

Mohamed Boudiaf
University
-M'sila-



Faculty of Mathematics
and Computer Science



Computer Science
Department

OPERATING SYSTEMS II

CHAPTER IV

DEADLOCK

3rd Year, Computer Science - SI

Dr. Hanene CHERAIT

2025-2026

INTRODUCTION

- ◉ In a **multiprogrammed** environment, multiple processes may compete for a finite number of resources.
- ◉ A process requests resources; if they are not available at that moment, the process enters a **waiting state**.
- ◉ It can happen that waiting processes **never change state**, because the resources they request are held by other waiting processes
 - → this situation is called a **deadlock**.
- ◉ The methods used by an operating system to handle deadlock are:
 - **Prevention**
 - **Avoidance**
 - **Detection/Recovery**
 - **Combined approach**

SYSTEM MODEL

- ⊙ A system consists of a finite set of resources that must be allocated among a number of concurrent processes.
- ⊙ Resources are grouped into several types:
 - **Physical**: printer, CPU cycles, memory space, ...
 - **Logical**: files, semaphores, monitors, ...
- ⊙ Resources can have multiple identical instances.
 - **Example**: An operating system has three processors and three memory units.

SYSTEM MODEL

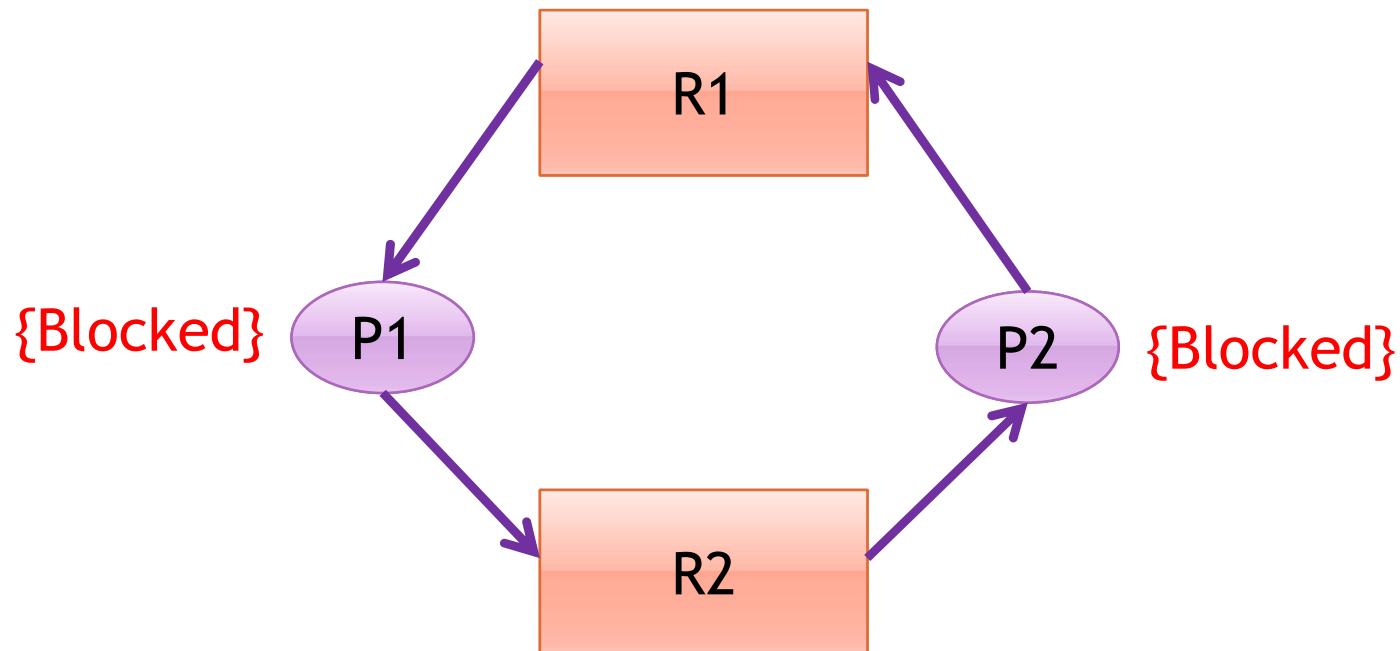
- A process can use a resource only according to the following sequence of events:
 - **Request**: If the request cannot be satisfied immediately, the requesting process must wait until it can acquire the resource.
 - **Use**: The process has acquired the resource and is using it.
 - **Release**: The process releases the resource.
- The Request and Release operations are system calls:
 - **Device** (e.g., printer): Request / Release device
 - **File**: Open / Close file
 - **Memory**: Allocate / Free memory

SYSTEM MODEL

For each use, the operating system checks that the process has requested and been allocated the resource. A system table records the state of resources (free/occupied) and which process they are allocated to.

DEADLOCK SITUATION

A set of processes is in a **deadlock situation** when each process in the set is waiting for an event that can only be caused by another process. The events are the **acquisition or release of resources**.



CONDITIONS FOR DEADLOCK

- ◉ A deadlock situation can occur if the following four conditions hold simultaneously in a system:
 - **Mutual Exclusion:** Only one process can use a resource at a time. Other requesting processes must wait.
 - **Hold and Wait:** A process may be holding at least one resource while waiting to acquire additional resources held by other processes.
 - **No Preemption:** Resources that are held cannot be forcibly taken away. Only the process holding the resource can release it.
 - **Circular Wait:**
 - There exists a chain containing at least two processes and two resources.
 - Each process holds a resource that the next process in the chain needs to continue execution.

GRAPHICAL DESCRIPTION OF DEADLOCKS

- ◉ A directed graph called a Resource Allocation Graph is used to model deadlock conditions.

- ◉ There are two types of nodes:

- processes



- resources



- ◉ An edge from a process to a resource means that the process is blocked waiting for that resource.



- ◉ An edge from a resource to a process means that the resource is allocated to that process.

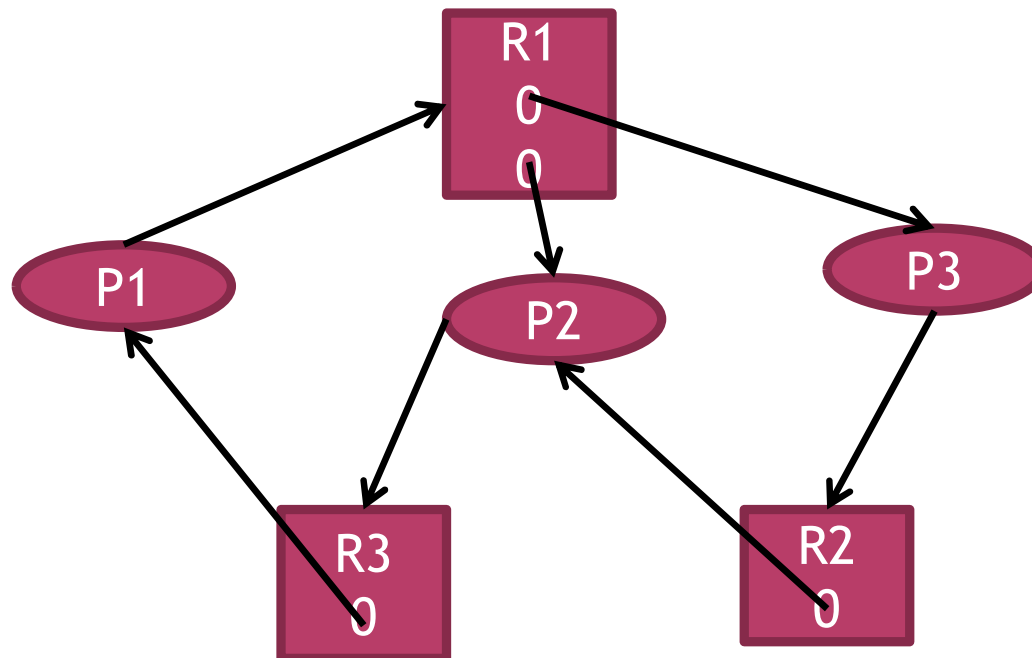


A resource allocation graph that **does not contain a cycle** implies that the system **is not in a deadlock**. If a **cycle exists**, the system **may or may not be in a deadlock**.

GRAPHICAL DESCRIPTION OF DEADLOCKS

◉ Example of Deadlock:

- Consider three processes, **P1**, **P2**, and **P3**, that use three resources, **R1**, **R2**, and **R3**, where **R1** has two instances, and **R2** and **R3** each have one instance.



There are two deadlock situations (cycles) in the system:

1. **P1** → **R1** → **P2** → **R3** → **P1**
2. **P1** → **R1** → **P3** → **R2** → **P2** → **R3** → **P1**

DEADLOCK HANDLING METHODS

1

Prevention

2

Avoidance

3

Detection/Recovery

4

Combined Approach

DEADLOCK HANDLING METHODS

1

Prevention

2

Avoidance

3

Detection/Recovery

4

Combined Approach

PREVENTION

- It is possible to avoid deadlock by implementing a policy that makes **one of the four necessary conditions unsatisfied**.
- However, such an implementation can **introduce more problems** than it solves, as shown in the following analysis:
 - **Mutual Exclusion**: Removing exclusive access to any resource does not appear to be a practical solution.
 - Some resources cannot be shared.
 - **Hold and Wait**: To prevent this condition, the system must guarantee that a process can **never request a resource while holding others at the same time**.

PREVENTION

- **No Preemption:** The following protocol can be applied
 - if resources cannot be immediately allocated to a requesting process even though it holds some resources, all currently allocated resources are forcibly reclaimed (**forced preemption**).
- **Circular Wait:** This represents the most promising prevention technique.
 - To break the cycle, one method is to assign a **unique priority number to each resource**.
 - Processes can request resources **only if the priority is higher than all resources they currently hold**.
 - If a resource does not have a higher priority than all currently held resources, the resource with the **highest priority among those held** must be released first.

DEADLOCK HANDLING METHODS

1

Prevention

2

Avoidance

3

Detection/Recovery

4

Combined Approach

AVOIDANCE

- ◉ Using additional information about how resources will be requested:
 - With knowledge of the complete sequence of requests for each process, we can decide for each request whether the process **should wait or not**.
 - The system must take into account
 - the **currently available resources**
 - the resources **allocated to each process**
 - the **future requests and releases** of each process.

AVOIDANCE

◎ The algorithms require:

- Each process declares the **maximum number of resources** of each type it may request.
 - Using this information, it is possible to construct an algorithm that ensures a system **will never enter a deadlock state**.
- The algorithms dynamically examine the **current state of resource allocation** to ensure that a **circular wait condition can never occur**.
 - The allocation state is defined according to the number of **available and allocated resources**, as well as the **maximum requests of each process**.

SAFE STATE

- ⦿ A system is said to be in a **safe state** if there exists a **safe execution sequence**.
- ⦿ The state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid deadlock.
 - $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence of processes for the current allocation state if, for each P_i :
 - ⦿ The resource requests of P_i can be satisfied by the currently available resources,
 - ⦿ **Plus** the resources held by all P_j with $j < i$.
- ⦿ **A safe state is not a deadlock, and vice versa.**

SAFE STATE / EXAMPLE

Let's consider a system with **12 tape drives** and **3 processes: P0, P1, and P2**.

Process	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

At time t_0 , the system is in a safe state. The sequence **<P1, P0, P2>** is a safe sequence.

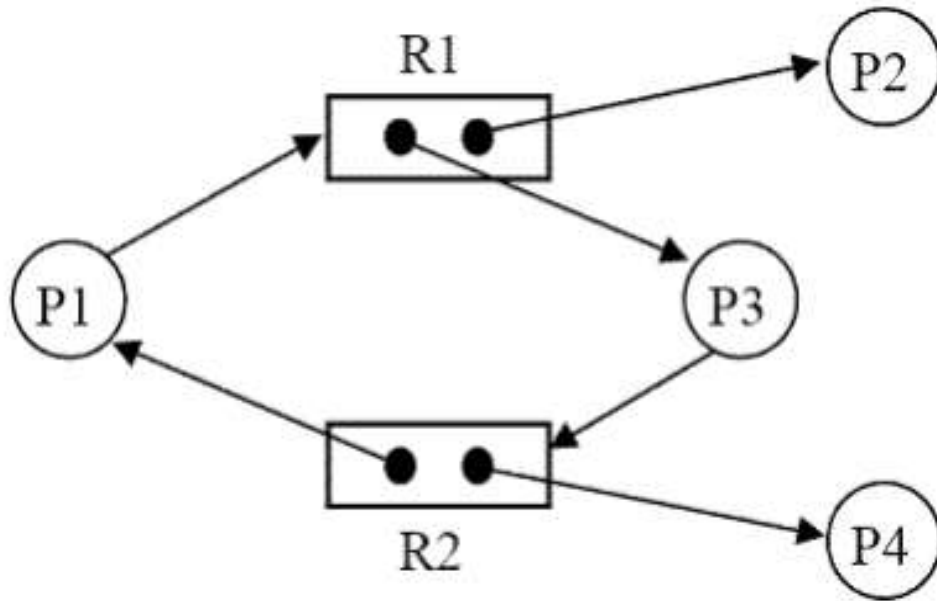
Principle: Initially, the system is in a **safe state**. Each time a process requests a currently available resource, the system must decide whether the resource can be **allocated immediately** or whether the process should wait. The request is granted **only if the allocation leaves the system in a safe state**.

BANKER'S ALGORITHM

- ◉ It mainly applies to a resource allocation system with multiple instances of the same type of resource.
- ◉ If **n** is the number of processes and **m** is the number of types of resources.

Variable	Description
Available[m]	A vector that defines the number of available resources for each type. Available[i] = K $\Rightarrow$ there exist K instances of resource R_i .
Max[n] [m]	A matrix that indicates the maximum requests for each process. Max[i, j] = the maximum number of instances of resource R_j required for the execution of process P_i .
Allocation[n][m]	A matrix that describes the number of resources of each type currently allocated to each process. Allocation[i, j] = the number of instances of resource R_j currently allocated to process P_i .
Need[n][m]	A matrix that defines the remaining resources requested by each process. Need[i, j] = Max[i, j] - Allocation[i, j]

BANKER'S ALGORITHM



Need

	R1	R2
P1	1	0
P2	0	0
P3	0	1
P4	0	0

Available

R1	R2
0	0

Allocation

	R1	R2
P1	0	1
P2	1	0
P3	1	0
P4	0	1

BANKER'S ALGORITHM

- The Banker's Algorithm involves two related components or "algorithms" in practice:
 - **Safety Algorithm**
 - Checks whether the system is in a safe state.
 - Determines if there exists a safe sequence of processes such that all processes can complete without causing a deadlock.
 - Uses the **Available**, **Allocation**, **Max**, and **Need** matrices to simulate resource allocation and verify safety.
 - **Resource-Request Algorithm**
 - Handles resource requests from processes.
 - When a process requests resources, the algorithm pretends to allocate them and then calls the **Safety Algorithm** to check if the system remains safe.
 - If the state is safe, the request is granted; otherwise, the process must wait.

SAFETY ALGORITHM

1

- Let **Work[m]** and **Finish[n]** be defined:
 - **Work** = Available
 - **Finish[n]** initialized to **false**

2

- Find an **i** such that:
 - **Finish[i] == false**
 - **Need[i] <= Work**
 - If no such **i** exists, go to **step 4**.

3

- **Work = Work + Allocation[i]**
- **Finish[i] = true**
- Go back to **step 2**.

4

- If **Finish[i] == true** for all **i**, then the system is in a **safe state**.
- Else, it is not in a **safe state**

SAFETY ALGORITHM - SCANNING METHODS

○ Restart-from-Beginning

- After a process executes, scan again from P0.
- Simple, guaranteed to check all processes.
- May recheck processes multiple times.

○ Next-Process (Continuous)

- After a process executes, scan from the next process.
- Wraps around at the end.
- **More efficient**, safe sequence depends on start.

○ Both ensure correctness; difference is scanning order.

SAFETY ALGORITHM / EXAMPLE

- Processes: P0, P1, P2
- Resources: A, B, C
- Available: [3, 2, 2]

Process	Max (A,B,C)	Allocation (A,B,C)	Need (A,B,C)
P0	3, 3, 2	1, 0, 1	2, 3, 1
P1	2, 3, 1	1, 1, 0	1, 2, 1
P2	1, 2, 1	0, 1, 0	1, 1, 1

- Initialization:
 - Work = Available = [3, 2, 2]
 - Finish = [false, false, false]

SAFETY ALGORITHM / EXAMPLE

Restart-from-Beginning

Work : [3, 2, 2]

Step	Process Selected	Test	Allocation	Work After	Finish	Next Start
0	-	-		[3,2,2]	[F,F,F]	P0
1	P0	$[3,2,2] \geq [2,3,1]$	[1,0,1]	[3,2,2]	[F,F,F]	P1
2	P1	$[3,2,2] \geq [1,2,1]$	[1,1,0]	[4,3,2]	[F,T,F]	P0
3	P0	$[4,3,2] \geq [2,3,1]$	[1,0,1]	[5,3,3]	[T,T,F]	P2
4	P2	$[5,3,3] \geq [1,1,1]$	[0,1,0]	[5,4,3]	[T,T,T]	-

Safe sequence = <P1, P0, P2>

SAFETY ALGORITHM / EXAMPLE

Next-Process (Continuous)

Work : [3, 2, 2]

Step	Process Selected	Test	Allocation	Work After	Finish	Next Start
0	-	-		[3,2,2]	[F,F,F]	P0
1	P0	$[3,2,2] \geq [2,3,1]$	[1,0,1]	[3,2,2]	[F,F,F]	P1
2	P1	$[3,2,2] \geq [1,2,1]$	[1,1,0]	[4,3,2]	[F,T,F]	P2
3	P2	$[4,3,2] \geq [1,1,1]$	[0,1,0]	[4,4,2]	[F,T,T]	P0
4	P0	$[4,4,2] \geq [2,3,1]$	[1,0,1]	[5,4,3]	[T,T,T]	-

Safe sequence = <P1, P2, P0>

RESOURCE-REQUEST ALGORITHM

- ◉ Let $\text{Request}[m]$ be the request vector for process P_i .
- ◉ If $\text{Request}[j] = k$, then process P_i wants k instances of the resource type R_j .
- ◉ If P_i requests resources, then:

1

- If $\text{Request}[i] \leq \text{Need}[i]$, then go to **step 2**.
- Otherwise, there is an error because P_i has exceeded its maximum claim.

2

- If $\text{Request}[i] \leq \text{Available}[i]$, then go to **step 3**.
- Otherwise, P_i must wait for resources to be released.

3

- Allocate the requested resources by performing:
 - $\text{Available} = \text{Available} - \text{Request}[i];$
 - $\text{Allocation}[i] = \text{Allocation}[i] + \text{Request}[i];$
 - $\text{Need}[i] = \text{Need}[i] - \text{Request}[i];$

4

- Call the **safety algorithm**

RESOURCE-REQUEST ALGORITHM

If the resource allocation state is safe

The transaction is completed and process P_i is allocated its resources

If the new state is not safe

P_i must wait and the previous resource-allocation state is restored:

- $Available = Available + Request[i];$
- $Allocation[i] = Allocation[i] - Request[i];$
- $Need[i] = Need[i] + Request[i];$

RESOURCE-REQUEST ALGORITHM

⦿ Drawbacks:

- It requires knowing the maximum number of resources each process may need.
 - This type of information is rarely available in real systems.
- **Pessimistic:** The algorithm may delay a resource request as soon as there is a risk of deadlock
 - even though in practice the deadlock might never occur.

DEADLOCK HANDLING METHODS

1

Prevention

2

Avoidance

3

Detection/Recovery

4

Combined Approach

DETECTION/RECOVERY

- ◉ Unlike the previous methods, this method consists of **detecting and recovering from deadlocks**.
 - Instead of preventing deadlocks, the system allows them to occur, then attempts to detect them and remedy the situation afterward.
- ◉ **Deadlock detection algorithm:**

Variable	Description
Available[m]	vector that defines the number of available resources for each type
Request[n][m]	matrix that indicates the current requests for each process
Allocation[n][m]	matrix that describes the number of resources of each type currently allocated to each process

DEADLOCK DETECTION ALGORITHM

1

- Let $Work[m]$ and $Finish[n]$
- $Work = Available$
- For $i = 0..n-1$, if $Allocation[i] \neq 0$, then $Finish[i] = false$; otherwise $Finish[i] = true$.

2

- Find an i such that:
- $Finish[i] == false$
- $Request[i] \leq Work$
- If no such i exists, go to step 4.

3

- $Work = Work + Allocation[i]$
- $Finish[i] = true$
- Go to step 2.

4

- If $Finish[i] == false$ for some i , then the system is in a deadlock state. **Each process P_i such that $Finish[i] = false$ is in a deadlock.**

DETECTION/RECOVERY

- Once a deadlock is detected by the detection algorithm, it can be resolved simply by applying one of the following techniques:
 - **Manual correction:** The system alerts the operator that a deadlock has occurred and prompts them to handle it manually, for example by restarting the system.
 - **Process termination:** Stop one or more processes. You can choose to terminate all processes, or terminate them one by one until the deadlock is eliminated.
 - **Resource preemption:** Preempt one or more resources by taking them from one process and giving them to another until the deadlock is resolved.

DEADLOCK HANDLING METHODS

1

Prevention

2

Avoidance

3

Detection/Recovery

4

Combined Approach

COMBINED APPROACH

- ⦿ Researchers have shown that none of these deadlock-handling methods alone is suitable for all resource allocation problems in operating systems.
- ⦿ These methods can be **combined**, allowing the selection of an **optimal technique** for **each class of resources** in a system.

THANK YOU FOR YOUR
ATTENTION

