# Operating System

**Prof. Prachi Verma, Prof. Manish Kumar** Assistant Professor, Computer Science & Engineering

# CHAPTER-4

## Deadlocks

# Deadlock Definition

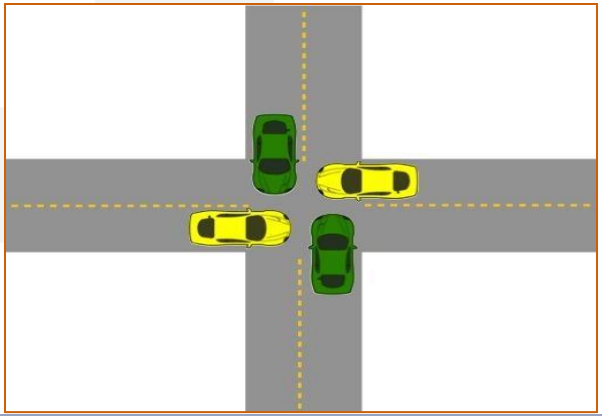
**Deadlock:**

- In a multiprogramming system, several process compete for limited number of resources and if resource is not available at that instance then process enters waiting state

- If a process is unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process, then system is said to be in deadlock.

# Example: Bridge Crossing

- Assume traffic in all four direction
- Each section of the bridge is viewed as a resource.
- If a deadlock occurs, it can be resolved only if one car backs up (pre-empt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible



# System Model

- It is three step model

i. Every process will request for the resource

ii. If entertained, then process will use the resource

iii. Process must release the resource after use

# Necessary and sufficient conditions for Deadlock

- There are 4 necessary and sufficient conditions for deadlock to occur
- **Mutual Exclusion**: Atleast one resource type in the system which can be used in non sharable mode i.e., mutual exclusion(one at a time/ one by one). Example: Printer

- **Hold & Wait**: A process is currently holding at least one resource and requesting additional resources which are being held by other processes.

# Necessary and sufficient conditions for Deadlock cont...

- **No pre-emption**: A resource cannot be pre-empted that is a resource will be released by the process after completion of its task, voluntarily,

- **Circular Wait**: Each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

# Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly using **Resource-Allocation Graphs**, having the following properties:
- A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. ( E.g. two dots might represent two laser printers. )
- A set of processes, { P1, P2, P3, . . ., PN }
- **Request Edges -** A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.

# Resource-Allocation Graph

- **Assignment Edges -** A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted.

# Example

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are ***directed*** graphs. )
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock but does not guarantee one.

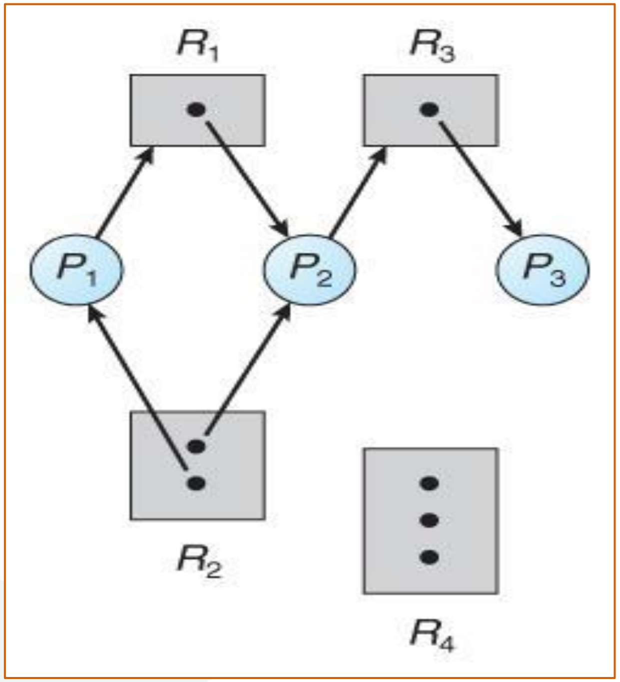



Fig: **Resource allocation graph**

# Example

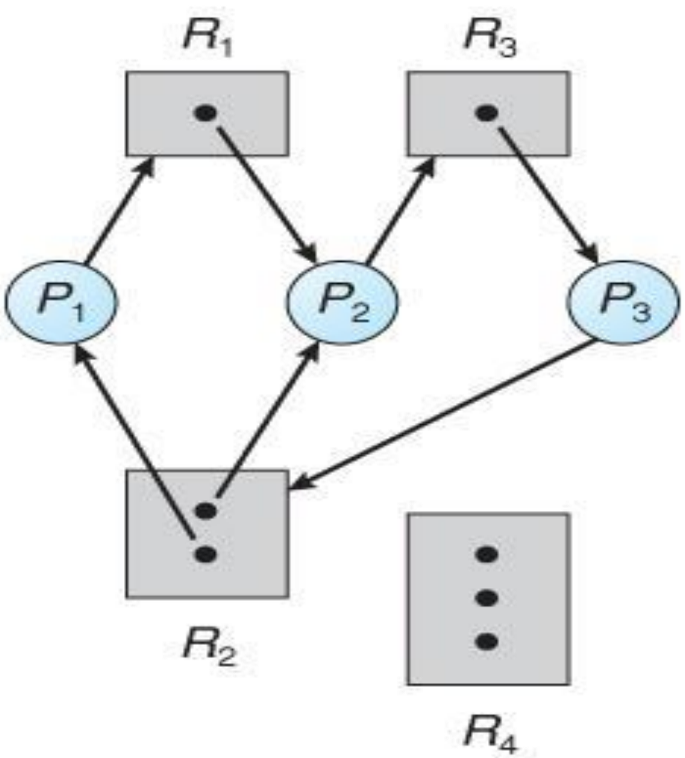



Fig: **Resource allocation graph with a deadlock**

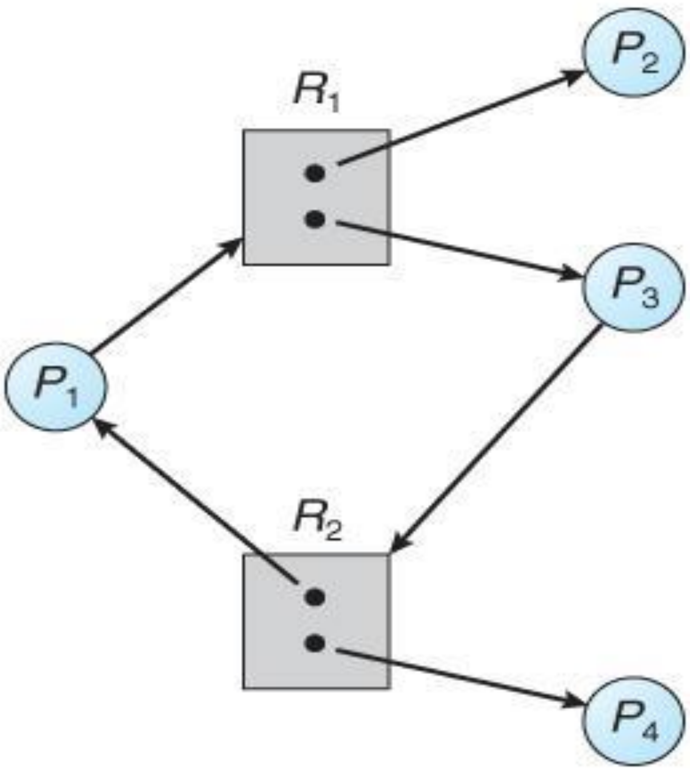



Fig: **Resource allocation graph with a cycle but no deadlock**

# Deadlock handling methods

- Prevention: It means to design such a system which violate atleast one of the four necessary condition of deadlock and ensure that deadlock should not occur.
- Avoidance: System maintains a set of data using which it takes a decision whether to entertain a new request or not, to be in safe state.
- Detection & recovery: In this we wait untill the deadlock occurs and once we detect it, we recover from it.
- Ignorance: We ignore the problem as if it does not exist.

# Deadlock Prevention

Mutual Exclusion:

- If a resource is assigned to more than one process, i.e., if a resource is made sharable then deadlock will not occur

- However based on hardware some resources cannot be shared among several processes at a time. For example: Printer, CD recorder, etc...

- So this prevention technique is not feasible.

# Deadlock Prevention

Hold & Wait:

- Conservative approach: Process is allowed to start execution if and only if it has acquired all the resources(less efficient, not implementable, easy, deadlock independence).
- Do not hold: Process will acquire only desired resources, but before making any fresh request it must release all the resources that is currently held. (efficient, implementable).
- Wait timeouts: We place a maximum time upto which a process can wait. After which process must release all the holding resources.

# Deadlock Prevention

No pre-emption:

- Forcefull pre-emption: We allow a process to forcefully pre-empt the resource holding by other processes.
- This method may be used by high priority process or system process.
- The process which are in waiting state must be selected as a victim instead of process in the running state.

# Deadlock Prevention

Circular wait:

- Circular wait can be eliminated by just giving a natural number to every resource

$$f:N\rightarrow R$$

- Allow every process to make request either only in the increasing or decreasing order of the resource number.
- If a process require a resource of lesser number (in case of increasing order), than it must first release all the resources larger than required number.

# Deadlock Avoidance

- If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - Resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

# Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence <P1, P2, …Pn> is safe, if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by Pj with j<i.

# Safe State

Possible states are:

**Deadlock** No forward progress can be made.

**Unsafe state** ***Possibility*** of deadlock.

**Safe state** A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.

# Safe State

The rule is simple: **If a request allocation would cause an unsafe state, do not honor that request.**

**NOTE: All deadlocks are unsafe, but all unsafe are NOT deadlocks.**

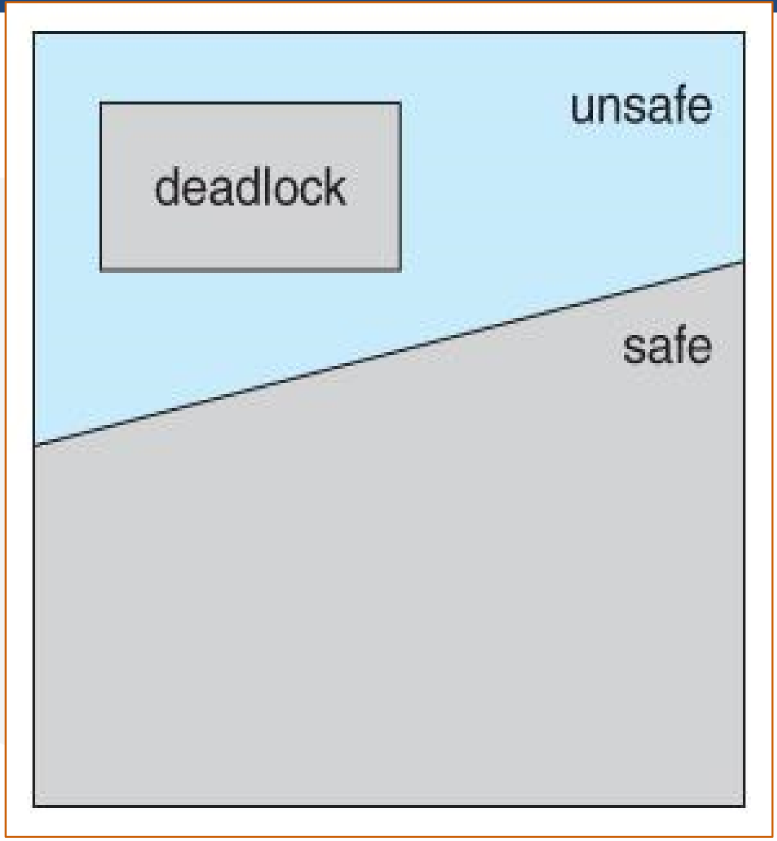



Fig:**Safe, unsafe, and deadlocked state spaces**

# Banker's Algorithm

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

# Data Structure for the Banker's Algorithm

- The banker's algorithm relies on several key data structures: ( where n is the number of processes and m is the number of resource categories. )
  - Available[ m ] indicates how many resources are currently available of each type.
  - Max[ n ][ m ] indicates the maximum demand of each process of each resource.
  - Allocation[ n ][ m ] indicates the number of each resource category allocated to each process.

# Banker's Algorithm

- Need[ n ][ m ] indicates the remaining resources needed of each type for each process. ( Note that Need[ i ][ j ] = Max[ i ][ j ] - Allocation[ i ][ j ] for all i, j. )

- For simplification of discussions, we make the following notations / observations:
  - One row of the Need vector, Need[ i ], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
  - A vector X is considered to be <= a vector Y if X[ i ] <= Y[ i ] for all i.

# Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
- Let Work and Finish be vectors of length m and n respectively.

▪ Work is a working copy of the available resources, which will be modified during the analysis.

▪ Finish is a vector of booleans indicating whether a particular process can finish.

▪ Initialize Work to Available, and Finish to false for all elements.

## Safety Algorithm

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

- *Work* := *Available*
- *Finish*[*i*] := *false* for *i* = 1,2,…,n.

Step 2:Find an *i* (i.e. process *Pi*) such that both:

- *Finish*[*i*] = *false*
- *Need_i* <= *Work*
- If no such *i* exists, go to step 4.

# Safety Algorithm

*Step 3: Work* := *Work* + Allocation_*i*

- *Finish*[*i*] := *true*
- go to step 2

Step 4: If *Finish*[*i*] = *true* for all *i*, then the system is in a safe state.

# Resource-Request Algorithm(Banker’s Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made, pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
  - Let Request[ n ][ m ] indicate the number of resources of each type currently requested by processes. If Request[ i ] > Need[ i ] for any process i, raise an error condition.

# Resource-Request Algorithm(Banker's Algorithm)

- If Request[ i ] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
- Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely. The procedure for granting a request is:
  - Available = Available - Request
  - Allocation = Allocation + Request
  - Need = Need - Request

# Banker's algorithm for single resource

- What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied.
- If granting the request leads to a safe state, it is carried out.
- If we have situation as per figure
  - then it is safe state
  - because with 10 free units
  - one by one all customers can be served.

Parul® University

DIGITAL LEARNING CONTENT

# Banker's algorithm for single resource

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |
| Free : 2 | | |

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 4 | 4 |
| D | 4 | 7 |
| Free : 0 | | |

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| ~~C~~ | ~~0~~ | ~~0~~ |
| D | 4 | 7 |
| Free : 4 | | |

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| ~~C~~ | ~~0~~ | ~~-~~ |
| D | 7 | 7 |
| Free : 1 | | |

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| ~~C~~ | ~~0~~ | ~~-~~ |
| ~~D~~ | ~~0~~ | ~~-~~ |
| Free : 8 | | |

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 5 | 5 |
| ~~C~~ | ~~0~~ | ~~-~~ |
| ~~D~~ | ~~0~~ | ~~-~~ |
| Free : 4 | | |

# Banker's algorithm for single resource

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| ~~B~~ | ~~0~~ | - |
| ~~C~~ | ~~0~~ | - |
| ~~D~~ | ~~0~~ | - |
| Free : 9 | | |

| Process | Has | Max |
|---|---|---|
| A | 6 | 6 |
| ~~B~~ | ~~0~~ | - |
| ~~C~~ | ~~0~~ | - |
| ~~D~~ | ~~0~~ | - |
| Free : 4 | | |

| Process | Has | Max |
|---|---|---|
| ~~A~~ | ~~0~~ | - |
| ~~B~~ | ~~0~~ | - |
| ~~C~~ | ~~0~~ | - |
| ~~D~~ | ~~0~~ | - |
| Free : 10 | | |

# Banker’s algorithm for single resource

- The order of execution is C, D, B, A. So if we can find proper order of execution then there is no deadlock.

| Process | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 2 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |
| Free : 1 | | |

# Examples of Banker's Algorithm

Consider the following situation

| | 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 |

# Examples of Banker's Algorithm

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. ( Request[ 1 ] = ( 1, 0, 2 ) )
- What about requests of ( 3, 3,0 ) by P4? or ( 0, 2, 0 ) by P0? Can these be safely granted? Why or why not?

| | *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 | |

# Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources pre-empted

# Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.
- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

# Single Instance of Each Resource Type

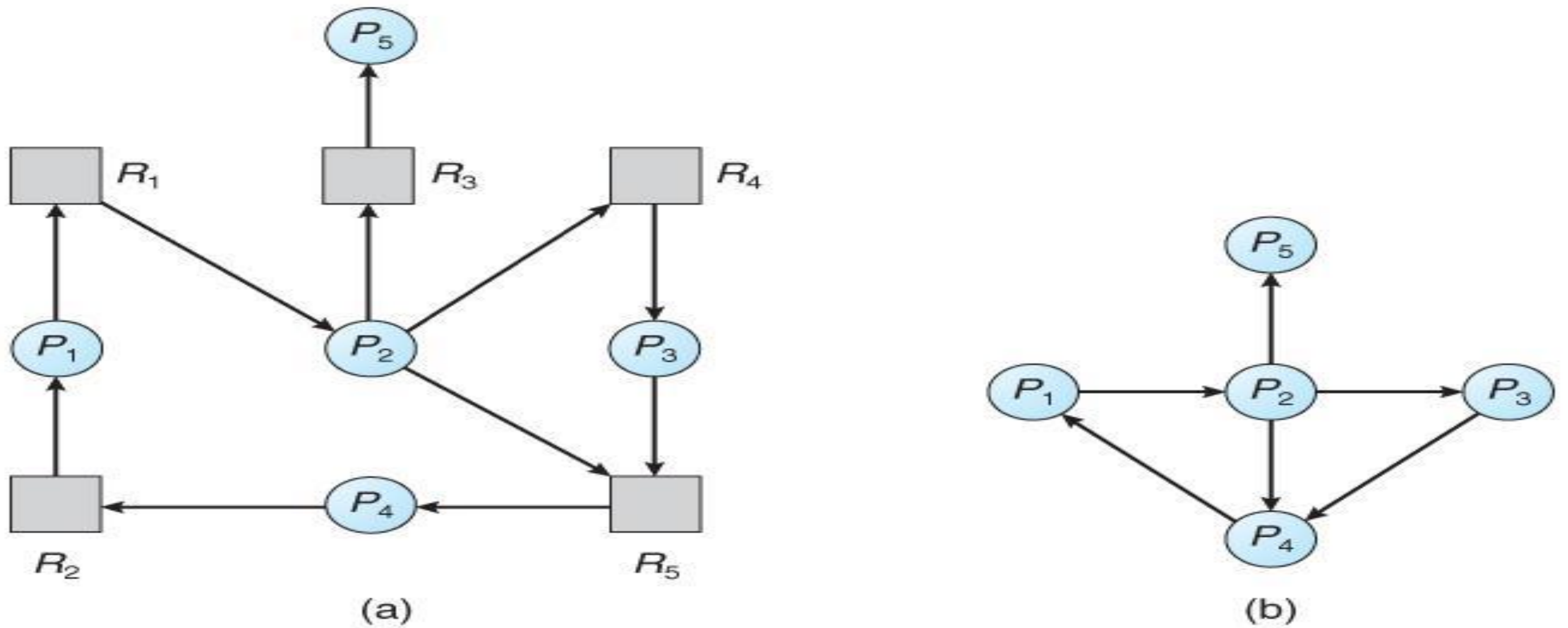



Fig. **(a) Resource allocation graph. (b) Corresponding wait-for graph**

# Several Instance of Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
- In step 1, the Banker's Algorithm sets Finish[ i ] to false for all i. The algorithm presented here sets Finish[ i ] to false only if Allocation[ i ] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[ i ] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resource allocated cannot be involved in a deadlock, and so can be removed from any further consideration.

## Several Instance of Resource Type

- Steps 2 and 3 are unchanged
- In step 4, the basic Banker's Algorithm says that if Finish[ i ] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[ i ] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

# Example

Consider, for example, the following state, and determine if it is currently deadlocked

| | 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 | |

Ubuntu OS Source: Google

# Example

Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

| | 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 1 | |
| $P_3$ | 2 1 1 | 1 0 0 | |
| $P_4$ | 0 0 2 | 0 0 2 | |

# Detection-Algorithm Usage

- When should the deadlock detection be done? Frequently, or infrequently? The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately.
- There are two obvious approaches, each with trade-offs:
- Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. The downside of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.

# Detection-Algorithm Usage

- Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the downside is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.

# Deadlock Recovery

- When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock.
- There are two approaches of breaking a Deadlock:

1. **Process Termination**
2. **Resource Pre-emption**

# Deadlock Recovery

1. **Process Termination:**

- To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

a) **Abort all the Deadlocked Processes:**

- Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.

# Deadlock Recovery

**b) Abort one process at a time until deadlock is eliminated:**

Abort one deadlocked process at a time, until deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

# Deadlock Recovery

**2. Resource Preemption:**

- To eliminate deadlocks using resource preemption, we preepmt some resources from processes and give those resources to other processes. This method will raise three issues –

**a) Selecting a victim:**

- We must determine which resources and which processes are to be preempted and also the order to minimize the cost.

b) **Rollback:**

- We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.

# Deadlock Recovery

**(c). Starvation:**

- In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called Starvation and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

Parul® University

DIGITAL LEARNING CONTENT

# References


[1] Silberschatz, A., Galvin, P. B., & Gagne, G. (2005). Operating system concepts. Hoboken, NJ: J. Wiley & Sons.

[2] Stallings, W. (2018). Operating systems: Internals and design principles. Prentice-Hall

[3] Tanenbaum, A. (2014). Modern operating systems. Harlow: Pearson.

[4] Nutt, G. J. (2004). Operating systems: A modern perspective. Boston: Pearson/Addison Wesley.

[5] Bower T. Operating System Structure. K–State Polytechnic. http://faculty.salina.k-state.edu/tim/ossg/Introduction/struct.html

[6] Bower T. Basic Operating System Concepts. K–State Polytechnic. http://faculty.salina.k-state.edu/tim/ossg/Introduction/OSrole.html

# DIGITAL LEARNING CONTENT


Parul® University

www.paruluniversity.ac.in