Deadlocks

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

System Model

- System consists of finite no of resources
  - To be distributed among a number of competing processes
- Resources are partitioned into several types,
  - Each consisting of some number of identical instances.

System Model

- Resource types –
  - Memory space
  - CPU cycles
  - Files
  - I/0 devices (such as printers and DVD drives)
- If a system has two CPUs,
  - then the resource type *CPU* has two instances

System Model

- Resource types $R_1, R_2, . . ., R_m$
- Each resource type R_i has W_i instances.
- A process may utilize a resource in only the following sequence:
  - **request**
  - **use**
  - **release**

System Model

- Request
  - The process requests the resource.
  - If the request cannot be granted immediately
  - If the resource is being used by another process
  - then the requesting process must wait until it can acquire the resource
- Use
  - The process can operate on the resource
  - If the resource is a printer, the process can print on the printer
- Release
  - The process releases the resource.

Deadlock

- A set of processes is in a deadlocked state when
  - every process in the set is waiting for an event that can be caused only by another process in the set.

- The events with which we are concerned are
  - resource acquisition and
  - release

Deadlock

- The resources may be either
  - physical resources (for example, printers, tape drives, memory space, and CPU cycles) or
  - logical resources (for example, files, semaphores, and monitors).

Deadlock involving the same resource type

- Consider a system with 3 CD RW drives and 3 processes.
  - Each process holds one CD RW drives.
  - If each process now requests another drive
  - The three processes will be in a deadlocked state.
  - Each is waiting for the event "CD RW drive is released," which can be caused by one of the other waiting processes.

- Deadlock involving the same resource type.

Deadlock involving different resources

- Deadlocks may also involve different resource types.

- Consider a system with one printer and one DVD drive. Suppose that
  - Process *Pi* is holding the DVD drive
  - Process *Pj* is holding the printer.
  - If *Pi* requests the printer
  - *If Pj* requests the DVD drive,
  - A deadlock occurs.

Deadlock Characterization

Deadlock can arise if the following four conditions hold simultaneously.

- **Mutual exclusion**
- **Hold and wait**
- **No preemption**
- **Circular wait**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion:

- Only one process at a time can use a resource
- If another process requests that resource, the requesting process must be delayed until the resource has been released

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

- Resources cannot be preempted
- 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,
- P_n is waiting for a resource that is held by P_0.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.

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

Resource-Allocation Graph

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

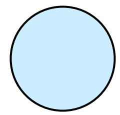
request edge –

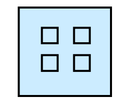
- directed edge $P_i \rightarrow R_j$
- P_i *Process* requested an instance of Resource type R_j
- P_i *is currently waiting for that resource*

assignment edge –

- directed edge $R_j \rightarrow P_i$
- R_j *Resource has been allocated to Process* P_i

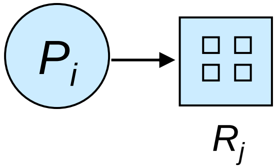
Resource-Allocation Graph (Cont.)

- Process 

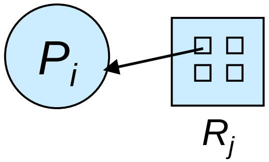
- Resource Type with 4 instances- 
  - Resource may have more than 1 instance
  - Each such instance is a dot within the square

Resource-Allocation Graph (Cont.)

- P_i requests instance of R_j
  - *Request edge points only to the square*

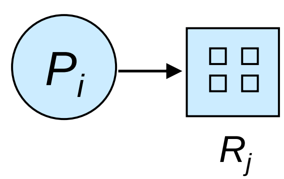


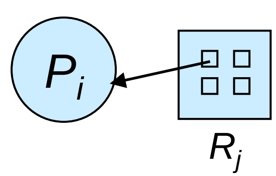
- P_i is holding an instance of R_j
  - *Assignment edge also designates one of the dots in the square*



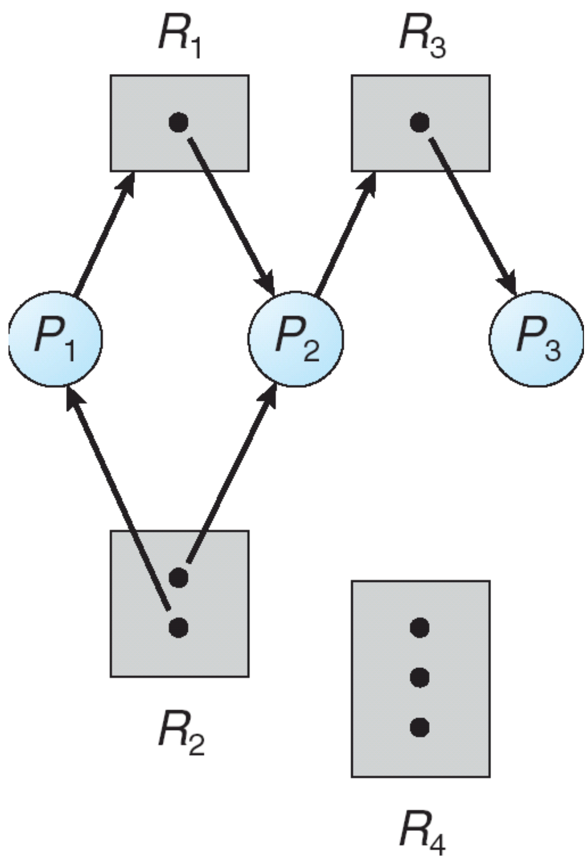
Resource-Allocation Graph (Cont.)

- P_i requests instance of R_j
  - *When process requests , the request edge is inserted in the graph*
  - *When request is granted, request edge is transformed to assignment edge the resource*
  - *When Resource not needed, assignment edge is deleted*

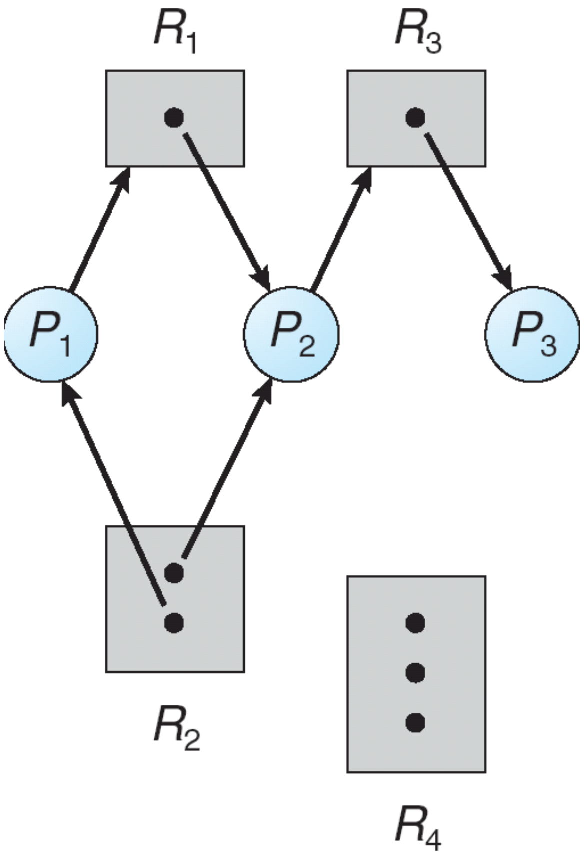




Example of a Resource Allocation Graph



Example of a Resource Allocation Graph



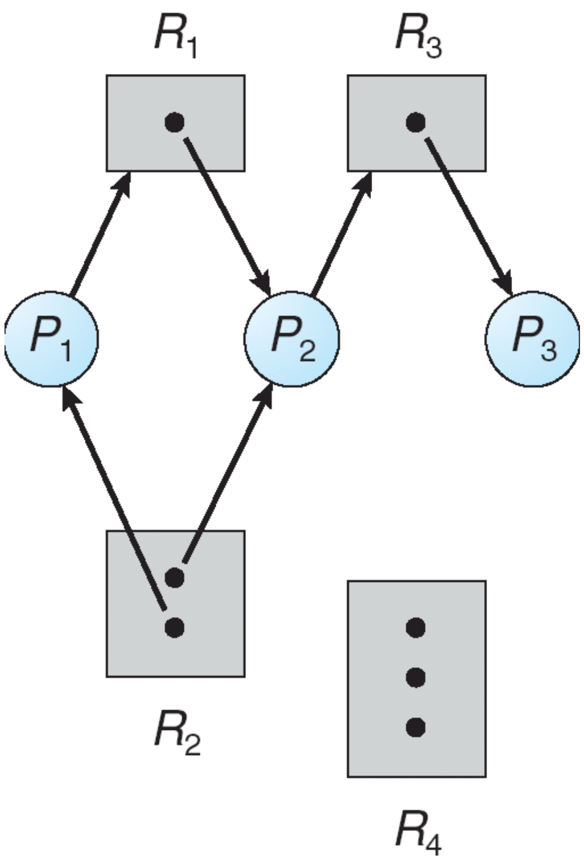
Sets P,R,E

- P={P1,P2,P3}
- R={R1,R2,R3}

Resource Instances

- R1=1 Instance
- R2=2 instances
- R3=1 instances
- R4=3 instances

Example of a Resource Allocation Graph



Process states:

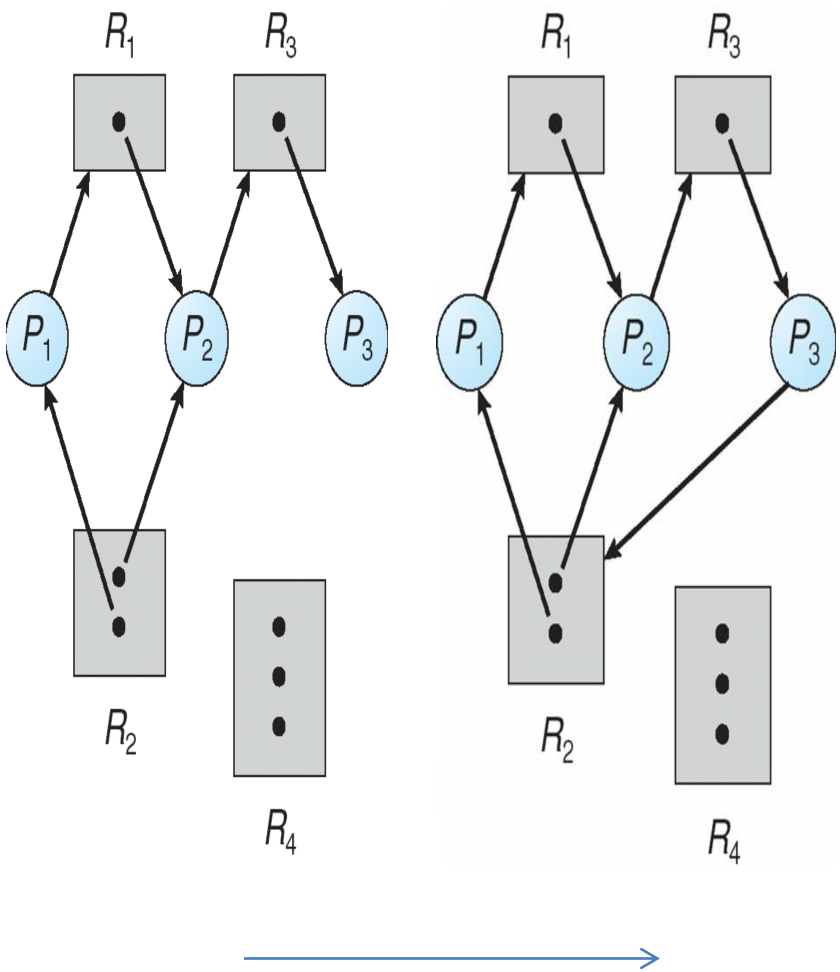
- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1 .
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process *P3* is holding an instance of R3 .

- E={P1->R1,P2->R3,R1->P2,
  R2->P2,R2->P1,R3->P3}

Basic Facts

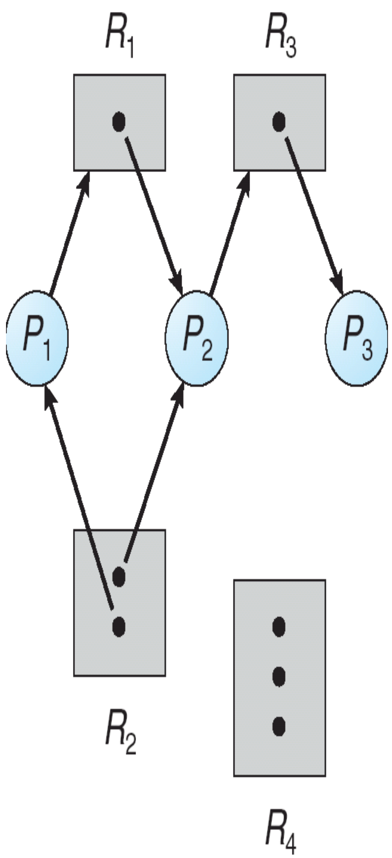
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - Deadlock may exist
  - if only one instance per resource type, then deadlock
    - In this case, cycle is both necessary and sufficient condition for deadlock
  - if several instances per resource type, possibility of deadlock
    - In this case, cycle is necessary but not sufficient condition for existence of deadlock

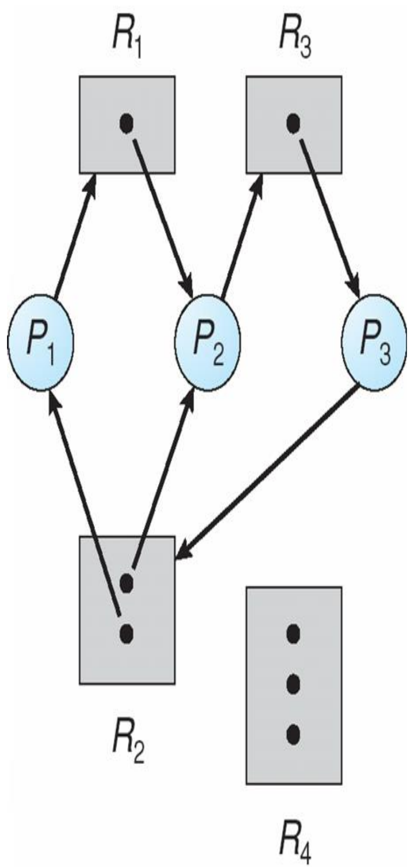
Resource Allocation Graph With A Deadlock



- If P3 requests an instance of resource R2?
- Since no resource instance is free, a request edge P3->R2 is added

Resource Allocation Graph With A Deadlock





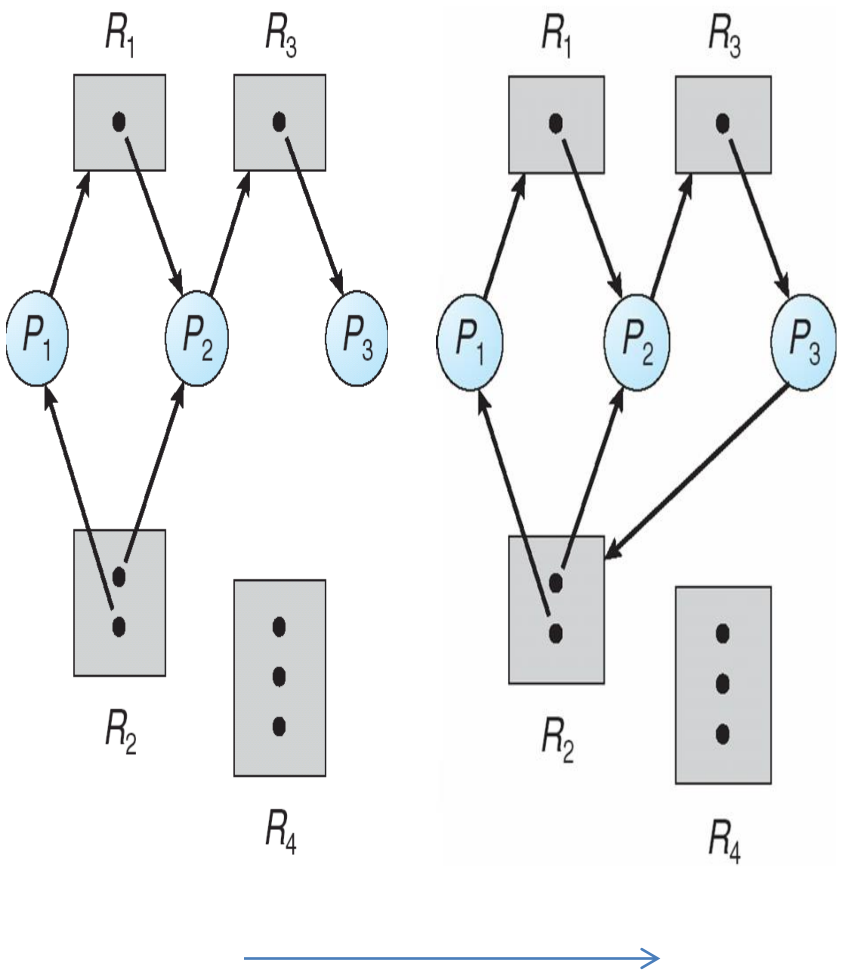
- So, Now Two cycles exist –

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

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

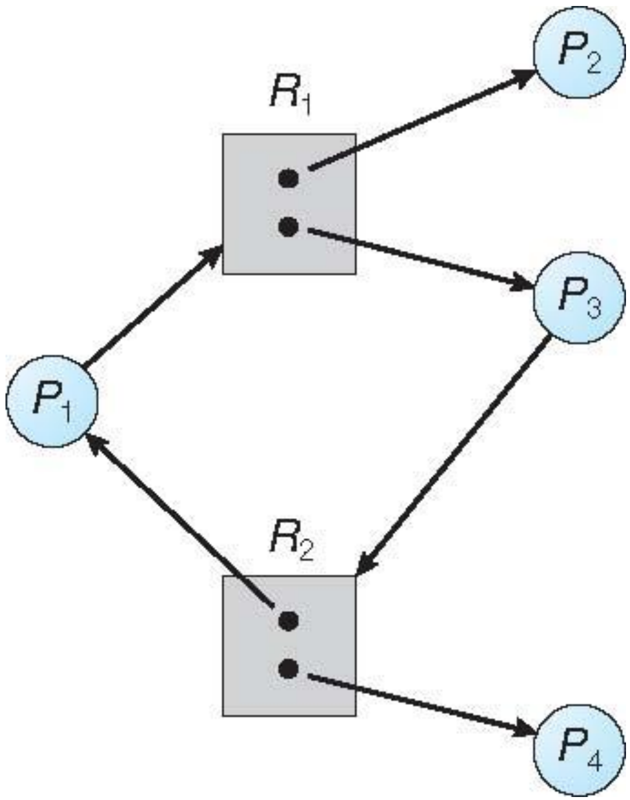
- P1,P2,P3 are deadlocked

Resource Allocation Graph With A Deadlock



- P1,P2,P3 are deadlocked
- Process *P2* is waiting for the resource *R3,* which is held by process *P3.* Process *P3* is waiting for either process *P1* or process *P2* to release resource R2.
- In addition, process *P1* is waiting for process *P2* to release resource R1.
- Circular Wait

Resource Allocation Graph With A Cycle But No Deadlock

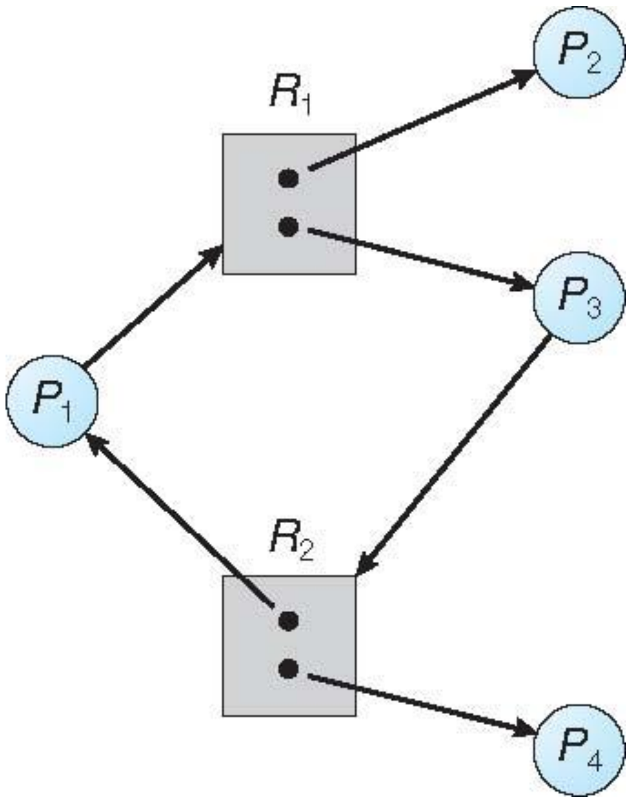


- One cycle exist –

P1->R1->P3->R2->P1

- Still No Deadlock
- ?

Resource Allocation Graph With A Cycle But No Deadlock



- Still No Deadlock
- P4 may release its instance of resource R2, which can be allocated to P3
- Thus, Breaking the cycle

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance

Methods for Handling Deadlocks

- If the system does not apply either a Deadlock prevention or Deadlock avoidance,
  - then deadlock may occur

Methods for Handling Deadlocks

- Then 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

Deadlock Prevention

Restrain the ways request can be made

- Prevent the occurrence of a deadlock
- Provides a set of methods for ensuring that
- **at least one of the necessary conditions cannot hold.**
- **by constraining how requests for resources can be made.**

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** –
- ME not required for sharable resources (e.g., read-only files);
  - Eg-If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** –
- ME must hold for non-sharable resources
  - Eg- A printer cannot be simultaneously shared by several processes.

Deadlock Prevention

Restrain the ways request can be made

- **Hold and Wait**
  - Must guarantee that whenever a process requests a resource,
  - it does not hold any other resources

Deadlock Prevention

Hold and Wait

- Method 1
  - Require process to request and be allocated all its resources
    - before it begins execution,
  - Implement this provision by requiring that system calls requesting resources for a process
    - precede all other system calls.

Deadlock Prevention

Hold and Wait

- Method 2
  - allow process to request resources
    - only when the process has none allocated to it.
  - A process may request some resources and use them
    - Before it can request any additional resources
    - It must release all the resources that it is currently allocated.

Deadlock Prevention

Hold and Wait- Difference between these two protocols

- 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.

Deadlock Prevention

Hold and Wait- Difference between these two protocols

Method 1

- <u>If all resources must be requested at the beginning of the process,</u>
- <u>The process must initially request the DVD drive, disk file, and printer.</u>
- It will hold the printer for its entire execution, even though it needs the printer only at the end.

Deadlock Prevention

Hold and Wait- Difference between these two protocols

Method 2

- 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
- Terminates.

Deadlock Prevention

Hold and Wait- Difference between these two protocols

- Two main disadvantages.
- First Method, resource utilization may be low,
  - since resources may be allocated but unused for a long period.
- Second Method, starvation is possible.
  - A process that needs several popular resources may have to wait indefinitely,
  - because at least one of the resources that it needs is always allocated to some other process.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - Method 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**
    - **Preempted resources are added to the list of resources**
  - Process will be restarted only when
    - it can regain its old resources, as well as
    - the new ones that it is requesting

Deadlock Prevention (Cont.)

- **No Preemption** –
  - Method 2-
- If a process requests some resources,
  - If they are not available,
  - check If 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.

Deadlock Prevention (Cont.)

- **No Preemption** –
  – Method 2-
- 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.

Deadlock Prevention (Cont.)

- **Circular Wait** –
- Impose a total ordering of all resource types,
- Requires that each process requests resources in an increasing order of enumeration

Deadlock Prevention (Cont.)

- **Circular Wait**
- *Resources R* = { R1, R2, ... , Rm}
- Assign to each resource type a unique integer number,
  - which allows us to compare two resources
  - to determine whether one precedes another in our ordering.
- Define a one-to-one function

  F: *R* -> *N*

- where *N* is the set of natural numbers.

Deadlock Prevention (Cont.)

- **Circular Wait**
- For example, if the set of resource types *R* includes tape drives, disk drives, and printers, then the function *F* might be defined as follows:

  F (tape drive) = 1

  F (disk drive) = 5

  F (printer) = 12
- 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.

Deadlock Prevention (Cont.)

- **Circular Wait**
- A process can initially request any number of instances of a resource type, *Ri.*
- After that, the process can request instances of resource type *Rj*
- If and only if F(Rj) > F(Ri)

Deadlock Avoidance

Deadlock Avoidance

Disadvantages of Deadlock prevention??

- Possible side effects of preventing deadlocks, are
  - <u>low device utilization and</u>
  - <u>reduced system throughput.</u>

- An alternative method for avoiding deadlocks
  - Deadlock Avoidance
  - is to require additional information about how resources are to be requested

A priori

- In Latin *a priori* means “what comes first.”

- *A priori* understandings are the assumptions that come before the rest of the assessment, argument, or analysis.

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

Deadlock Avoidance

- Given this a priori information,
  - it is possible to construct an algorithm
  - That ensures that the system will never enter a deadlocked state.
  - Such an algorithm defines the deadlock-avoidance approach.

Deadlock Avoidance

- 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 resources,
  - the number of allocated resources,
  - and the maximum demands of the processes

Safe State

- A state is *safe* if
  - the system can allocate resources to each process (up to its maximum) in some order and
  - still avoid a deadlock.

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

Safe State

- System is in **safe state** if there exists a sequence <P_1, P_2, …, 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 $j < i$

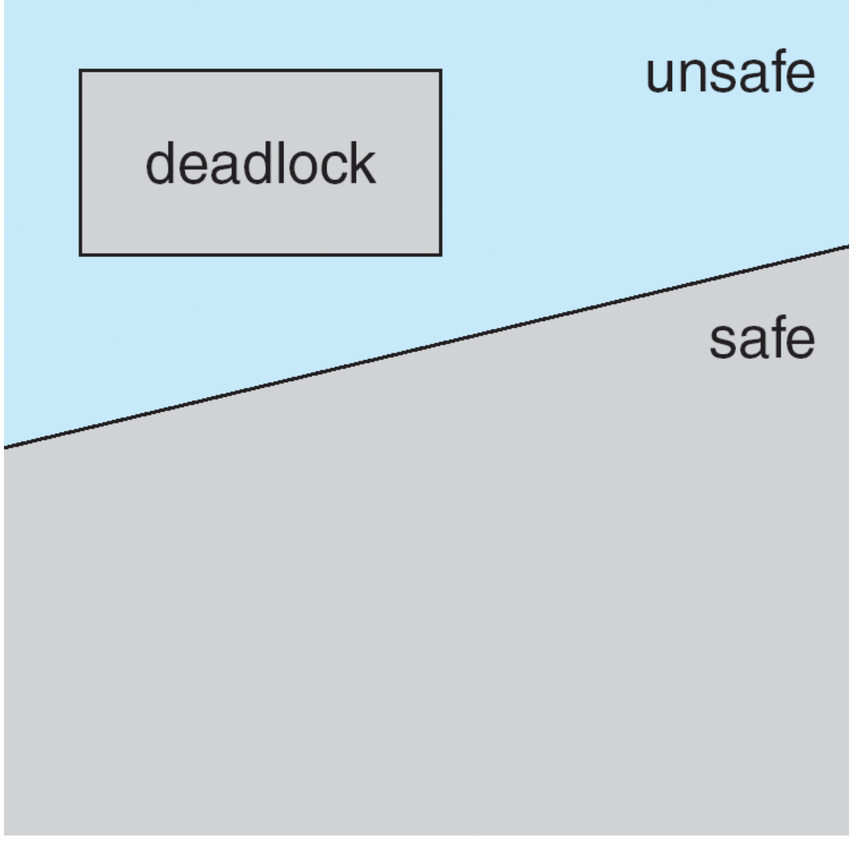
Safe State

- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished ($j < i$)
  - 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



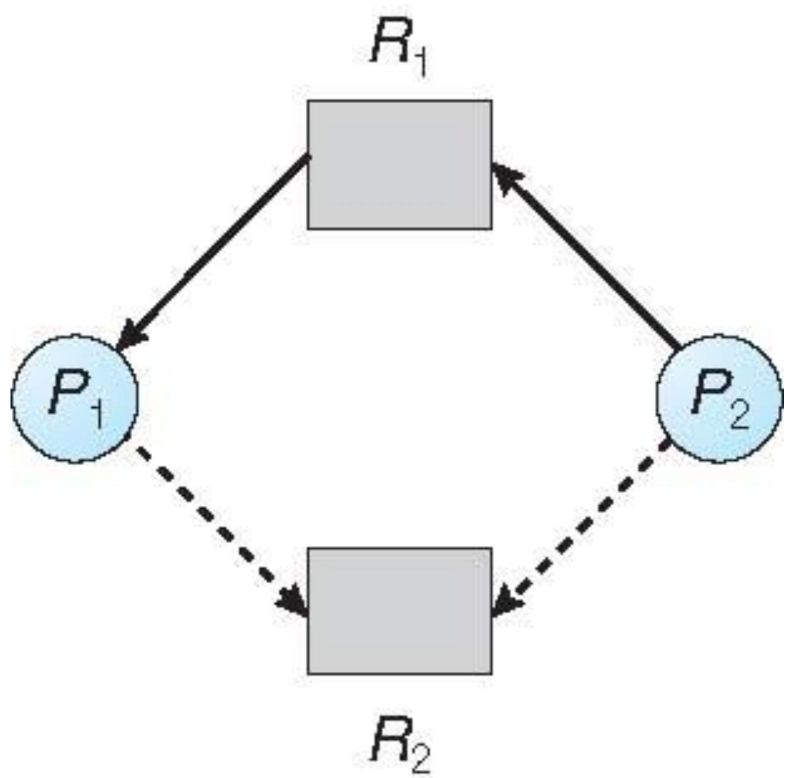
Avoidance Algorithms

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

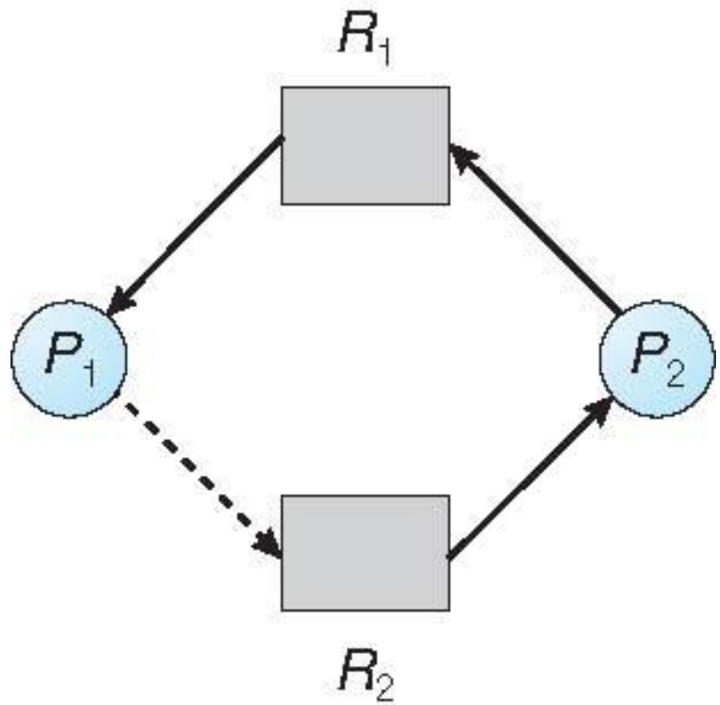
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph

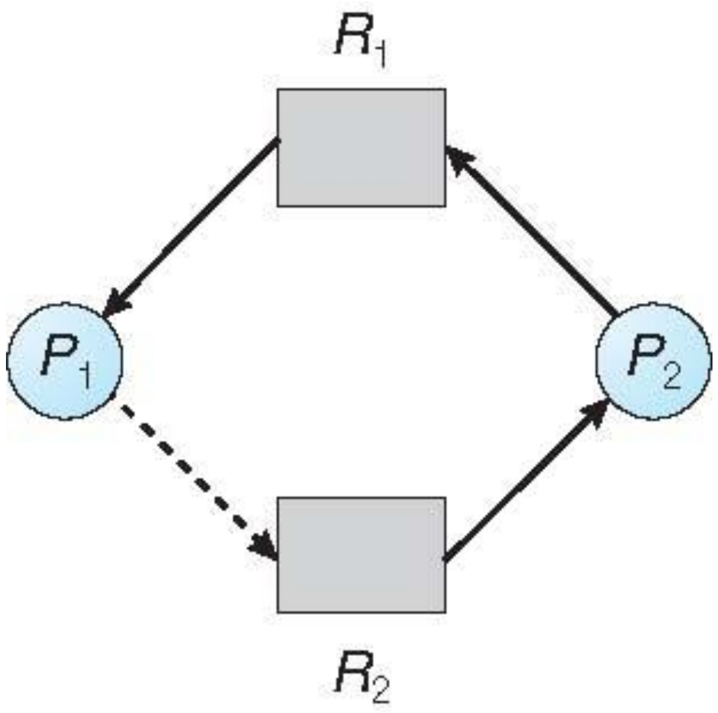


Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

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



Banker's Algorithm

- Multiple instances

- Each process must a priori claim maximum use
  - This number may not exceed the total number of resources in the system.

Banker’s Algorithm

- 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

- A resource allocation and deadlock avoidance algorithm.
- This algorithm test for safety simulating
  - the allocation for predetermined maximum possible amounts of all resources,
  - then makes an “s-state” check to test for possible activities,
  - before deciding whether allocation should be allowed to continue.

Data Structures for the Banker's Algorithm

n = number of processes

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, …, *n*- 1**

2. Find an ***i*** such that both:
   (a) ***Finish* [*i*] = *false***
   (b) ***Need**$_i$* $\leq$ ***Work***
   (Add the process in the safe sequence)
   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

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	

Example (Cont.)

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

Need [i, j] = Max [i, j] – Allocation [i, j]

	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

Applying the Safety algorithm on the given system,

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

n **= number of processes**

m **= number of resources types**

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**

Process	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			

Process	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

Applying the Safety algorithm on the given system,

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0 > Work$ (7,4,3 > 3,3,2)

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $Need_1 < Work$ (1,2,2 < 3,3,2)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0)

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2 > Work$ (6, 0, 0 > 5,3, 2)

So P_2 must wait

n = number of processes

m = number of resources types

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**$_i$* ≤ ***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

Process	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	

Process	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

Applying the Safety algorithm on the given system,

Step 2

For i=3
$Need_3 = 0, 1, 1$
Finish [3] = false and $\overset{0,\,1,\,1}{Need_3} < \overset{5,\,3,\,2}{Work}$
So P_3 must be kept in safe sequence

Step 3

$Work = \overset{5,\,3,\,2}{Work} + \overset{2,\,1,\,1}{Allocation_3}$

Work =

A	B	C
7	4	3

Finish =

0	1	2	3	4
false	true	false	true	false

Step 2

For i = 4
$Need_4 = 4, 3, 1$
Finish [4] = false and $\overset{4,\,3,\,1}{Need_4} < \overset{7,\,4,\,3}{Work}$
So P_4 must be kept in safe sequence

Step 3

$Work = \overset{7,\,4,\,3}{Work} + \overset{0,\,0,\,2}{Allocation_4}$

Work =

A	B	C
7	4	5

Finish =

0	1	2	3	4
false	true	false	true	true

Step 2

For i = 0
$Need_0 = 7, 4, 3$
Finish [0] is false and $\overset{7,\,4,\,3}{Need} < \overset{7,\,4,\,5}{Work}$
So P_0 must be kept in safe sequence

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) $\boldsymbol{Need_i \le Work}$
   If no such ***i*** exists, go to step 4
3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2
4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Process	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			

Process	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

Applying the Safety algorithm on the given system,

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1, 0)

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

Step 2

For i = 2
$Need_2$ = 6 , 0, 0
Finish [2] is false and $Need_2 <$ Work (6, 0, 0 < 7, 5, 5)
So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for $0 \le i \le n$
Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

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) $\boldsymbol{Need_i \le Work}$
   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

Process	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			

Process	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

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 $\boldsymbol{Request_i \leq Need_i}$ 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:

$$\boldsymbol{Available = Available - Request_i;}$$
$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$
$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- **Apply Safety Algorithm**
  - If safe $\Rightarrow$ the resources are allocated to ***P**$_i$*
  - If unsafe $\Rightarrow$ ***P**$_i$* must wait, and the old resource-allocation state is restored

Example: P_1 Request (1,0,2)

Process	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	

Process	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

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

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

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Example: *Explanation*

- What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

Process	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			

Process	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

$$\text{Request}_1 = \overset{A}{1}, \overset{B}{0}, \overset{C}{2}$$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$$\text{Request}_1 < \text{Need}_1 \quad (1, 0, 2 < 1, 2, 2)$$

Step 2

$$\text{Request}_1 < \text{Available} \quad (1, 0, 2 < 3, 3, 2)$$

Step 3

Available = Available – Request_1

Allocation_1 = Allocation_1 + Request_1

Need_1 = Need_1 - Request_1

Process	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			

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm again

m=3, n=5 Step 1 of Safety Algo

Work = Available

Work =	2	3	0

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ > Work (7, 4, 3 > 2, 3, 0)

So P_0 must wait

But Need ≤ Work

For i = 1 Step 2

$Need_1$ = 0, 2, 0

Finish [1] is false and $Need_1$ < Work (0, 2, 0 < 2, 3, 0)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (2, 3, 0 + 3, 0, 2)

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ > Work (6, 0, 0 > 5,3, 2)

So P_2 must wait

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$_i$*** **≤ *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

Process	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			

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm again

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work ($0, 1, 1 < 5, 3, 2$)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ ($5, 3, 2 + 2, 1, 1$)

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ < Work ($4, 3, 1 < 7, 4, 3$)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ ($7, 4, 3 + 0, 0, 2$)

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work ($7, 4, 3 < 7, 4, 5$)

So P_0 must be kept in safe sequence

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) ***$Need_i \le Work$***
   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

Process	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			

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm again

Step 3

$Work = \overset{7,4,5}{Work} + \overset{0,1,0}{Allocation_0}$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

Step 2

For i = 2

$Need_2 = 6, 0, 0$

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

$Work = \overset{7,5,5}{Work} + \overset{3,0,2}{Allocation_2}$

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for $0 \leq i \leq n$

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Hence the new system state is safe, so we can immediately grant the request for process P_1.

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$_i$* ≤ *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

Process	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			

Why Banker's algorithm is named so?

Why Banker's algorithm is named so?

- Banker's algorithm is named so because it is used in banking system to check **whether loan can be sanctioned to a person or not.**
- Suppose there are n number of account holders in a bank and the total sum of their money is S.
- If a person applies for a loan then the bank
  - first subtracts the loan amount from the total money that bank has and
  - if the remaining amount is greater than S then only the loan is sanctioned.

Why Banker's algorithm is named so?

- It is done because if all the account holders comes to withdraw their money then the bank can easily do it.
- Bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers.
- The bank would try to be in safe state always.

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Consider the following snapshot of a system-

Answer the following questions using the Banker's algorithm-

(i) What are the total instances of Resources

(ii) What is the content of the matrix need?

(iii) Is the system in a safe state?

(iv) If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

Exercise 1

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

(i) What are the total instances of Resources

For A=1+1+1=3

For B=3+6+5=14

For C=1+5+3+1+2=12

For D=2+4+2+4=12

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P1	1	0	0	0	1	7	5	0					0	7	5	0
P2	1	3	5	4	2	3	5	6					1	0	0	2
P3	0	6	3	2	0	6	5	2					0	0	2	0
P4	0	0	1	4	0	6	5	6					0	6	4	2

Content of the matrix need

Step 1: in row of process P0, use formula

Need=Max – Allocation

Step 2: Follow step 1 above for all other processes i.e. P1, P2, P3, P4, P5.

Result given above.

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P1	1	0	0	0	1	7	5	0					0	7	5	0
P2	1	3	5	4	2	3	5	6					1	0	0	2
P3	0	6	3	2	0	6	5	2					0	0	2	0
P4	0	0	1	4	0	6	5	6					0	6	4	2

Work=[1,5,2,0]
For P) Need $\leq$ Work, so Safe
Sequence=<P0>
Work=[1,5,2,0]+[0,0,1,2]
Work=[1,5,3,2]
For P1, Need is not $\leq$ Work
P1 must wait
For P2, Need $\leq$ Work i.e.
[1,0,0,2]<[1,5,3,2]
So Safe Sequence=<P0,P2>
Work=[1,5,3,2]+[1,3,5,4]
=[2,8,8,6]

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*$_i$ ≤ *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

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P1	1	0	0	0	1	7	5	0					0	7	5	0
P2	1	3	5	4	2	3	5	6					1	0	0	2
P3	0	6	3	2	0	6	5	2					0	0	2	0
P4	0	0	1	4	0	6	5	6					0	6	4	2

Work=[2,8,8,6]

For P3, Need ≤Work

[0,0,2,0]<Work

Safe Sequence=<P0,P2,P3>

Work=[2,8,8,6]+[0,6,3,2]

=[2,14,11,8]

For P4, Need ≤ Work

[0,6,4,2]<Work

Safe Sequence=<P0,P2,P3,P4>

Work=[2,14,11,8]+[0,0,1,4]

=[2,14,12,12]

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**$_i$* **≤ *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

Process	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P1	1	0	0	0	1	7	5	0					0	7	5	0
P2	1	3	5	4	2	3	5	6					1	0	0	2
P3	0	6	3	2	0	6	5	2					0	0	2	0
P4	0	0	1	4	0	6	5	6					0	6	4	2

Work=[2,14,12,12]
Now for P1,
Need $\leq$ Work
[0,7,5,0]<Work
Safe Sequence=<P0,P2,P3,P4,P1>
Work=[2,14,12,12]+[1,0,0,0]
=[3,14,12,12]
Thus, System is in a safe state

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) $\boldsymbol{Need_i \leq Work}$
   If no such ***i*** exists, go to step 4
3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2
4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Deadlock Detection

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - examines the state of the system to determine whether a deadlock has occurred
- Recovery scheme

Detection Algorithm

Detection Algorithm

- Single Instance of all Resources Types
- Several Instances of Resource Types

Single Instance of Each Resource Type

Detection Algorithm

- If only a single instance of all resources are there,
  - Use a variant of the resource-allocation graph, called **a *wait-for* graph.**
  - **Obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges**

Single Instance of Each Resource Type

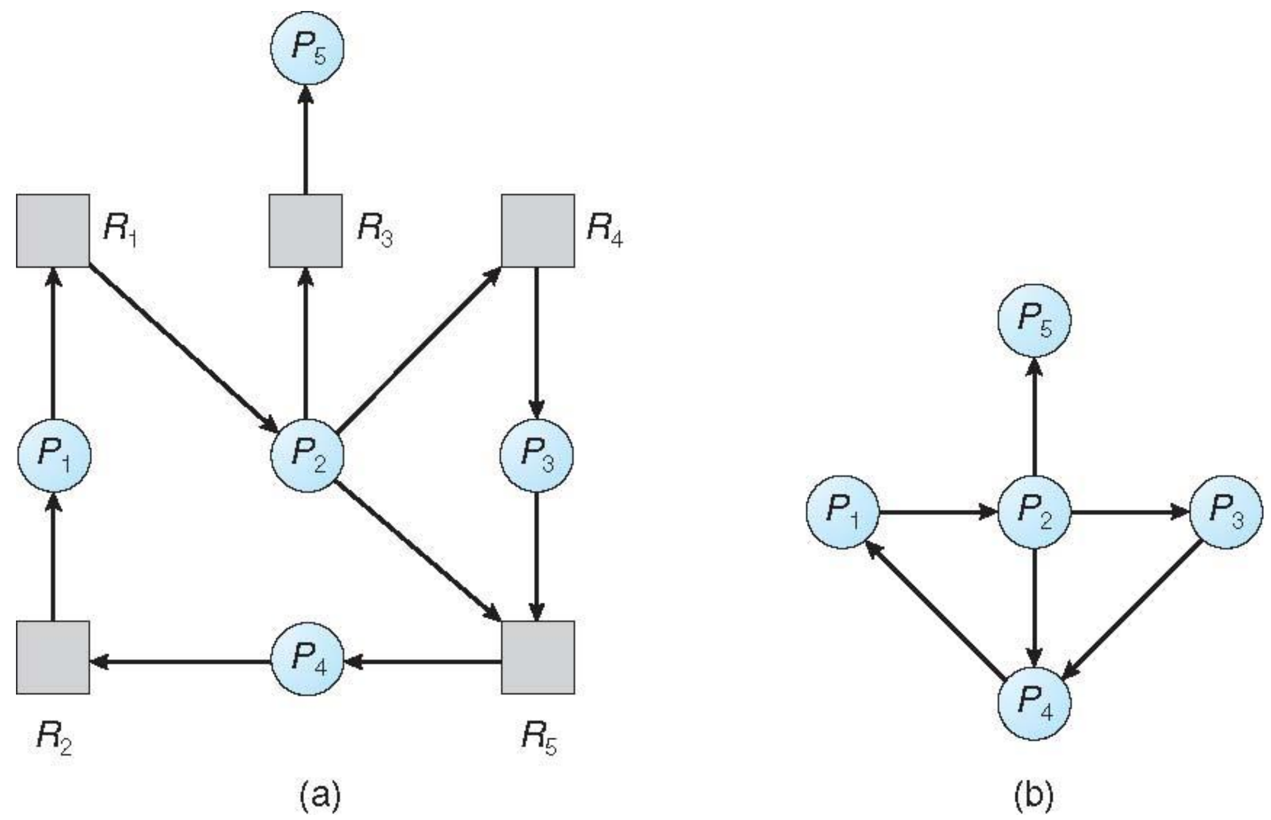
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- 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

m=no of resources

n=no of processes

- **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.

Several Instances of a Resource Type

m=no of resources

n=no of processes

- **Available**
- **Allocation**
- **Request**

m

n

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

n x m

Detection Algorithm

m=no of resources

n=no of processes

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work = Available***
   (b) For $\boldsymbol{i = 0,1,2, \ldots, n\text{-}1}$, if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

   Initialize Work to Available and Finish to False

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i \leq Work}$

   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

Deadlock Detection

m=no of resources

n=no of processes

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 0,1,2, …, *n-1***, if ***Allocation$_i$* ≠ 0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

   Initialize Work to Available and Finish to False

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

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$_i$* ≤ *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

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 of Detection Algorithm

Initially

1. Work = [0, 0, 0] &
2. Finish = [false, false, false, false, false]

3. i=0 is selected as both
4. Finish[0] = false and [0, 0, 0]<=[0, 0, 0]
5. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
6. Finish = [true, false, false, false, false]

7. i=1 ,
8. Finish[1]=false and [2,0,2]!<=[0,1,0]
9. P1 should wait

10. *i=2* is selected as both
11. Finish[2] = false and [0, 0, 0]<=[0, 1, 0]
12. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3]
13. Finish = [true, false, true, false, false]

Check If
(a) ***Finish*[*i*] == *false***
(b) Request$_i$ ≤ Work
Then
Work = Work + Allocation$_i$
Finish*[*i*] = *true

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Example of Detection Algorithm

1. i=3 is selected as both
2. Finish[3] = false and [1, 0, 0]<=[3, 1, 3]
3. Work =[3, 1, 3]+[2, 1, 1] =>[5, 2, 4] &
4. Finish = [true, false, true, true, false]

5. *i=4, is selected as both*
6. *F*inish[4]=[false] and [0,0,2]<=[5,2,4]
7. *Work=[5,2,4]+[0,0,2]=[5,2,6]*
8. Finish = [true, false, true, true, true]

9. *i=1* is selected as both
10. Finish[1] = false and [2, 0, 2]<=[5, 2, 6]
11. Work =[5, 2, 6]+[2, 0, 0] =>[7, 2, 6]
12. Finish = [true, true, true, true, true].

No Finish[i] == false

No deadlock

Check If
(a) ***Finish*[*i*] == *false***
(b) Request$_i$ ≤ Work
Then
Work = Work + Allocation$_i$
Finish*[*i*] = *true

If ***Finish[i] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state.

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

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

Snapshot at time T0:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 1	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

- State of system?

Example of Detection Algorithm

1. Initially
2. Work = [0, 0, 0]
3. Finish = [false, false, false, false, false]

4. i=0 is selected as both
5. Finish[0] = false and [0, 0, 0]<=[0, 0, 0]
6. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
7. Finish = [true, false, false, false, false]

8. i=1 ,
9. Finish[1]=false and [2,0,2]!<=[0,1,0]
10. P1 should wait
11. Finish = [true, false, false, false, false]

12. i=2 is selected as both
13. Finish[2] = false and [0, 0, 1]!<=[0, 1, 0]
14. P2 should wait
15. Finish = [true, false, false, false, false]

Check If
(a) ***Finish*[*i*] == *false***
(b) Request$_i$ ≤ Work
Then
Work = Work + Allocation$_i$
Finish*[*i*] = *true

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Example of Detection Algorithm

1. i=3
2. Finish[3] = false and [1, 0, 0]!<=[0, 1, 0]
3. P3 should wait
4. Finish = [true, false, false, false, false]

5. *i=4,*
6. *F*inish[4]=[false] and [0,0,2]<=[0,1,0]
7. P4 should wait
8. Finish = [true, false, false, false, false]

- ***Finish[i] of P1,P2,P3,P4 = false***
- ***So deadlock***
- ***Thus process P1,P2,P3,P4 are deadlocked.***

Check If
(a) ***Finish*[*i*] == *false***
(b) Request$_i$ ≤ Work
Then
Work = Work + Allocation$_i$
Finish*[*i*] = *true

If ***Finish[i] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state.

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Example (Cont.)

- 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 should we invoke the detection algorithm?
- Depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

Detection-Algorithm Usage

- If deadlocks occur frequently,
  - then the detection algorithm should be invoked frequently.
  - Resources allocated to deadlocked processes will be idle until the deadlock can be broken.
  - In addition, the number of processes involved in the deadlock cycle may grow.

Detection-Algorithm Usage

- 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.**

Detection-Algorithm Usage

- Invoking the deadlock-detection algorithm **for every resource request**
  – will incur **considerable overhead** in computation time.

Detection-Algorithm Usage

- A less expensive alternative is simply to
  - **invoke the algorithm at defined intervals-**
  - **for example, once per hour or whenever CPU utilization drops below 40 percent.**

Recovery from Deadlock

- When a detection algorithm determines that
  - a deadlock exists,

- Several alternatives are available-
  - Let the operator deal with the deadlock manually.
  - Let the system *recover* from the deadlock automatically.

Recovery from Deadlock

- There are two options for breaking a deadlock
  - Process Termination
    - Abort one or more processes to break the circular wait.
  - Resource Preemption
    - Preempt some resources from one or more of the deadlocked processes

Recovery from Deadlock: Process Termination

- Eliminate deadlocks by aborting processes

- The system reclaims all resources allocated to the terminated processes.

- 2 Methods-
  - Abort all deadlocked processes

  - Abort one process at a time until the deadlock cycle is eliminated

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - Breaks the deadlock cycle, but **at great expense;**
  - The deadlocked processes **may have computed for a long time,**
  - The results of these partial computations must be discarded and probably will have **to be recomputed later.**

Recovery from Deadlock: Process Termination

- Abort one process at a time until the deadlock cycle is eliminated.
  - Incurs **considerable overhead**, since **after each process is aborted,**
  - **Deadlock-detection algorithm must be invoked** to determine whether any processes are still deadlocked.

Recovery from Deadlock: Process Termination

Examples-

- Aborting a process may not be easy.

1. If the process was in the midst of updating a file, terminating it
   - will leave that **file in an incorrect state**.
2. If the process was in the midst of printing data on a printer,
   - the **system must reset the printer** to a correct state before printing the next job.

Recovery from Deadlock: Process Termination

- Abort those processes whose termination will **incur the minimum cost.**

- In **which order** should we choose to abort?

Recovery from Deadlock: Process Termination

- Many factors may affect -
  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

- Successively preempt some resources from processes
- Give these resources to other processes until the deadlock cycle is broken.

- If preemption is used, 3 issues need to be addressed:
  - **Selecting a victim**
  - **Rollback**
  - **Starvation**

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** –
  - Which resources and Which Processes are to be pre-empted
  - determine the order of preemption to minimize cost.
  - Cost factors
    - as the number of resources a deadlocked process is holding
    - Amount of time the process has so far consumed during its execution

Recovery from Deadlock: Resource Preemption

- **Rollback** – return to some safe state, restart process from that state

Recovery from Deadlock: Resource Preemption

Rollback –

- If we pre-empt a resource from a process
- Clearly, it cannot continue with its normal execution;
- It is missing some needed resource.

What should be done with that process?

- We must roll back the process to some safe state and restart it from that state.

Recovery from Deadlock: Resource Preemption

Safe State-

- It is difficult to determine what a safe state is,
- The simplest solution is a total rollback: abort the process and then restart it.
- Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

Recovery from Deadlock: Resource Preemption

Starvation –

- **Same process may always be picked as victim,**
- How can we guarantee that resources will not always be pre-empted from the same process?
- Ensure that a process can be picked as a victim" only a (small) finite number of times
- **Include number of rollback in cost factor**