# الملزمة السادسة

# Deadlock

- If the waited process request a resource held by other waiting process, this situation called a ***deadlock***.
- ***Resource instance:*** is the number of resources in the system of that type.
- Ex: if the system has two CPUs, then the resource type of CPU is two.
- A process use a resource in only the following sequence:

1. *Request:* if the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. *Use:* the process can operate on the resource.
3. *Release:* the process release the resource.

- a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by only another process in the set.
- Consider a system that has one printer and one tape drive. Suppose that process P1 is holding the tape drive and process P2 is holding the printer. If P1 requests the printer and P2 requests the tape drive, deadlock occurs.

## Deadlock Conditions

A deadlock ban arise if the following four conditions hold simultaneously in a system:

***1. Mutual exclusion:*** only one process at a time can use the resource. if another process requests that resource, it must wait until that resource has been released.

***2. Hold-and-Wait:*** there must exist a process that is holding at least one resource and is waiting additional resources that are currently being held by other processes.

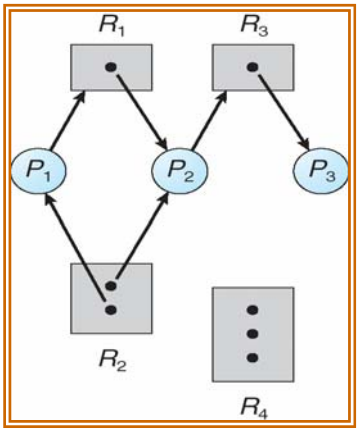
3***. Nopreemption:*** a resource can be released by the process holding it, after that process has completely its task.

***4. Circular wait:*** There must exist a set { $P_0$, $P_1$, ..., $P_n$} of waiting processes, such that $P_0$ waiting $P_1$, and $P_1$ waiting $P_2$, ..... $P_{n-1}$ waiting $P_n$, and $P_n$ waiting $P_0$.

## Resource-Allocation Graph

- This graph used to describe deadlock more precisely.
- It consist of:
  1. set of processes P={$P_1$, $P_2$, ...., $P_n$}
  2. set of resources R={ $R_1$, $R_2$, ..., $R_m$}
  3. set of edges E
     - $P_i \rightarrow R_j$ : denote that process $P_i$ request $R_j$.
     - $R_j \rightarrow P_i$ : denote that resource $R_j$ allocates to process $P_i$.

Ex: Let we have the following Resource Allocation Graph:



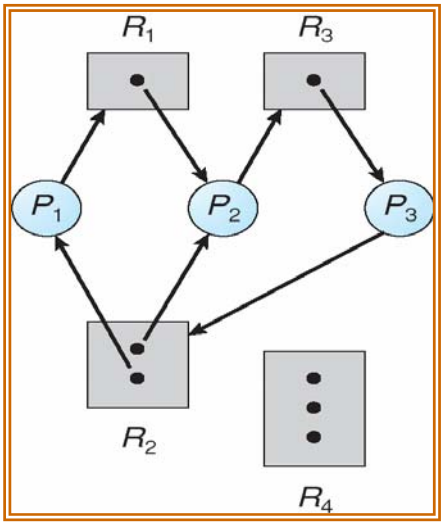

- The sets of P, R, and E are:
  - P={$P_1$, $P_2$, $P_3$}
  - R={$R_1$, $R_2$, $R_3$, $R_4$}
  - E={$P_1 \rightarrow R_1$, $P_2 \rightarrow R_3$, $R_1 \rightarrow P_2$, $R_2 \rightarrow P_2$, $R_2 \rightarrow P_1$, $R_3 \rightarrow P_3$}
- Resource instances:
  - One instance of $R_1$.
  - Two instances of $R_2$.
  - One instance of $R_3$.
  - Three instances of $R_4$.

- Process States
  - Process $P_1$ is holding an instance of resource $R_2$, and is waiting for instance of $R_1$.
  - Process $P_2$ is holding an instance of $R_1$ and $R_2$, and is waiting for instance of $R_3$.
  - Process $P_3$ is holding an instance of $R_3$.

**Notes:**

1. If the graph contains no cycle, then no process in the system is deadlocked.
2. If the graph contains a cycle, then:
   - if each resource type has exactly one instance, then deadlock occur.
   - If each resource has several instances, then deadlock may occur.

- Now consider that $P_3$ request an instance of $R_2$ ($P_3 \rightarrow R_2$)




At this point, two cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

So, processes $P_1$, $P_2$, and $P_3$ are deadlocked. $P_2$ is waiting for $R_3$, which is held by $P_3$. $P_3$ is waiting for either $P_1$ or $P_2$ to release $R_2$. $P_1$ is waiting $P_2$ to release $R_1$

# Deadlock Handling

1. Ensure that the system will ***never*** enter a deadlock state.
2. Allow the system to enter a deadlock state and then ***recover***.

## Deadlock Prevention

We prevent the deadlock by ensuring that at least one of the fourth conditions cannot hold.

1. *Mutual Exclusion*: it is not possible to denying this condition, since there is non sharable resources.
2. *Hold-and-wait*: we must guarantee that a process requests a resource, it does not hold any other resources.
3. *Nopreemption*: if a process that is holding some resources and request another resource that cannot be immediately to it, then all resources currently being her are released.
4. *Circular wait*: we order all the resources. And each process can request resources in an increasing order.
   Ex: F(tape drive)=1, F(disk drive)=2, F(printer)=3.

## Deadlock Avoidance

- Require that the system has some additional information about the processes.
- 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.

## 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 $Rj$ available.

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

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

**Safety Algorithm**

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

*Work* = *Available*

*Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find *i* such that both:

(a) *Finish* [*i*] = *false*

(b) *Needi* ≤ *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.

**Resource-Request Algorithm for Process *Pi***

*Request* = request vector for process *Pi*. If *Requesti* [*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*.

1. If *Requesti* ≤ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If *Requesti* ≤ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available.

3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

*Available = Available – Request;*

*Allocationi = Allocationi + Requesti*;

*Needi = Needi – Requesti;*

- *If safe ⇒ the resources are allocated to Pi.*

- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

***EX:*** Lets have 5 processes *(P*0- *P*4); 3 resource types: *A* (10 instances), *B* (5instances), and *C* (7 instances).

| | *Allocation* | *Max* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| *P*0 | 0 1 0 | 7 5 3 | 3 3 2 |
| *P*1 | 2 0 0 | 3 2 2 | |
| *P*2 | 3 0 2 | 9 0 2 | |
| *P*3 | 2 1 1 | 2 2 2 | |
| *P*4 | 0 0 2 | 4 3 3 | |

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

| | *Need* |
|---|---|
| | *A B C* |
| *P*0 | 7 4 3 |
| *P*1 | 1 2 2 |
| *P*2 | 6 0 0 |
| *P*3 | 0 1 1 |
| *P*4 | 4 3 1 |

- The system is in a safe state since the sequence < *P*1, *P*3, *P*4, *P*2, *P*0> satisfies safety criteria.

**Ex: *P*1 Request (1,0,2)**

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

| | *Allocation* | *Need* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| *P*0 | 0 1 0 | 7 4 3 | 2 3 0 |
| *P*1 | 3 0 2 | 0 2 0 | |
| *P*2 | 3 0 1 | 6 0 0 | |
| *P*3 | 2 1 1 | 0 1 1 | |
| *P*4 | 0 0 2 | 4 3 1 | |

- Executing safety algorithm shows that sequence < *P*1, *P*3, *P*4, *P*0, *P*2> satisfies safety requirement.
- Can request for (3,3,0) by *P*4 be granted?
- Can request for (0,2,0) by *P*0 be granted?