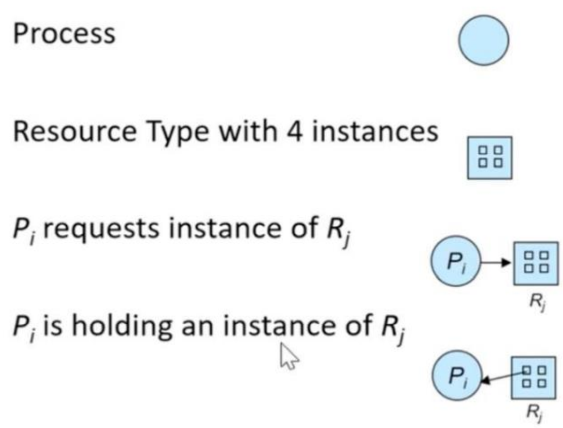
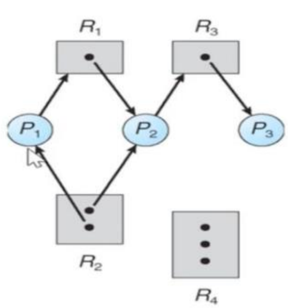
Deadlock

By : Mohamed Gamal Maklad

- **System Model** :
  - A **system consists of a finite number of resources** to be distributed among a number of competing processes.
  - The **resources** may be partitioned into several **types**:
    - CPU cycles, memory space, I/O devices (such as printers and DVD drives ())
  - Each process utilizes a resource as: **Request ➔ Use ➔ Release**
- **Deadlock can arise if four conditions hold simultaneously (لازم الأربعه يتحققوا مع بعض):**
  - **Mutual exclusion**: **only one process at a time** can use a resource
  - **Hold and wait**: a process **holding at least one resource is waiting to acquire additional resources** held by other processes
  - **No preemption**: a resource can **be released only voluntarily** by the process holding it, after that process has completed its task
  - **Circular wait**: there exists a set {P0, P1, P2 } of waiting processes such that P0 is waiting for a resource that is held by PI, PI is waiting for a resource that is held by P2, **..., P_{n_I} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P0.**
- **Resource-Allocation Graph** :
  - Deadlocks can be described more precisely in terms of a directed **graph called a system resource-allocation graph (A set of vertices V and a set of edges E.)**
  - **V is partitioned into two types:**
    - P ={P1,P2,….Pn}the set **consisting of all the processes** in the system
    - R={R1,R2,..Rm}, the set **consisting of all resource types** in the system
  - **E is partitioned into two types:**
    - **request edge** — directed edge Pi➔Rj
    - **assignment edge** — directed edge Rj ➔ Pi



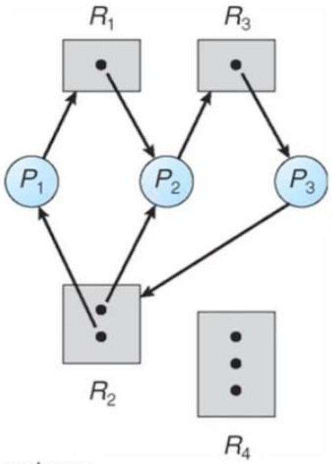
- **Basic Facts** :
  - If graph contains **no cycles no deadlock**
  - If graph contains a cycle :
    - **if only one instance per resource type, then deadlock**
    - **if several instances per resource type, possibility of deadlock**
- **Example of a Resource Allocation :**



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

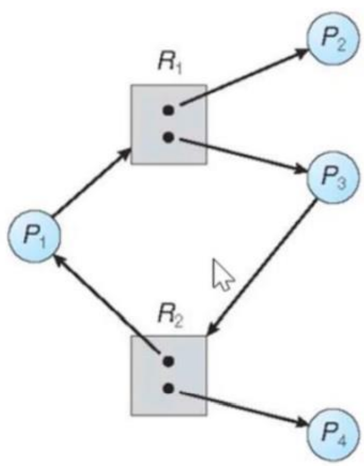
No Deadlock

There is no cycles So no deadlock



There is cycle and only one instance per resource type, then deadlock
So there **is Dead Lock**

Two cycles exist in the system:
P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2



There is cycle But several instances per resource type So There is no deadlock

ı cycle is exist:
P1 → R1 → P3 → R2 → P1

❖ **Methods for Handling Deadlocks**:
- Ensure that the system will **never enter a deadlock state**:
  - Deadlock prevention
  - Deadlock avoidance
- **Deadlock detection and recovery**: **Allow the system to enter a deadlock state and then recover**
- **Ignore the problem** and **pretend that deadlocks never occur** in the system; used by most operating systems, including UNIX

❖ **Deadlock Prevention**:
- **Mutual Exclusion** **not required for sharable** resources (e.g., read only files); must **hold for non-sharable resources**
- **Hold and Wait** must **guarantee that whenever a process requests a resource**, it does not hold any other resources:
  - **Require process to request** and be **allocated all its resources before it begins execution**,
  - allow process to request resources **only when the process has none allocated to it.**
  - **Low resource utilization; starvation possible**

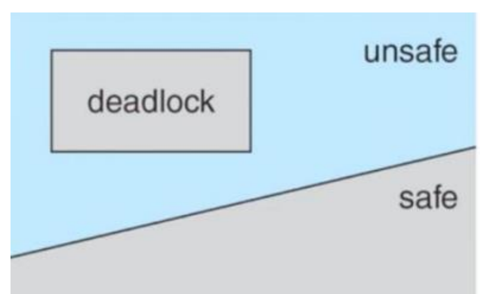
- **No Preemption:**
    - If a process **that is holding some resources requests another resource that cannot be immediately allocated to it**, then all resources currently being held are released
    - **Preempted resources are added to the list of resources** for which the process is **waiting**
    - **Process will be restarted only when it can regain its old resources**, as well as the new ones that it is requesting
- **Circular Wait** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance:

- Requires that the system has some additional a priori information available
- 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 sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
    - If Pi resource needs are not immediately available, then **Pi can wait until all Pj have finished**
    - When **Pj is finished**, Pi can obtain needed resources, execute, return allocated resources, and terminate
    - When *Pi terminates, Pi +1 can obtain its needed resources*, and so on



Basic Facts :

- If a system **is in safe state no deadlocks**
- If a system is in **unsafe state possibility of deadlock**
- **Avoidance ensure that a system will never enter an unsafe state.**

Avoidance Algorithms:

- Single instance of a resource type→ Use a **resource-allocation graph**
- Multiple instances of a resource type→Use the **banker's algorithm**

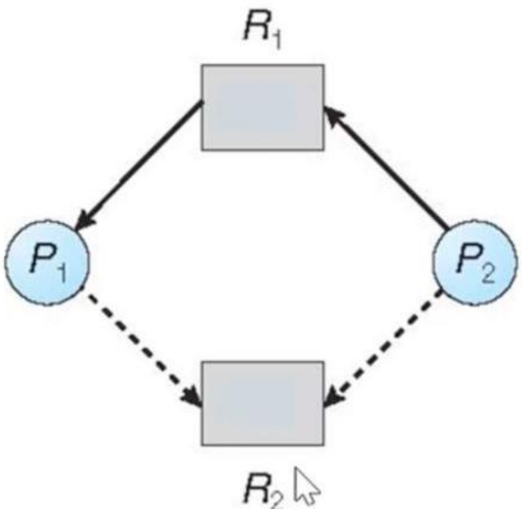
Resource-Allocation Graph Scheme:

- **Claim edge** Pi —> Rj indicated that **process Pi may request resource RJ**, **represented by a dashed line**
- **Claim edge converts to request** edge when a **process requests a resource**

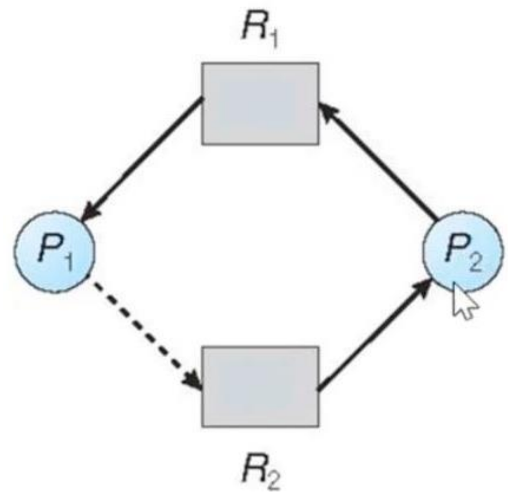
- **Request edge converted to an assignment** edge when **the resource is allocated to the process**
- **When a resource is released by a process, assignment edge reconverts to a claim edge**
- **Resources must be claimed a priori** (مسبقا)in the system

❖ Resource-Allocation Graph Algorithm :

- The **request can be granted only if converting the request edge to an assignment** edge does not result in the formation of a cycle in the resource allocation graph



Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph

❖ Banker's Algorithm:

- Multiple instances
- Each **process must a priori claim maximum use**
- When a process requests a set of resources, the **system must determine whether the allocation of these resources will leave the system in a safe state**
- If it will, the **resources are allocated; otherwise, the process must wait until some other process releases enough resources**

❖ Data Structures for the Banker's Algorithm:

n = number of processes **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:

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)
- Snapshot at time T_0:

	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	

اول حاجه هنعملها هنجيب need عن طريق

Need=Max -Allocation

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
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	

بعد كده هنشوف need<available لو لقينا هنعمل update للavailable

Available =Available + Allocation

- P0=>743<332
- P1=> 112<332 ➔ update variable➔ Available =200+332=532
- P2=>600<532
- P3=> 011<532 update variable➔ Available =211+532=743
- P4=>431<743 update variable➔ Available =002+743=745
- P0=>743<745 update variable➔ Available =010+745=755
- P2=>600<755 update variable➔ Available =302+755=1057

System is safe state since the sequence is < p1,p3,p4,p0,p2>

احنا هنا فضلنا نكرر لحد ما الشرط اتحقق ف بقت safety لو فضلنا كرر ان need<available و متحققش كده هيكون unsafty

---

❖ Resource Request Algorithm:

- If **Request <= Needi go to step 2**. Otherwise, raise error condition, since process has exceeded its maximum claim
- If **Requesti <= Available**, go to step 3. Otherwise Pi must wait, since resources are not available
- Pretend to allocate requested resources to Pi by modifying the state as:

  Available = Available — Request;

  Allocationi = Allocationj + Request

  Needi = Needi — Request;
- If **safe** the resources are allocated to Pi
- If **unsafe** Pi must **wait**, and the old resource-allocation state is restored

- Check that Request ≤ $Need_1$ (that is, (1,0,2) ≤ (1,2,2) ⇒ true
- 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	**3 3 2**
P_1	**2 0 0**	**1 2 2**	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Available = Available – Request;

102<122 بيتحقق

102<200 بيتحقق

كدده تمام الشرطين اتحققوا نخش ع الخطوه 3

Available =332-102= 230

Allocation = 200+ 102=302

Need = 122-102=20

	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	

كده بعد ما عملنا update للavailable ,allocation,need

هنكرر نفس الخطوات اللي كانت في safty

و نشوف هل هيكون في وضع safety و لأ

وهيطلع عندنا <p1,p3,p4,p0,p2>

P1 can request(1,0,2