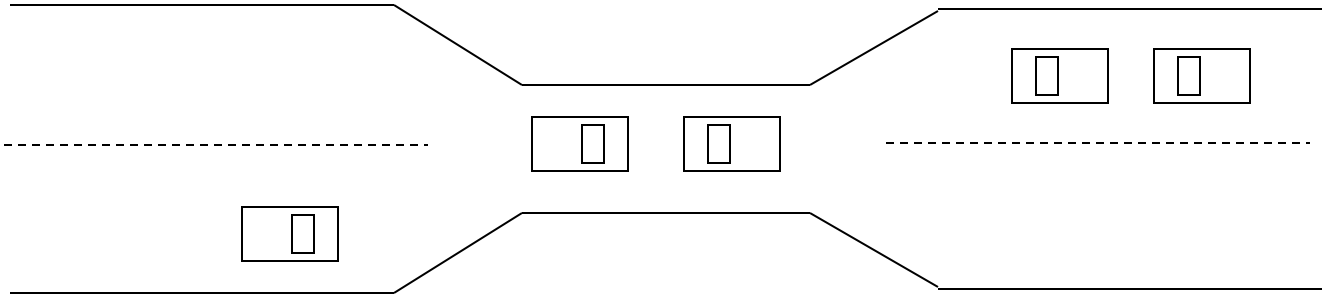
- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlock

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

Deadlock Characterization

Necessary Conditions:

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 $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

No Deadlock Situation

If you can *prevent at least one of the necessary deadlock conditions* then you won't have a DEADLOCK

Resource-Allocation Graph

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

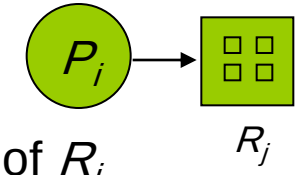
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge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

Resource-Allocation Graph (Cont.)

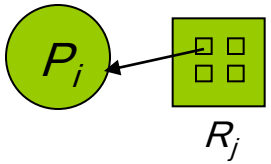
- Process

- 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

1- The sets P,R,and E

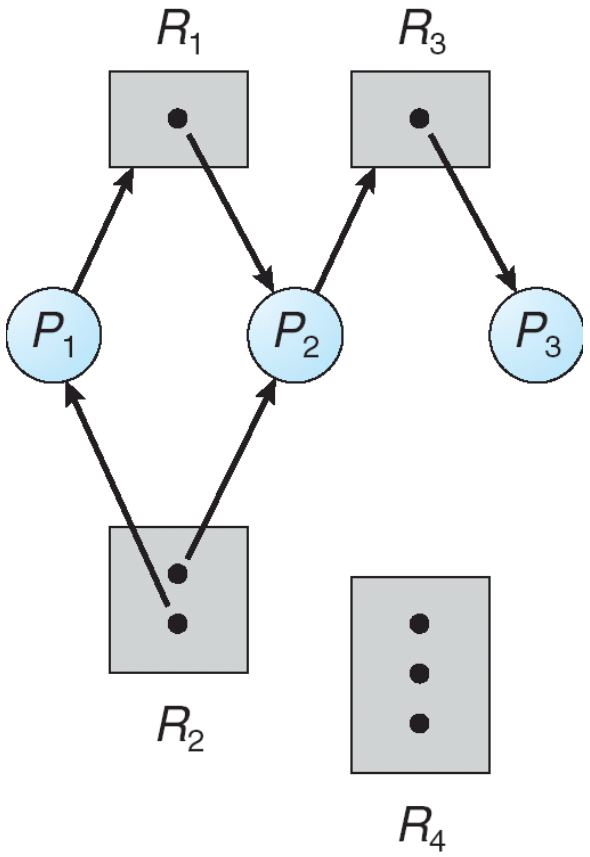
P= {P1,P2,P3}
R={R1,R2,R3,R4}
E={P1→ R1, P2→R3 ,R1→P2, R2→P2,
R2→P1, R3→P3}

2- Resource instances

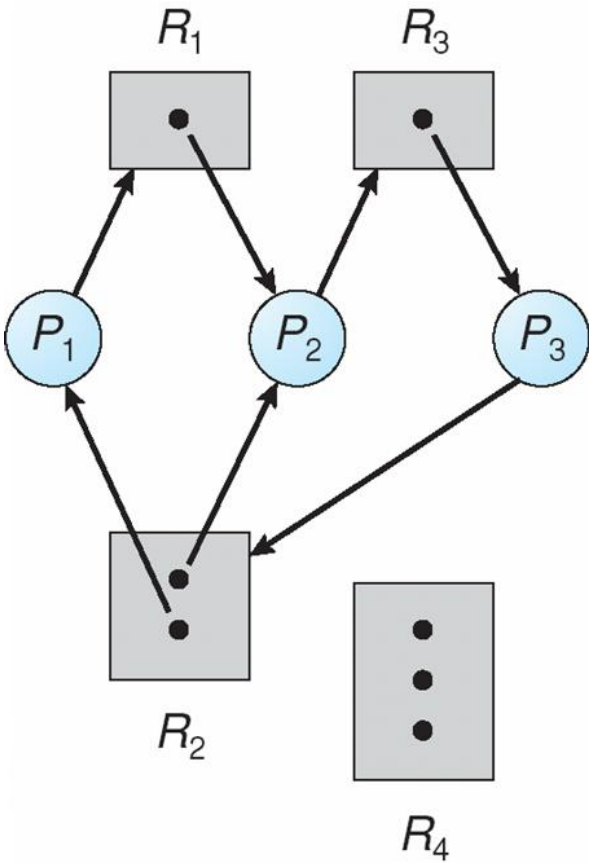
- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

3- Process states

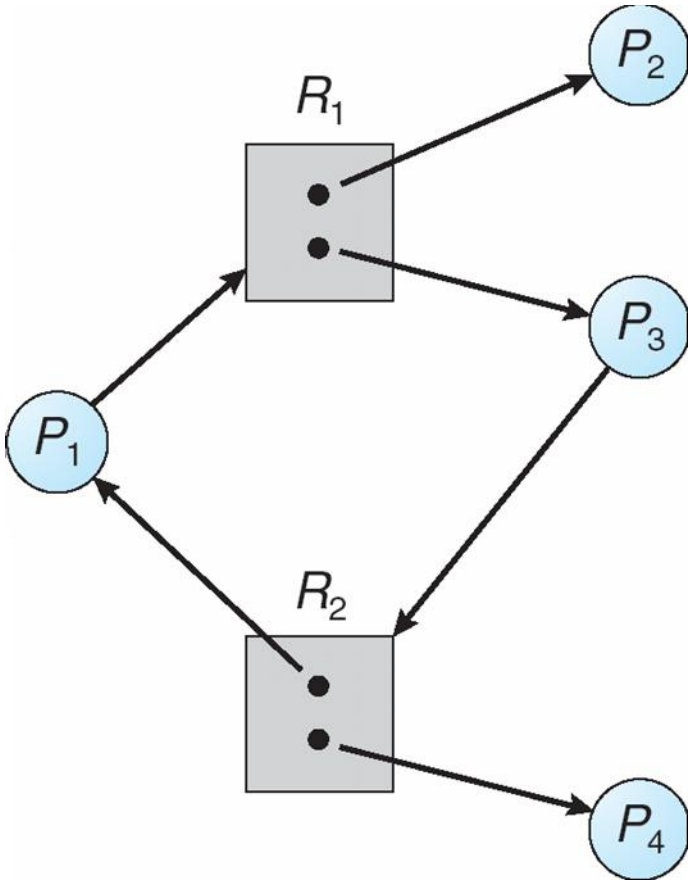
- Process P1 is holding an instances of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and waiting for an instances of R3.
- Process P3 is holding an instance of R3.



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

- Use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state, **detect** it, and **recover**
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Remove the possibility of deadlock occurring by denying one of the four necessary conditions:

- **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 by using one of the following protocols:
  1. Require process to request and be allocated all its resources before it begins execution.
  2. 0r allow process to request resources only when the process has none
  - Problems
    - Low resource utilization;
    - starvation possible

Deadlock Prevention (Cont.)

Remove the possibility of deadlock occurring by denying one of the four necessary conditions:

- **No Preemption** –
  1. 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
  2. Preempted resources are added to the list of resources for which the process is waiting
  3. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
  - It is applied to resources whose state easily saved and restored later, such as CPU registers and memory space
  - The main *Advantages* is more better resource utilization
  - Problems
    - The cost of removing a process's resources
    - starvation possible

Deadlock Prevention (Cont.)

Remove the possibility of deadlock occurring by denying one of the four necessary conditions:

- **Circular Wait** –
  - Resources are uniquely numbered
  - Processes can only request resources in linear ascending order
  - Problems
    - Resources must be requested in ascending order of resource number rather than as needed
    - Resource numbering must be maintained by someone and must reflect every addition to the OS
    - Difficult to sit down and write code

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

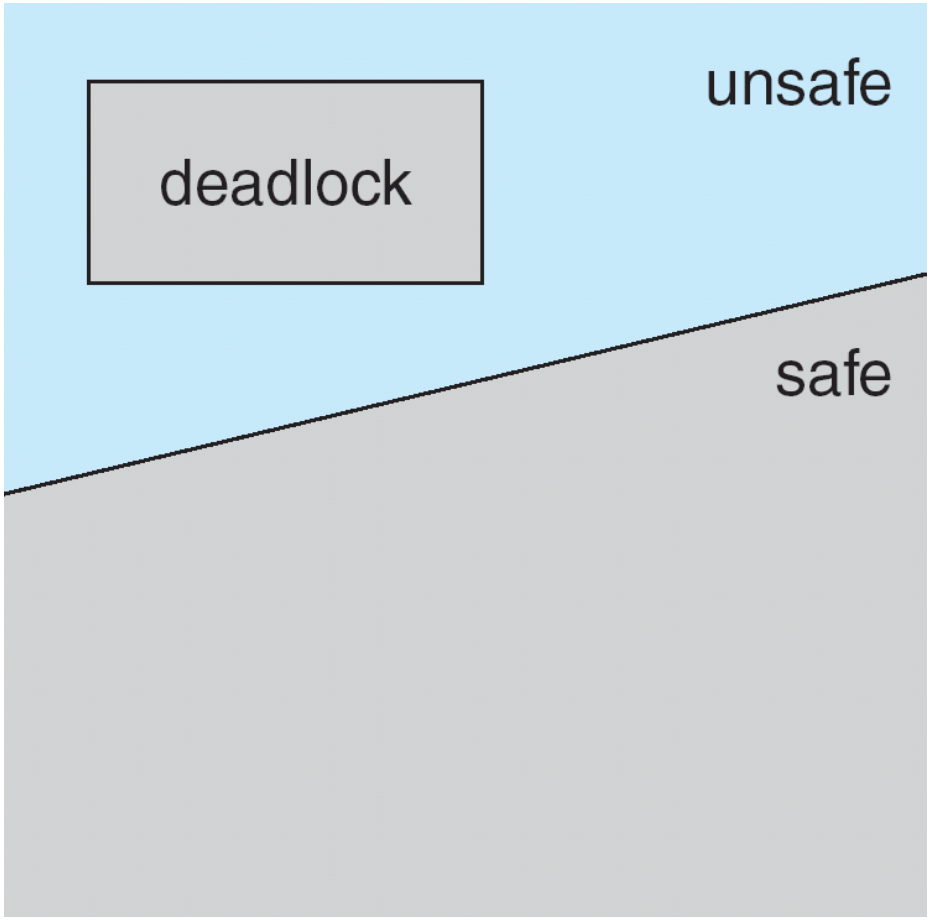
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ 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

- 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



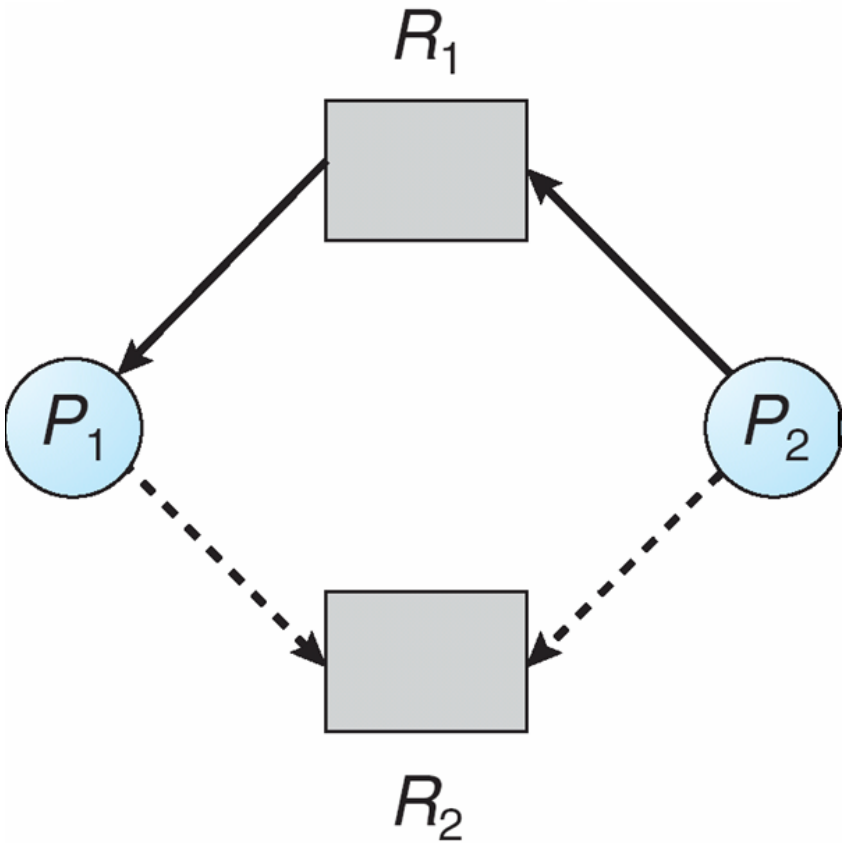
Avoidance algorithms

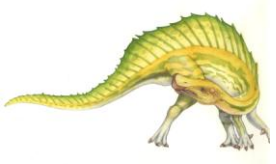
- The avoidance algorithms ensure that the system will always remain in a safe state.
  - Single instance of a resource type
    - Use a resource-allocation graph
  - Multiple instances of a resource type
    - Use the banker's algorithm

Resource-Allocation Graph Scheme

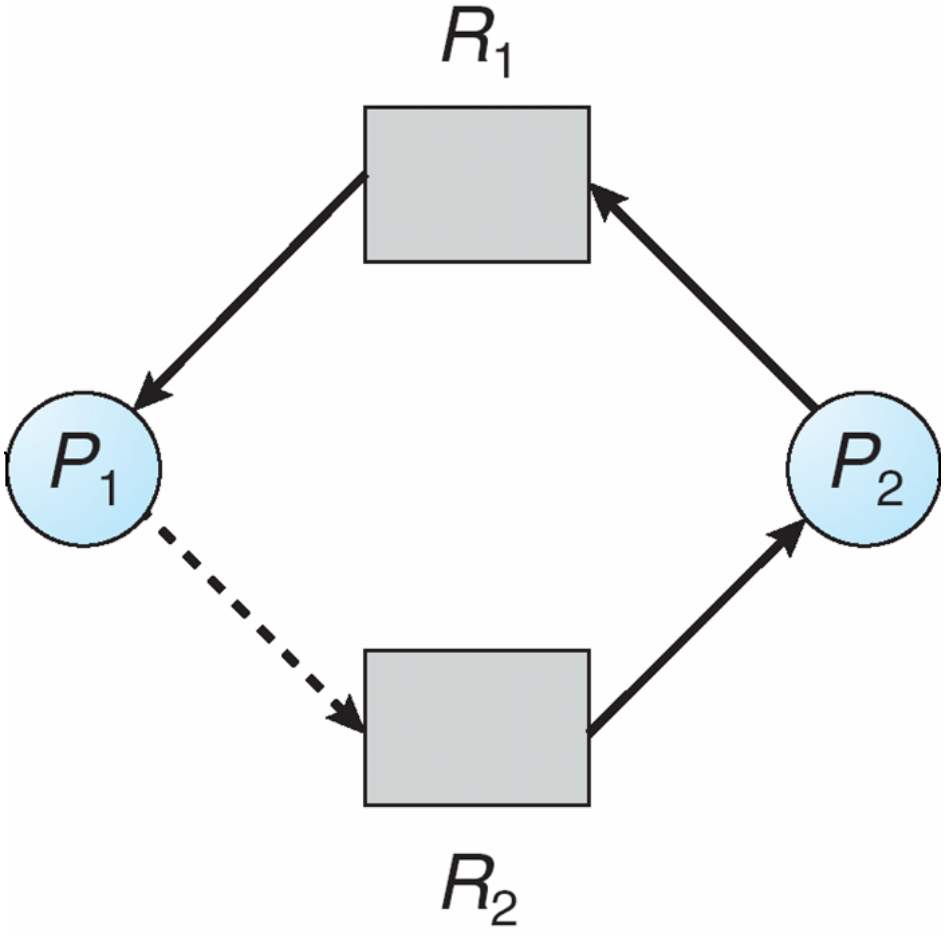
- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource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

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use, When a new process enters a system, it must declare the maximum number of instances of each resource type that may not exceed the total number of resources in the system
- 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 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

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

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 index *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

Resource-Request Algorithm for Process P_i

Request = 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 $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \leq Available}$, 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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *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

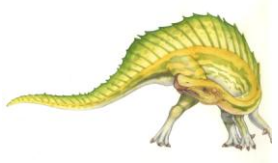
- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5 instances), 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	

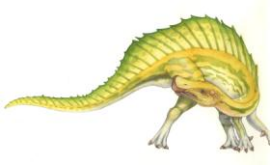


Example (Cont.)

- 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



Example: P_1 Request (1,0,2)

1- Check that Request ≤ Need (that is, (1,0,2) ≤ (1,2,2) ⇒ true

2- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

3- *Available* = *Available* – *Request* => (3,3,2) - (1,0,2) = (2,3,0)

Allocation = *Allocation* + *Request* => (2,0,0) - (1,0,2) = (3,0,2)

Need = *Need* – *Request* => (1,2,2) - (1,0,2) = (0,2,0)

	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	

Example: P_1 Request (1,0,2) cont.

- Executing safety algorithm:
  - Check P_0
    - Step 1: Work = 2 3 0

      Finish[0] = False
    - Step 2 : (a) *Finish* [*0*] = *false*

      (b) $Need_0 \leq Work$

      7 4 3 ≤ 2 3 0

      so there is no i exist
    - Step 3 : skip it
    - P_0 no finish (no terminate)

Example: P_1 Request (1,0,2) cont.

- Executing safety algorithm (cont.):
  - Check P_1
    - Step 1: Work = 2 3 0

      Finish[1] = False
    - Step 2 : (a) *Finish* [*1*] = *false*

      (b) $Need_1 \leq Work$

      0 2 0 $\leq$ *2 3 0*

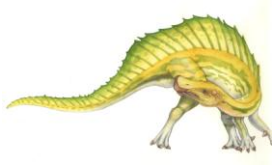
      yes there is i exist
    - Step 3 : $Work = Work + Allocation_1$

      Work = *2 3 0* + *3 0 2*

      Work = *5 3 2*

      Finish[*1*] = *true* { P_1 is terminate }

      go to step 2



Example: P_1 Request (1,0,2) cont.

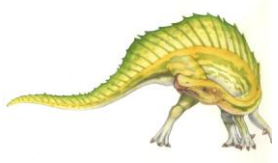
- Executing safety algorithm (cont.) :
  - Check P_2
    - Step 1: Work = *5 3 2*

      Finish[2] = False
    - Step 2 : (a) *Finish* [*2*] = *false*

      (b) $Need_2 \leq Work$

      6 0 0 $\leq$ *5 3 2*

      so there is no i exist
    - Step 3 : skip it
    - P_2 no finish (no terminate)



Example: P_1 Request (1,0,2) cont.

- Executing safety algorithm (cont.) :
  - Check P_3
    - Step 1: Work = 5 3 2

      Finish[3] = False
    - Step 2 : (a) *Finish* [*3*] = *false*

      (b) $Need_3 \leq Work$

      $0\ 1\ 1 \leq 5\ 3\ 2$

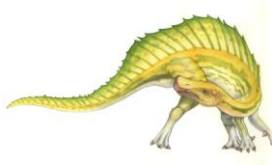
      yes there is i exist
    - Step 3 : $Work = Work + Allocation_3$

      $Work = 5\ 3\ 2 + 2\ 1\ 1$

      $Work = 7\ 4\ 3$

      Finish[*3*] = *true* { P_3 is terminate }

      go to step 2



Example: P_1 Request (1,0,2) cont.

- Executing safety algorithm (cont.) :

  - Check P_4
    - Step 1: Work = 7 4 3

      Finish[4] = False
    - Step 2 : (a) *Finish* [*4*] = *false*

      (b) $Need_4 \leq Work$

      4 3 1 ≤ 7 4 3

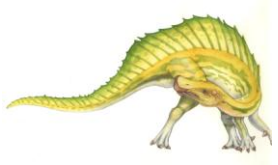
      yes there is i exist
    - Step 3 : *Work* = *Work* + $Allocation_4$

      Work = *7 4 3* + *0 0 2*

      Work = *7 4 5*

      Finish[*4*] = *true* { P_4 is terminate }

      go to step 2



Example: P_1 Request (1,0,2) cont.

- Executing safety algorithm (cont.) :
  - Check P_0
    - Step 1: Work = 7 4 5

      Finish[0] = False
    - Step 2 : (a) $Finish[0] = false$

      (b) $Need_0 \leq Work$

      $7\ 4\ 3 \leq 7\ 4\ 5$

      yes there is i exist
    - Step 3 : $Work = Work + Allocation_0$

      $Work = 7\ 4\ 5 + 0\ 1\ 0$

      $Work = 7\ 5\ 5$

      $Finish[0] = true$ { P_0 is terminate }

      go to step 2

Example: P_1 Request (1,0,2) cont.

- Executing safety algorithm (cont.) :
  - Check P_2
    - Step 1: Work = 7 5 5

      Finish[2] = False
    - Step 2 : (a) *Finish* [*2*] = *false*

      (b) $Need_2 \leq Work$

      6 0 0 ≤ 7 5 5

      yes there is i exist
    - Step 3 : $Work = Work + Allocation_2$

      Work = *7 5 5* + *3 0 2*

      Work = *10 5 7*

      Finish[*2*] = *true* { P_2 is terminate }

      go to step 2

  Step 4 : ***all process in the system is terminated in sequence***

  < P_1 , P_3 , P_4 , P_0 , P_2 >

Example: P_1 Request (1,0,2) cont.

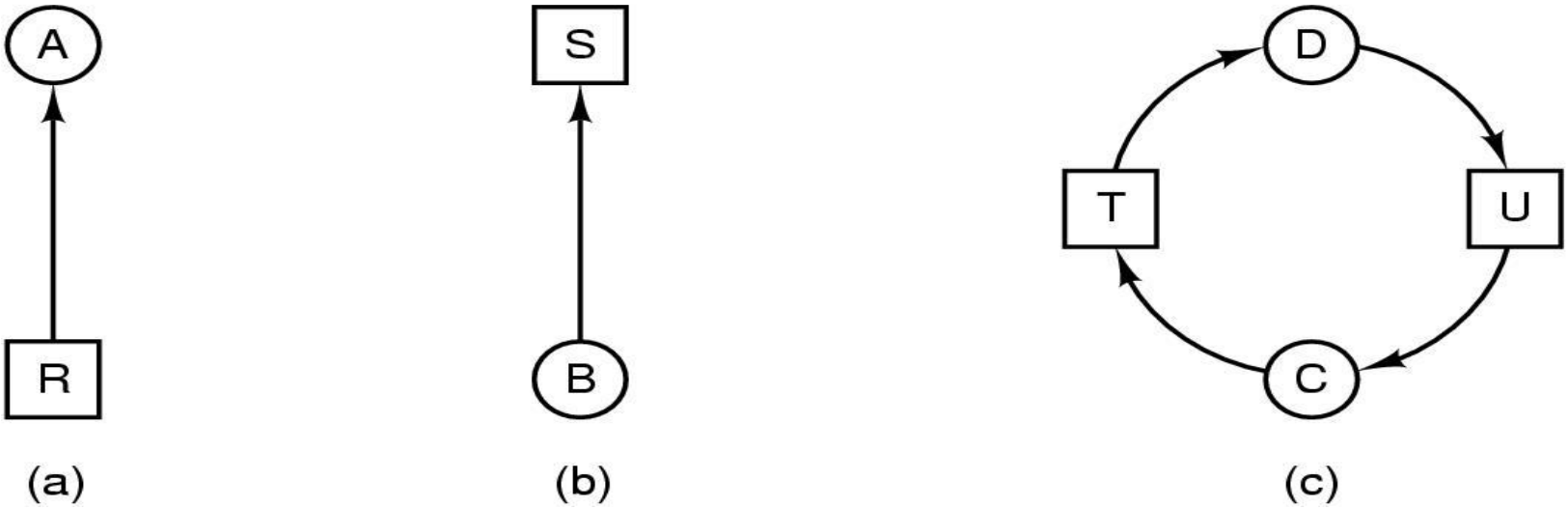
- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

- Methods by which the occurrence of deadlock, the processes and resources involved are detected.
- Generally work by detecting a circular wait
- The cost of detection must be considered
- One method is resource allocation graphs

A Resource Allocation Graph Example



- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U

Recovery from Deadlock: Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods:
  1. Abort all deadlocked processes
  2. 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?

Recovery from Deadlock: Resource Preemption

- To eliminate deadlocks by resource preemption, we preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

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

End of Chapter 7