



# Operating Systems ICS 431

Weeks 8-9

**Deadlock Handling**

**Dr. Tarek Helmy El-Basuny**

## Reminder: Ch. 6 Process Synchronization

- To practice collaborative learning as we agreed, to foster self learning skill, and to help us progress in the course.
- You need to study Ch. 6 “Process Synchronization” by yourself and it will be included in the Major Exam II.
- Office hours can be used to answer any inquires about this chapter.
- Your objectives out of reading this chapter is to know:
  - Why synchronization? Or the necessity of synchronization by the OS.
  - How do processes work with resources that must be shared between them?
  - What is a critical section?
  - Dangers of handling the critical section without synchronization.
  - How to ensure that only one process can get access to the critical section?
  - What is atomic operation? **It executes without interruptions, all or none.**
  - Different algorithms to synchronize processes entering into the critical section.
  - Synchronization tools.
  - Semaphores, and types of Semaphores.
  - Incorrect usages of Semaphores.
  - Monitors.
  - Classical problems of synchronization.
  - Synchronization methods in different OSs.
- Evaluating synchronization algorithms of handling a critical section where every **algorithm should allow Mutual Exclusion, Progress and bounded waiting.**

# Deadlocks

- Computer's System Resources
- **Deadlock**, **Live lock/Busy waiting**, **Starvation** definitions,
- Deadlock examples in computer systems and real life,
- Fundamental causes of Deadlock
- Computer System Model (Processes, Recourses, Requests)
- Resource Allocation Graph to visualize the computer system.
- **Methods for Handling Deadlocks**
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Deadlock Recovery
- **Integrated approaches to handle Deadlock in current OSs.**

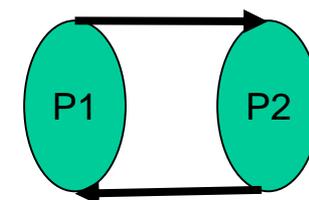
# Computer's System Resources

- **Reusable resources**
  - A reusable resource is the one that can be safely used by the process at a time and then reused by another process at different time.
  - The process **requests** the resource, **acquires** it, and then **releases** it.
    - **Examples:** Processors, I/O channels, main and auxiliary memory blocks, I/O devices, files (i.e. data bases), semaphores, etc...
- **Consumable resources**
  - A consumable resource is the one that will be created, used and destroyed. **It can not be reused by another process.**
  - The number of consumable resources is usually unlimited.
    - **Examples:** Interrupts, Signals, Messages, Information in I/O buffers, etc.
- **Preemptable resources.**
  - Can be taken away from a process with no ill effects (e.g. Memory, CPU,..).
- **Nonpreemptable resources**
  - Will cause the process to fail if it is taken away (e.g. CD, ...)
- **Deadlock is the permanent blocking of a set of processes that either compete for system resources or communicate with each other.**

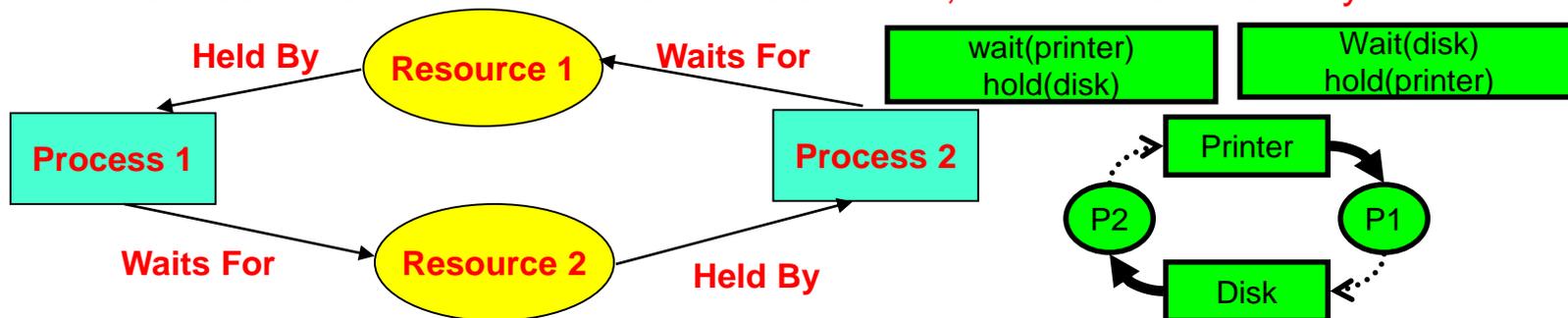
# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- **Example 1**
  - System has a Printer and a Disk.
  - $P_1$  and  $P_2$  each holds one resource and needs another one.
  - $P_1$  and  $P_2$  cannot progress, **will wait for each other**.
- Starvation vs. **deadlock** vs. Livelock
  - **Deadlock**: circular waiting (**without progressing**) of processes for resources.
  - **Live lock/Busy waiting**: processes run but make no progress, i.e. looping
  - A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.
  - **Starvation**: a process/thread **is temporary waiting** to gain regular access to shared resources and is unable to make progress now **but will latter progress**.
  - Both deadlock and livelock lead to starvation, but not the other way

“P1 is waiting for P2”



“P2 is waiting for P1”

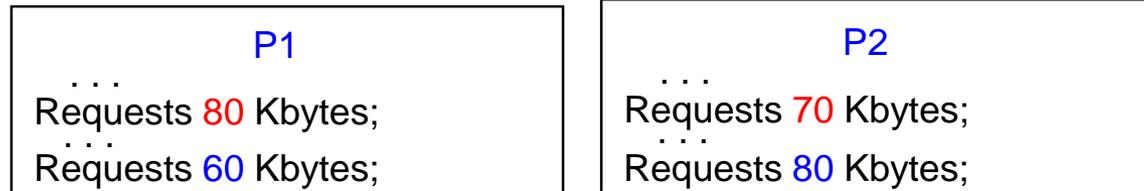


## Deadlock Types

- People sometimes classify deadlock into the following types:
  - **Resources deadlocks:** when processes are waiting for each other due to limited number of resources.
    - A process needs multiple resources for an activity.
    - Deadlock occurs if each process in a set requests a resource held by another process in the same set, and it must receive the requested resource to move further.
  - **Communication deadlocks:** when processes are waiting for each other due to lost of communicating messages.
    - Processes wait to communicate with other processes in a set.
    - Each process in the set is waiting on another process's message, and no process in the set initiates a message until it receives a message for which it is waiting.

## Example of Resources Deadlock: Due to shortage of memory

- 200K bytes of memory space is available for allocation, and the following sequence of events occur.



- Deadlock occurs if both processes progress to their second request. Why?

**Fig. 6.2** A simple deadlock. This system is deadlocked because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.

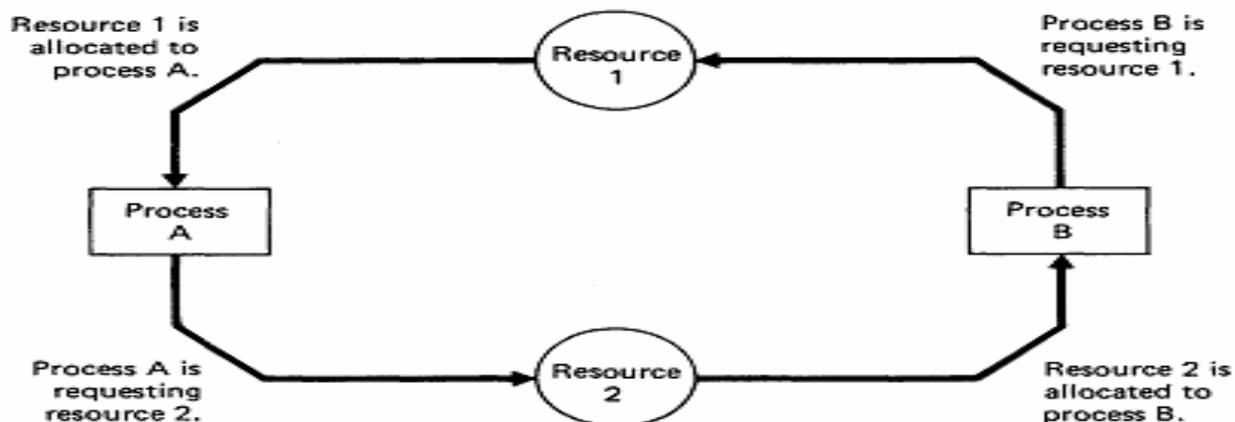
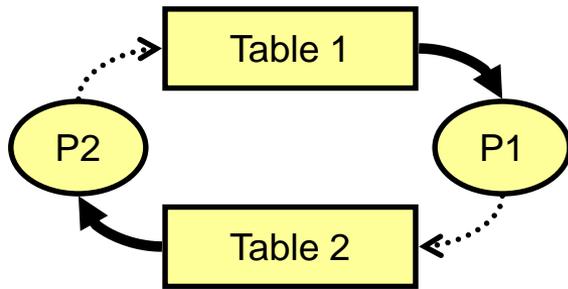
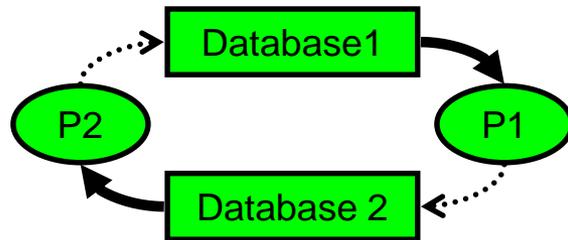


Figure 6.2 (page 157)

Miller 30

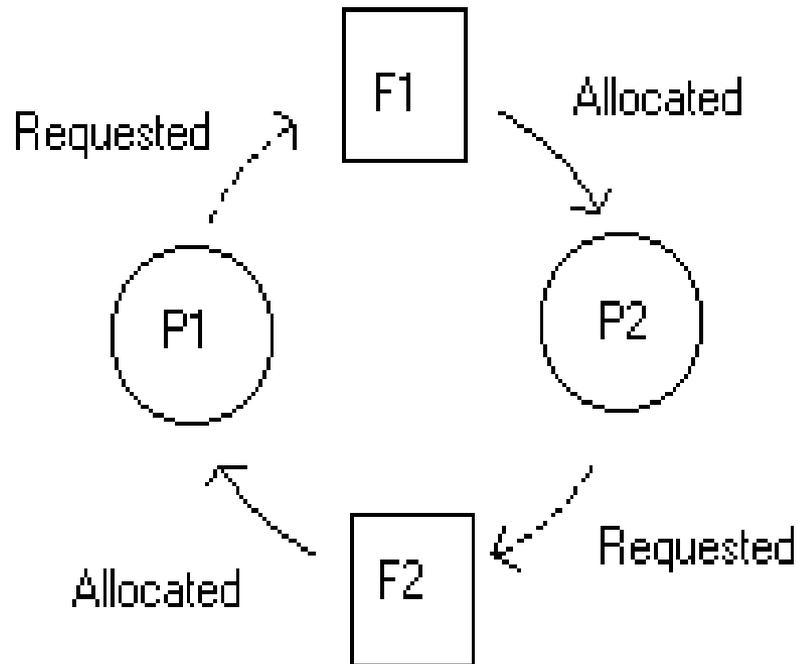


## Example: Deadlocks in Databases Access



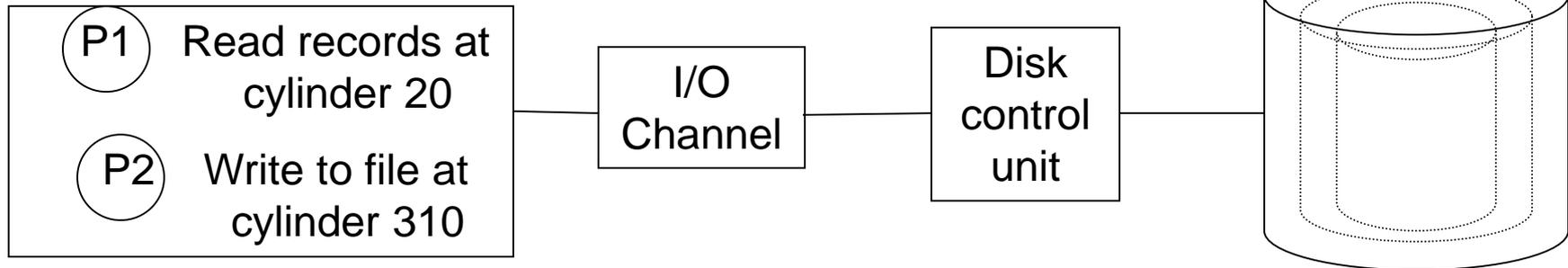
- Deadlock can occur if 2 processes access & lock records or Tables or even the whole database.
- 3 different levels of locking :
  - The entire database for duration of request.
  - A subsection of the database i.e. Table.
  - Individual record until the process is completed.
- If don't use locks, can lead to a **race condition**.
- This particular type of deadlock is easily prevented by using an all-or-none resource allocation algorithm.

## Example: Deadlocks on File Requests



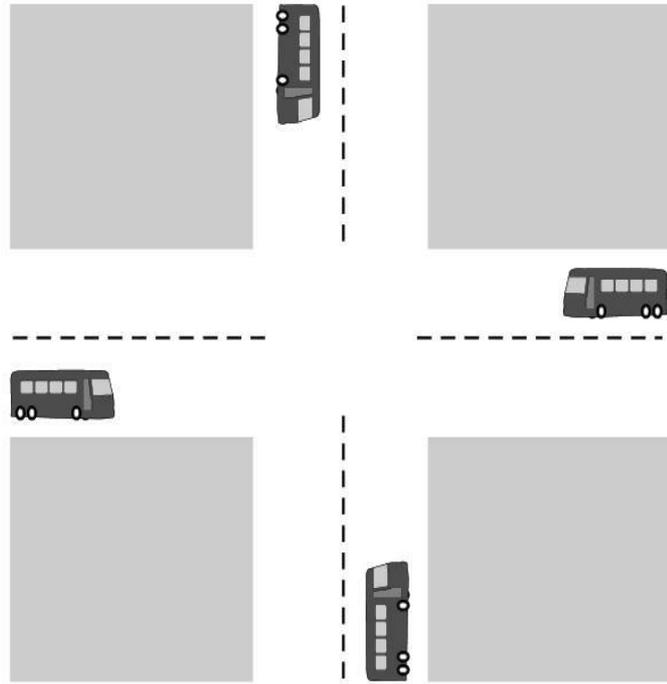
- If processes can request and hold files for duration of their execution, deadlock can occur.
- If only two files and two processes, this may lead to starvation !!
- But, if other processes that require F1 or F2 are put on hold as long as this situation continues, this will lead to deadlock !
- Deadlock remains until a process is withdrawn or powerfully removed and its file is released.

## Deadlocks in Disk Sharing



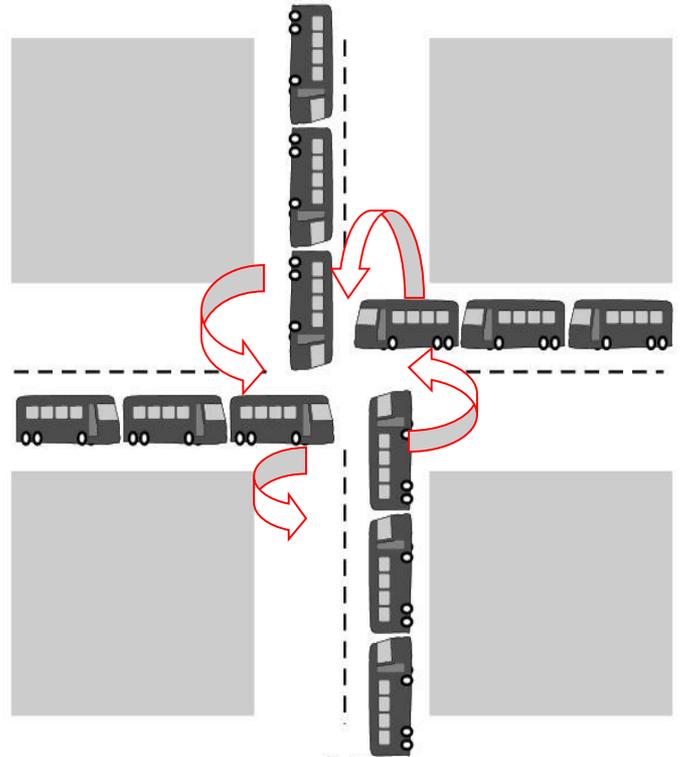
- Disks are designed to be shared, so it is common for processes to access different areas of same disk.
- Without controls to regulate use of disk drive, **competing processes could send conflicting commands and deadlock the system.**

# Real Deadlock in the daily life



(a)

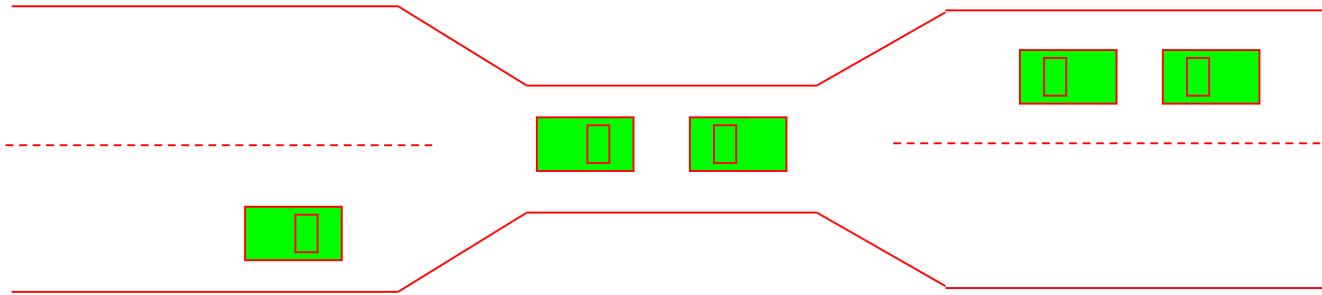
(a) Potential deadlock



(b)

(b) Actual deadlock.

## Bridge Crossing Deadlock Example



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

# Deadlock Causes

**Fundamental causes of deadlocks:** Four necessary and sufficient conditions for a deadlock to occur. All of these conditions must be present for a deadlock to occur.

- If one of these conditions is absent, **no deadlock is possible**.

## 1. Mutual exclusion condition:

- Each resource is either currently assigned to exactly one process or a resource that cannot be used by more than one process at a time.

## 2. Hold and wait condition:

- Processes currently holding resources granted earlier can request new resources or processes already holding resources may request new resources.

## 3. No preemption condition:

- Resources previously granted cannot be taken away from a process. They must be explicitly released by the process holding them.

## 4. Circular wait condition:

- There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource held by  $P_1$ ,  $P_1$  is waiting for a resource held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource held by  $P_0$ .

# Computer System Model

- Any computer system consists of:
  - Processes running in the system, i.e.  $P_1, P_2, \dots, P_n$
  - Resource available (HW or SW), i.e.  $R_1, R_2, \dots, R_m$ 
    - CPU cycles, memory spaces, I/O devices, signals, messages, ....
- Each resource type  $R_i$  may have  $W_i$  instances.
  - The blocks of the main memory or the disk can be given to more than processes.
- Each process utilizes a resource as follows:
  - Requests it, if the request can not be granted immediately, then the process must wait.
  - Uses it, the process can operate on the resource.
  - Releases it, the process releases the resource either voluntarily or preempted.



(a) Resource is requested



(b) Resource is held

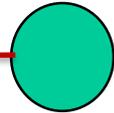
## Resource-Allocation Graph (RAG)

- A graphical way to visualize the computer system to determine if a deadlock may occur or no.
- Basic components of any RAG are set of vertices  $V$  and a set of edges  $E$ .
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system.
- $E$  can be either:
  - **Request edge** – directed edge  $P_i \rightarrow R_j$
  - **Assignment edge** – directed edge  $R_j \rightarrow P_i$
- An arrow from the **process** to resource indicates the process is **requesting** the resource.
- An arrow from a resource to a process means an instance of the resource has been **allocated/assigned** to the process.

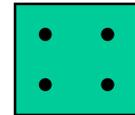
# Resource Allocation Graph

- In the RAG, the process is represented by a circle, and the resource is represented by a square.

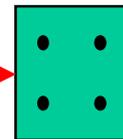
A Process



A Resource type with 4 instances

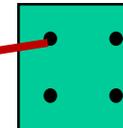
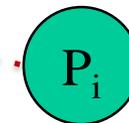


$P_i$  requests instance of  $R_j$



$R_j$

$P_i$  is holding an instance of  $R_j$



$R_j$

Dots represent number of instances of a resource type.

- A request edge points to the resource, and an assignment edge comes from the resource instances (dots) to the process.

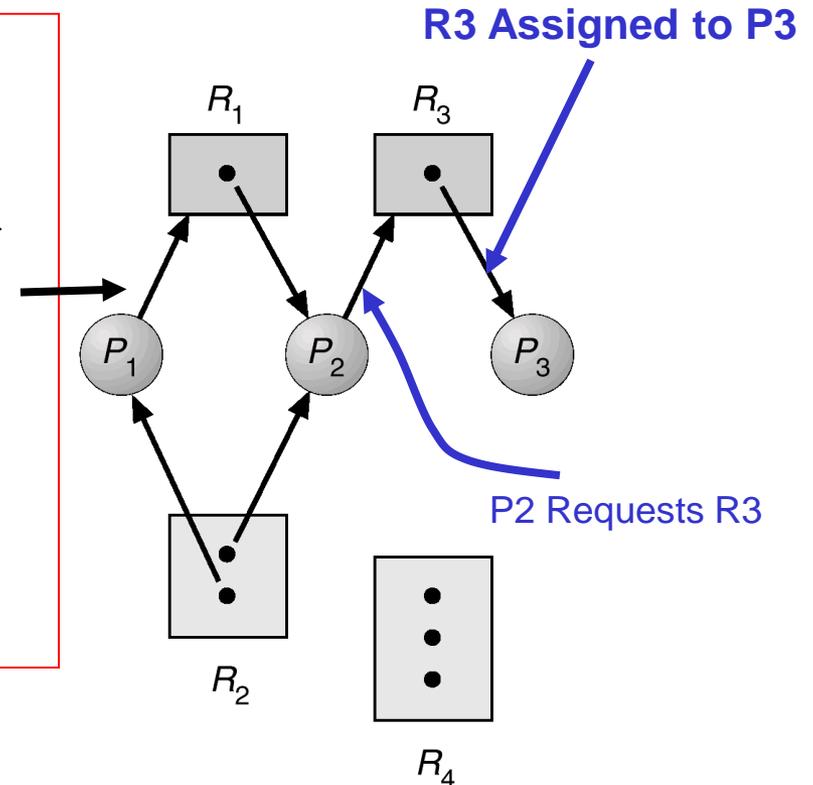
## Resource Allocation Graph: Example

### Resource allocation graph:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

### Resource instances:

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

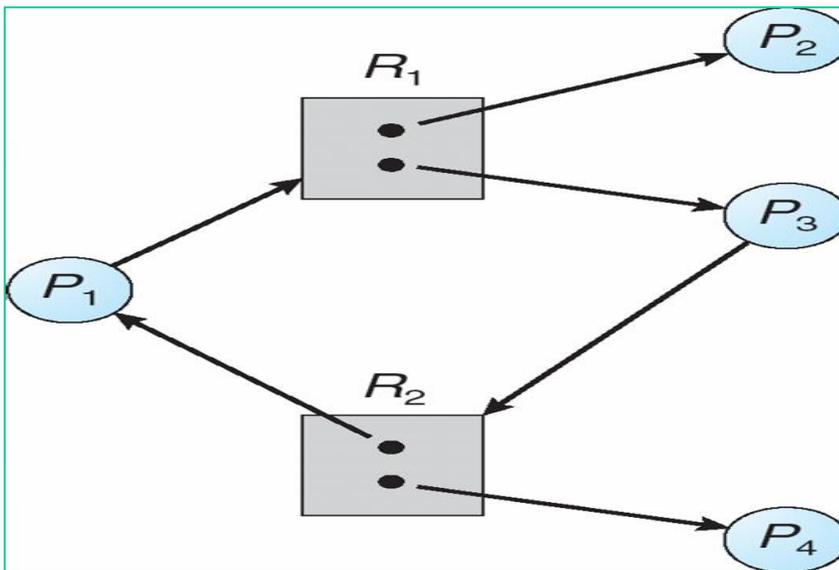


### The above RAG can be interpreted as followings:

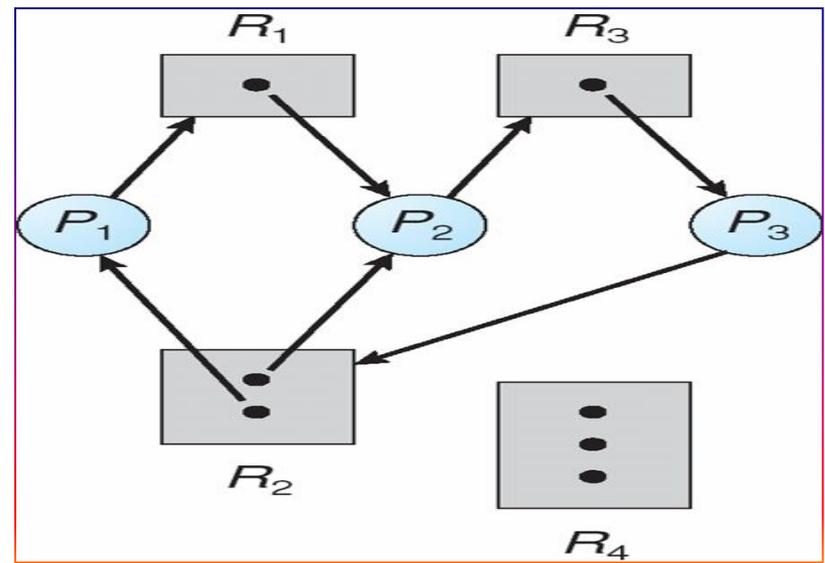
- $P_1$  is holding an instance of R2 and is waiting for an instance of R1.
- $P_2$  is holding an instance of R1, an instance of R2 and is waiting for an instance of R3.
- $P_3$  is holding an instance of R3.

## Basic Facts of a RAG

1. If the RAG contains no cycles, then no process is deadlocked.
2. If the RAG contains a cycle, then:
  - If the resource types have multiple instances, then deadlock **MAY** exist.
  - If each resource type has one instance or the number of assignment and requesting is more than the number of instances, then deadlock has occurred.



RAG Example 1

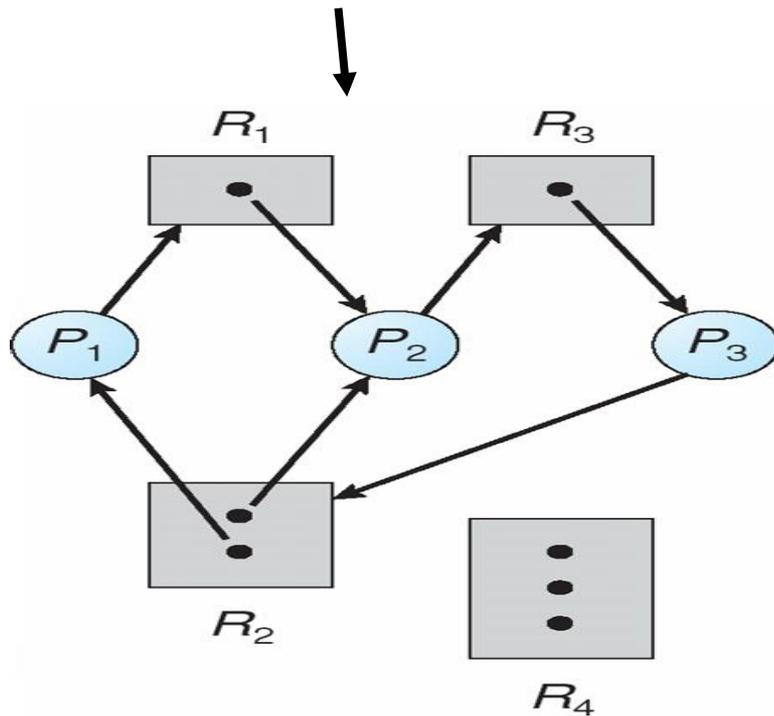


RAG Example 2

# Resource Allocation Graph Examples

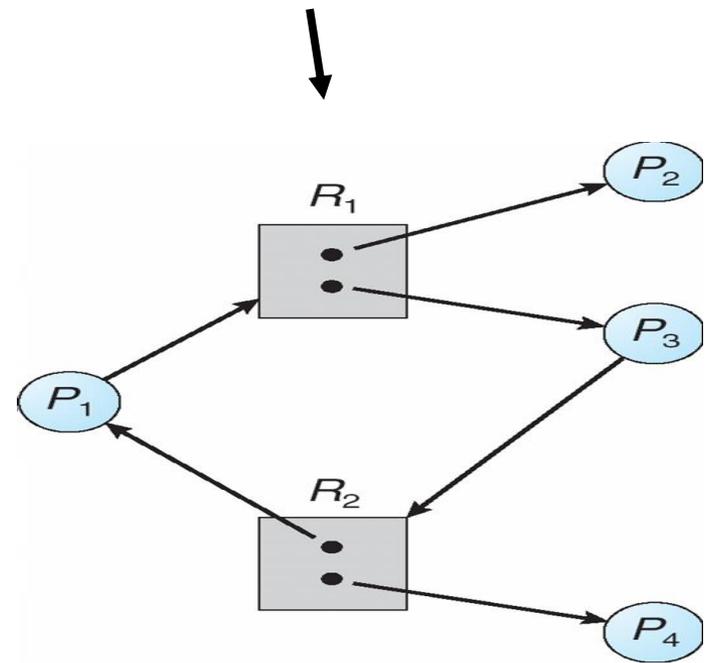
RAG with a deadlock: **Two cycles:**

- Cycle 1:  $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Cycle 2:  $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



RAG with a cycle but **no deadlock:**

- Cycle:  $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- P4 may release its instance of R2 that can be allocated to P3 breaking the cycle.



## How to Handle Deadlock

There are three strategies:

1. Ignores the deadlock handling. If it occurs, resolved by explicit user intervention. It is used by some operating systems, i.e. traditional UNIX.
2. Ensures deadlock never occurs using either:
  - **Prevention strategies:** Preventing the main causes/conditions of deadlock so that deadlock is impossible. If we can not prevent the causes then try,
  - **Avoidance strategies:** Avoiding at least one of the four conditions. Do not allow the system to get into a deadlocked state by disallowing dangerous allocations that may cause deadlock to happen.
3. If we can not prevent its causes or avoid its occurrence. This requires using:
  - **Detection strategies:** To know that deadlock has occurred, and which processes are in deadlock.
  - **Recovery strategies:** Abort a process or preempt some resources (using some polices) when deadlock is detected, to recover it.

Do not allow one of the four conditions to occur.

## 1. Preventing Mutual exclusion:

- **Example:** Allow the printer to be concurrently shared?
  - What are the consequences?
- **Conclusion:**
  - Prevention not possible, since some resources are naturally non-sharable.
  - Automatically holds for printers, CD-RW and other non-sharable devices.
- **Solutions:**
  - Make resources concurrently sharable wherever possible e.g. read-only file access (don't need mutual exclusion and aren't easily to deadlock).
  - Better to have a spooler process to which print jobs are sent (completed output file must be printed first).

### 2. Preventing Hold and wait:

- ❑ Process should request all the resources it will ever need at once.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Must guarantee that whenever a process requests a resource, it does not hold any other resources.
- ❑ **Conclusion:**
  - Inefficient - not all resources needed all the time
  - Processes probably will not know in advance what resources they will need or may have to wait excessive time to get all resources at once.
  - Utilization is low: many resources may be allocated but unused for long time.
  - Starvation possible: the process requests popular resource may wait indefinitely as it might be allocated to another process.

## 3. Preventing No Preemption (means support preemption):

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, **then all resources currently being held are released.**
- **Need to save state of resource if a process is forced to release it**

### Not practical for some resources, i.e.:

- Cannot take a printer away from a process in the middle of printing
- Cannot take a semaphore away from a process randomly (**might be in the middle of updating a shared area**)
- Cannot take open streams, pipes and sockets away
- process would need to be written very carefully, probably using signals
- very undesirable if possible at all

### Occasionally possible:

- Processes resident in main memory (**one or more processes can be swapped out to VM to release their pages and allow remaining processes to continue**)

## 4. Preventing Circular Wait:

- ❑ Impose order of all resource types and require that each process requests resources in an increasing order of enumeration.
- ❑ Supply information about how resources are to be used at process startup. i.e.
  - Use CD and disk, then release both
  - Use and release CD, then use disk
  - Prioritize processes and assign resources in the order of priorities, i.e. hard drive has higher order than printer.
- ❑ Impractical due to:
  - It will depend on programmer to follow the order (one program may not follow the order and causes deadlock system)
  - Adding a new resource that upsets ordering requires all code ever written for system to be modified!
  - Resource numbering affects the efficiency of utilization.

# Deadlocks

- **We have presented last class:**
- Computer's System Resources: Consumable resources, reusable resources, preemptable, non-preemptable. Here we will treat everything (interrupts, signals, messages, information in I/O buffers, I/O devices) as resources.
- Deadlock, Live lock/Busy waiting, Starvation Definitions,
- Types of Deadlock: Communication deadlocks, Resources deadlocks, what does it mean?
- Deadlock Examples: due to limited resources, due to lost of communicating messages, due to sharing of resources, due to mutual exclusion.
- Deadlock causes: Mutual exclusion, no-preemption, hold and wait, circular wait.
- Computer system modeling,
- Constructing the RAG, facts in a RAG.
- Methods for Handling Deadlocks
  - Deadlock Prevention: can we prevent the deadlock? If yes how and if no why?
- **We are going to present:**
  - Deadlock Avoidance: can we avoid the deadlock? If yes, How? if no why?
  - Deadlock Detection: can we detect the deadlock? If yes, How?
  - Recovery from Deadlock
- Integrated Approaches to Deadlock Handling.

## Deadlock Avoidance

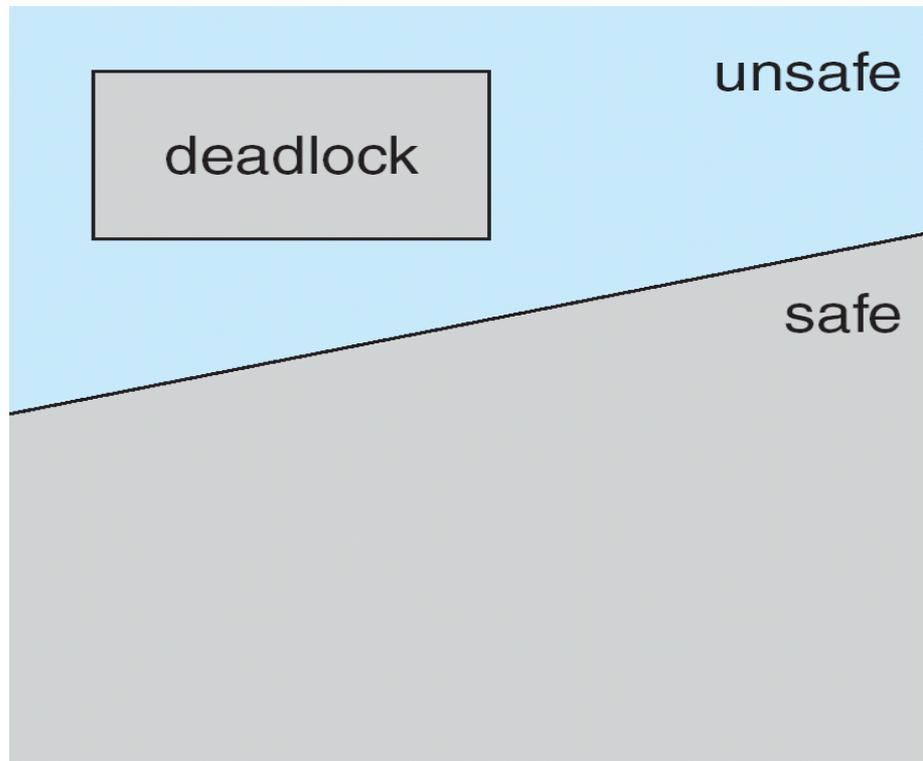
- **Since we can't prevent the causes of deadlock,**
- **Is it possible to avoid its occurrence?** Through careful allocation of resources to processes to **avoid entering the system into unsafe state.**
- If we have prior knowledge of how resources will be requested, then will be possible to determine **what will be the state of the system?** if the OS allows each process to get the requested resources,
- **Possible states are:**
  1. **Deadlock:** Processes will be waiting each other without progress.
  2. **Unsafe state:** If some requests are allocated deadlock **may** occur.
  3. **Safe state:** there are enough resources such that all processes will be able to finish without waiting for each other.
- **As we will conclude “avoidance may be inefficient”:**
  - Must know resources requirements of all processes in advance.
  - Resource request per each process has to be known and fixed.
  - Complex analysis for every request.

## Safe, Unsafe , Deadlock State

- If a system is in **safe state**  $\Rightarrow$  then the deadlock never occurs.
- If a system is in **unsafe state**  $\Rightarrow$  then there is possibility of a deadlock to occur.
- **Avoidance**  $\Rightarrow$  means ensure that a system will never enter into an **unsafe state**.

**Only with luck will processes avoid deadlock.**

**OS avoids deadlock**



## Deadlock Avoidance

- Simplest and most useful model, it requires that each process declares in advance the maximum number of each resource type it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition will never be there.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum needs of the processes.
- System makes a decision based on
  - The resources currently available
  - The resources currently allocated
  - Future requests for resources

## Safe State

- When a process requests an available resource, the OS must decide if the immediate allocation leaves the system in a *safe state* or *no*.
- System is in *safe state* if there exist a *safe sequence* of all processes.
- Sequence  $\langle P_1, P_2, \dots, P_n \rangle$  is safe if for each  $P_i$ , the resources that  $P_i$  requests can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$ .
  - 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.

## Example

- System with 12 memory blocks available and three processes:
  - P0: currently has 5, maximum needs of 10, future needs 5.
  - P1: currently has 2, maximum needs of 4, future needs 2.
  - P2: currently has 2, maximum needs of 9, future needs 7.
  - State is ??????.
  - However, if P2 were allocated 1 additional block, the state would become ??????.
  - However, if P2 were to request 1 additional block, the state would become ??????.

## Example

- System with 12 disks and three processes:
  - P0: currently has 5, maximum need of 10, **needs 5**.
  - P1: currently has 2, maximum need of 4, **needs 2**.
  - P2: currently has 2, maximum need of 9, **needs 7**.
  - State is **currently safe**
    - 3 disks are available, 2 can be given to P1, when it terminates there will be 5 drives available, P0 can use them and terminates, this makes 10 drives available, P2 can get 7 of them and finish. **Safe sequence is (P1, P0, P2)**.
  - However, if P2 were allocated **1** additional disk, **the state would become unsafe**.
    - Because all of the 3 processes could not obtain their needed disks to complete.
  - However, if P2 were to request **1** additional disk, **the state would become safe**.

## Resource-Allocation Graph Algorithm

- Add to the RAG an extra link called a **claim edge**.
- Claim edge  $P_i \text{ ---}\rightarrow R_j$  indicated that process  $P_i$  may request resource  $R_j$  in future.
- Claim edges represented by **dashed line** arrows  $\text{---}\rightarrow$ .
- Claim edge converts to a request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.
- If no cycle comes into existence, then the state is still safe
- A cycle means an unsafe state (**although not necessarily deadlock**)

## RAG For Deadlock Avoidance

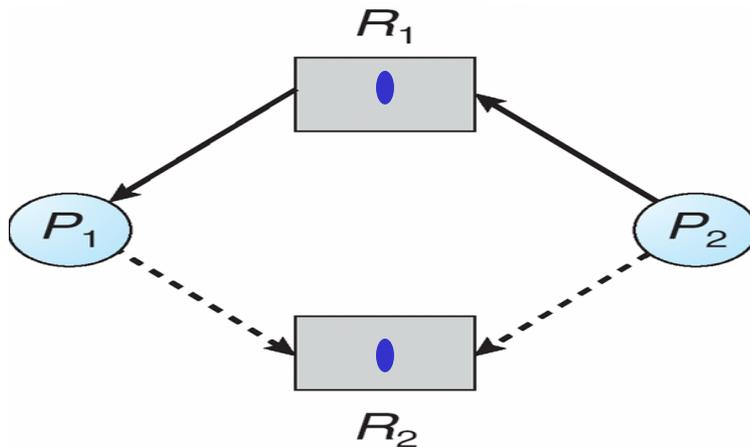
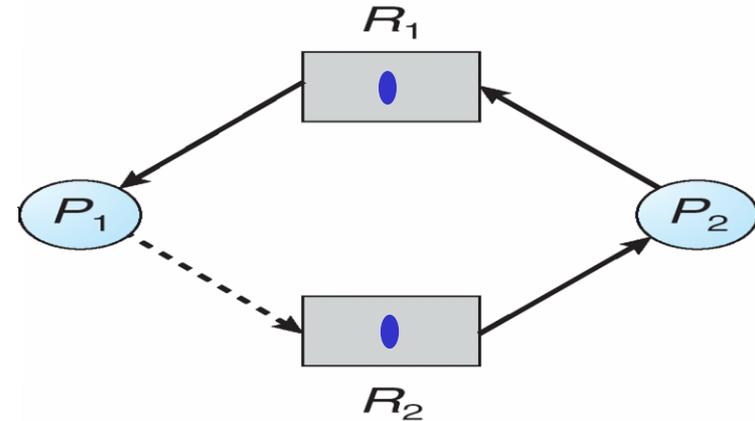


Fig. 1



Unsafe State in Resource-Allocation Graph

Fig. 2

1. Suppose that  $P_2$  requests  $R_2$ , Fig. 1. Although  $R_2$  is free, we can not allocate it to  $P_2$  since this will create a cycle and indicates of unsafe state, Fig. 2.
2. If  $P_1$  requests  $R_2$  and  $P_2$  requests  $R_1$ , then a deadlock will occur.

## RAG For Deadlock Avoidance: Multiple Instances

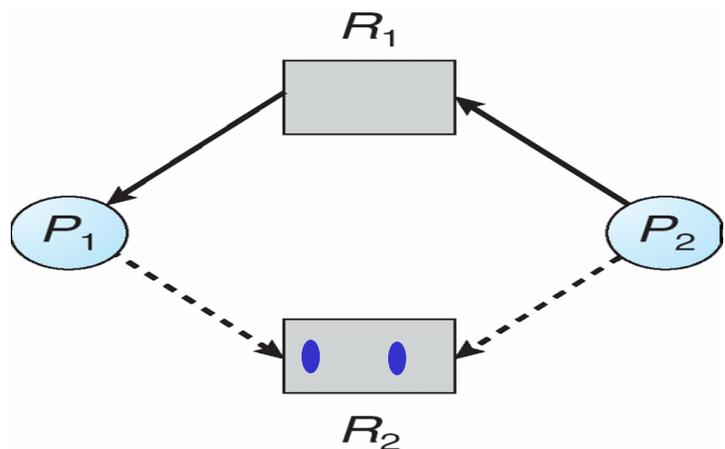
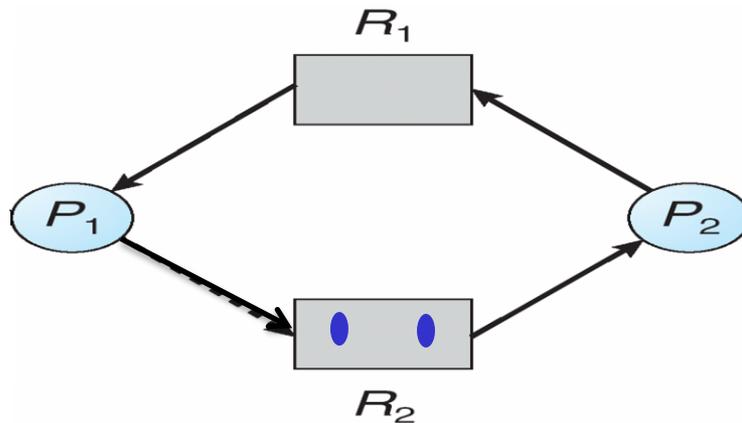


Fig. 1



Safe State In Resource-Allocation Graph

Fig. 2

1. Suppose that  $P_2$  requests  $R_2$ , Fig. 1. because  $R_2$  has two instances, we can allocate them to  $P_1$  and  $P_2$  even there will be a cycle and indicates of safe state, Fig. 2.
2. That means the RAG is not applicable in case of having recourses with many instances.
3. There should be another algorithm fro that situation.

## Banker (Dijkstra's) Algorithm

- The Resource Allocation Graph algorithm is **not applicable** to avoid the deadlock in systems **with multiple instances of each resource type**.
- **The banker algorithm is applicable to multiple resource instances.**
- This algorithm could be used in the banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.
- Each process must declare the maximum number of each resource type it may need in future.
- **This number should not exceed the total number of available resources in the system.**
- **When a process requests a resource, the system must check whether the allocation of these resources will leave the system in a safe state.**
- **If it will leave the system in a safe state:**
  - **Then, resources will be allocated,**
  - **Otherwise, the process must wait until other processes release enough resources.**
- **When a process gets all its resources it must return them in a finite amount of time.**

## Deadlock avoidance

- Examine each resource request and determine whether or not granting the request can lead to deadlock.
- Define a set of vectors and matrices that characterize the current state of all resources and processes.

➤ **Resource allocation state matrix**

$Alloc_{ij}$  = the number of units of resource  $j$  held by process  $i$

➤ **Maximum claim matrix**

$Max_{ij}$  = the maximum number of units of resource  $j$  that the process  $i$  will ever require simultaneously.

➤ **Available vector**

$Avai_j$  = the number of units of resource  $j$  that are available.

	$R_1$	$R_2$	$R_3$	...	$R_r$
$P_1$	$n_{1,1}$	$n_{1,2}$	$n_{1,3}$	...	$n_{1,r}$
$P_2$	$n_{2,1}$	$n_{2,2}$			
$P_3$	$n_{3,1}$		$\ddots$		$\vdots$
$\vdots$	$\vdots$				
$P_p$	$n_{p,1}$			...	$n_{p,r}$

$\langle n_1, n_2, n_3, \dots, n_r \rangle$

## Data Structures for the Banker Algorithm

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

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

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

- For simplification, let  $X$  and  $Y$  be vectors of length  $n$ , we say that  $X \leq Y$  iff  $X[i] \leq Y[i]$  for all  $i=1,2,\dots, n$ .
- If  $X=[0,3,2,1]$  and  $Y=[1,7,3,2]$ , then  $X < Y$ .
- **Available:** Vector of length  $m$ .  $\text{Available}[j] = k$ , means there are  $k$  instances of resource type  $R_j$  available.

## Safety Algorithm

This algorithm for finding out whether or not the system is in safe state

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

Initialize:

**Work = Available**

**Finish [i] = false** if there is a process  $i$  ( $i=1, 2, \dots, n$ ) not yet finish.

2. Find a process  $i$  such that both:

(a) **Finish [i] = false** (**not yet finish**)

(b) **Need<sub>i</sub> ≤ Work** (**needs less recourses than the available**)

If no such  $i$  exists, go to step 4. (**all processes have finished**)

3. **Work := Work + Allocation<sub>i</sub>** (**update the work and available**)

**Finish[i] = true**

go to step 2.

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

The algorithm may require an order of  $m*n^2$  operation to decide.

## 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  $Request_i \leq Need_i$  go to step 2. Otherwise, **raise error condition, since process has exceeded its maximum claim.**
2. If  $Request_i \leq Available$ , then 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:

$Available := Available - Request_i;$

$Allocation_i := Allocation_i + Request_i;$

$Need_i := Need_i - Request_i;$

- 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 of Banker's Algorithm

- Consider a system of 5 processes and 3 resources:
- 5 processes  $P_0$  through  $P_4$ ;
- Resource types: A (10 instances), B (5 instances, and C (7 instances).
- Snapshot at time  $T_0$ :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>	
	A B C	A B C	A B C		A B C
				$P_0$	7 4 3
$P_0$	0 1 0	7 5 3	3 3 2	$P_1$	1 2 2
$P_1$	2 0 0	3 2 2		$P_2$	6 0 0
$P_2$	3 0 2	9 0 2		$P_3$	0 1 1
$P_3$	2 1 1	2 2 2		$P_4$	4 3 1
$P_4$	0 0 2	4 3 3			

The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria.

The content of the matrix need is defined to be Max – Allocation.

## Example P1 Allocated (1, 0, 2)

- Suppose that P1 allocated 1 additional instances of type A, 2 instances of type C. **To decide whether this request can be granted, we:**
- Check that  $Need \leq Available$  (that is,  $(0,2,0) \leq (2,3,0) \Rightarrow true$ . **This request can be allocated and we arrive at the new state:**

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>	
	A B C	A B C	A B C		A B C
$P_0$	0 1 0	7 5 3	2 3 0	$P_0$	7 4 3
$P_1$	<b>3 0 2</b>	3 2 2		$P_1$	<b>0 2 0</b>
$P_2$	3 0 2	9 0 2		$P_2$	<b>6 0 0</b>
$P_3$	2 1 1	2 2 2		$P_3$	<b>0 1 1</b>
$P_4$	0 0 2	4 3 3		$P_4$	<b>4 3 1</b>

- Executing safety algorithm shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement.
- Can request for (3,3,0) by  $P_4$  be granted?
- Can request for (0,2,0) by  $P_0$  be granted?

## Deadlock Avoidance is not Practical

- It is very difficult or even impossible for some kind of processes to declare the maximum # of resources it requires in advance.
  - Many processes are interactive and/or dynamic where the users or the system cannot anticipate their eventual requirements.
  - Moreover, its point in time is also unknown.
- Resources may disappear or newly plug in
  - Some devices leave the available pool while others may plug in.
- New processes may appear or old processes may be killed.
  - The system is dynamic and processes are born and die at any moment
- The algorithm requires that processes release the resources within a finite time, but this may cause lengthy delay to other processes waiting in line.

## Deadlock Detection

- If a system does not employ either a deadlock prevention or a deadlock avoidance, then a deadlock may occur, and this requires:
- A detection algorithm to examine the state of the system and determines whether a deadlock has occurred or not. If occurred, which processes are in the deadlock?
- A recovery policy to recover from the deadlock.

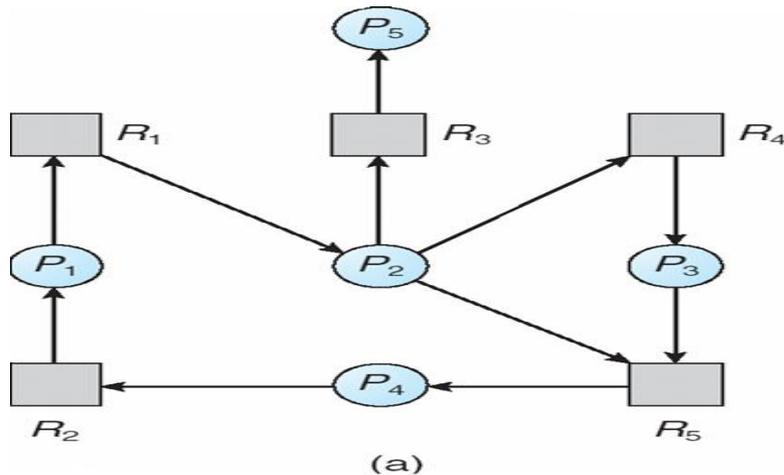
## Wait-For Graph: Deadlock Detection Algorithm

- If all resources have a single instance, then Wait-For Graph (WFG) or it may be named (Process Dependency Graph) will be used.
- 1- Create a WFG (Process Dependency Graph (PDG))
  - In a WFG/PDG, only nodes and edges are there where:
    - Nodes represent processes, and
    - An edge from  $P_i \rightarrow P_j$  means  $P_i$  is waiting for  $P_j$ .
- 2- Periodically check for a cycle in the WFG/PDG.
- 3- If there is a cycle, then processes in the cycle are deadlocked.
  - An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices (processes) in the graph.

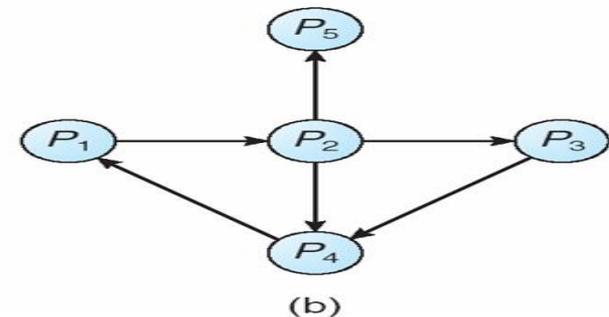
## Resource-Allocation Graph $\rightarrow$ Wait-for Graph

### How to create a WFG/PDG out of the RAG?

- Remove the nodes of type resources from the RAG.
- If the corresponding RAG contains a request edge  $P_i \rightarrow R_q$  and an assignment edge from  $R_q \rightarrow P_j$  for the same resource  $R_q$ .
  - Remove the request and assignment edges.
  - Create an edge from  $P_i \rightarrow P_j$  in the WFG
- An edge from  $P(i)$  to  $P(j)$  implies that  $P(i)$  is waiting for  $P(j)$  to release a resource that  $P(i)$  needs.



Resource-Allocation Graph

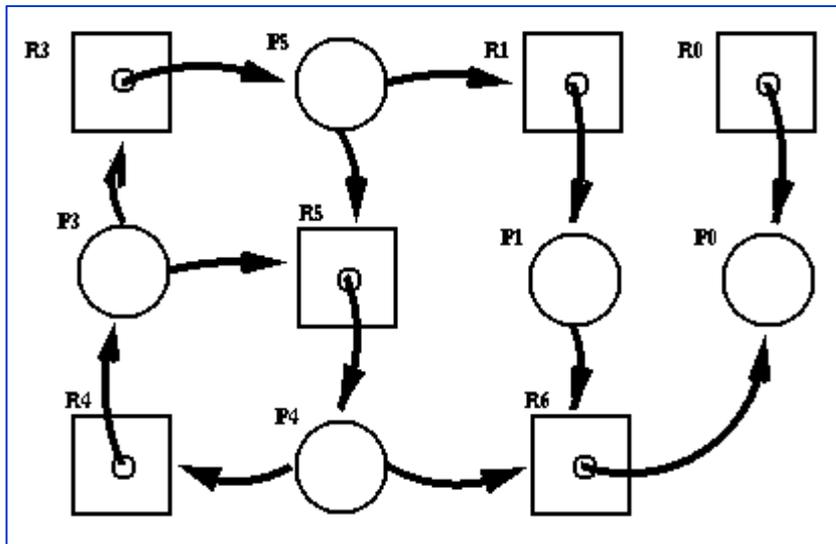


Corresponding WFG/PDG

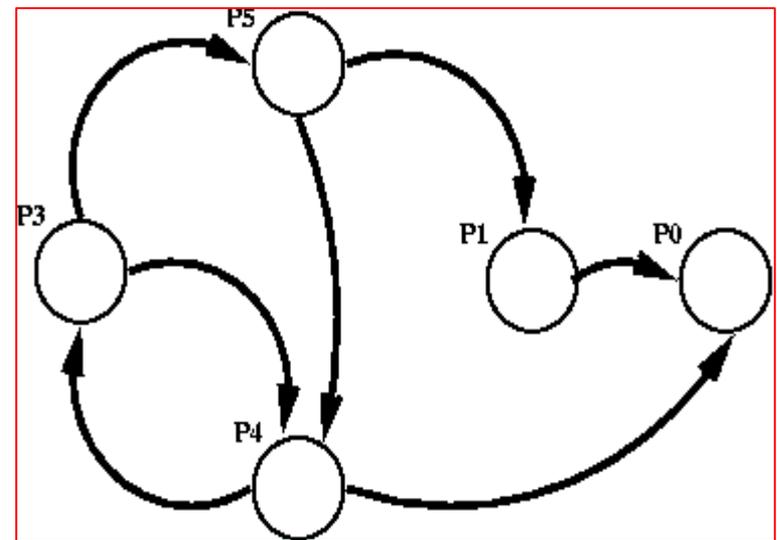
## Resource-Allocation Graph $\rightarrow$ Wait-for Graph

### How to create a WFG/PDG out of the RAG?

- Remove the nodes of type resources from the RAG.
- If the corresponding RAG contains a request edge  $P_i \rightarrow R_q$  and an assignment edge from  $R_q \rightarrow P_j$  for the same resource  $R_q$ .
  - Remove the request and assignment edges.
  - Create an edge from  $P_i \rightarrow P_j$  in the WFG
- An edge from  $P(i)$  to  $P(j)$  implies that  $P(i)$  is waiting for  $P(j)$  to release a resource that  $P(i)$  needs.



Resource-Allocation Graph



Corresponding WFG/PDG

## Exercises: try to solve

- **Suppose there is only one instance of each resource**
- **Example 1:** Is there a deadlock?
  1. P1 has R2 and R3, and is requesting R1
  2. P2 has R4 and is requesting R3
  3. P3 has R1 and is requesting R4
- **Example 2:** Is there a deadlock?
  1. P1 has R2, and is requesting R1 and R3
  2. P2 has R4 and is requesting R3
  3. P3 has R1 and is requesting R4
- **Use a wait-for graph/Process Dependency Graph:**

## With Multiple Instances of Resources

- The WFG algorithm is not applicable to systems with multiple instances of each resource type.
- Another algorithm is applicable to such kind of systems, where the following data structures will be created:
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An  $n \times m$  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  $R_j$ .
- **Available:** A vector of length  $m$  indicates the number of available instances of each resources type.

**Resource** =  $(R_1, R_2, \dots, R_m)$

**Allocation** =  $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$

**Request** =  $\begin{pmatrix} M_{11} & M_{12} & \dots & M_{1m} \\ M_{21} & M_{22} & \dots & M_{2m} \\ \dots & \dots & \dots & \dots \\ M_{n1} & M_{n2} & \dots & M_{nm} \end{pmatrix}$

**Available** = Total - Allocation

**Available** =  $\mathbf{V} = (V_1, V_2, \dots, V_m)$

## Detection Algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively  
Initialize:  
**Work** := **Available**. For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_i \neq 0$ , then  $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$ .
2. Find a process  $i$  such that both:
  - (a)  $\text{Finish}[i] = \text{false}$
  - (b)  $\text{Request}_i \leq \text{Available}$If no such  $i$  exists, go to step 4.
3. **Work** = **Work** +  $\text{Allocation}_i$   
 $\text{Finish}[i] = \text{true}$ , go to step 2.
4. If  $\text{Finish}[i] = \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] = \text{false}$ , then  $P_i$  is deadlocked.

Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state.

## Example of Detection Algorithm

- 5 processes,  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time  $T_0$ , we have this resource allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $\text{Finish}[i] = \text{true}$  for all  $i$ .

## Example of Detection Algorithm

- Suppose now  $P_2$  requests an additional instance of type C.

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	ABC	A B C	000
$P_0$	010	0 0 0	
$P_1$	200	2 0 1	
$P_2$	303	0 0 1	
$P_3$	211	1 0 0	
$P_4$	002	0 0 2	

- State of system?
  - We claim that the system is now deadlocked.
  - 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, and how often, to invoke the detection algorithm depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by the deadlock when it happens?
- If detection algorithm is invoked randomly, there may be many cycles in the WFG and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
- In the extreme, we could invoke the detection algorithm every time a request for allocation cannot be granted. But this makes an overhead computing.
- The detection algorithm can be invoked whenever the CPU utilization drops below 40 % or once per hour.

## Recovery from Deadlock: **By Process Termination**

- If deadlock has been detected by the OS, the OS should be able to recover it.
- Terminate all deadlocked processes. It is too expensive as these processes may have computed for a long time and later will be repeated, if the process was in the middle of updating or printing a file, terminating it makes an errors.
- Abort one process at a time until the deadlock cycle is eliminated. **This means the detection algorithm should be invoked many times to check and this makes overhead computing.**
- **Partial termination means there should be a mechanism to select the process to be terminated, like the CPU scheduling.**
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - Is the process interactive or batch?
  - How many processes will need to be terminated.
- **Starvation:** Same process may always be picked as victim, we should include number of rollback as a cost factor.

## Recovery from Deadlock: **By Resource Preemption**

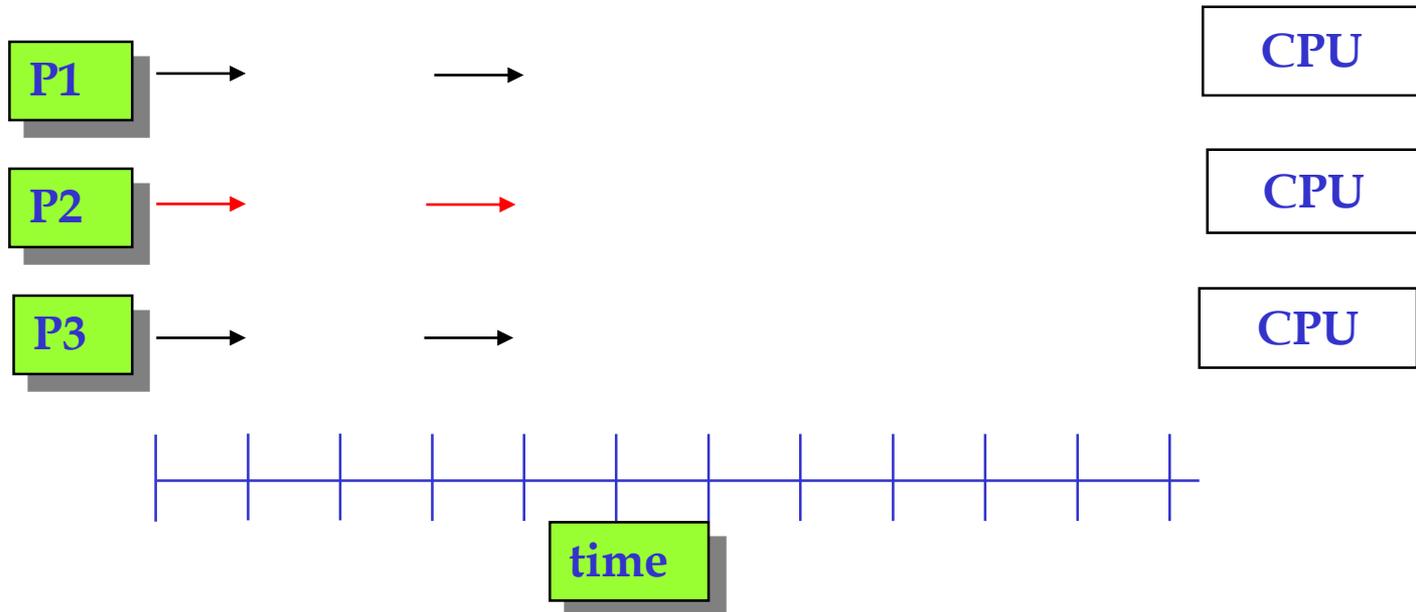
- Preempt some resources from the processes and give them to other processes until the deadlock cycle is broken. **Three issues need to be addressed.**
  - **Selecting a victim:** we must determine the order of preemption to minimize the cost. **i.e. priority, age of the process with the resource.**
  - **Rollback:** We must return the process to a safe state, and restart the process from that state. **Need to save the state of the process with the preempted resource.**
  - **Starvation:** Same process's resources may always be picked as victim, **we should include number of rollback as a cost factor.**

## An Integrated Deadlock Handling Strategy

- Each deadlock strategy has its strengths and weaknesses.
- Using a single strategy may be inefficient.
- Modern OSs combine the three basic approaches (Prevention, Avoidance, Detection)
- Allowing the use of the optimal approach for each of resources in the system.
  - For I/O devices, and files as resources use Prevention through resource ordering, no selection among pending processes.
  - For main memory use Prevention through Preemption, a process can always be swapped-out.
  - For assignable devices: if device-requirement information is available use Avoidance.
  - For swap space use Avoidance, since maximal storage requirements are known in advance.
- Use most appropriate technique for handling deadlocks within each class.
  - For database records that need locking first and then updating
  - Deadlocks occur frequently because records are dynamically requested by competing processes.
  - DBMSs, therefore, need to employ deadlock detection and recovery procedures.

# Concurrency Vs. Parallelism

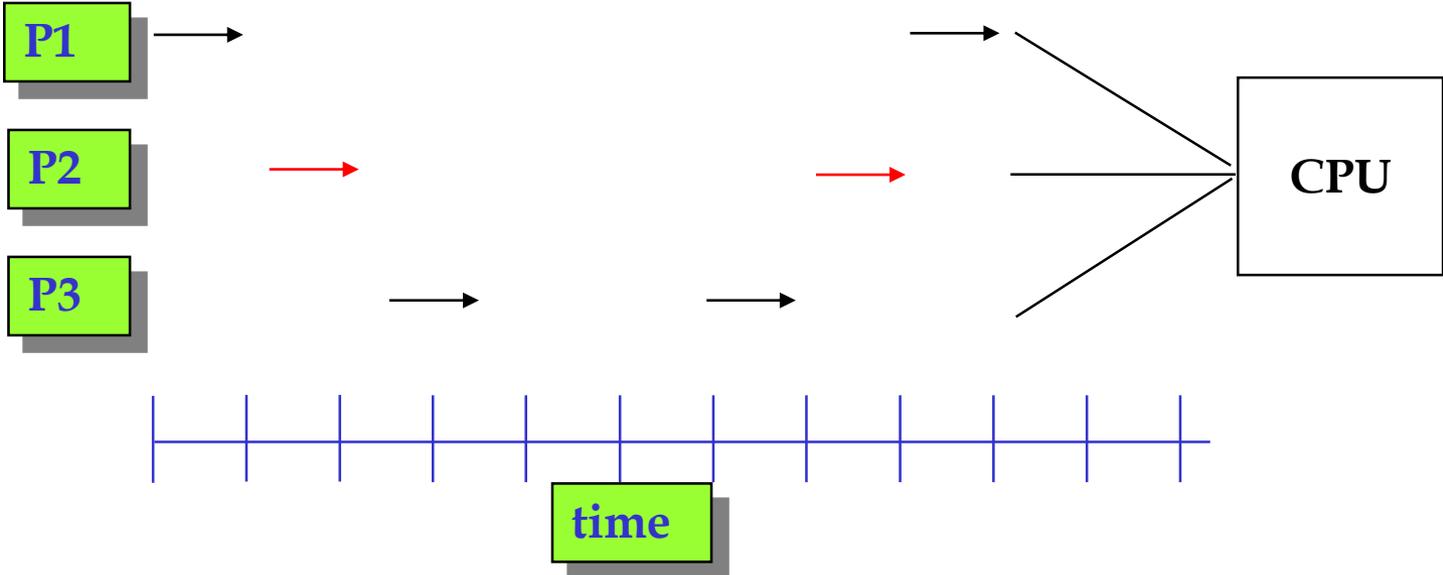
## Parallel Processing



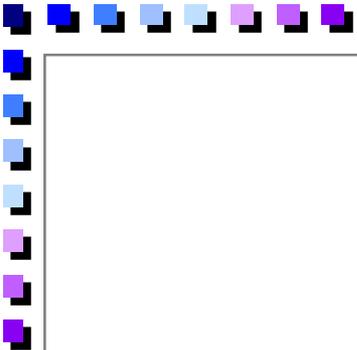
No of executing processes  $\leq$  the number of CPUs

# Concurrency Vs. Parallelism

## Concurrent Processing



Number of simultaneously executing processes > number of CPUs



**The End!!**

**Thank you**

**Any Questions?**

