CHAPTER 7

1. Introduction

- In a multiprogrammed environment, **multiple processes** may compete for a **finite** number of resources. A process requests resources; if those resources are not available at that time, the process enters a waiting state. It may happen that **waiting** processes **never change state** again, because the resources they request are held by other waiting processes. This situation is called a **deadlock**.
- The methods used by the system to address the deadlock problem are :
  - Prevention
  - Detection

System model

- A system consists of a finite set of resources that must be distributed to a certain number of concurrent processes.
  - Resources are grouped into several types.
  - **Resources:**
    - Physical: printer, CPU cycle, memory space.
    - Logical: files, semaphores, monitors.
  - Resources may have multiple instances, such as memory space, files, I/O devices, and CPU cycles.

System model

- Resources may have multiple instances, such as memory space, files, I/O devices, and CPU cycles.
- A process can use a resource only according to the following sequence of events :
  1. **Request**: If the request cannot be satisfied immediately, the requesting process must wait until it can acquire the resource.
  2. **Use**: The process has the resource and is using it.
  3. **Release**: The process releases the resource.
- **Request and Release are system calls.**

System model

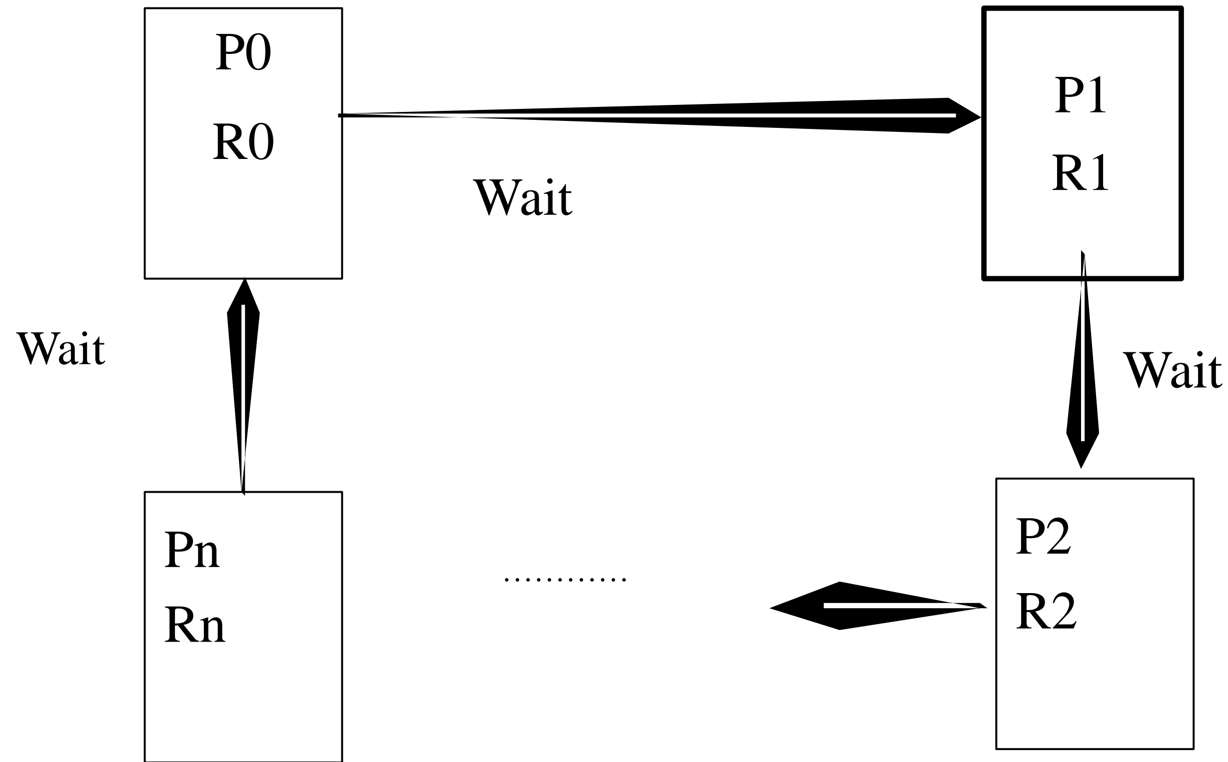
- **Request and Release are system calls.**
- **<u>Exemple :</u>**
  1. Request/Release ==> **device**
  2. Open/close ==> **file**
  3. Allocate/free ==> **memory**
- Therefore, for each use, the operating system (OS) must verify that the process has requested and been allocated the resource. A system table records the state of the resources (free/occupied) and to which process they are allocated.

DEADLOCK

- Un ensemble de processus est dans une situation d'interblocage, quand **chaque processus** de l'ensemble **attend** un événement qui ne peut être produit que par un **autre processus**. Les évènements sont **l'acquisition**/ **la libération** de ressources.
-
- A deadlock can occur if the following **four conditions** occur simultaneously in a system:

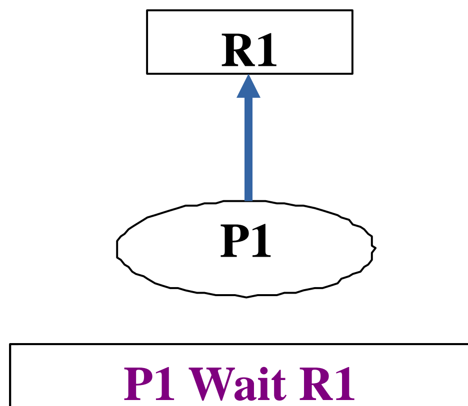
1. **Mutual exclusion:** Only one process can use a resource at a time. Other requesting processes are placed in waiting.
2. **Hold and wait:** There may exist a process holding at least one resource and waiting to acquire additional resources held by others.
3. **No preemption:** Busy resources cannot be preempted. Only the process holding the resource can release it.
4. **Circular wait:** A circular chain of two or more processes exists, where each process is waiting for a resource held by the next process in the chain.

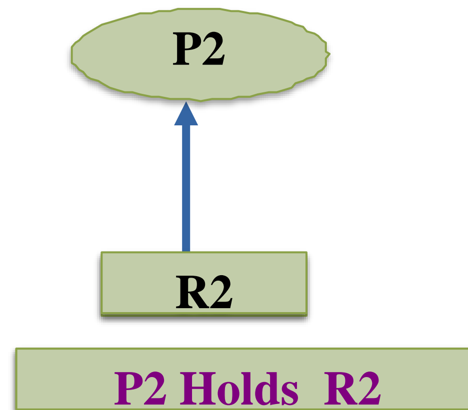
- **Circular wait:** {P0, P2,……,Pn} set of processes waiting

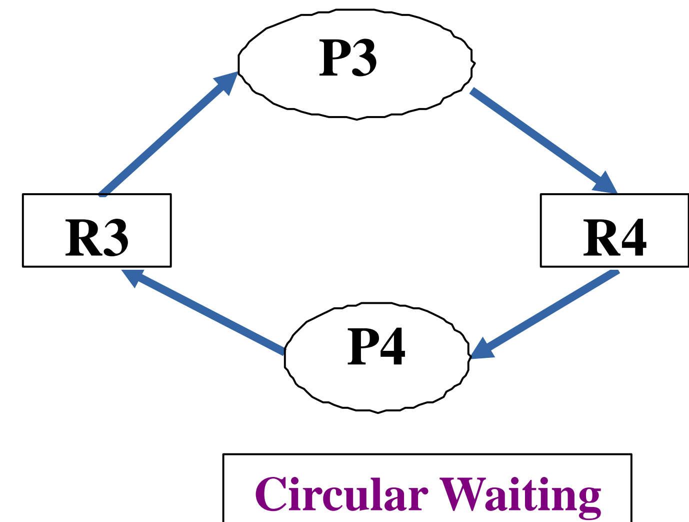


Modeling Deadlocks: Resource Allocation Graph

- Resource allocation graphs are used to model deadlock conditions. Two types of nodes are distinguished :
  - Processes
  - Resources
- An arc from a process to a resource indicates that the process is blocked waiting for that resource.

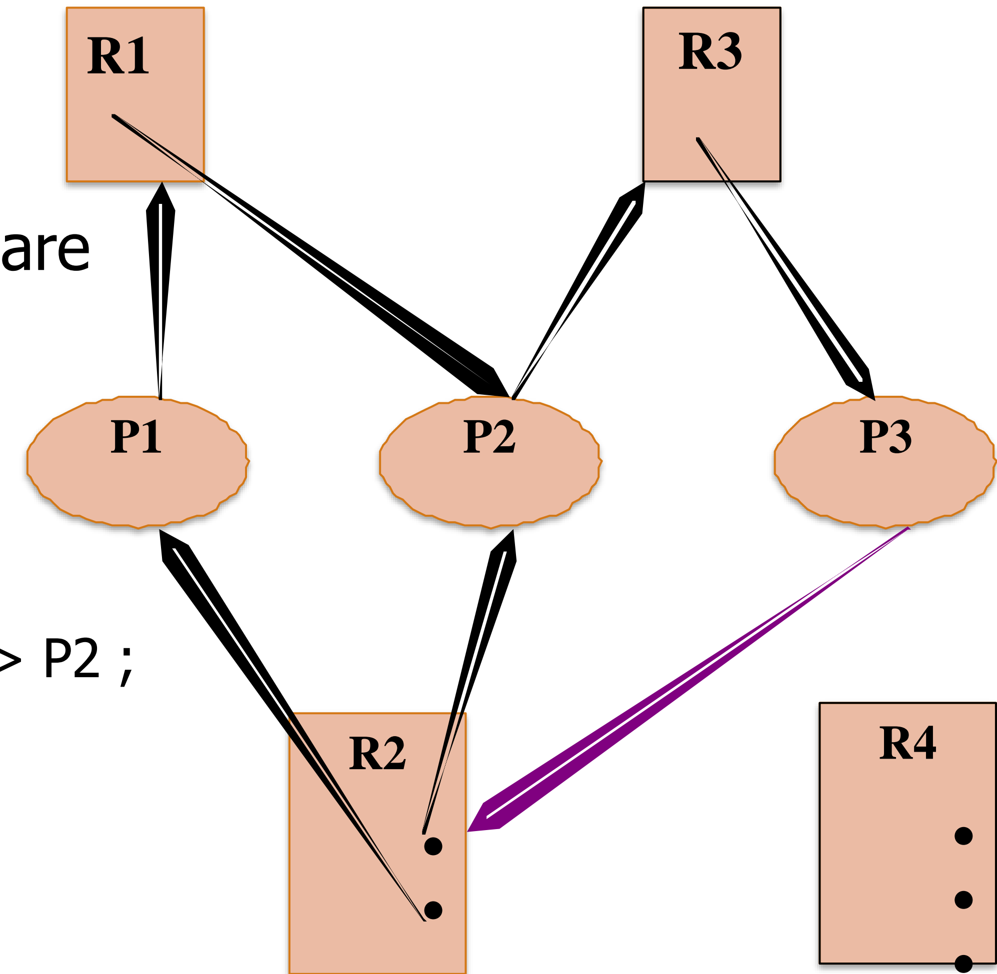






- A resource allocation graph that does not contain a cycle implies that the system is not in a deadlock state. If there is a cycle, then the system **may** or may not be in a **deadlock** state.

- **Exemple:**
  -
  - **P**= {P1, P2, P3}
  - **R**= {R1, R2, R3, R4}
  - The available instances of the resources are as follows :
    - R1 **->** 1 instance ; R2 **->** 2instances ;
      R3 **->** 1 instance ; R4 **->** 3 instances.
    - A= {P1 -> R1 ; P2 -> R3 ; R1 -> P2 ; R2 -> P2 ;
      R2 -> P1 ; R3 -> P3}
  - **P3 -> R2**



Methods for Handling Deadlocks

- Deadlocks can be handled using the following methods :
  - Preventing deadlocks
  - Avoiding deadlocks
  - Detecting and recovering from deadlocks

5.1 Preventing deadlocks

- **Principle**: This method prevents one of the four conditions from occurring..
- **Prevention**: It is possible to prevent deadlocks if the four conditions are not met. This can be done by implementing a policy that makes one of the conditions impossible. However, such an implementation may introduce more problems than it solves, as the following analysis will demonstrate. :

1. **Mutual exclusion:**

Eliminating exclusive access to all resources is not a practical solution. Some resources cannot be shared. Example: Spooling.

2. **Hold and wait:**

Processes request resources without already holding any resources.

5.1 Preventing deadlocks

- **First strategy**: One strategy that enforces this restriction is to require all possible resources before a process starts its execution..

- Disadvantages :
  1. The resources required by a process are not known in advance.
  2. Resources may be dependent on the execution of other processes. For example, a process that needs to print a document may need to wait for a printer to become available.
  3. Resources may be held for a long time before they are actually needed. This can waste resources and lead to performance problems.

5.1 Preventing deadlocks

Second strategy: Processes must release all held resources when a request is issued.

- **No-preemption condition:**

  Preemption of a resource is equivalent to forced sharing. The condition of no-preemption is a requirement of the resource release strategy. It state that resources cannot be preempted from processes. This is because preempting a resource would require the process to be suspended, which could lead to deadlocks.

- **Circular wait:**

  Eliminating circular wait is the most promising deadlock prevention technique. To break this cycle, one method is to assign each resource a unique priority number. Processes can only request resources if the priority is higher than all held resources. If the resource does not have a priority higher than all held resources, the one with the highest priority must be released first.

5.2 Avoiding Deadlocks

The prevention of deadlocks is a complex problem, and there is no single solution that works in all cases. The avoidance approach is one of several possible solutions. It works by using additional information about how resources will be requested to make decisions about whether to grant or deny requests.

- **Principle**: The use of additional information about how resources will be requested. With this knowledge of the complete sequence of requests for each process, we can decide for each request whether the process should or should not wait.

- Decision: Whether the current request can be satisfied or if it must wait to avoid a possible future deadlock. The system must take into account the currently available resources, those allocated to each process, and the future requests and releases of each process.

5.2 Preventing Deadlocks

- **Requirements**. The algorithms require::

1. **Each process declares the maximum number of resources of each type**. With this information about each process, it is possible to construct an algorithm that ensures that a system will never be in a deadlock situation.
2. **Other algorithms dynamically examine the resource allocation state** to ensure that a circular wait condition can never exist. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of processes..

5.2.1 Healthy State

- A healthy state is one in which the system can allocate resources to each process (up to its maximum) in a certain order and still avoid deadlock.
- <P1, P2 ,...,Pn> Is a healthy sequence of processes for the current allocation state if for each process Pi, the resource requests of Pi can be satisfied by the currently available resources plus the resources held by all processes Pj with j < i.
- A healthy state is not a deadlock state.
- **Example.**

Imagine a system with 13 printers and 3 processes P0, P1, and P2.

Process	Maximum Resource Needs	Current Resource Needs
P0	11	5
P1	5	2
P2	10	2

At time t0, the system is in a healthy state. The sequence <P1, P0, P2> is healthy.

5.2.1 Healthy State

- However, the following state is an unhealthy state because there does not exist a healthy sequence.

Process	Maximum Resource Needs	Current Resource Needs
P0	11	5
P1	5	2
P2	10	3

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

Resource Allocation Graph Algorithm.

- **Constraint**: There is only one instance of each resource type.
- **Principe.** In addition to request and allocation arcs, we introduce a new type of arc called **a claim arc**.

- A claim arc **Pi - - -> Rj** indicates that process **Pi** may request resource **Rj** at some point in the future (having the same direction as the request arc and written in dotted lines).
- When process **Pi** requests resource **Rj**, the claim arc **Pi - - -> Rj** is transformed into a request arc.
- When **Rj** is released by **Pi**, the allocation arc **Rj ===> Pi** is converted back into a claim arc **Pi - - -> Rj**.
- Before a process starts its execution, all of its claim arcs must already appear in the resource allocation graph.
- When process Pi requests resource Rj, the request can only be granted if the transformation of the request arc Pi------>Rj into allocation arc Rj===>Pi does not cause **a cycle in the resource allocation graph**.
- We check whether a state is healthy or not using a cycle detection algorithm.

- **Constraint**: There can be multiple instances of each resource type.
- **Principle.**
  - Each process must declare the maximum number of instances of each resource type that it needs.
  - This number must be less than the total number of resources in the system.
  - If the allocation of the requested set of resources leaves the system in a healthy state, then the resources are allocated.
  - Otherwise, the process must wait for other processes to release enough resources.

Data Structures.

- • ***Available*** indicates the number of resources of each type.

  Available : Array[0..N-1] of integer;

  Available[i]=K ⇒ ∃ K ressources Ri.

- • ***Max*** indicates the maximum demand of each process.

  Max : Array[0..N-1, 0..M-1] of integer;

- • ***Allocation*** indicates the number of resources allocated.

  Allocation : Array[0..N-1, 0..M-1] of integer;

- • ***Need*** indicates the resources needed by each process to complete its work.

  Need : Array[0..N-1, 0..M-1] of integer;

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

Algorithm for Determining the Safe State.

1. Work := available;

   Finish[i]:= False; i:= 1..M

2. Find a process i such that:
   a. Finish[i] = False;
   b. Need[i] <= Work
   c. If no such process 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 processes i, then the system is in a safe state.
   a. Else the system is not in a safe state.

Resource Request Algorithm

Request i: A vector of resource requests for process i.

If Requesti[j] = k then the process Pi needs K instances of the resource type Rj.

1. If Request[i] <= Need[i], then go to step 2 Else error, the system has exceeded its maximum claim.
2. If Request[i] <= Available, then go to step 3 Else process i must wait
3. Allocate the requested resources to process i

- Available:= Available – $request_i$;
- $Allocation_i$:= $Allocation_i$ + $Request_i$;
- $Need_i$:= $Need_i$ - $Request_i$

- **If the resource allocation state is safe, the transaction is complete and the requested resources are allocated to process Pi. However, if the new state is not safe, Pi must wait and the old resource allocation state is restored.**

- **The disadvantages of the algorithm are as follows:**

  - The algorithm is indeed very expensive in terms of execution time and memory for the system. This is because it requires maintaining several matrices and triggering at every resource request.
  - Knowledge of the maximum number of resources required for each process. This type of information is rarely available on systems.
  - The algorithm may delay a resource request as soon as there is a risk of deadlock (but in reality deadlock may not occur).

5.3 Deadlock Detection

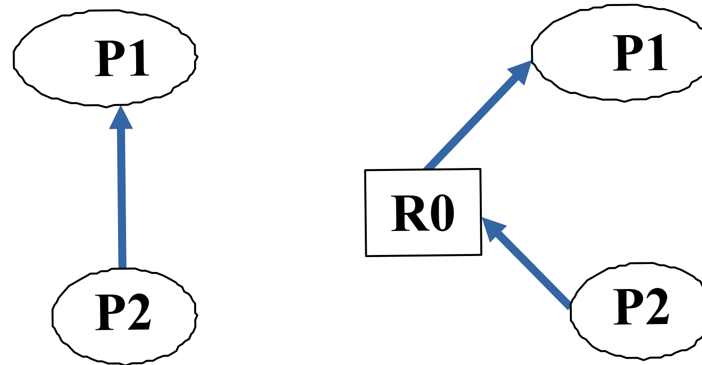
(Detection and Recovery)

- Systems may allow deadlocks to occur (assuming that they are rare); then recover from them when they do occur. In this environment, the system must provide:

1. An algorithm that examines the system state to determine if a deadlock has occurred.
2. An algorithm to recover from the deadlock.

5.3.1 System with a Single Instance of Each Resource Type

- **Constraint**: One instance of each resource type. The deadlock detection algorithm uses a variant of the resource allocation graph, called the **wait graph**. We obtain this graph from the resource allocation graph by removing resource-type nodes and folding appropriate arcs.



- An arc from process Pi to process Pj in a wait graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- **A deadlock occurs if the wait graph contains a cycle.**

Remark.

- **Other algorithms are used to detect deadlocks when the system contains multiple instances of the same resource type.**

5.3.2 Recovering from Deadlocks

There are several alternatives when a deadlock detection algorithm determines that one has occurred.

- One option is to inform the operator that a deadlock has occurred and let them handle it manually.
- **Another option is to let the system automatically recover from the deadlock**. There are two options for undoing a deadlock. One is to simply **terminate** one or more **processes** to break the circular wait. The second option is to **preempt resources** from one or more processes in the deadlocked situation.