Operating Systems
CS220

Lecture 14

Deadlocks

22nd June 2021

By: Dr. Rana Asif Rehman

Deadlocks

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

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

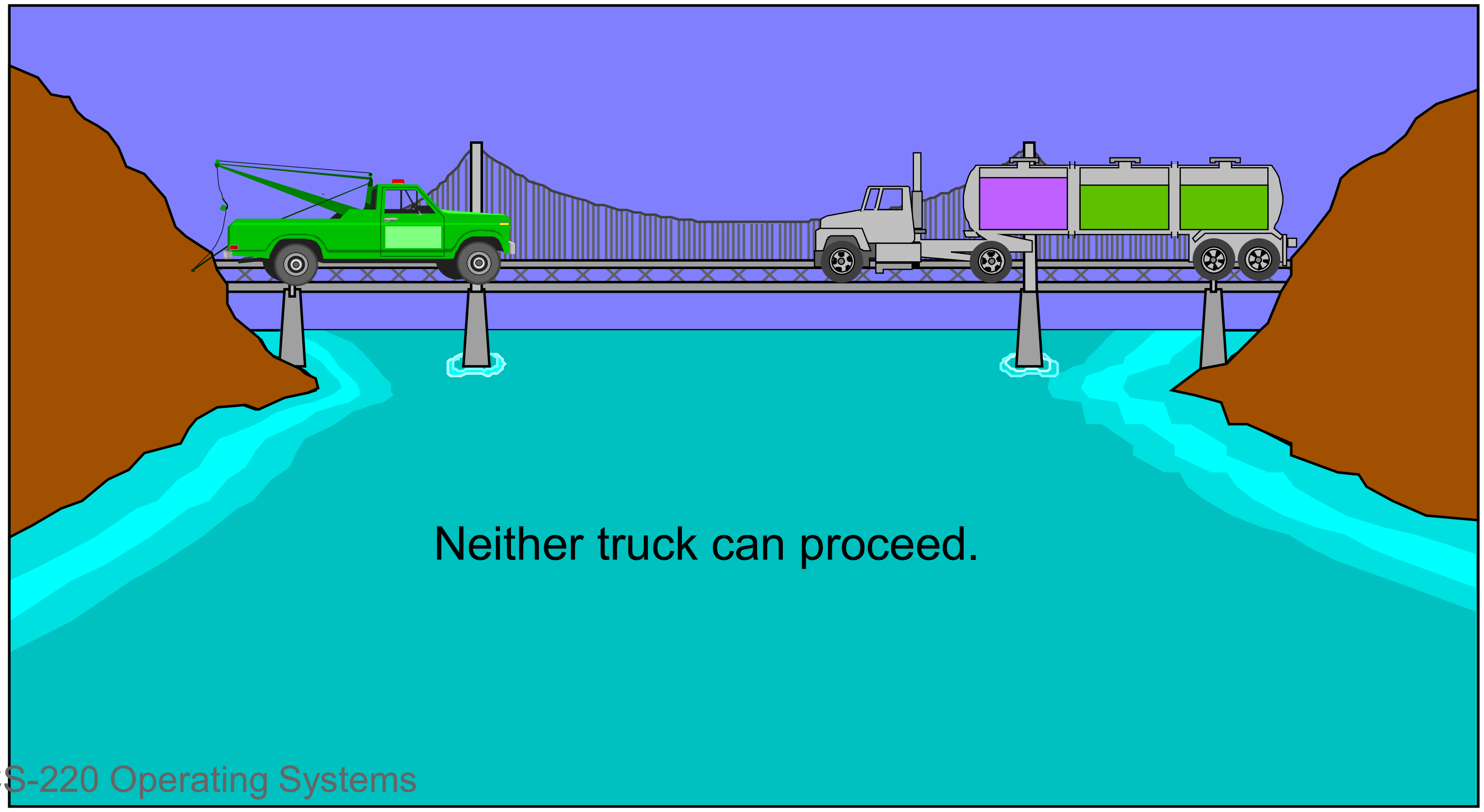
The Deadlock Problem

- 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

```
    P0                P1
wait (A);         wait(B)
wait (B);         wait(A)
```

Deadlock

"When two trucks approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."



System Deadlock

- A process must request a resource before using it, and must release the resource after finishing with it.

  A set of processes is in a *deadlock state* when every process in the set is waiting for a resource that can only be released by another process in the set.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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.

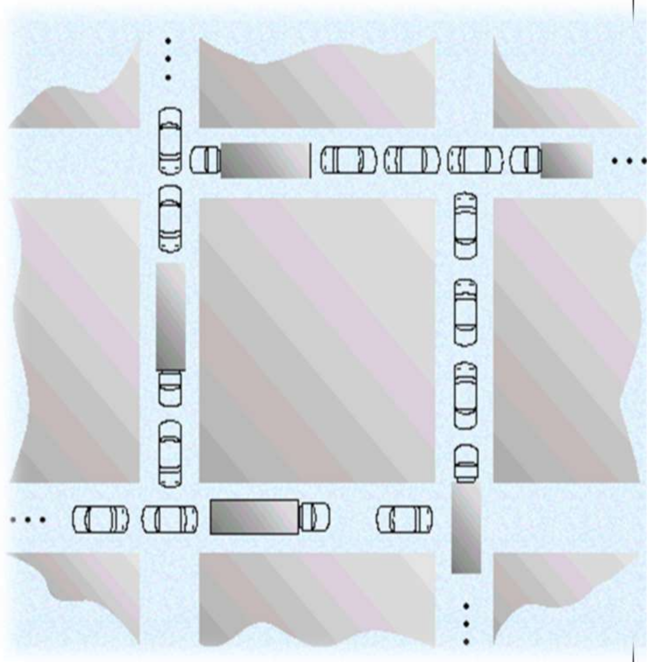
  P0 → P1 → P2 → … → Pn → P0

Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.

Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.

No-preemptive condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.

Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.



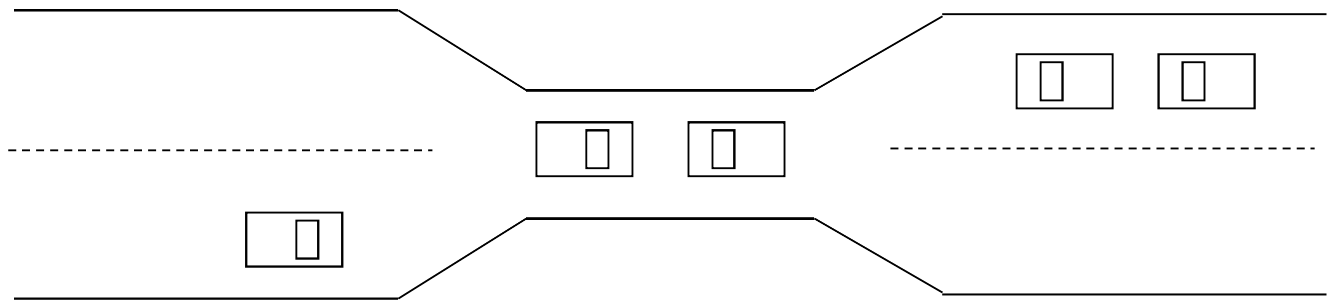
Resources

- Can be a piece of hardware

  (Tape drive, Disk drive, Printer)
- Can be a piece of information

  (File, Shared variable, Critical section)
- Preemptible Resources
  - Such as memory, buffers, CPU
- Nonpreemptible Resources
  - Such as printer

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

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**

request → use → release

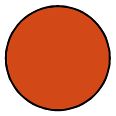
Resource-Allocation Graph

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

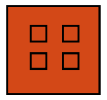
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

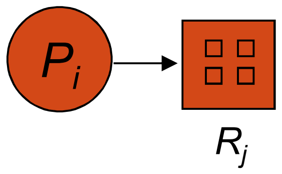
- Process



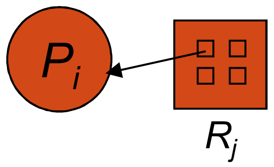
- Resource Type with 4 instances



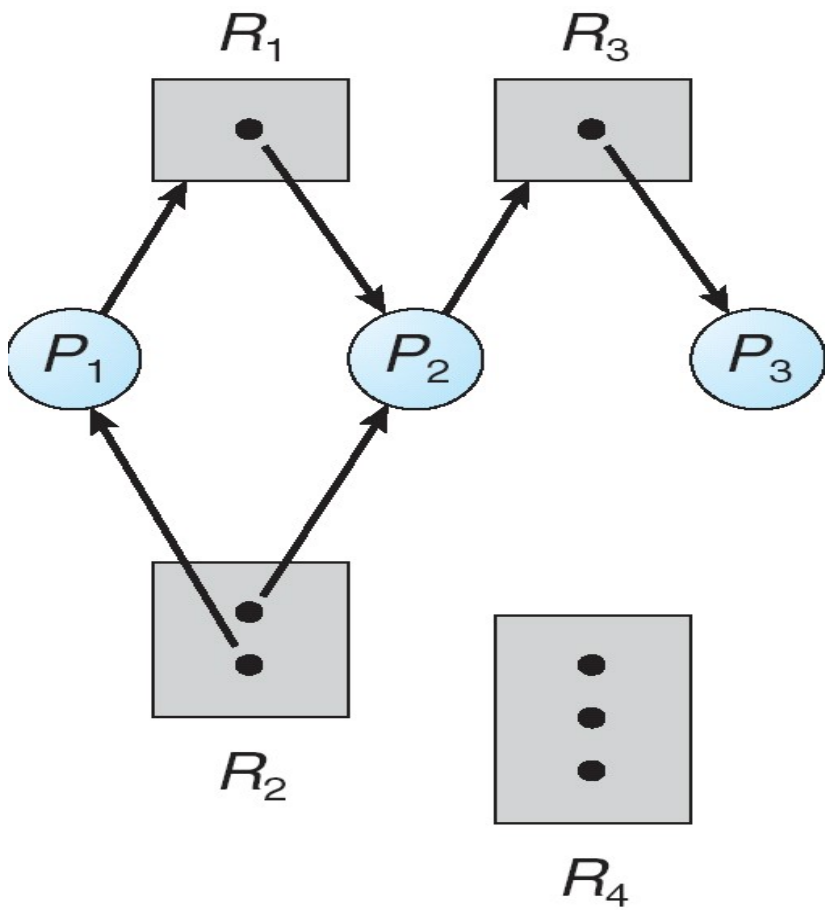
- P_i requests instance of R_j



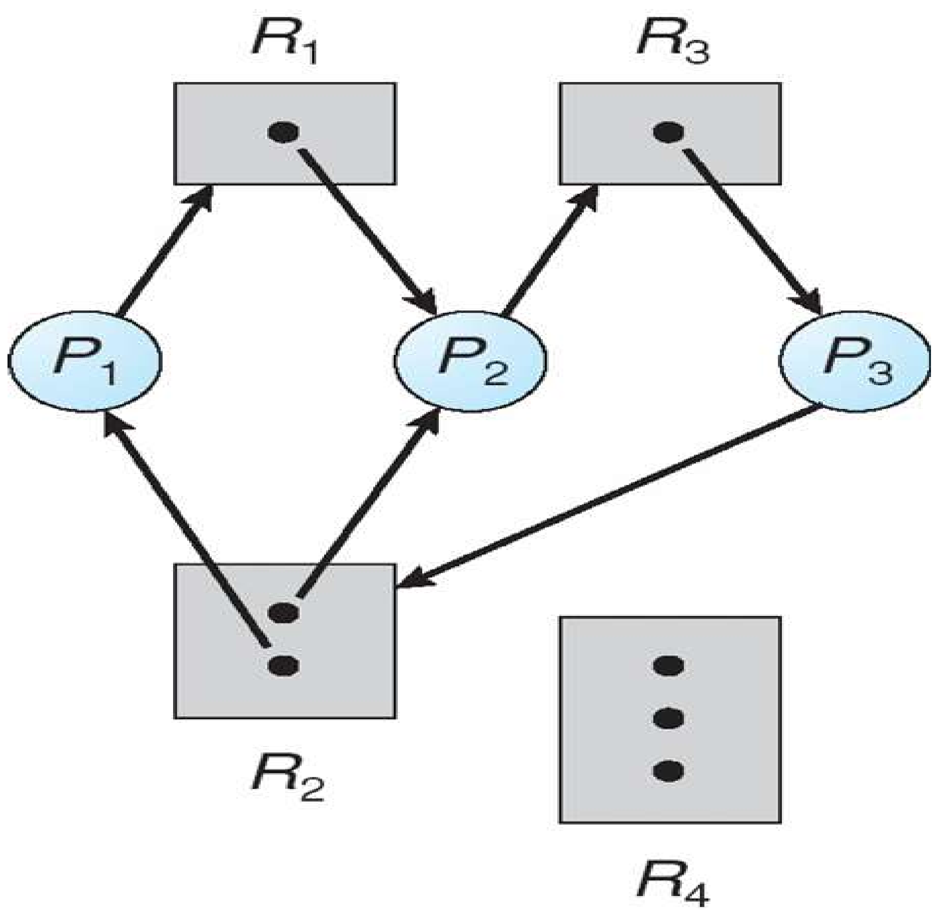
- P_i is holding an instance of R_j



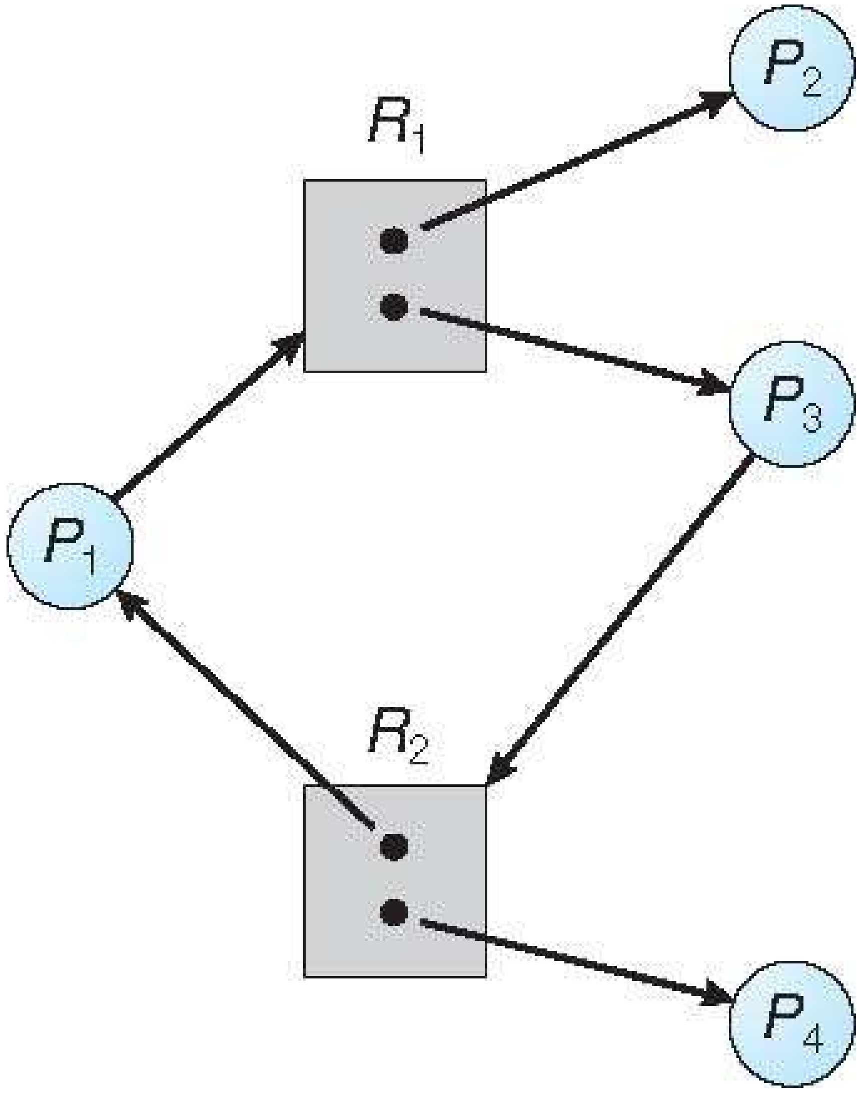
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock

- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

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

Deadlock Handling

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

1. Deadlock Prevention

Restrain the ways request can be made to insure that at least one of the four necessary conditions is violated.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
  Cannot be prevented for all resources. Some resources are inherently non-sharable, such as a printer.

- **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

Deadlock Prevention (Cont.)

- We assign a unique number to each resource type by using function

  F: R $\rightarrow$ N

  and make sure that processes request resources in an increasing order of enumeration.

  For example, tape drive = 1, disk drive = 5, and printer = 12.

- **Proof**

  Let's assume that there is a cycle

  $\mathbf{P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_k \rightarrow P_0}$

  $R_0 \quad R_1 \quad R_2 \quad\quad R_k \quad R_0$

  $\Rightarrow F(R_0) < F(R_1) < \ldots F(R_k) < F(R_0)$

  $\Rightarrow F(R_0) < F(R_0)$, which is impossible

  $\Rightarrow$ There can be no circular wait.

2. Deadlock Avoidance

Requires that the system has some additional *a priori* information available about the use of resources by processes.

- 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

A state is said to be a safe state if the system may allocate the required resources to each process up to the maximum required in a particular sequence, without facing deadlock.

Safe State

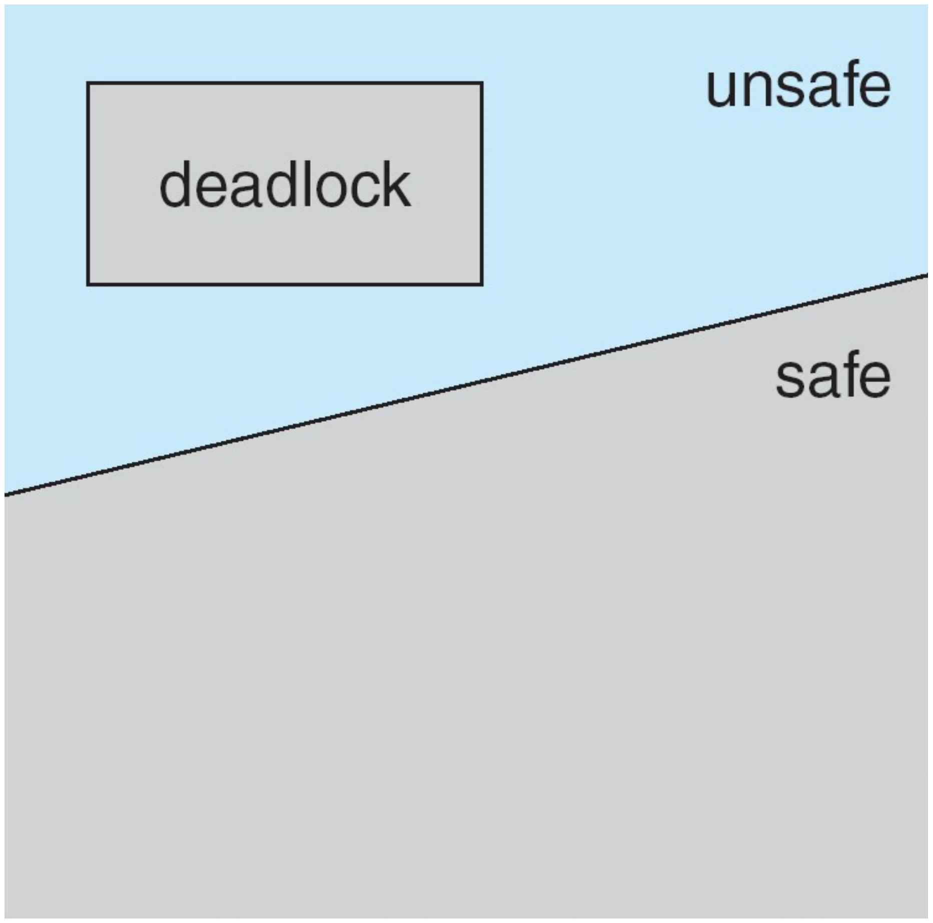
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state System is in **safe state** if there exists a **safe sequence** of all processes.

  Sequence <P_1, P_2, …, P_n> is **safe** if for each P_i, the resources that P_i can still request can be satisfied by the currently available resources, plus the resources held by all the P_j, with $j<i$.
- In other words, a safe sequence specifies the order in which processes can be finished.
- 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 $\Rightarrow$ 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



Example

- System with 24 tape drives and three processes
- P0 may require 20 tape-drives during execution, P1 may require 8, and P2 may require up to 18.
- Suppose, P0 is holding 10 tape drives, P1 holds 5 and P2 holds 4 tape drives. The system is said to be in safe state, since there is a safe sequence that avoids the deadlock.
- Current system state:

Process	Max Need	Allocated
P_0	20	10
P_1	8	5
P_2	18	4

- System is in a safe state with the safe sequence $<P_1, P_0, P_2>$

Example

- System with 12 tape drives and three processes
- Current system state:

Process	Max Need	Allocated
P_0	10	5
P_1	4	2
P_2	9	2

- System is in a safe state with the safe sequence $<P_1, P_0, P_2>$

Example

- P_2 requests and is allocated one more tape drive.
- Assuming the tape drive is allocated to P_2, the new system state will be:

Process	Max Need	Allocated
P_0	10	5
P_1	4	2
P_2	9	3

- System gets into 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

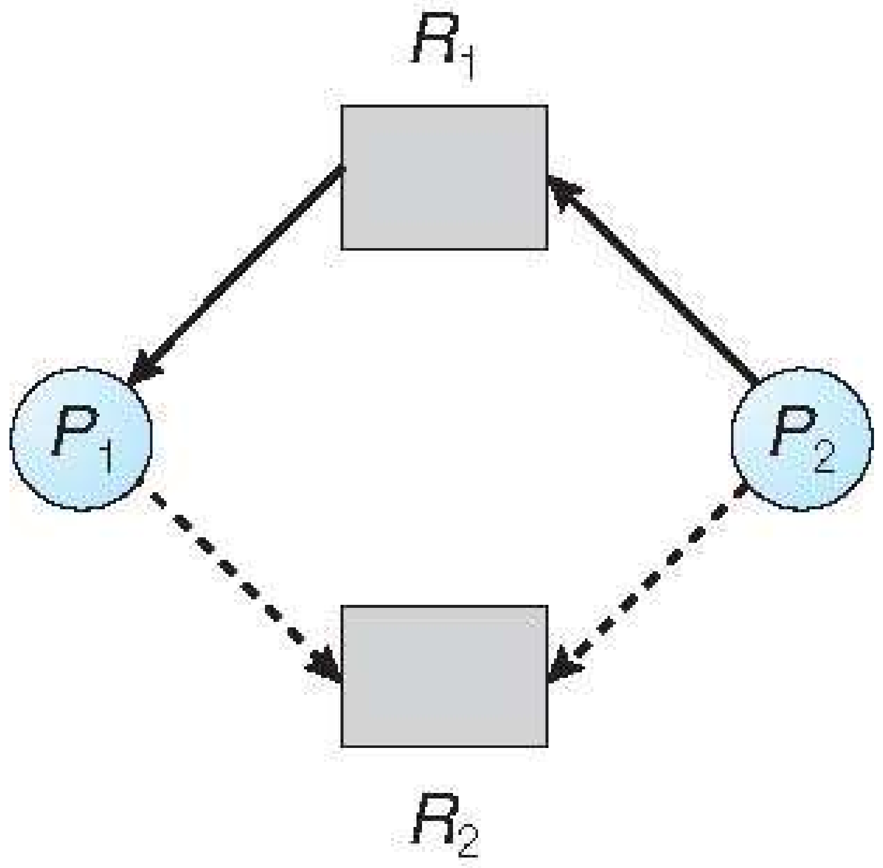
Resource-Allocation Graph (RAG) Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicates 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

Request edge – directed edge $P_i \rightarrow R_j$

Assignment edge – directed edge $R_j \rightarrow P_i$

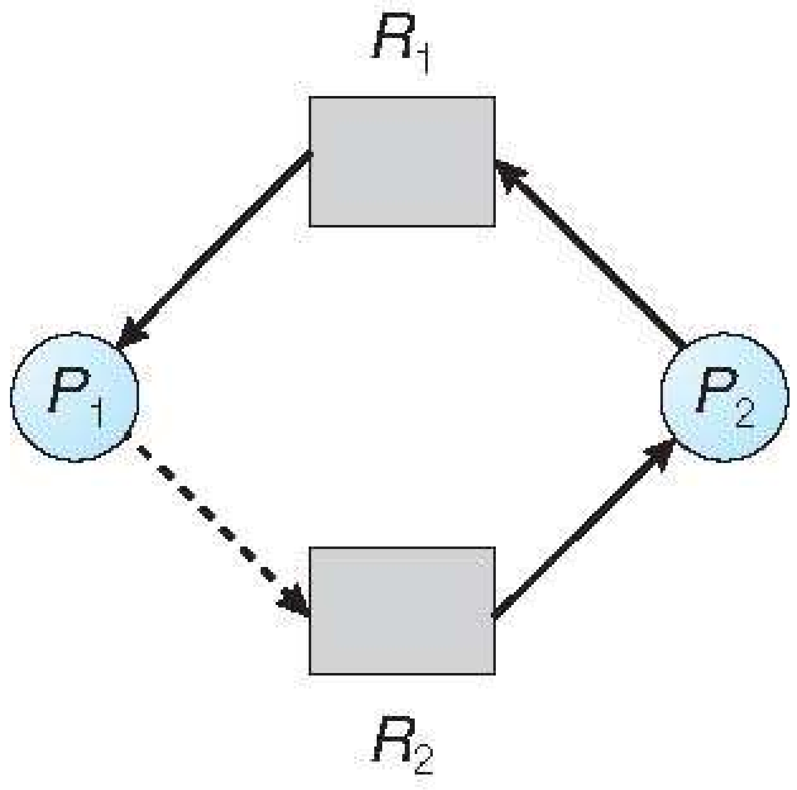
Resource-Allocation Graph (RAG)



Request edge – directed edge $P_i \rightarrow R_j$

Assignment edge – directed edge $R_j \rightarrow P_i$

Unsafe State In Resource-Allocation Graph



Suppose, P2 requests R2. If we allocate R2 to P2, it will create a cycle in the graph and the system will enter an unsafe state. It should not be allocated to P2.

The algorithm cannot be applied to a resource allocation system with multiple instances of each resource.

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Banker’s Algorithm

- The Banker’s algorithm allows the following
  - Mutual exclusion
  - Wait and hold
  - No preemption

It prevents the Circular wait

Data Structures for the Banker's Algorithm

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

- **Available**: A vector of length m indicates number of available resources of each type.
- **Max**: *n x m* matrix indicates the maximum requirement of each process
- **Allocation**: *n* x *m* matrix indicates the number of resources of each type currently allocated to each process
- **Need**: *The n* x *m matrix indicates the remaining resources need of each process*

Banker's Algorithm

Safety Algorithm

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Resource Request Algorithm

```
Step 1:  If (Request ≤ Need )
             Go to Step 2
         else  Error
Step 2:  If (Request ≤ Available)
             Go to Step 3
         else  Wait
Step 3:  Available = Available – Request
         Allocation = Allocation + Request
         Need = Need – Request
Step 4:  Check new state is Safe or Not?
         (by using Safety Algorithm)
```

Example

Assume we have the following resources:

5 tape drives, 2 graphic displays, 4 printers, 3 disks

We can create a vector representing our total resources. **Total** = (5, 2, 4, 3). Consider we have already allocated these resources among four process processes as demonstrated by the following matrix named **Allocated.**

Allocated = (4, 2, 2, 3)

We also need to a matrix to show the number of each resources still needed for each process. We call this matrix **Need**

- Find a row in the **Need** matrix, which is less than the Available vector. If such a row exists, then the process represented by that row may complete with those additional resources. If no such row exists, deadlock is possible

- Need (Process D) = (0,0,1,0) < (1,0,2,0)

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:

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

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max – Allocation*

Process	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

Example (Cont.)

Safety Algorithm

```
if ($Need \le Available$) then
    Execute Process
        new Available = Available + Allocation
else
    Do not execute, go forward
```

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

- Safe Sequence : < >

Example (Cont.)

Safety Algorithm

$if\ (Need \leq Available)\ then$

Execute Process

new Available $=$ Available $+$ Allocation

$else$

Do not execute, go forward

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	3	3	2
P_1	1 2 2	2 0 0	5	3	2
P_2	6 0 0	3 0 2			
P_3	0 1 1	2 1 1			
P_4	4 2 1	0 0 2			

- Safe Sequence : $< P_1 >$

Example (Cont.)

Safety Algorithm

```
if (Need ≤ Available) then
    Execute Process
        new Available = Available + Allocation
else
        Do not execute, go forward
```

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1			
P_4	4	2	1	0	0	2			

- Safe Sequence : $< P_1, P_3 >$

Example (Cont.)

Safety Algorithm

```
if ($Need \le Available$) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	3	3	2
P_1	1 2 2	2 0 0	5	3	2
P_2	6 0 0	3 0 2	7	4	3
P_3	0 1 1	2 1 1	7	4	5
P_4	4 2 1	0 0 2			

- Safe Sequence : $< P_1, P_3, P_4 >$

Example (Cont.)

Safety Algorithm

```
if (Need ≤ Available) then
  Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	3	3	2
P_1	1 2 2	2 0 0	5	3	2
P_2	6 0 0	3 0 2	7	4	3
P_3	0 1 1	2 1 1	7	4	5
P_4	4 2 1	0 0 2	7	5	5

- Safe Sequence : $< P_1, P_3, P_4, P_0 >$

Example (Cont.)

- **Final safe sequence:**

  <P1, P3, P4, P0, P2>

- **Not a unique sequence**

- **Possible safe sequences for the this example:**

  <P1,P3,P4,P0,P2>, <P1,P3,P4,P2,P0>, <P1,P3,P2,P0,P4>, <P1,P3,P2,P4,P0>, <P1,P3,P0,P2,P4>, <P1,P3,P0,P4,P2>

Example: P_1 Request (1,0,2)

Check that : Is $Request_1 \leq Need_1$?

$(1,0,2) \leq (1,2,2) \Rightarrow$ true

Check that : Is $Request \leq Available$?

$(1,0,2) \leq (3,3,2) \Rightarrow$ true

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

Step 1: *If (Request ≤ Need)*

Go to Step 2

else Error

Step 2: *If (Request ≤ Available)*

Go to Step 3

else Wait

Step 3: Available = Available – Request

Allocation = Allocation + Request

Need = Need – Request

Step 4: Check new state is Safe or Not?

(by using Safety Algorithm)

Example: P_1 Request (1,0,2)

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	2	3	0
P_1	0	2	0	3	0	2			
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

```
Step 1: If (Request ≤ Need)
            Go to Step 2
        else Error
Step 2: If (Request ≤ Available)
            Go to Step 3
        else Wait
Step 3: Available = Available – Request
        Allocation = Allocation + Request
        Need = Need – Request
Step 4: Check new state is Safe or Not?
        (by using Safety Algorithm)
```

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Safe Sequence : < >

Example: P_1 Request (1,0,2)

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	2	3	0
P_1	0 2 0	3 0 2	5	3	2
P_2	6 0 0	3 0 2			
P_3	0 1 1	2 1 1			
P_4	4 3 1	0 0 2			

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocati
else
    Do not execute, go forward
```

$Safe\ Sequence : < P_1 >$

Example: P_1 Request (1,0,2)

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	2	3	0
P_1	0 2 0	3 0 2	5	3	2
P_2	6 0 0	3 0 2	7	4	3
P_3	0 1 1	2 1 1			
P_4	4 3 1	0 0 2			

Safe Sequence : $< P_1, P_3 >$

Example: P_1 Request (1,0,2)

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	2	3	0
P_1	0 2 0	3 0 2	5	3	2
P_2	6 0 0	3 0 2	7	4	3
P_3	0 1 1	2 1 1	7	4	5
P_4	4 3 1	0 0 2			

Safe Sequence : $< P_1, P_3, P_4 >$

Example: P_1 Request (1,0,2)

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	2	3	0
P_1	0 2 0	3 0 2	5	3	2
P_2	6 0 0	3 0 2	7	4	3
P_3	0 1 1	2 1 1	7	4	5
P_4	4 3 1	0 0 2	7	5	5

Safe Sequence : $< P_1, P_3, P_4, P_0 >$

Example: P_1 Request (1,0,2)

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 4 3	0 1 0	2	3	0
P_1	0 2 0	3 0 2	5	3	2
P_2	6 0 0	3 0 2	7	4	3
P_3	0 1 1	2 1 1	7	4	5
P_4	4 3 1	0 0 2	7	5	5

Safe Sequence : $< P_1, P_3, P_4, P_0, P_2 >$

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement. So Yes, P_1's request may be **granted immediately.**

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

Example: P_0 Request (0,2,0)

Check that : Is $\text{Request}_0 \le \text{Need}_0$?

$(0,2,0) \le (7,4,3) \Rightarrow$ true

Check that : Is $\text{Request} \le \text{Available}$?

$(0,2,0) \le (3,3,2) \Rightarrow$ true

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

Step 1: *If (Request* $\le$ *Need)*

Go to Step 2

else Error

Step 2: *If (Request* $\le$ *Available)*

Go to Step 3

else Wait

Step 3: Available = Available – Request

Allocation = Allocation + Request

Need = Need – Request

Step 4: Check new state is Safe or Not?

(by using Safety Algorithm)

Example: P_0 Request (0,2,0)

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 2 3	0 3 0	3	1	2
P_1	1 2 2	2 0 0			
P_2	6 0 0	3 0 2			
P_3	0 1 1	2 1 1			
P_4	4 3 1	0 0 2			

Safe Sequence : < >

```
Step 1: If (Request ≤ Need )
            Go to Step 2
        else  Error
Step 2: If (Request ≤ Available)
            Go to Step 3
        else  Wait
Step 3: Available = Available – Request
        Allocation = Allocation + Request
        Need = Need – Request
Step 4: Check new state is Safe or Not?
        (by using Safety Algorithm)
```

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available +Allocation
else
    Do not execute, go forward
```

Example: P_0 Request (0,2,0)

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 2 3	0 3 0	3	1	2
P_1	1 2 2	2 0 0	5	2	3
P_2	6 0 0	3 0 2			
P_3	0 1 1	2 1 1			
P_4	4 3 1	0 0 2			

Safe Sequence : $< P_3, >$

Example: P_0 Request (0,2,0)

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 2 3	0 3 0	3	1	2
P_1	1 2 2	2 0 0	5	2	3
P_2	6 0 0	3 0 2	7	2	3
P_3	0 1 1	2 1 1			
P_4	4 3 1	0 0 2			

Safe Sequence : $< P_3, P_1 >$

Example: P_0 Request (0,2,0)

```
if (Need ≤ Available) then
    Execute Process
    new Available = Available + Allocation
else
    Do not execute, go forward
```

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 2 3	0 3 0	3	1	2
P_1	1 2 2	2 0 0	5	2	3
P_2	6 0 0	3 0 2	7	2	3
P_3	0 1 1	2 1 1	10	2	5
P_4	4 3 1	0 0 2			

$Safe\ Sequence : < P_3, P_1, P_2 >$

Example: P_0 Request (0,2,0)

Process	Need	Allocation	Work		
	A B C	A B C	A	B	C
P_0	7 2 3	0 3 0	3	1	2
P_1	1 2 2	2 0 0	5	2	3
P_2	6 0 0	3 0 2	7	2	3
P_3	0 1 1	2 1 1	10	2	5
P_4	4 3 1	0 0 2	10	5	5

Safe Sequence : $< P_3 ,P_1,P_2 ,P_0,P_4>$

- Executing safety algorithm shows that sequence $< P_3 ,P_1,P_2 ,P_0,P_4>$ satisfies safety requirement. So Yes, P_0's request may be **granted immediately**.

Example: P_4 Request (3,3,0)

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

- **Do it yourself !**

3. Deadlock Detection & Recovery

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

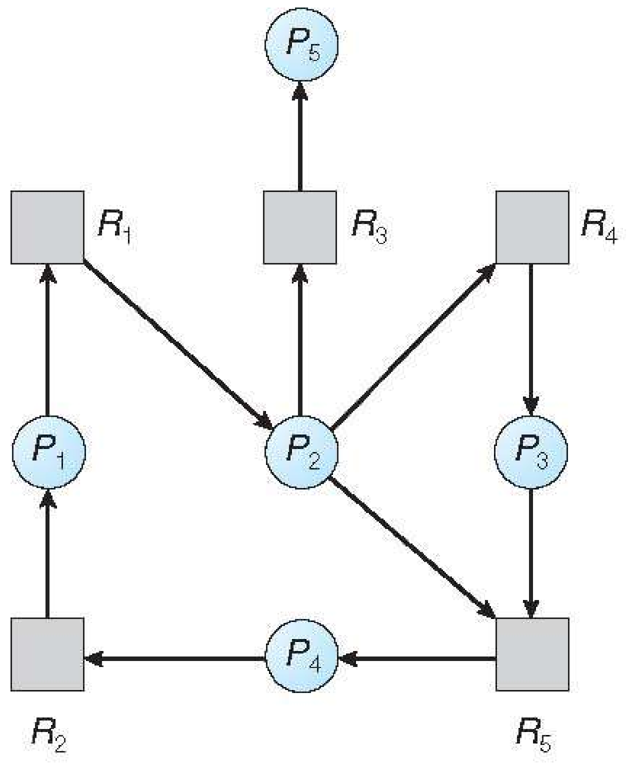
Single Instance of Each Resource Type: Wait-For Graph

- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{P_i} \rightarrow \boldsymbol{P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$

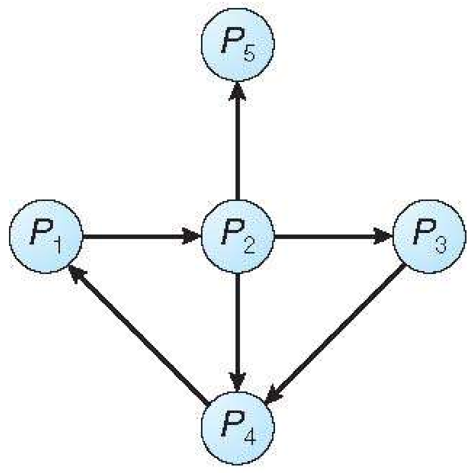
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of $\boldsymbol{n}^2$ operations, where $\boldsymbol{n}$ is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph   Corresponding wait-for graph

Several Instances of a Resource Type: Detection Algorithm

- **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

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

Initialize:

(a) $\textbf{\textit{Work}} = \textbf{\textit{Available}}$

(b) For $\boldsymbol{i} = \mathbf{1,2, \ldots,} \boldsymbol{n}$, if $\textbf{\textit{Allocation}}_i \neq \mathbf{0}$, then
$\textbf{\textit{Finish}}[\mathbf{i}] = \textbf{\textit{false}}$; otherwise, $\textbf{\textit{Finish}}[\mathbf{i}] = \textbf{\textit{true}}$

Step 2. Find an index ***i*** such that **both:**

(a) $\textbf{\textit{Finish}}[i] == \textbf{\textit{false}}$

(b) $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$

If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

Step 3. $Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2

Step 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	

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

Detection Algorithm

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	

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

Detection Algorithm

- 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

	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

```
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 ≠ 0, then
                Finish[i] = false; otherwise,
                Finish[i] = true

2.  Find an index i such that both:
        (a) Finish[i] == false
        (b) Request_i ≤ 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 ≤ i ≤ n, then the system is in deadlock
    state. Moreover, if Finish[i] == false, then P_i is deadlocked
```

Detection Algorithm

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?
    1. Priority of the process
    2. How long process has computed, and how much longer to completion
    3. Resources the process has used
    4. Resources process needs to complete
    5. How many processes will need to be terminated
    6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **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

References

- Operating System Concepts (Silberschatz, 9th edition) Chapter 7
- Chapter 6, Modern Operating Systems By Tenenbaum
- http://en.wikipedia.org/wiki/Deadlock#Prevention