Chapter 7: Deadlocks

Narzu Tarannum

BRACU

BRAC UNIVERSITY

Inspiring Excellence

Covered in chapter 7

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

BRAC UNIVERSITY
Inspiring Excellence

Chapter Objectives

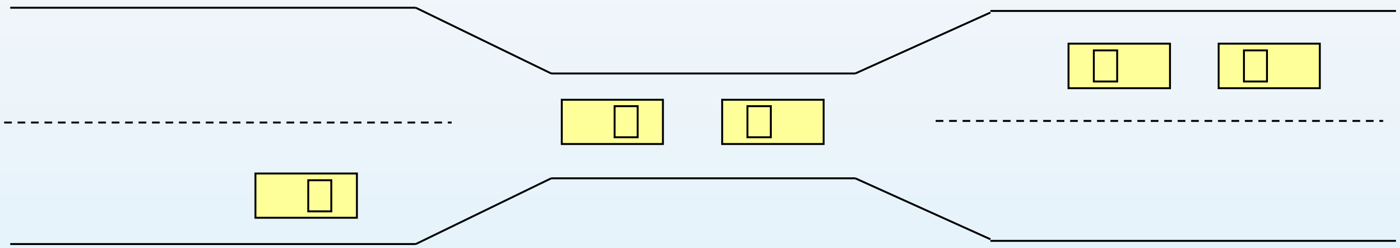
- To develop a description of **deadlocks**, which prevent sets of concurrent processes from completing their tasks.

- To present a number of different methods for **preventing or avoiding deadlocks** in a computer system.

BRAC UNIVERSITY
Inspiring Excellence

Bridge Crossing Example

Deadlock cause (a situation or opposing parties) to come to a point where no progress can be made because of fundamental disagreement.



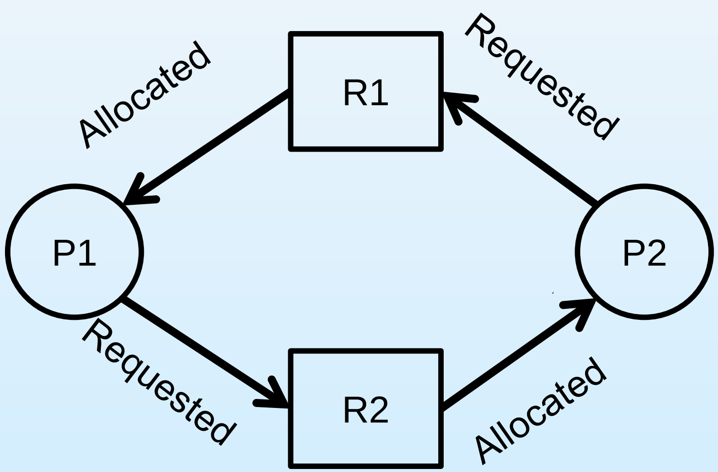
- Each section of a bridge can be viewed as a resource.
- Traffic moves only in one direction on the Bridge.
- Deadlock occurs!
- If deadlock occurs, it can be resolved if one car backs up.
- Several cars may have to be backed up.

- **Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.**

BRAC UNIVERSITY
Inspiring Excellence

The Deadlock Problem

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting process will never gain change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.



P1→R1 **Need R2**
P2→R2 **Need R1**

- Example
  - System has 2 Resources: R1, R2.
  - *P1* and *P2* each hold one resource and each needs another one.

BRAC UNIVERSITY
Inspiring Excellence

System Model

- A system consist of a finite number of resources to be distributed among a number of competing processes.
  - Resource types $R_1, R_2, . . ., R_m$
  - Process are $P_1, P_2, . . ., P_n$

- Resources are partitioned into several types

  - *Physical resource for example, CPU cycles, memory space, I/O devices .*
  - Logical resources for example, semaphores, mutex locks, and files.
- Each resource type consisting of some number of identical instance.
- Each resource type R_i has W_i instances.

BRAC UNIVERSITY
Inspiring Excellence

System Model

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  - **Request:** The process requests the resource.
  - **Use**: The process can operate on the resource
  - **Release**: The process releases the resource.

- **A process must request a resource before using it and must release the resource after using it.**

Necessary Conditions for Deadlock

Deadlock can arise if four conditions hold **simultaneously** in a system:

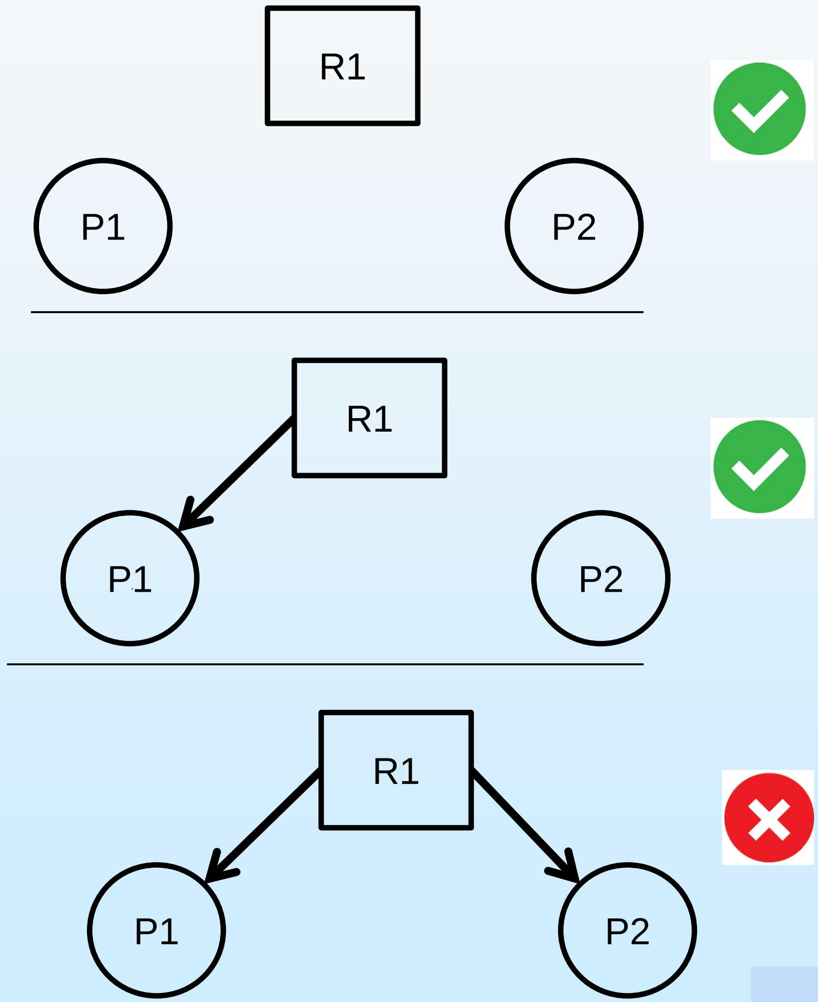
1. **Mutual exclusion**
2. **Hold and wait**
3. **No preemption**
4. **Circular wait**

If one of them is not present in a system, no deadlock will arise

BRAC UNIVERSITY
Inspiring Excellence

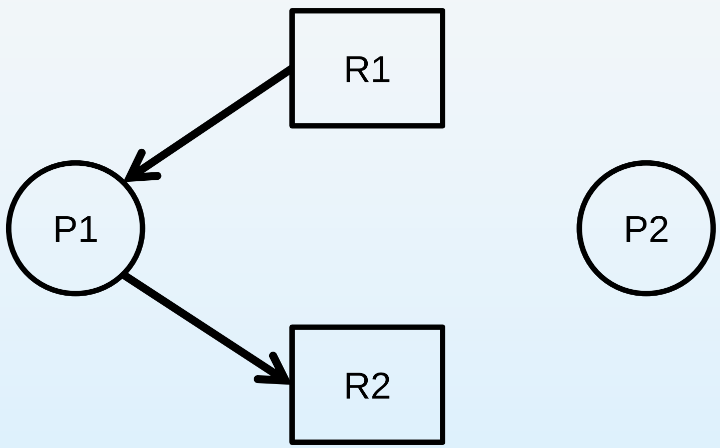
Necessary Conditions

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; only one process at a time can use a resource.



Necessary Conditions

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



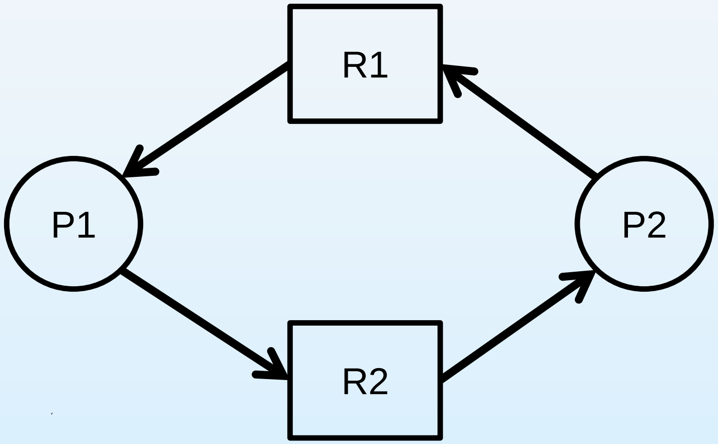
Necessary Conditions

3. No preemption:
a resource can be released only voluntarily by the process holding it, after that process has completed its task.

BRAC UNIVERSITY
Inspiring Excellence

Necessary Conditions

4. Circular wait:
There must be a circular chain of two or more processes each of which is waiting for a resource held by the next member of the chain.



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 $R_j \rightarrow P_i$

G → (*V*, *E*)

Process ← *V* → Resource

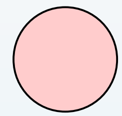
E → Request edge $P_i \rightarrow R_j$

E → Allocation edge $R_j \rightarrow P_i$

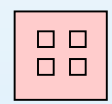
BRAC UNIVERSITY
Inspiring Excellence

Resource-Allocation Graph (Cont.)

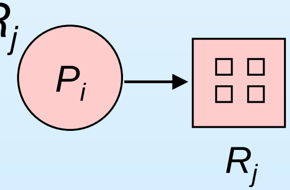
- Process



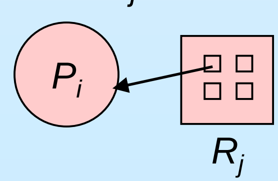
- Resource Type with 4 instances



- P_i requests instance of R_j

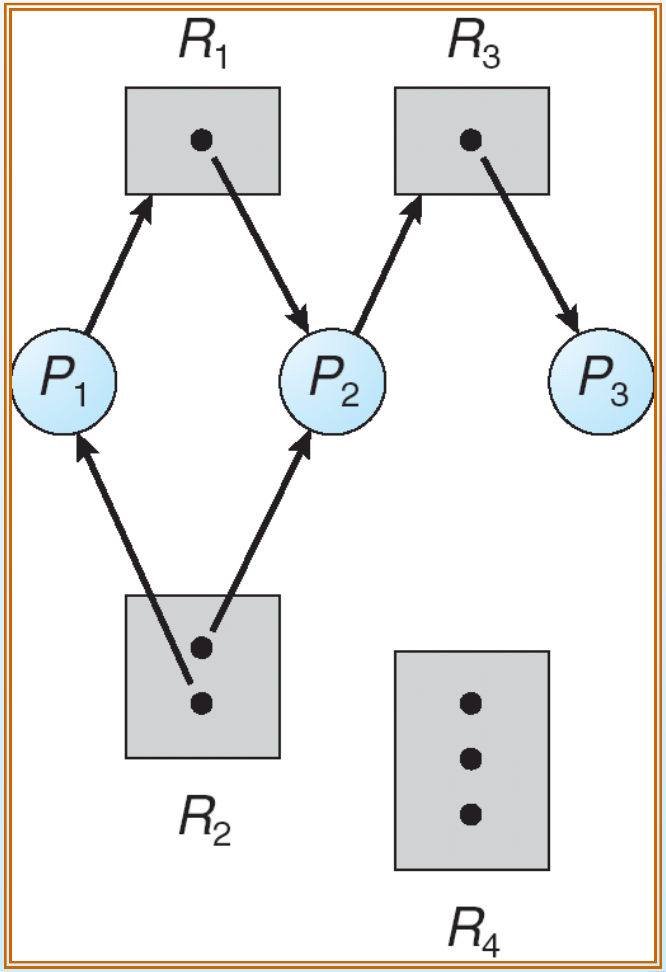


- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

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

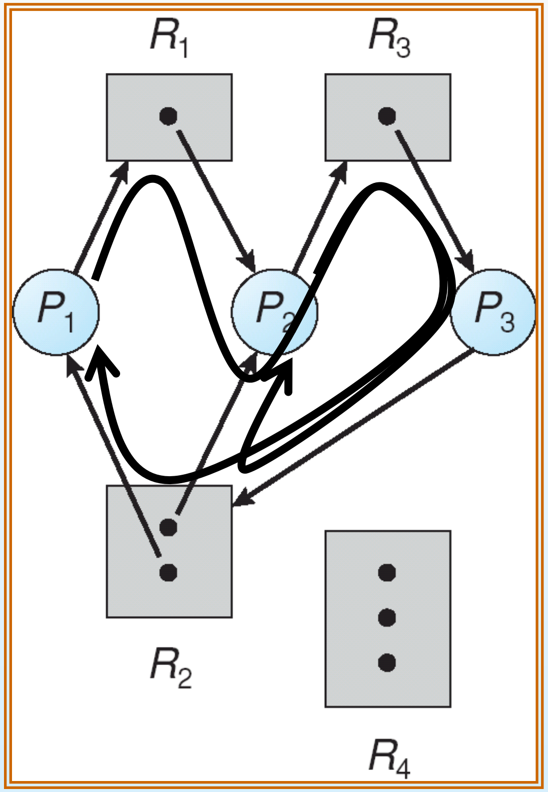


Basic Facts

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

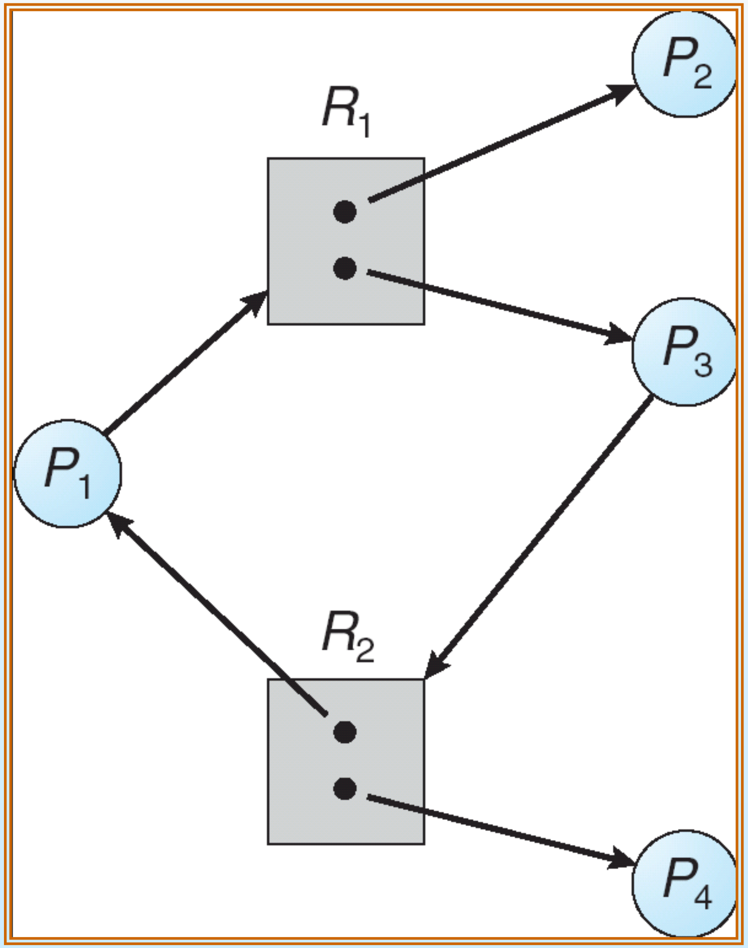
BRAC UNIVERSITY
Inspiring Excellence

Resource Allocation Graph With A Deadlock



- *In the figure At this point, two minimal cycles exist in the system:*
  - ***P1 → R1 → P2 → R3 → P3 → R2 → P1***
  - ***P2 → R3 → P3 → R2 → P2***

Resource Allocation Graph With A Cycle But No Deadlock



Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state (Deadlock prevention).
- In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.(Deadlock avoidance).
- Allow the system to enter a deadlock state and then recover (Deadlock detection and recovery).
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX. (Ignorance)

Methods for Handling Deadlocks

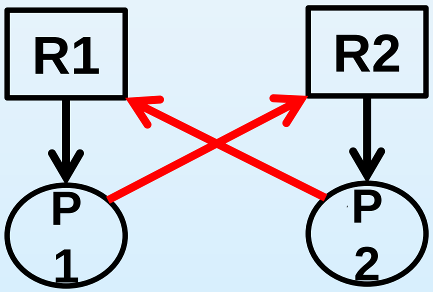
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery
- Ignorance

Deadlock Prevention

Eliminate one of the four conditions.

1.Mutual Exclusion –

- The mutual exclusion condition must hold for non-sharable resources. Example: a printer
- sharable resources, in contrast do not require mutually exclusive access and thus can not be involved in a deadlock. Example: read only files



In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-shareable.

Deadlock Prevention (Cont.)

2.Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.

- **First Protocol or conservative approach:** A process is allowed to start execution if and only if it has acquired all the resources.
- **Second Protocol or do not hold approach**: A process will acquire only desired resources but before making any fresh request, it must release all the resources that it currently hold.
  - Both this protocols have two main disadvantages:

    1. Poor resource utilization   2. starvation is possible
- **Third Protocol or wait-time out approach**: We place a maximum time bound up to which a process can wait for resources, after which they must release all the holding resources.

BRAC UNIVERSITY

Inspiring Excellence

R1, R2, … R4, R5, ………R10

Deadlock Prevention (Cont.)

3.No Preemption –

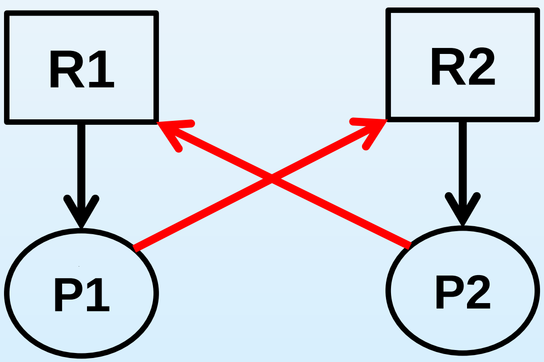
- If a process that is holding some resources requests another resource that cannot be immediately allocated to it(that is, the process must wait), then all resources currently being held are released or preempted.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

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

BRAC UNIVERSITY
Inspiring Excellence

Deadlock Prevention (Cont.)

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



P1	P2
R1	R2
R2	R1

P1	P2
R1	R1
R2	R2

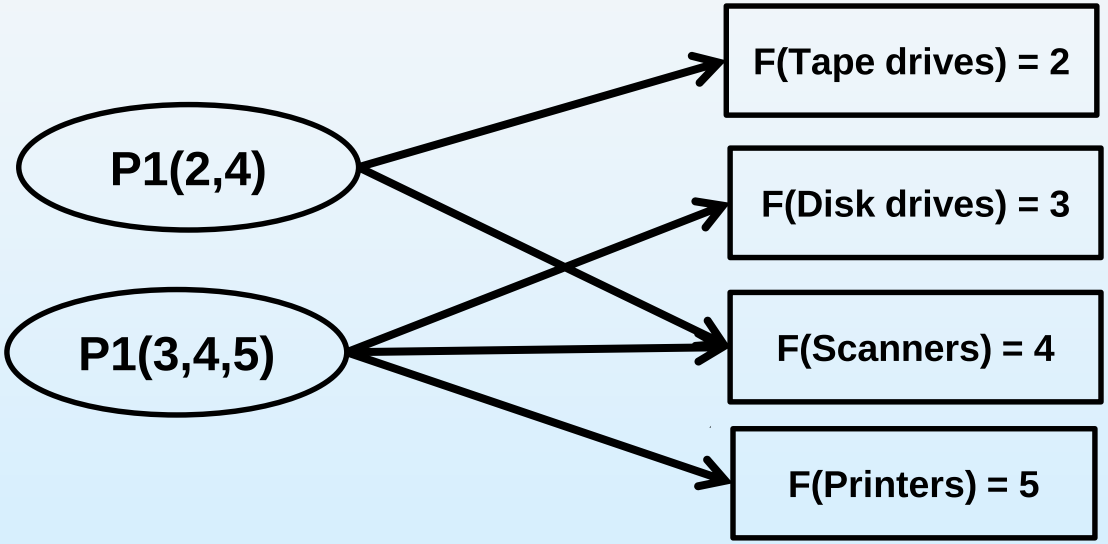
P1 → R2 → P2 → R1

BRAC UNIVERSITY
Inspiring Excellence

Deadlock Prevention (Cont.)

4.Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. Let R = R1, R2, ……., Rm are the resources.

-Circular wait can be eliminated by just giving the natural number of every resource. So, we can define the function *F: R → N*



Developing and ordering, or hierarchy, in itself does not prevent deadlock. It is up to application developer to write programs that follow the ordering.

BRAC UNIVERSITY
Inspiring Excellence

Methods for Handling Deadlocks

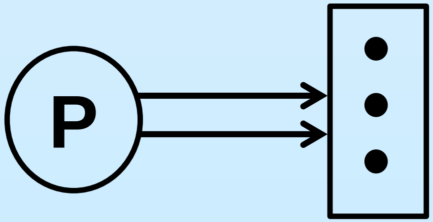
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery
- Ignorance

Deadlock Avoidance

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

- Banker's Algorithm requires the information of each process that 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.

With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.



Safe-Unsafe State

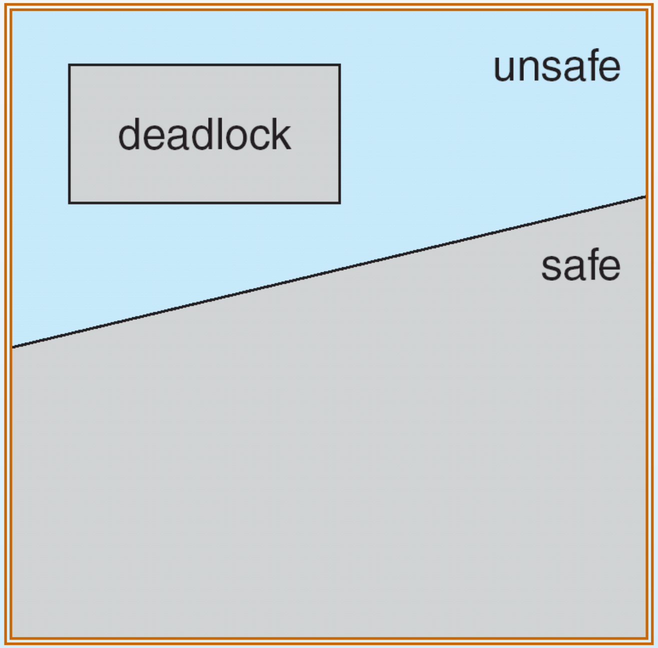
- A state is safe if the system can allocate resources to each process(up to its maximum) in some order and still avoid a deadlock.
  - A state is said to be safe, if there is some scheduling order in which every process can run to completion that means system is in safe state if there exists a safe sequence of all processes.
  - An unsafe state **does not have to** lead to a deadlock; it **could** lead to a deadlock.

ENSURE SYSTEM NEVER REACHES AN UNSAFE STATE

BRAC UNIVERSITY
Inspiring Excellence

Safe, Unsafe , Deadlock State

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock. Not all unsafe states are deadlocks, however.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



Example of a Safe-unsafe state

Let there are 3 processes (A, B, C) where total number of resources are 10. Every process has to declare maximum number of resource they need in advance. In present scenario system allocated 3 units of resources to A where maximum requirement of A is 9 units. In the same way system allocated 2 units to B where maximum requirement of B is 4 and system allocated 2 units to C where maximum requirement of C is 7

Total: 10 units

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3 units

BRAC UNIVERSITY
Inspiring Excellence

Safe state

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3 units

Allocate 2 units to B

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1 units

B completes

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5 units

Allocate 5 units to C

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0 units

C completes

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7 units

Allocate 6 units to A

	Has	Max
A	9	9
B	0	-
C	0	-

Free: 1 unit

BRAC UNIVERSITY
Inspiring Excellence

This is a safe state because there is some scheduling order in which every process executes. Here the order is: B→C→A

Unsafe state

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2 units

Allocate 2 units to B

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0 units

B completes

	Has	Max
A	4	9
B	0	-
C	2	7

Free: 4 units

5 units to C or A can not be allocated

	Has	Max
A	?	9
B	0	-
C	?	7

Free: 0 units

BRAC UNIVERSITY
Inspiring Excellence

This is an unsafe state because there exist no scheduling order in which every process executes.

Banker's Algorithm

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

BRAC UNIVERSITY
Inspiring Excellence

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

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

X<=Y if X=(1,7,3,2) and Y=(0,3,2,1) then X≠Y.

Banker's 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 and 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.

Example of Banker's Algorithm

- 3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances).
- 5 processes P_0 through P_4;
- Snapshot at time T_0:

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

BRAC UNIVERSITY
Inspiring Excellence

Example of Banker's 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 and 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.

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

Work = Available = 3 3 2

Finish[0]=Finish[1]=Finish[2]=Finish[3]=Finish[4]=False

Need(P0)>Work → Finish[0]=F → Do Nothing	7 4 3 ≤ 3 3 2 → F → do nothing
Need(P1)<=Work → Finish[1]=T → Work=Work + AllocationP1	1 2 2 ≤ 3 3 2 → T: Work = 3 3 2 + 2 0 0 = 5 3 2
Need(P2)>Work → Finish[2]=F → Do Nothing	6 0 0 ≤ 5 3 2 → F → do nothing
Need(P3)<=Work → Finish[3]=T → Work=Work + AllocationP3	0 1 1 ≤ 5 3 2 → T: Work = 5 3 2 + 2 1 1 = 7 4 3
Need(P4)<=Work → Finish[4]=T → Work=Work + AllocationP4	4 3 1 ≤ 7 4 3 → T: Work = 7 4 3 + 0 0 2 = 7 4 5
Need(P0)<=Work → Finish[0]=T → Work=Work + AllocationP0	7 4 3 ≤ 7 4 5 → T: Work = 7 4 5 + 0 1 0 = 7 5 5
Need(P2)<=Work → Finish[2]=T → Work=Work + AllocationP2	6 0 0 ≤ 7 5 5 → T: Work = 7 5 5 + 3 0 2 = 10 5 7

BRAC UNIVERSITY
Inspiring Excellence

Found the safe sequence <P1, P3, P4, P0, P2>

- Request for resources (1 0 2) for P1 will be granted at this state?

Banker's Resource-request Algorithm

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

$P_i \rightarrow Request_i$

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

4. Check Bankers safety Algorithm, if this new state is safe

If safe $\Rightarrow$ the resources are allocated to Pi.

If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored

BRAC UNIVERSITY

Inspiring Excellence

Example of Banker's Resource-request Algorithm

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

    $Available = Available - Request_i;$

    $Allocation_i = Allocation_i + Request_i;$

    $Need_i = Need_i – Request_i;$

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

Check a new additional request for P_1 (1 0 2) can be granted now?

1. if Request of P1 ≤ Need of P1	1 0 2 <= 1 2 2 → T
2. if Request of P1 ≤ Available	1 0 2 ≤ 3 3 2 → T
3. $Available = Available – Request_i$ $Allocation_i = Allocation_i + Request_i$ $Need_i = Need_i – Request_i$	Available = 3 3 2 – 1 0 2 = 2 3 0 Allocation = 2 0 0 + 1 0 2 = 3 0 2 Need = 1 2 2 – 1 0 2 = 0 2 0

BRAC UNIVERSITY

Inspiring Excellence

Example of Banker's Resource-request Algorithm

3. Available = 3 3 2 – 1 0 2 = 2 3 0
   Allocation = 2 0 0 + 1 0 2 = 3 0 2
   Need = 1 2 2 – 1 0 2 = 0 2 0

Update table after additional request for P_1 (1 0 2)

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

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

BRAC UNIVERSITY
Inspiring Excellence

Verify with Banker's 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 and 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.

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

Need(P0)>Work → Finish[1]=F → Do Nothing	7 4 3 ≤ 2 3 0 → F → do nothing
Need(P1)<=Work → Finish[1]=T → Work=Work + Allocation	0 2 0 ≤ 2 3 0 → T: Work = 2 3 0 + 3 0 2 = 5 3 2
Need(P2)>Work → Finish[2]=F → Do Nothing	6 0 0 ≤ 5 3 2 → F → do nothing
Need(P3)<=Work → Finish[3]=T → Work=Work + Allocation	0 1 1 ≤ 5 3 2 → T: Work = 5 3 2 + 2 1 1 = 7 4 3
Need(P4)<=Work → Finish[4]=T → Work=Work + Allocation	4 3 1 ≤ 7 4 3 → T: Work = 7 4 3 + 0 0 2 = 7 4 5
Need(P0)<=Work → Finish[0]=T → Work=Work + Allocation	7 4 3 ≤ 7 4 5 → T: Work = 7 4 5 + 0 1 0 = 7 5 5
Need(P2)<=Work → Finish[2]=T → Work=Work + Allocation	6 0 0 ≤ 7 5 5 → T: Work = 7 5 5 + 3 0 2 = 10 5 7

BRAC UNIVERSITY
Inspiring Excellence

Since there is a safe sequence <P1, P3, P4, P0, P2> ,
Additional request P_1 (1 0 2) can be granted

Are following requests get accepted?

- Request for (3 3 0) resources for P4
- Request for (0 2 0) resources for P0

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

Are following requests get accepted?

- Request for (3 3 0) resources for P4
- Request for (0 2 0) resources for P0

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

BRAC UNIVERSITY
Inspiring Excellence

Thank You