

# Les systèmes d'exploitation 2

**Adil ENAANAI**

[adil.enaanai@gmail.com](mailto:adil.enaanai@gmail.com)

Département d'Informatique  
Faculté des Sciences -Tétouan-

# Les systèmes d'exploitation II

## Chapitres

**Chapitre 1:** Le système de gestion des fichiers

**Chapitre 2:** Gestion des processus

**Chapitre 3:** L'ordonnancement des processus

**Chapitre 4:** La communication et la synchronisation interprocessus

**Chapitre 5:** La gestion de la mémoire

**Chapitre 6:** La gestion des périphériques

*Filière:* SMI

*Semestre:* S3

*Prof.* Adil ENAANAI

# **Chapitre 1**

# **Le système de gestion des fichiers**

# Introduction

- L'ordinateur est composé de trois composants essentiels:
- Le microprocesseur
- La mémoire
- Le support de stockage





## Introduction

### Les composants de base



Pour l'exécution des processus



Pour la mémorisation des processus en cours d'exécution



Pour le stockage des données et des programmes



# Introduction

## Fonctionnement



```
Public static void main(){  
...  
}
```

Lorsque l'utilisateur appelle le programme (Clic), ce dernier est transféré vers la mémoire



```
1001101101  
1001101101  
.....
```

Mettre les instructions dans une file d'attente



Programme stocké sur le disque dur

La RAM mémorise le programme et ses données

Le processus exécute les instructions

# Introduction

## Problèmes

Où et comment un programme est mémorisé?

Est-ce qu'on peut lancer plusieurs programmes en même temps?

Est-ce qu'on peut mémoriser plusieurs programmes dans une même mémoire?

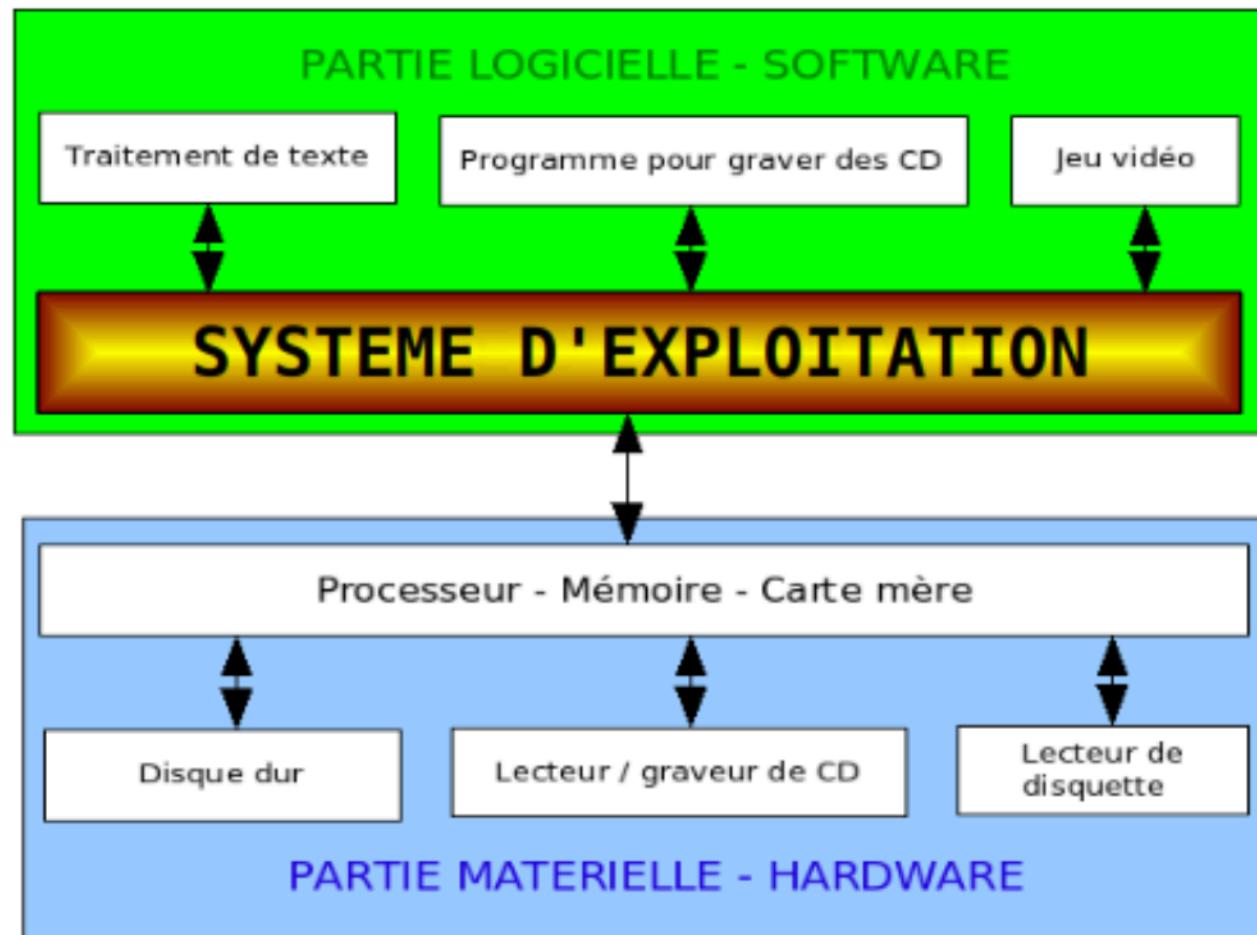
Est-ce que les autres périphériques peuvent demander les services du microprocesseur?

Comment donner une certaine priorité aux données et processus?



# Introduction

## Architecture





## Introduction

### Rôle d'un système d'exploitation

Le rôle principal d'un système d'exploitation est de cacher ce qui se passe dans la partie Hardware.

Il fournit une couche d'abstraction qui permet de simplifier l'utilisation du matériel

Les pilotes jouent le rôle d'interface entre le système d'exploitation et le matériel. Ils permettent de donner les fonctions de base pour bien utiliser le matériel.

## Introduction

### Rôle d'un système d'exploitation

C'est le noyau du système d'exploitation qui permet de:

- ✓ Gérer les processus
- ✓ Gérer la mémoire
- ✓ Gérer les fichiers
- ✓ Gérer les périphériques
- ✓ Gérer les utilisateurs





## Introduction

### Rôle du BIOS

Lorsqu'on appuie sur le bouton marche de l'ordinateur, le premier programme qui démarre s'appelle: BIOS



### Les étapes de démarrage d'un ordinateur

**Le Power-Good :** la mise sous tension envoie un signal à la carte mère afin d'initialiser le processeur.

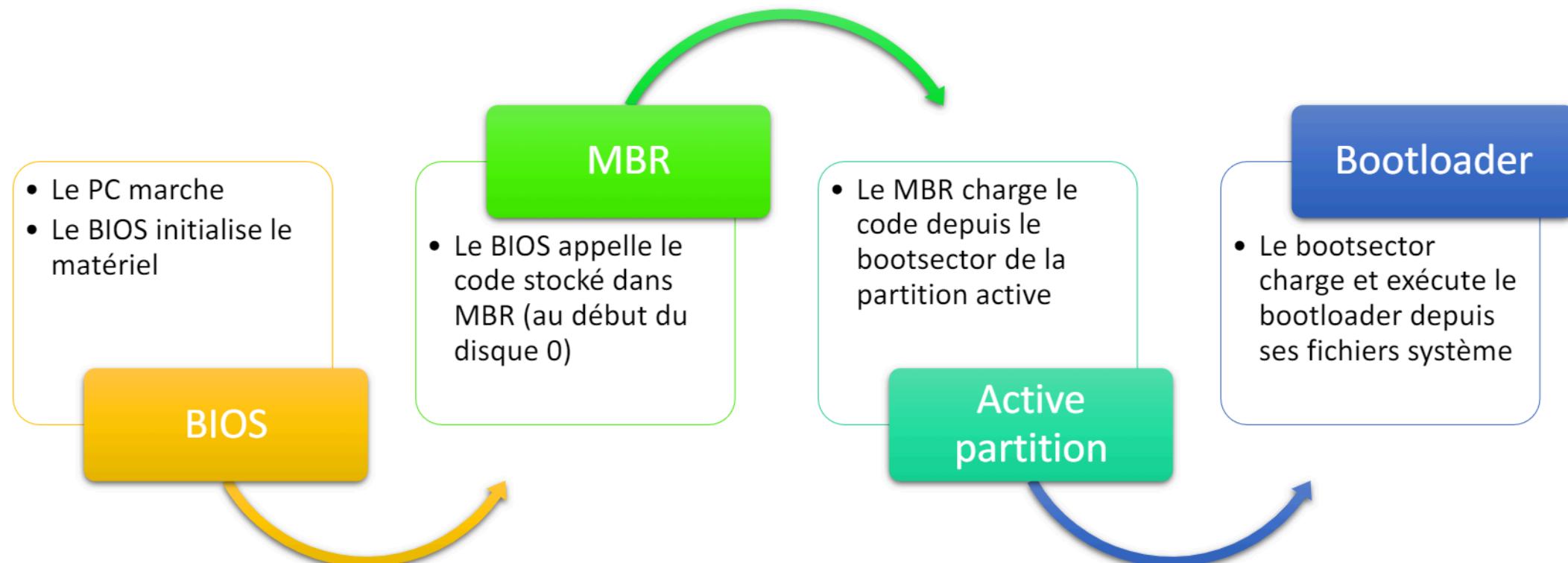
**Le POST :** un test de vérification sur l'ensemble des composants matériels

**Affichage des configurations :** le BIOS est paramétré pour afficher certaines informations liées au matériel et à l'état du système

**Lancement du système d'exploitation :** cette dernière étape concerne l'amorçage en lui-même du système d'exploitation, afin que celui-ci prenne le relais.

## Introduction

### Vue d'ensemble du processus de démarrage BIOS / MBR





## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Le BIOS

- Le BIOS inspecte le matériel
- Vérifie s'il y'a des problèmes
- Règle l'affichage
- Déetecte les périphériques USB
- Affiche des messages à l'écran dans certains cas
- Charge les 512 premiers octets du disque dur du périphérique de démarrage sélectionné

Ces 512 octets sont ce que l'on appelle communément le MBR, ou Master Boot Record

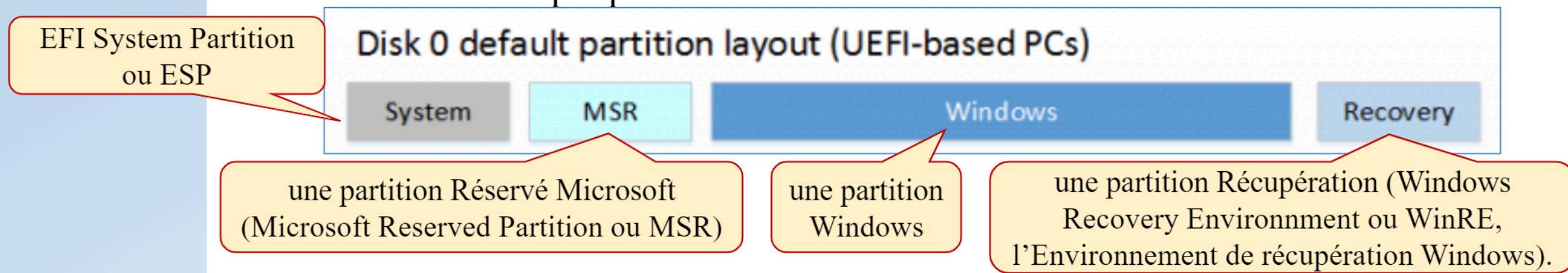
# Le démarrage d'un système d'exploitation

## Composants du processus de démarrage

### Table de partitionnement

Un disque a besoin d'être partitionné afin de pouvoir être utilisé : une ou plusieurs partitions doivent être créées dessus, c'est absolument indispensable.

Si on prend l'exemple d'un disque où le système d'exploitation Windows 10 est installé, quatre partitions essentielles au fonctionnement de l'OS sont créées sur le disque par l'installateur :





## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement

Toutes ces partitions sont définies dans la table des partitions, inscrite sur le disque. C'est cette table qui contient toutes les informations concernant le découpage du disque en partitions.

Il existe deux tables de partitionnement :

- ✓ **MBR** (Master Boot Record)
- ✓ **GPT** (GUID Partition Table)

## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -Le MBR (Master Boot Record)-

Le MBR (Master Boot Record) est le nom donné au premier secteur physique d'un disque.

Les informations qu'il contient sont capitales : c'est à l'intérieur de ce MBR que la table des partitions est définie et c'est grâce à lui qu'un système d'exploitation (Windows, Ubuntu...) peut démarrer.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le MBR (Master Boot Record)-**

**Le démarrage d'un système d'exploitation sur un disque MBR**

Au démarrage de l'ordinateur, après la séquence d'initialisation (Power-On Self Test ou POST), le BIOS lit le contenu du MBR du disque placé en première position dans l'ordre d'amorçage.

Le MBR – dont la taille est de 512 octets – contient :

- Un code d'amorçage (Master Boot Code).
- La table des partitions : les 4 partitions primaires.
- Une signature (0x55 AA).



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le MBR (Master Boot Record)-**

**Le démarrage d'un système d'exploitation sur un disque MBR**

Après la lecture du MBR du disque, le BIOS va exécuter le code d'amorçage situé dedans.

L'objectif du code d'amorçage est de lancer le chargeur d'amorçage (bootloader) du système d'exploitation situé sur la partition du disque marquée comme active. Sur Windows, le chargeur d'amorçage est Windows Boot Manager (**bootmgr**) ; sur Linux, c'est **GRUB**.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

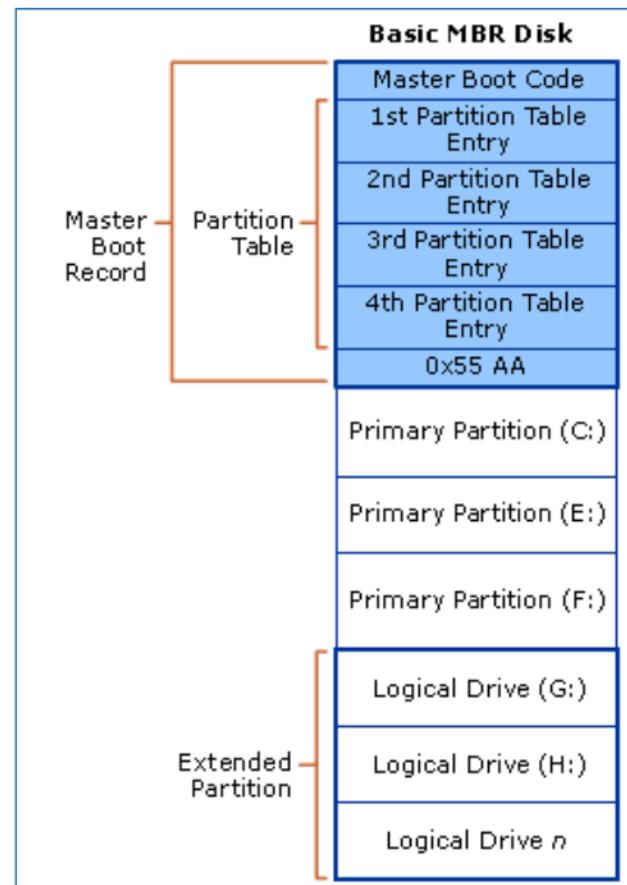
#### Table de partitionnement -Le MBR (Master Boot Record)-

##### Structure et caractéristiques

La table de partitionnement du MBR possède néanmoins des limitations, devenues de sérieux inconvénients à l'heure d'aujourd'hui :

- *4 partitions primaires maximum.*
- *Taille d'une partition limitée à 2,2 To.*

Ces limitations ont poussé les fabricants à se tourner vers une nouvelle table de partitionnement plus performante, introduite par Intel : **GPT** (GUID Partition Table). Le **GPT** est d'ailleurs une spécification de l'UEFI, le remplaçant du BIOS.





## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -Le MBR (Master Boot Record)-

##### Question1

Quels systèmes d'exploitation peuvent lire et écrire sur un disque MBR ?

**Réponse:** Tous les systèmes d'exploitation qui existent !

##### Question2

Quels systèmes d'exploitation peuvent démarrer sur un disque MBR ?

##### Réponse:

Sur un PC BIOS : toutes les versions de Windows et toutes les distributions GNU/Linux.

Sur un PC UEFI : toutes les versions de Windows après avoir activé le CSM (Compatibility Support Module) qui émule un environnement BIOS ; toutes les distributions GNU/Linux.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le MBR (Master Boot Record)-**

#### Question3

A quoi ressemble un disque MBR sur lequel on installe Windows ?

#### Réponse:

Lorsque vous installez Windows en mode BIOS/MBR, l'installateur de Windows crée :

- une partition Réservé au système (System Reserved) ;
- une partition Windows ;
- une partition Récupération (Recovery) [Windows 10 uniquement].



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -Le GPT (GUID Partition Table)-

Le GPT (GUID Partition Table) est un nouveau standard pour décrire la table de partitionnement d'un disque. Il est amené à remplacer celle du MBR à cause des limitations que l'on a vues tout à l'heure.

Le GPT fait parti du standard UEFI.

Voici les principaux avantages du GPT :

- Jusqu'à 128 partitions par disque.
- Jusqu'à 256 To par partition.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -Le GPT (GUID Partition Table)-

Le GPT a également quelques particularités :

Pour garantir une compatibilité avec les logiciels gérant uniquement le MBR, le GPT possède un MBR protecteur (Protective MBR).

Deux GPT sont présents sur le disque : l'un primaire, l'autre secondaire (qui est une sauvegarde du premier). Le primaire se situe au début du disque alors que le secondaire se situe à la fin du disque.

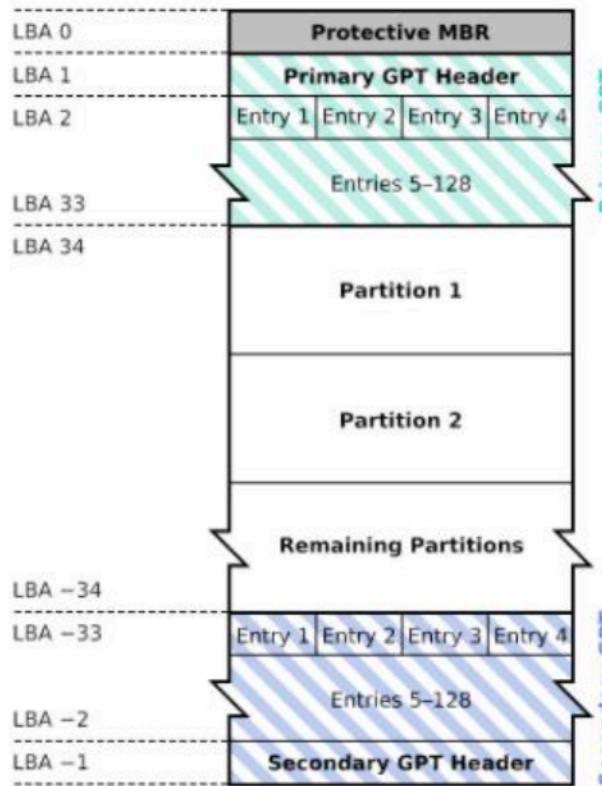
La table des partitions est précédée d'un entête qui contient les informations sur le disque : blocs utilisables, GUID, nombre et taille des partitions...

# Le démarrage d'un système d'exploitation

## Composants du processus de démarrage

### Table de partitionnement -Le GPT (GUID Partition Table)-

GUID Partition Table Scheme





## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le GPT (GUID Partition Table)-**

*Le démarrage d'un système d'exploitation sur un disque GPT*

Le GPT fonctionne de pair avec l'UEFI, au même titre que le MBR fonctionne de pair avec le BIOS. Mais contrairement au MBR, **le GPT ne contient pas de code d'amorçage.**

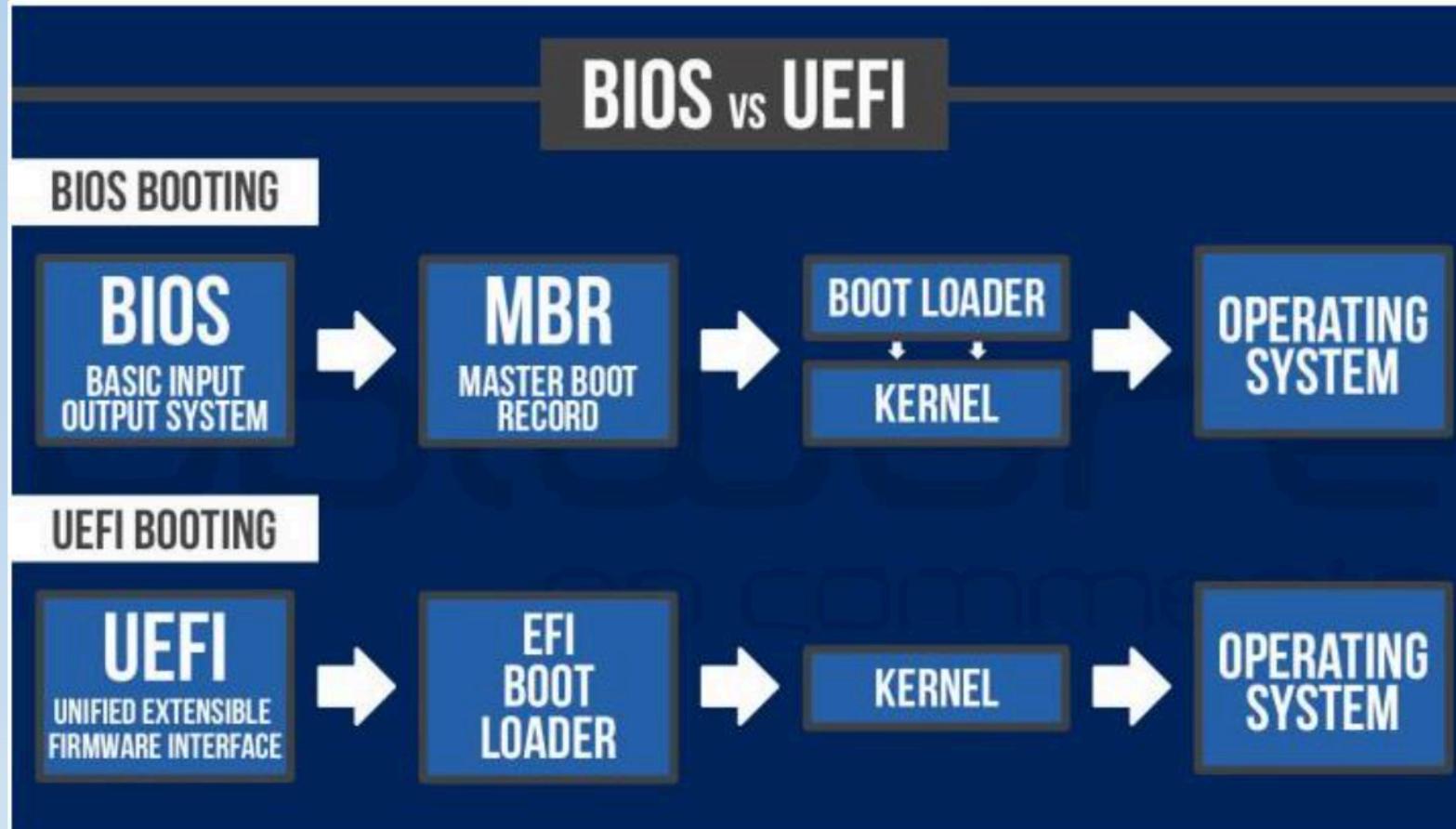
L'UEFI+GPT fonctionne différemment : l'UEFI charge directement le chargeur d'amorçage (**bootloader**) du système d'exploitation, enregistré sous la forme d'un fichier **.efi** sur la partition EFI du disque GPT. Par exemple, Windows Boot Manager (le bootloader de Windows) est situé dans le répertoire \EFI\Microsoft\Boot sous le nom « **bootmgfw.efi** »



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -Le GPT (GUID Partition Table)-



Si aucun fichier .efi n'est spécifié dans l'ordre d'amorçage, l'UEFI recherchera par défaut un fichier \EFI\Boot\bootx64.efi sur chaque disque GPT.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le GPT (GUID Partition Table)-**

#### Question1

Quelles versions de Windows peuvent lire un disque GPT ?

#### Réponse:

Toutes les versions 32 bits et 64 bits de Windows Vista, Windows 7, Windows 8, Windows 8.1 et Windows 10 ; et Windows XP Professionnel Édition 64 bits.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le GPT (GUID Partition Table)-**

#### Question2

Quelles versions de Windows peuvent démarrer sur un disque GPT ?

#### Réponse:

Toutes les versions 64 bits de Windows Vista SP1, Windows 7, Windows 8, Windows 8.1 et Windows 10.

Aucune version 32 bits de Windows, ni Windows XP Professionnel Édition 64 bits, ne peut démarrer sur un disque GPT.



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**Table de partitionnement -Le GPT (GUID Partition Table)-**

**Question3**

Peut-on utiliser à la fois un disque MBR et un disque GPT sur Windows ?

**Réponse:**

OUI

**Question4**

Peut-on convertir un disque MBR en GPT et inversement ?

**Réponse:**

OUI



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -Le GPT (GUID Partition Table)-

##### Question 5

A quoi ressemble un disque GPT sur lequel on installe Windows ?

##### Réponse:

Lorsque vous installez Windows en mode UEFI/GPT, l'installateur de Windows crée :

- une partition Système EFI (EFI System Partition ou ESP) ;
- une partition Réservé Microsoft (MSR) ;
- une partition Windows ;
- une partition Récupération

Disk 0 default partition layout (UEFI-based PCs)

System

MSR

Windows

Recovery



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

#### Table de partitionnement -les différences entre MBR et GPT-

MBR	GPT
<b>Ancienne</b> table de partitions	<b>Nouvelle</b> table de partitions
Lecture : <b>toutes</b> les versions de Windows	Lecture : Windows <b>Vista, 7, 8 et 10</b>
Démarrage : <b>toutes</b> les versions de Windows	Démarrage : uniquement les versions <b>64 bits</b> de Windows <b>Vista, 7, 8 et 10</b>
Jusqu'à <b>4</b> partitions primaires (ou <b>3</b> partitions primaires + <b>1</b> partition étendue permettant d'aller jusqu'à 128 partitions logiques)	Jusqu'à <b>128</b> partitions primaires
<b>2 To</b> maximum par partition	<b>256 To</b> maximum par partition



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

savoir si un disque est au format MBR ou GPT sur Windows

Méthode n°1 : avec DiskPart

Ouvrez une invite de commandes.

Ouvrez DiskPart puis listez les Disques

```
> | diskpart  
> | list disk
```

Si un astérisque est affiché dans la colonne GPT, c'est que le disque est au format GPT. S'il n'y a pas d'astérisque, c'est que le disque utilise la table de partitionnement du MBR.

Nº disque	Statut	Taille	Libre	Dyn	GPT
Disque 0	En ligne	238 G octets	0 octets		*
Disque 1	En ligne	931 G octets	0 octets		*
Disque 2	En ligne	119 G octets	0 octets		*



## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

**savoir si un disque est au format MBR ou GPT sur Windows**

**Méthode n°1 : avec DiskPart**

Pour avoir plus de détails sur un disque, entrez la commande suivante (remplacez 0 par le numéro du disque à analyser) :

```
> | select disk 0  
> | detail disk
```

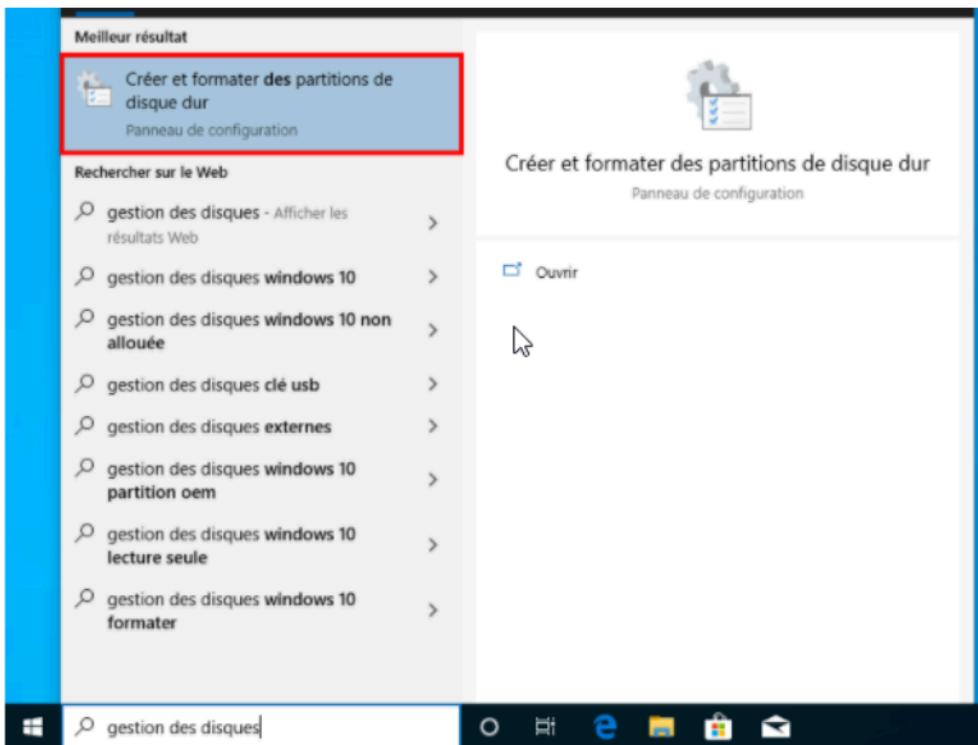
# Le démarrage d'un système d'exploitation

## Composants du processus de démarrage

savoir si un disque est au format MBR ou GPT sur Windows

Méthode n°2 : avec l'outil Gestion des disques

Ouvrez l'outil Gestion des disques : via le menu Démarrer ou la Recherche :





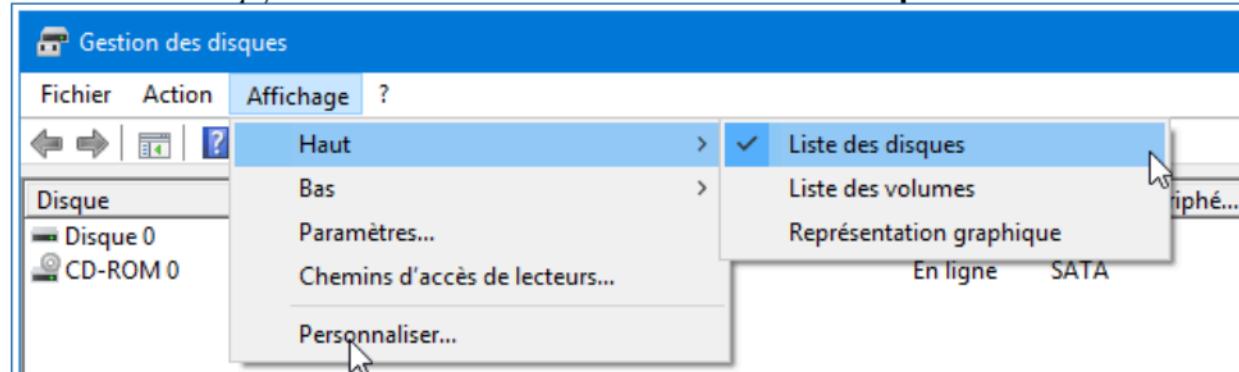
## Le démarrage d'un système d'exploitation

### Composants du processus de démarrage

savoir si un disque est au format MBR ou GPT sur Windows

Méthode n°2 : avec l'outil Gestion des disques

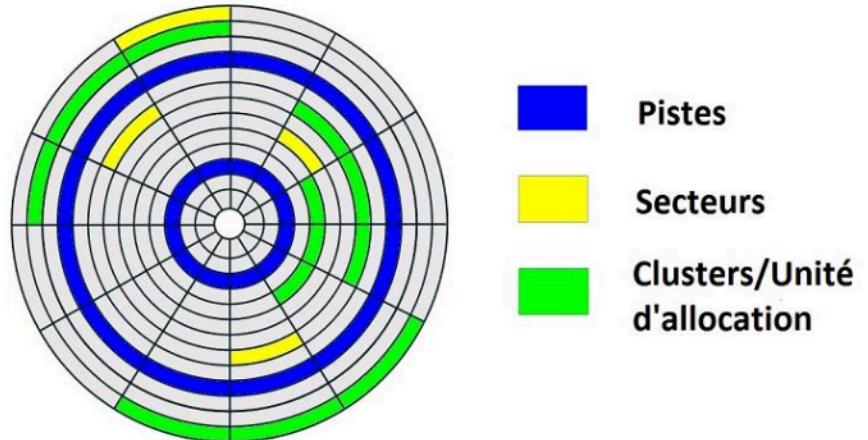
Sélectionnez Affichage > Haut > Liste des disques.



Regardez la colonne Type de partitions pour savoir si un disque est au format MBR ou GPT.

Disque	Type	Capacité	Espace non alloué	État	Type de périphé...	Type de partitions
Disque 0	De base	255,98 Go	3 Mo	En ligne	SATA	GPT
CD-ROM 0	CD-ROM	4,40 Go	0 Mo	En ligne	SATA	MBR

## Les clusters



- Pour rappel, toutes les mémoires de masse sont découpées en secteurs de taille fixe, qui possèdent tous une adresse.
- Les clusters sont des groupes de plusieurs secteurs consécutifs
- La taille idéale des clusters dépend du nombre de fichiers à stocker et de leur taille. Pour de petits fichiers, il est préférable d'utiliser une taille de cluster faible, alors que les gros fichiers ne voient pas d'inconvénients à utiliser des clusters de grande taille.
- En effet, un fichier doit utiliser un nombre entier de clusters.
- Le débit en lecture et écriture sera nettement amélioré avec des clusters suffisamment gros.

## Allocation des secteurs

- Le système d'exploitation doit mémoriser pour chaque fichier, quels sont les secteurs du HDD qui correspondent (leur adresse)
- Cette liste de secteurs est stockée dans une table de correspondance, située au tout début d'une partition : la Master File Table
- la Master File Table. Celle-ci contient, pour chaque fichier, son nom, ses attributs, ainsi que la liste de ses clusters

chaque partition est décomposée en quatre portions :

- ✓ le secteur de boot proprement dit, qui contient le chargeur de système d'exploitation ;
- ✓ la Master File Table, qui contient la liste des fichiers et leurs clusters ;
- ✓ et enfin, les fichiers et répertoires contenus dans le répertoire racine.

## Allocation des secteurs

Lors de la création ou de l'agrandissement d'un fichier, l'O.S doit lui allouer un certain nombre de secteurs.

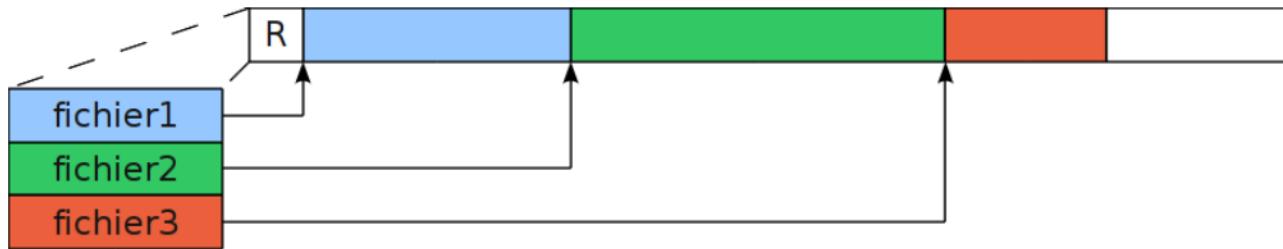
Pour cela, il existe diverses méthodes d'allocation:

- **Allocation contiguë**
- **Allocation par liste chainée**
- **Allocation par liste chainée et table**
- **Allocation par i-nodes**



## Allocation des secteurs

### Allocation contiguë



- Un fichier est une suite de blocs contigus sur le disque
- Localisation d'un fichier = adresse du premier bloc et nombre de blocs
- Lecture très performante, il suffit de se placer au premier bloc. Pas besoin de chercher les blocs

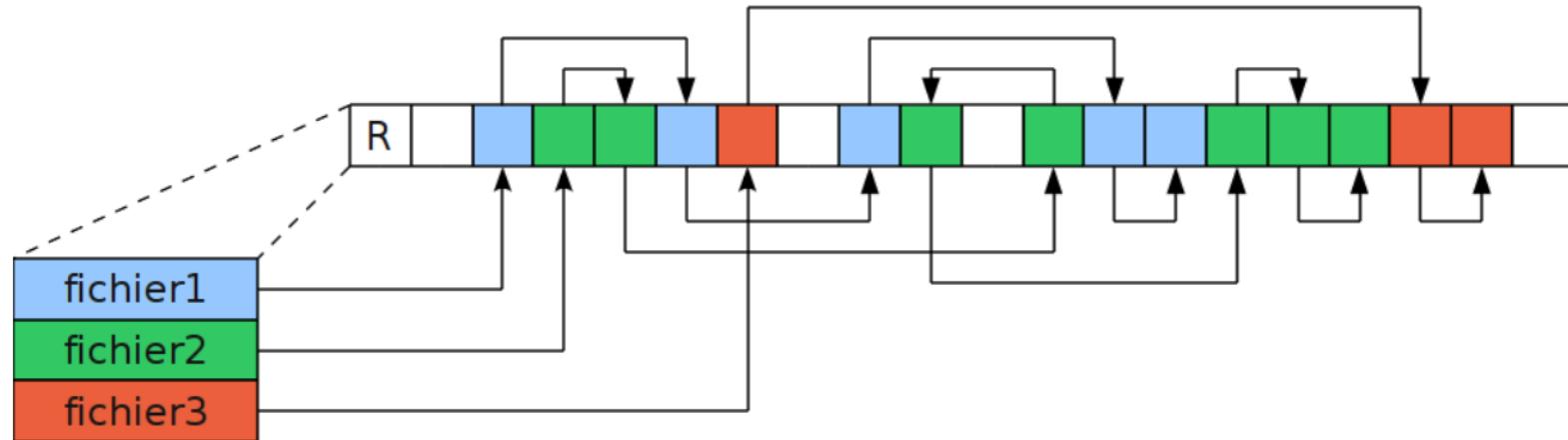
### Problèmes

- Au cours du temps, des trous apparaissent (fichiers supprimés):fragmentation
- On peut défragmenter le disque mais très coûteux
- On peut réutiliser les espaces libres mais il faut maintenir une liste, et connaître la taille finale d'un fichier
- Impraticable si les fichiers sont appelés à croître en taille



## Allocation des secteurs

### Allocation par liste chaînée

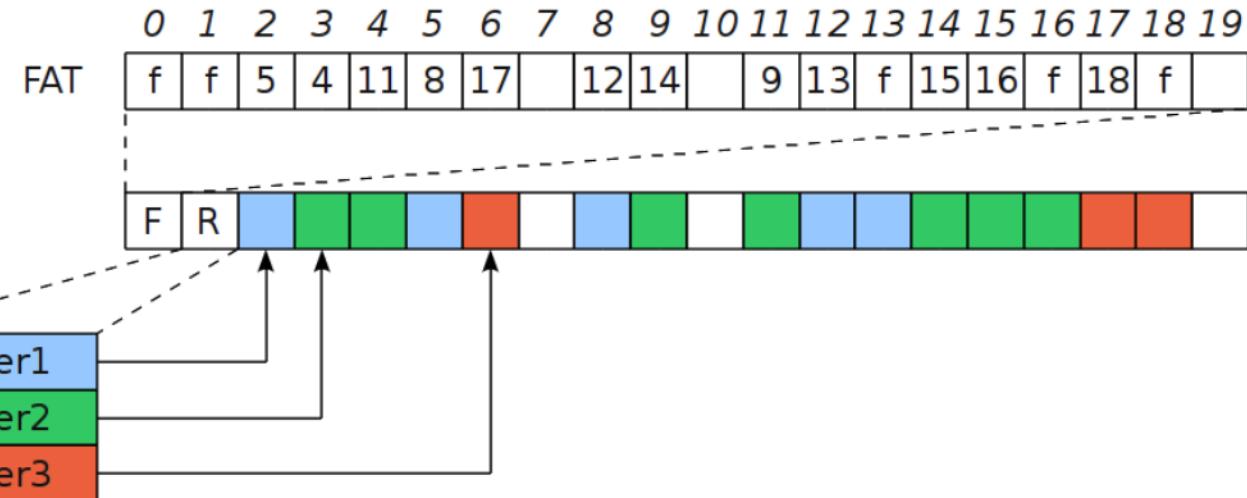


- Un fichier est une série de blocs maintenus dans une liste chaînée
- Le début d'un bloc est utilisé comme pointeur vers le suivant
- Pas de fragmentation externe
- Fragmentation interne sur le dernier bloc
- Un fichier est trouvé par son adresse de premier bloc
- La lecture séquentielle est très rapide
- Accès aléatoire très lent



## Allocation des secteurs

### Allocation par liste chainée et table



- Les pointeurs sont placés dans une table en mémoire
  - ✓ C'est une File Allocation Table (FAT)
- Le bloc redevient donc entièrement disponible pour les données
- Mais nécessite beaucoup de mémoire
- ✓ Un disque de 20GB avec des blocs de 1KB a une table de 20 millions d'entrées
- ✓ Chaque entrée nécessite 3 ou 4 octets => 60-80MB

## Allocation des secteurs

### Tables d'index

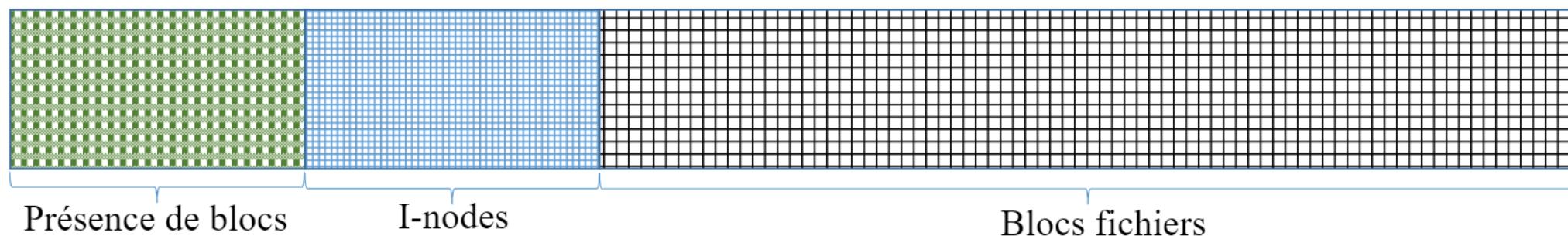


- Principe : une table contient la liste ordonnée des blocs contenant les données du fichier.
- on ne charge que les tables des fichiers ouverts
- la taille du fichier est limité par la taille de la table d'index

## Allocation des secteurs

### Allocation par i-node

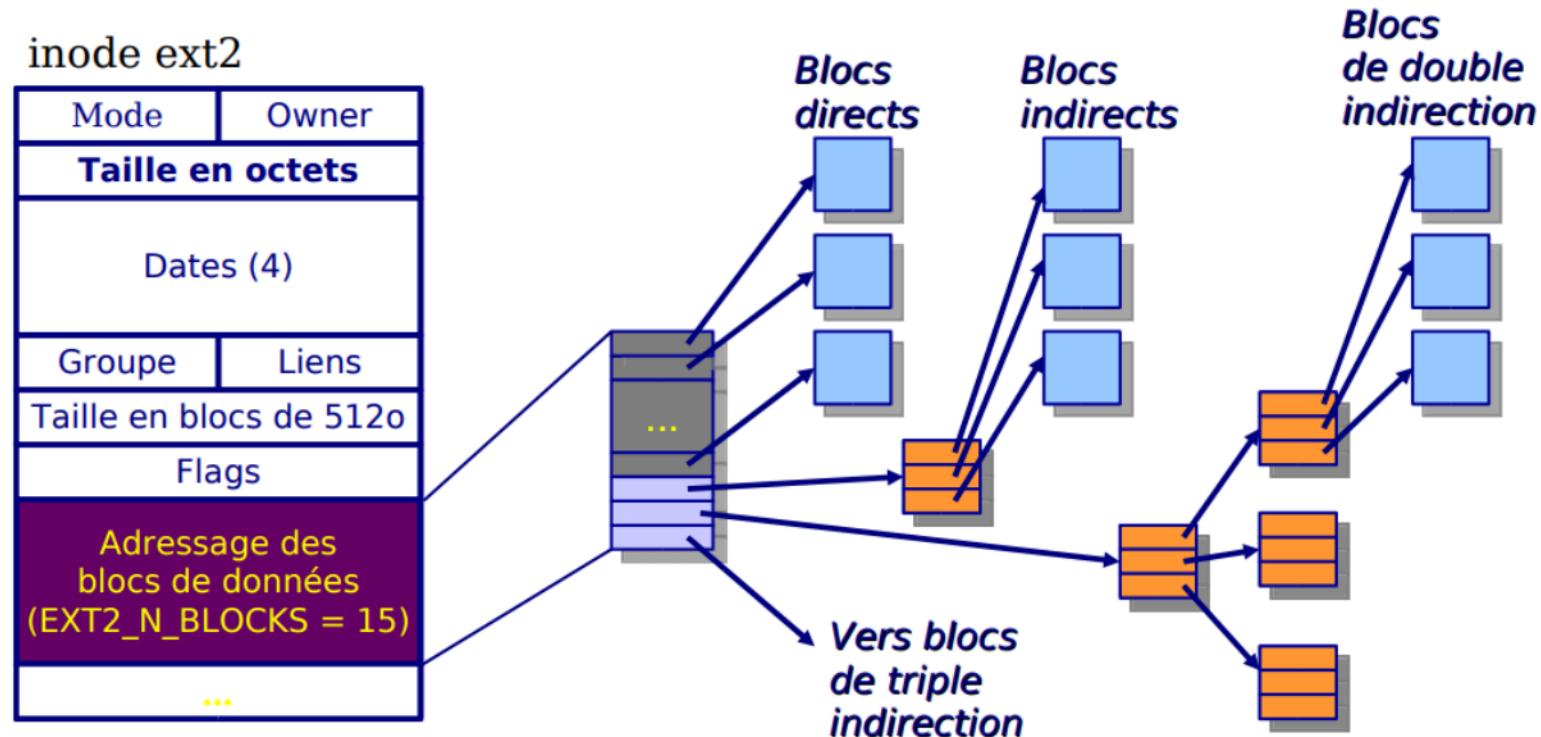
- I-node est une structure stockée sur le disque, allouée à la création du fichier et repérée par un numéro
- Contient les attributs du fichier :
  - ✓ Nom
  - ✓ Type : fichiers normaux, répertoires, périphériques, tubes nommés , sockets
  - ✓ Droits d'accès
  - ✓ Heures diverses
  - ✓ Taille du fichier en octets
  - ✓ Table des adresses des blocs de données



# Allocation des secteurs

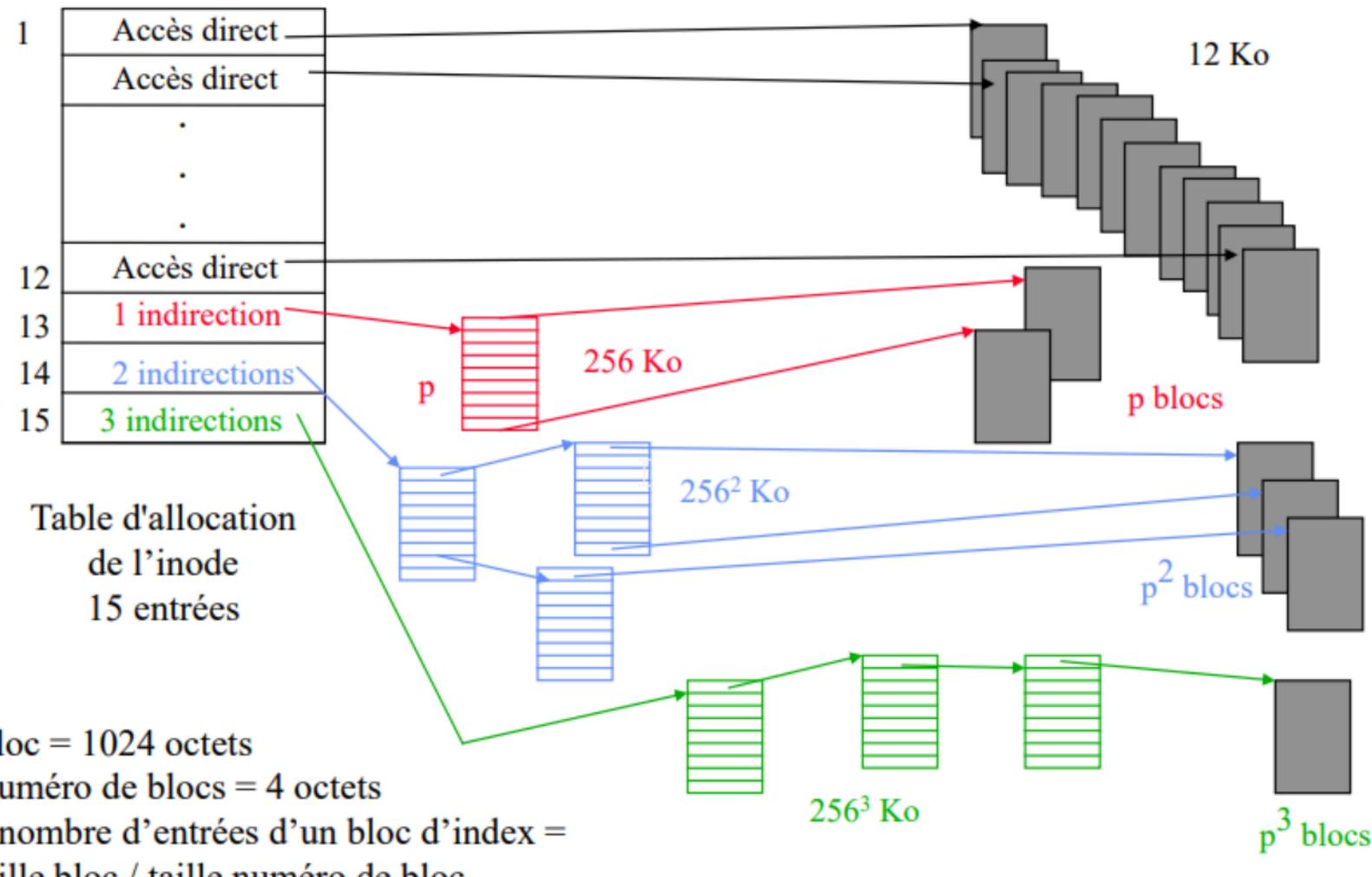
## Allocation par i-node

- L'inode contient les attributs du fichier
- Il y a ensuite un certain nombre de références vers des blocs
- Et finalement une indirection vers un autre groupe d'adresses



# Allocation des secteurs

## Allocation par i-node

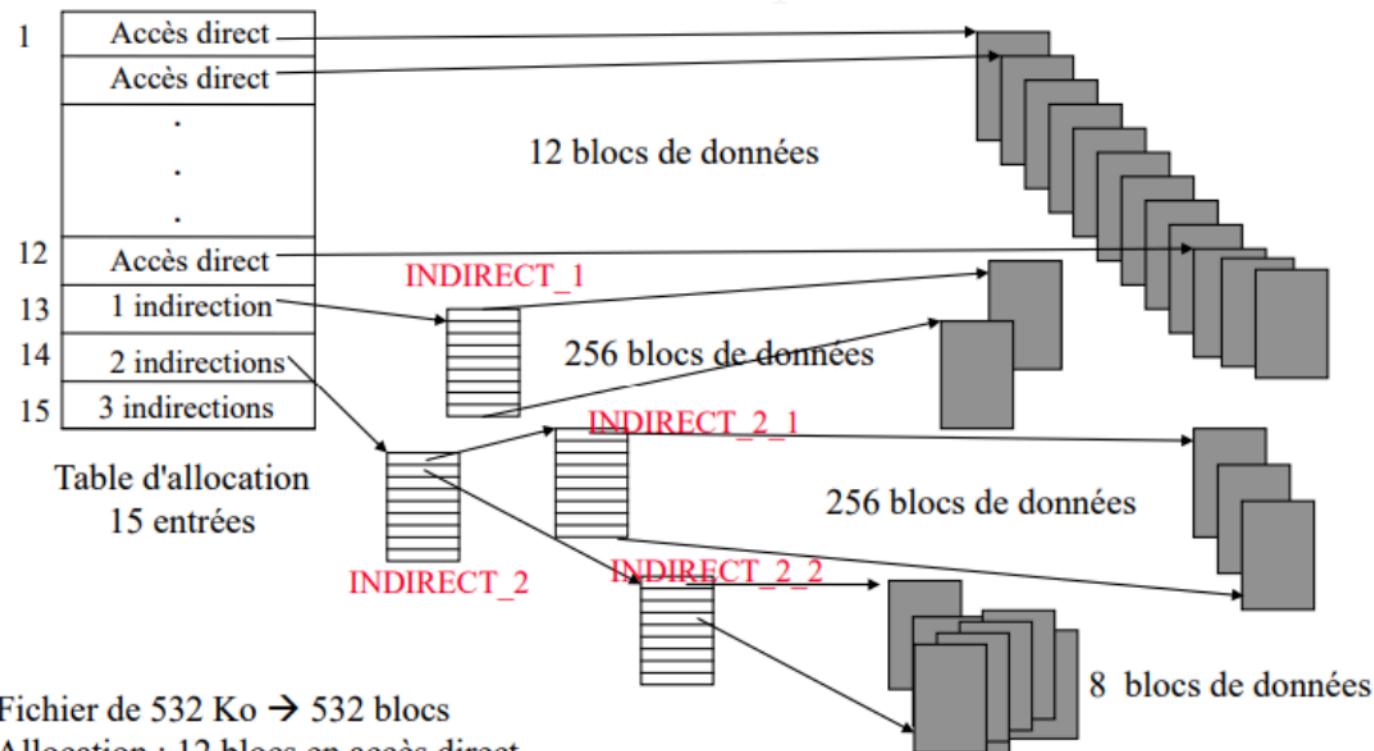


# Allocation des secteurs

## Allocation par i-node

### EXEMPLE

Bloc = 1024 octets ; adresse de bloc = 4 octets → 256 entrées dans le bloc d'index



Fichier de 532 Ko → 532 blocs

Allocation : 12 blocs en accès direct

: 256 blocs de données pointés par le bloc index INDIRECT 1

: restent  $532 - 12 - 256 = 264$  blocs . Tous ces blocs sont pointés à partir du bloc d'index INDIRECT\_2. 2 blocs d'index INDIRECT\_2\_1 et INDIRECT\_2\_2 sont nécessaires à ce niveau

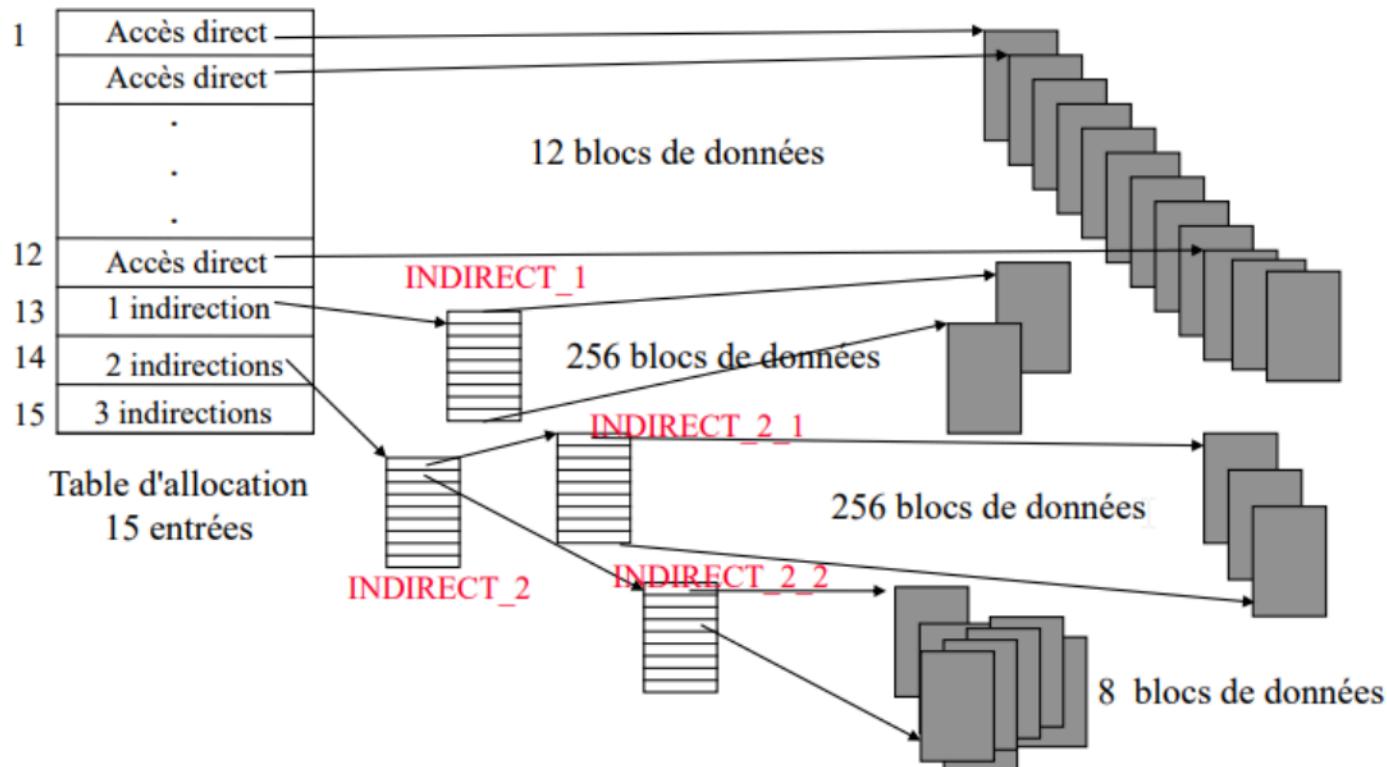


# Allocation des secteurs

## Allocation par i-node

### EXEMPLE

Bloc = 1024 octets ; adresse de bloc = 4 octets → 256 entrées dans le bloc d'index



Nombre d'accès disque pour lire le fichier :

$$12 + (2 * 256) + (3 * 256) + (3 * 8) = 1316 \text{ AD}$$

Temps de lecture =  $1316 * 10 \text{ ms} = 13160 \text{ ms} = 13 \text{ s}$

## Allocation des secteurs

### Allocation par i-node (exercice1)

On rappelle que sous Unix, un fichier est représenté de façon interne par la structure de données i-node. Cette structure comprend:

- un ensemble de 10 pointeurs directs qui pointent vers des blocs contenant les premières données du fichier
  - un pointeur indirect simple,
  - un pointeur indirect double,
  - et enfin un pointeur indirect triple (un degré de plus d'indirection).
- En supposant que la taille d'un bloc est de 512 octets et que la taille d'un index est de 4 octets, indiquer (en nombre de blocs et en octets) :
- 1) la taille minimale d'un fichier ?
  - 2) la taille maximale d'un fichier ? (réponse: 1.082.201.087 octets)



## Allocation des secteurs

### Allocation par i-node (Correction de l'exercice1)

1) La taille minimale d'un fichier ?

– Un fichier une fois créé va occuper au minimum 1 bloc ; sa taille est au moins égale à 512 octets

2) La taille maximale d'un fichier ?

– Avec 128 entrées pour chaque table de pointeurs et 512 octets par bloc, cela donne des fichiers d'au maximum

$$10 + 128 + 128*128 + 128*128*128 = 2.113.674$$

blocs soit une taille très confortable de  $512*2.097.152 = 1.082.201.087$  octets.

## Allocation des secteurs

### Allocation par i-node (exercice2)

La solution Unix (et GNU/Linux) est basée sur les blocs d'index, mais avec un raffinement. La structure de base (appelée inode, information node), contient les données suivantes :

- la taille
- l'identité du propriétaire et du groupe
- les droits d'accès
- la date de création
- le nombre de références existant pour ce fichier dans le système (liens physiques)
- les adresses des 10 premiers blocs de données
- l'adresse d'un bloc contenant 128 ou 256 adresses de blocs de données (simple indirection)
- l'adresse d'un bloc contenant 128 ou 256 adresses de blocs de données (double indirection).
- l'adresse d'un bloc contenant 128 ou 256 adresses de blocs contenant 128 ou 256 adresses de blocs de données (triple indirection)

## Allocation des secteurs

### Allocation par i-node (exercice2)

On se place dans le cadre d'un SGF de type Unix basé sur des inodes.

La structure est telle que celle décrite précédemment. Dans un inode donné, on note:  
TAB\_DIRECT la table de 10 éléments donnant les numéros des 10 premiers blocs du fichier

INDIRECT\_1 le pointeur vers la table de  $p$  numéros de blocs suivants

INDIRECT\_2 le pointeur vers la table de  $p$  numéros de bloc des tables de  $p$  numéros de blocs suivants (donc les  $p^2$  blocs suivants du fichier).

INDIRECT\_3 le pointeur vers la table de  $p$  numéros de bloc des tables de  $p$  numéros de blocs des tables de  $p$  numéros de blocs suivants (donc les  $p^3$  blocs suivants du fichier).

On supposera que les blocs font 1024 octets et qu'un numéro de bloc occupe 4 octets, et donc  $p = 256$ .

Par ailleurs, le temps d'accès moyen du disque est de 20 ms.

## Allocation des secteurs

### Allocation par i-node (exercice2)

Un processus lit séquentiellement un fichier de 8 MB, à raison de 256 octets à chaque lecture. Il fait donc 32768 demandes de lectures successives.

1. Décrire ce qui se passe lors des deux premières demandes de lecture de 256 octets, puis lors de la 5<sup>ème</sup> demande.
2. Décrire ce qui se passe lors des 41<sup>ème</sup> et 45<sup>ème</sup> demande.
3. Décrire ce qui se passe lors des 1065<sup>ème</sup> et 1066<sup>ème</sup> demande.
4. Décrire ce qui se passe lors des 2089<sup>ème</sup> et 2090<sup>ème</sup> demande.
5. En déduire le nombre total d'accès disque nécessaires et le temps d'attente en entrée-sortie.

## Allocation des secteurs

### Allocation par i-node (Correction de l'exercice2)

#### 1) 1<sup>re</sup>, 2<sup>ème</sup> et 5<sup>ème</sup> demande de lecture :

- 1<sup>re</sup>: lecture du 1<sup>er</sup> quart du bloc 1. bloc direct. 1 E/S
- 2<sup>ème</sup>: lecture du 2<sup>ème</sup> quart du bloc 1. bloc direct. 1 E/S
- 5<sup>ème</sup>: lecture du 1<sup>er</sup> quart du bloc 2. bloc direct. 1 E/S

Notez qu'on peut déduire de ce qui précède que

- a) 4 lectures étant nécessaires pour lire un bloc complet, cela générera 4 E/S ;
- b) la lecture des 10 blocs à adressage direct nécessitera 40 E/S.

#### 2) 41<sup>ème</sup> et 45<sup>ème</sup> demande de lecture :

- 41<sup>ème</sup>: lecture du 1<sup>er</sup> quart du 1<sup>er</sup> bloc en simple indirection : 2 E/S.
- 45<sup>ème</sup>: lecture du 1<sup>er</sup> quart du 2<sup>ème</sup> bloc en simple indirection : 2 E/S.

Donc, pour lire la totalité des 256 blocs en simple indirection, il faudra  $256 \times 4 \times 2 = 2048$  E/S.

## Allocation des secteurs

### Allocation par i-node (Correction de l'exercice2)

#### 3) 1065<sup>ème</sup> et 1066<sup>ème</sup> demande de lecture :

sachant qu'il a fallu 1064 demandes de lecture pour lire l'intégralité des blocs en adressage direct et en adressage à simple indirection, cela signifie que :

— 1065<sup>ème</sup>: lecture du 1<sup>er</sup> quart du 1<sup>er</sup> bloc en double indirection : 3 E/S.

— 1066<sup>ème</sup>: lecture du 2<sup>ème</sup> quart du 1<sup>er</sup> bloc en double indirection : 3 E/S.

#### 4. 2089<sup>ème</sup> et 2090<sup>ème</sup> lecture :

sachant qu'il faut  $256 \times 4 = 1024$  lectures pour lire les 256 blocs adressés par la première table de second niveau, on en déduit que :

— 2089<sup>ème</sup>: lecture du 1<sup>er</sup> quart du 1<sup>er</sup> bloc de la deuxième table en double indirection : 3 E/S

— 2090<sup>ème</sup>: lecture du 2<sup>ème</sup> quart du 1<sup>er</sup> bloc de la deuxième table en double indirection : 3 E/S

## Allocation des secteurs

### Allocation par i-node (Correction de l'exercice2)

Le fichier a une taille de 8 MB, soit 8192 blocs. En utilisant les calculs que nous avons fait en début de correction, on en déduit que la double indirection suffira. Notre fichier utilisera les 10 blocs à adressage direct, les 256 blocs à adressage en simple indirection, et  $8192 - 256 - 10 = 7926$  blocs à adressage en double indirection. Au vu de ce qui précède, ces 7926 blocs nécessiteront  $7926 \times 4 \times 3 = 95112$  E/S.

Au total, la lecture complète du fichier nécessitera  $40 + 2048 + 95112 = 97200$  E/S. Sachant que le temps d'accès moyen à un bloc est de 20 ms, le temps total d'E/S pour lire l'intégralité du fichier sera de  $97200 \times 20 = 1\,944\,000$  ms = 1944 s = 32 min24 s.

**Fin de chapitre 1**

**Chapitre 2**

**La gestion des processus et  
Ordonnancement**

## Définitions

**Programme:** C'est une suite d'instructions écrites dans un fichier en utilisant un langage de programmation

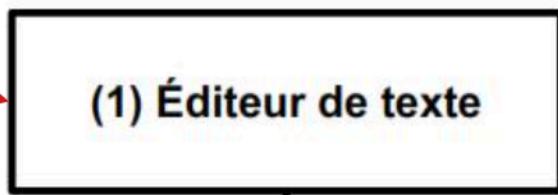
**Processus:** C'est un programme en cours d'exécution. Il est en langage machine et réside dans une mémoire. Le processeur lit ses instructions et les exécute.



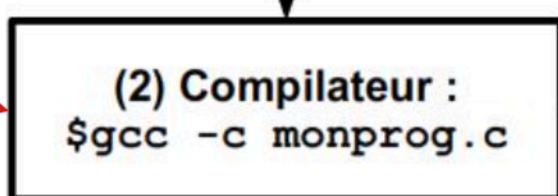
## Chaîne de production d'un programme exécutable

création un fichier sur le disque que l'on appelle le fichier source

- Le compilateur vérifie que la syntaxe du langage est respectée
- Le compilateur génère ensuite l'équivalent du programme source en langage machine



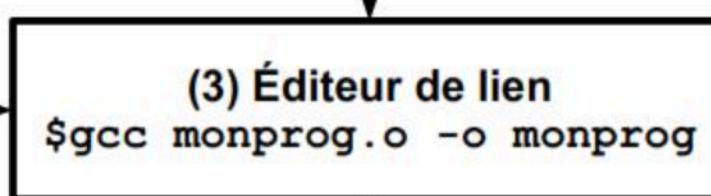
Source : monprog.c



objet: monprog.o

résovurer les références externes, c'est-à-dire par exemple, d'associer les appels système inclus dans le programme à leur adresse dans le système.

**Bibliothèque**



Ficher exécutable sur le disque :

Le système alloue de la place mémoire pour placer le code et les données du programme

**monprog : Exécutable en mémoire processus**

**(4) Chargeur**



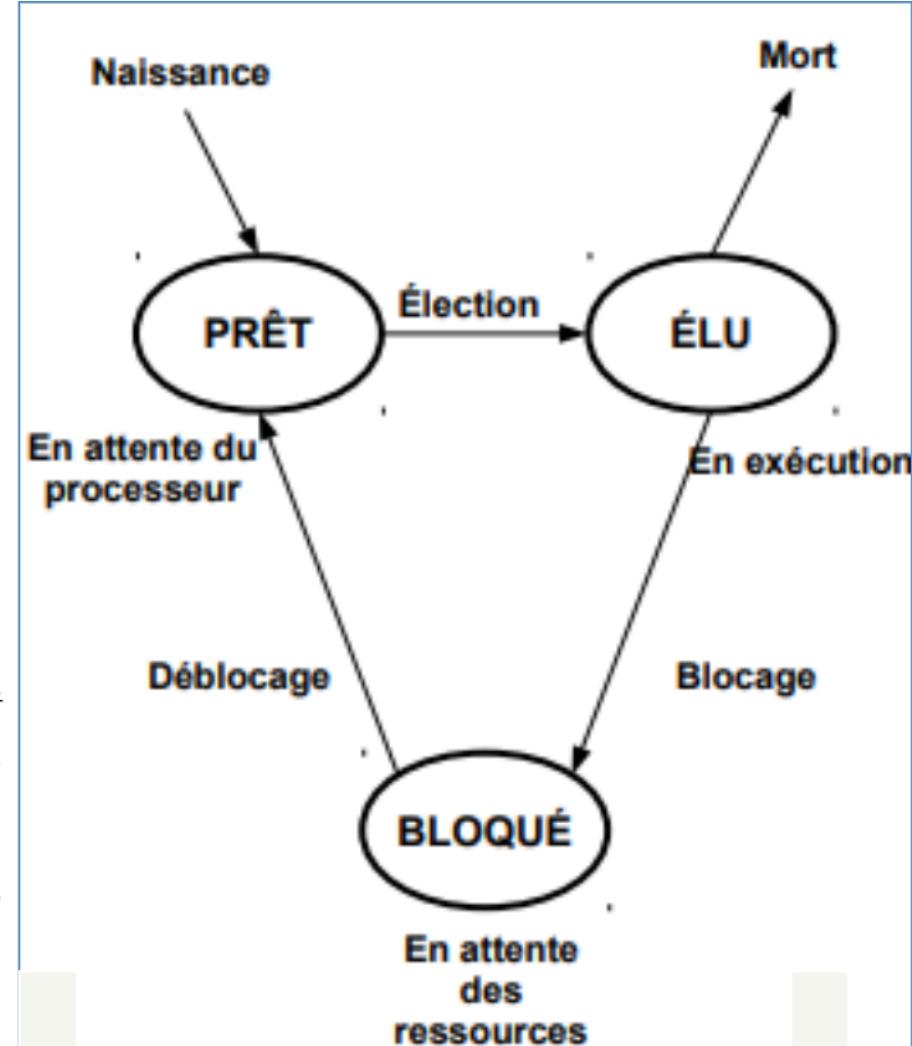
## Du monotâche vers le multitâche

- Les premiers OS autorisaient un seul programme à être exécuter à la fois.
  - *Un tel programme avait un contrôle complet du système et un accès à toutes les ressources du système*
  
- Les OS actuels permettent à plusieurs programmes d'être charger en mémoire et exécuter en même temps.
  - *Cette évolution a nécessité un contrôle plus strict et un cloisonnement plus rigoureux des différents programmes*

## Les états d'un processus

Lors de son exécution, un processus est caractérisé par un état

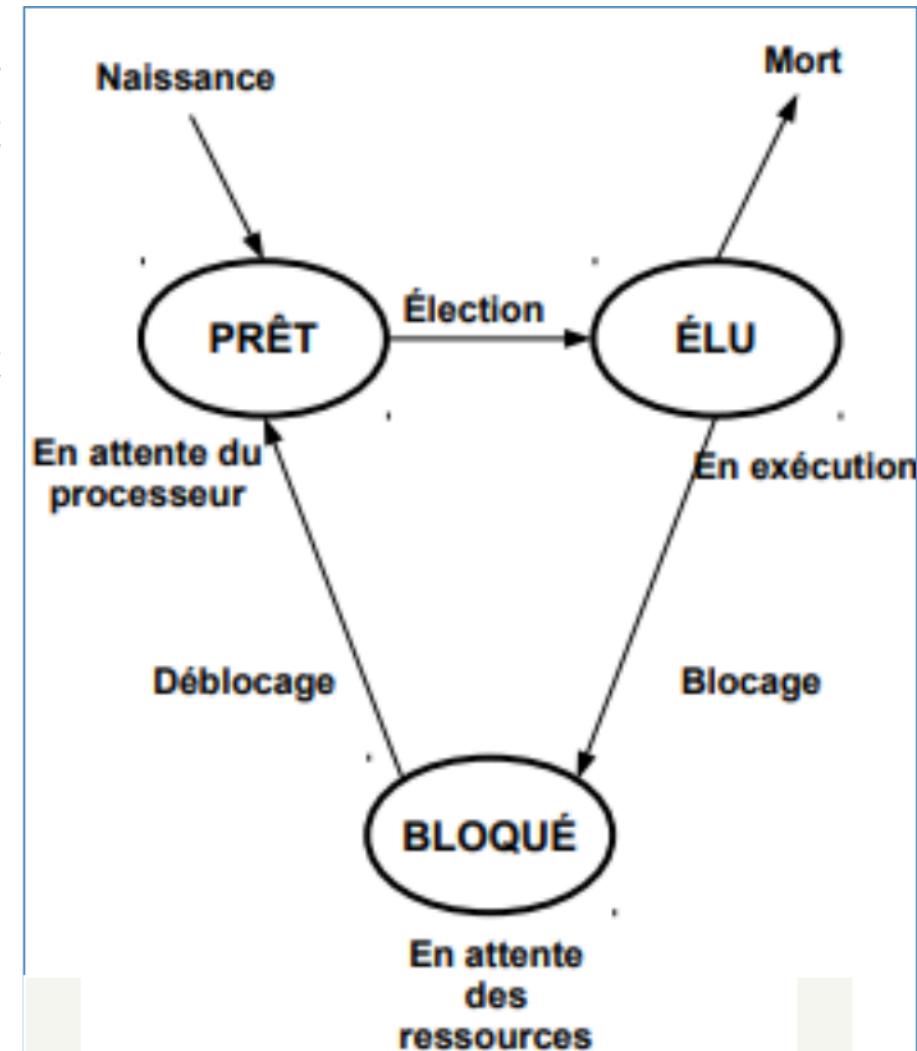
- Lorsque le processus obtient le processeur et s'exécute, il est dans l'état élu.
  - *L'état élu est l'état d'exécution du processus.*
- lors de cette exécution, le processus peut demander à accéder à une ressource qui n'est pas immédiatement disponible
  - *le processus ne peut pas poursuivre son exécution tant qu'il n'a pas obtenu la ressource le processus quitte alors le processeur et passe dans l'état bloqué.*
  - *L'état bloqué est l'état d'attente d'une ressource autre que le processeur.*



## Les états d'un processus

Lorsque le processus a enfin obtenu la ressource qu'il attendait, celui-ci peut potentiellement reprendre son exécution.

- Or, lorsque le processus est passé dans l'état bloqué, le processeur a été alloué à un autre processus.
  - *Le processeur n'est donc pas forcément libre.*
- Le processus passe alors dans l'état Prêt.
  - *L'état Prêt est l'état d'attente du processeur*





## Process Control Block (PCB)

Chaque processus est représenté dans le système d'exploitation par un bloc de contrôle de processus (Process Control Block - PCB) appelé aussi un task-control block

- Le système d'exploitation maintient dans une table appelée «table des processus»
  - *les informations sur tous les processus créés (une entrée par processus : PCB)*

process state
process number
program counter
registers
memory limits
list of open files
• • •

# Process Control Block (PCB)

Table des processus

PID	PCB
1	
2	
...	
n	



PCB du processus n

Compteur ordinal
Registres
État
Priorité
Espace d'adressage
Père
Fils
Fichiers ouverts
Actions associées aux signaux
....



PCB du processus 2

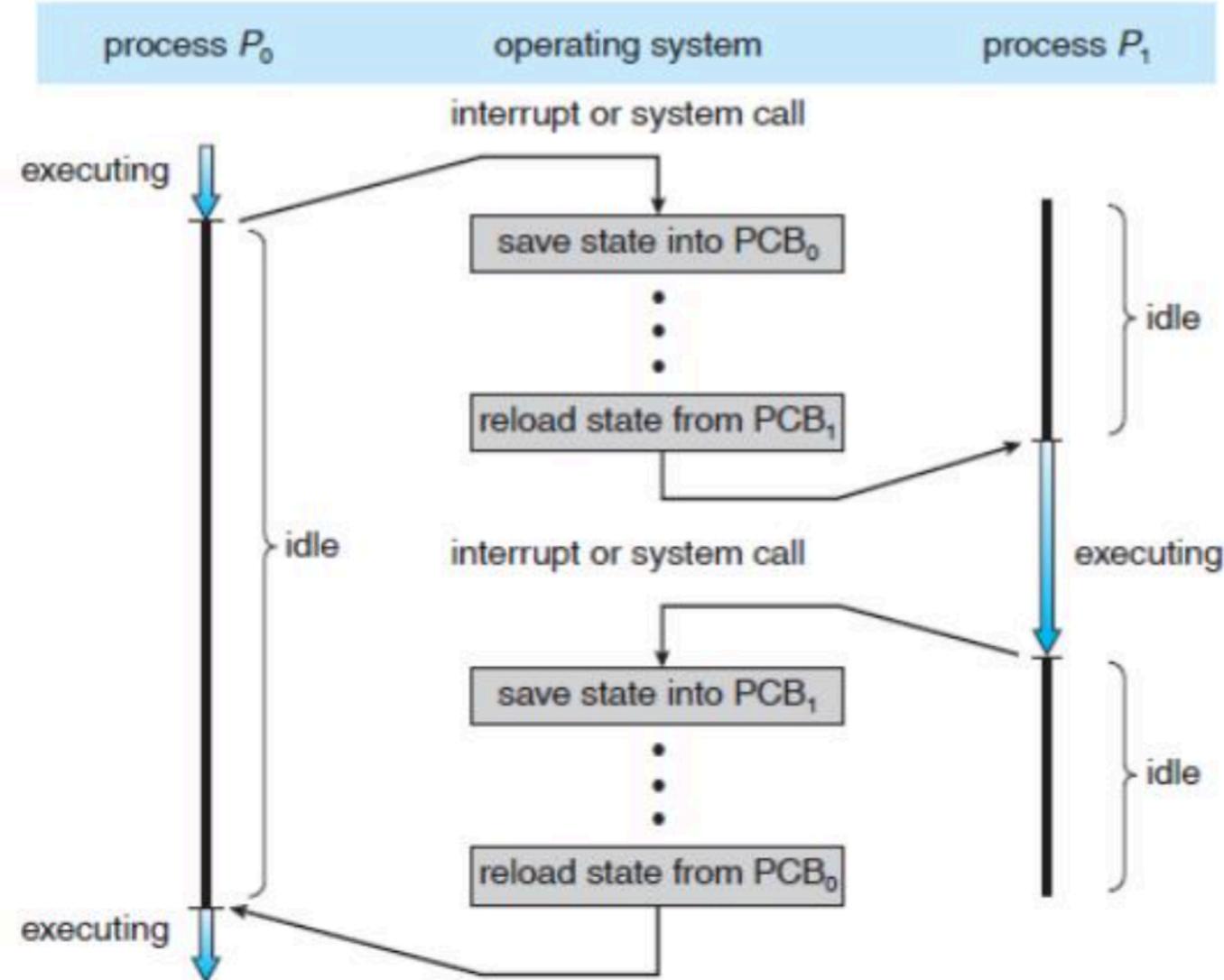
Compteur ordinal
Registres
État
Priorité
Espace d'adressage
Père
Fils
Fichiers ouverts
Actions associées aux signaux
....



PCB du processus 1

Compteur ordinal
Registres
État
Priorité
Espace d'adressage
Père
Fils
Fichiers ouverts
Actions associées aux signaux
....

# Commutation de contextes de processus





## La mémoire associée à un processus -Allocation de la mémoire

Au cours de son exécution, un processus alloue de la mémoire. Il existe trois types d'allocations :

- 1)allocation statique
- 2)allocation sur la pile (« stack allocation »)
- 3)allocation sur le tas (« heap allocation »)

# La mémoire associée à un processus -Allocation de la mémoire

## Exemple d'allocation statique

```
variable-statique.c ✘
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 void proc_1()
6 {
7     static int nb_appel_1 = 0;
8     nb_appel_1++;
9     printf("proc_1> Le nombre d'appel = %d\n",nb_appel_1);
10 }
11
12 void proc_2()
13 {
14     int nb_appel_2 = 0;
15     nb_appel_2++;
16     printf("proc_2> Le nombre d'appel = %d\n",nb_appel_2);
17 }
18
19
20 int main(int argc, char** argv)
21 {
22     proc_1();
23     proc_1();
24     proc_1();
25     proc_2();
26     proc_2();
27     proc_2();
28     return 0;
29 }
30
```

```
proc_1> Le nombre d'appel = 1
proc_1> Le nombre d'appel = 2
proc_1> Le nombre d'appel = 3
proc_2> Le nombre d'appel = 1
proc_2> Le nombre d'appel = 1
proc_2> Le nombre d'appel = 1
-----
(program exited with code: 0)
Press return to continue
```

### Allocation statique :

- Exemple : static int exemple=1 ;
- Espace mémoire nécessaire précisé dans le code avant l'exécution
- Espace réservé au moment de la compilation dans le fichier binaire résultant
- Espace accessible lors du chargement du binaire avant l'exécution
- Pas d'allocation lors de l'exécution



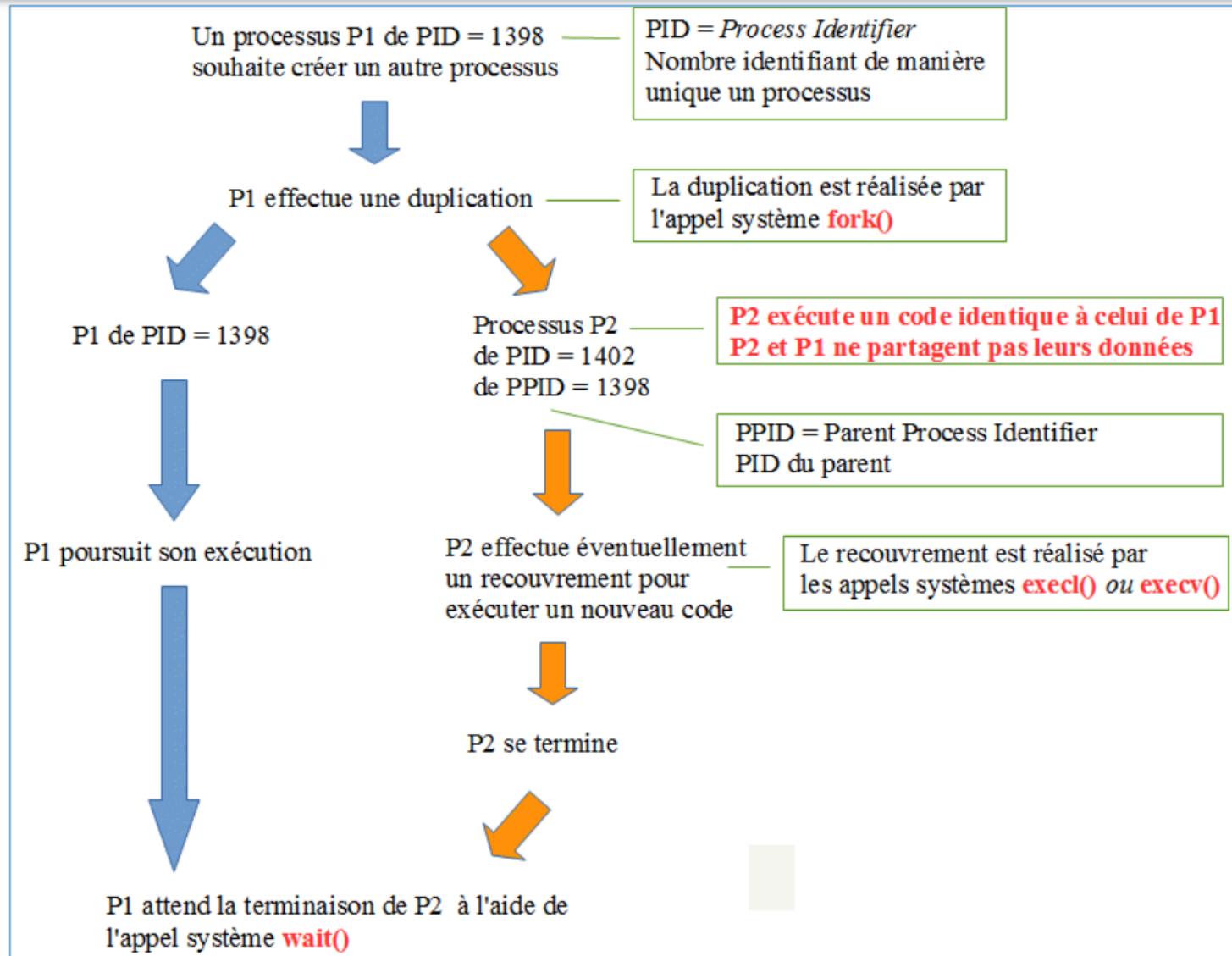
## La mémoire associée à un processus -Allocation de la mémoire

### Exemple d'allocation statique

Allocation dynamique de mémoire du processus

- Se fait pendant l'exécution du programme
- Espace non prévue dans le binaire du programme
- Demande d'allocation se fait durant l'exécution

# Cycle de vie d'un processus





## Syntaxe des appels systèmes utilisés

### Appel système fork()

```
#include <unistd.h>
pid_t fork(void);
```

Crée un nouveau processus par duplication du processus appelant. Le fils reçoit une copie de la zone mémoire du père.

#### Valeur de retour :

- Le PID du fils dans le père
- 0 dans le fils
- -1 si le fils n'a pas été créé.

# Syntaxe des appels systèmes utilisés

## Appel système fork()

### Exemple

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int retour = fork();
    if (retour==0){
        printf("Je suis le fils, mon PID est: %d, le PID de mon père est: %d\n",
               getpid(),getppid());
        return 0;
    }
    printf("Je suis le père, mon PID est: %d\n",getpid());
    sleep(3);
    return 0;
}
```

Pour utiliser les appels système Linux

Le processus fils démarre et commence à partir de la ligne qui suit **fork()**

Pour le processus fils, retour = 0

Pour le processus fils, retour = PID

Faire une pause de 3 secondes pour que le parent attend la fin de son fils d'abord

## Syntaxe des appels systèmes utilisés

### Appel système `wait()`

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Attend la fin d'un fils.

- **\*status** : entier court apportant une indication sur la façon dont le fils s'est terminé :
  - Le poids faible contient le numéro du signal reçu si le fils s'est terminé sur réception d'un signal.
  - Le poids fort contient la valeur de retour du processus fils s'il s'est terminé par exécution de l'instruction **return** ou de la fonction **exit()**.
- **Valeur de retour** :
  - PID du fils qui vient de se terminer
  - -1 en cas d'erreur

## Syntaxe des appels systèmes utilisés

### Exemples

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main(){
    int retour = fork();
    if (retour==0){
        printf("Je suis le fils, mon PID est: %d, le PID de mon père est: %d\n",
               getpid(),getppid());
        return 0;
    }
    int codeFinfils;
    int retourWait = wait(&codeFinfils);
    printf("Je suis le père, mon PID est: %d, mon fils est : %d\n",getpid(),
           retourWait);
    return 0;
}
```

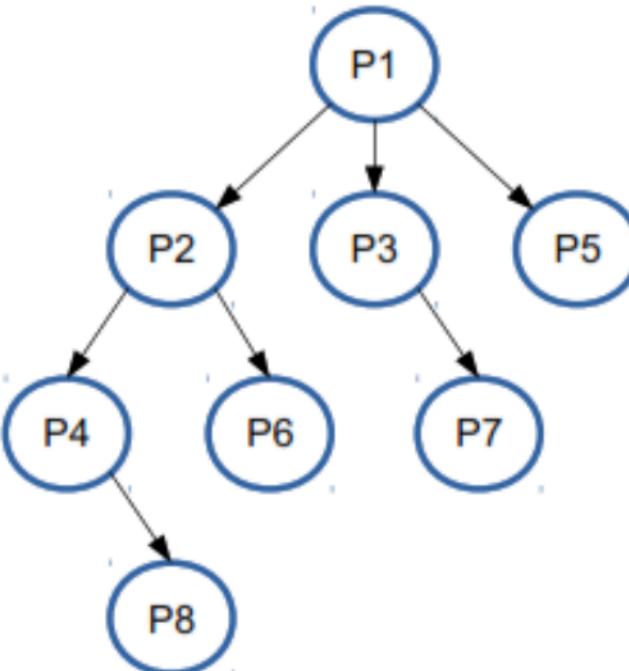


## Syntaxe des appels systèmes utilisés

### Exercice

Ecrire un programme en langage C qui permet de créer l'arborescence des processus suivantes.

Il faut noter que chaque processus parent ne doit se terminer avant ces fils



## Syntaxe des appels systèmes utilisés

### Duplication suivi d'un recouvrement

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Remplace le code du processus appelant par un nouveau code. Les fonctions `execl()` et `execv()` recherchent le nouveau programme à exécuter dans le répertoire courant, les fonctions `execlp()` et `execvp()` le recherchent dans le PATH.

- `const char *path` : nom d'un fichier exécutable avec son chemin
- `const char *file` : nom d'un fichier exécutable se trouvant dans la liste des chemins des fichiers exécutable (variable PATH)
- `const char *arg, ...` : liste variable d'arguments de la commande à exécuter dont le dernier est NULL
- `char *const argv[]` : tableau d'arguments de la commande à exécuter dont le dernier est NULL
- Valeur de retour : Ces fonctions retournent une valeur seulement si elles ont échouée et dans ce cas cette valeur est -1. En effet, lorsqu'elles réussissent, le code se trouvant en-dessous a été remplacé par un nouveau code.



## Syntaxe des appels systèmes utilisés

### Duplication suivi d'un recouvrement

Exemple avec l'appel système `execl()`

```
#include<stdio.h>
#include<unistd.h>
int main(int argc, char *argv[]){
    execl("./programme","Programme","SupTech","Casablanca","Maroc", NULL);
    return 0;
}
```

**Liste des arguments**

On appelle un autre programme à la place du processus en cours

```
#include<stdio.h>
int main(int argc, char *argv[]){
    printf("Argument1: %s\n", argv[0]);
    printf("Argument2: %s\n", argv[1]);
    printf("Argument3: %s\n", argv[2]);
    printf("Argument4: %s\n", argv[3]);
    return 0;
}
```

```
user@machine: ~/Bureau
user@machine:~/Bureau$ ./execl
Argument1: Programme
Argument2: SupTech
Argument3: Casablanca
Argument4: Maroc
user@machine:~/Bureau$ █
```

Équivalent à:

`user@machine:~/Bureau$ ./programme SupTech Casablanca Maroc`

## Syntaxe des appels systèmes utilisés

### Duplication suivi d'un recouvrement

Exemple avec l'appel système **execlp()**

Exécute un programme qui se situe dans la variable d'environnement PATH

```
#include<stdio.h>
#include<unistd.h>

int main(int argc, char *argv[]){
    execlp("ls","ls","-l","Rep1", NULL);
    return 0;
}
```

Même syntaxe que execl  
NULL signifie:  
fin d'arguments

```
user@machine:~/Bureau$ ./execlp
total 4
-rw-rw-r-- 1 user user    0 déc.   4 15:54 file1
drwxrwxr-x 3 user user 4096 déc.   4 15:56 Rep2
```



## Mort naturelle et Zombie

Un processus peut se terminer normalement ou anormalement.

- Dans le premier cas, l'application est abandonnée à la demande de l'utilisateur, ou la tâche à accomplir est finie.

- Dans le second cas, ***un dysfonctionnement est découvert***, qui est si sérieux qu'il ne permet pas au programme de continuer son travail

Un programme peut se finir de plusieurs manières. La plus simple est de revenir de la fonction main( ) en renvoyant un compte rendu d'exécution sous forme de valeur entière.

- Cette valeur est lue par le processus père, qui peut en tirer les conséquences adéquates.

- Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle, tandis que les cas d'échec sont indiqués par des codes de retour non nuls (et qui peuvent être documentés avec l'application).



## Orphelin et Zombie

### Processus orphelins

- si un processus père meurt avant son fils ce dernier devient orphelin.

### Processus zombie

- Si un fils se termine tout en disposant toujours d'un PID celui-ci devient un **processus zombie**
  - Le cas le plus fréquent : le processus s'est terminé mais son père n'a pas (encore) lu son code de retour.
- 
- Un processus est dit zombie s'il s'est achevé, mais qui dispose toujours d'un identifiant de processus (PID) et reste donc encore visible dans la table des processus. On parle aussi de processus défunt.

## Orphelin et Zombie

Au moment de la terminaison d'un processus, le système désalloue les ressources que possède encore celui-ci mais ne détruit pas son bloc de contrôle.

- Le système passe ensuite l'état du processus à la valeur TASK\_ZOMBIE (représenté généralement par un Z dans la colonne « statut » lors du listage des processus par la commande ps).
- Le signal SIGCHLD est alors envoyé au processus père du processus qui s'est terminé, afin de l'informer de ce changement.
  - Dès que le processus père a obtenu le code de fin du processus achevé au moyen des appels systèmes, le processus terminé est définitivement supprimé de la table des processus.



## Orphelin et Zombie

- Il est primordial dans les scripts de pouvoir déterminer si une commande a réussi à effectuer son travail correctement ou non.
- On imagine donc l'importance qui peut être portée à la lecture du code de retour d'un processus. Cette importance est telle qu'un processus qui se termine passe automatiquement par un état spécial, zombie, en attendant que le processus père ait lu son code de retour.
- Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l'état zombie.
- Voici un exemple, dans lequel le processus fils attend deux secondes avant de se terminer. Tandis que le processus père affiche régulièrement l'état de son fils en invoquant la commande ps.



## Orphelin et Zombie

### Processus orphelins

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main (){
pid_t child_pid;child_pid = fork ();
if (child_pid > 0) {
sleep (60);
}
else {
exit (0);
}
return 0;
}
```

Lancer cette commande pour voir les processus zombies:  
**ps axo stat,ppid,pid,comm | grep -w defunct**



## Exercices

### Exercice1

Répondre par OUI ou NON en justifiant vos réponses.

1) Un processus est une entité produite après compilation

2) Un processus est une entité produite après chargement d'un binaire en mémoire



## Exercices

### Correction de l'exercice 1

1. Un processus est une entité produite après compilation
  - **Non, car un processus est une image d'un programme en exécution**
  
2. Un processus est une entité produite après chargement d'un binaire en mémoire
  - **Oui, car une fois terminer le chargement d'un programme en mémoire un processus est créé**

## Exercices

### Exercice2

Préciser le nombre de processus créer par les programmes ci-dessous :

Code 1	Code 2	Code 3
<pre>int main() {     fork();     fork();     fork(); }</pre>	<pre>int main() {     if (fork() &gt; 0)         fork(); }</pre>	<pre>int main() {     int cpt=0;     while (cpt &lt; 3) {         if (fork() &gt; 0)             cpt++;         else             cpt=3;     } }</pre>



## Exercices

### Correction de l'exercice 2 –Code1

8 processus sont créés :

L'exécution du programme crée un processus P1.

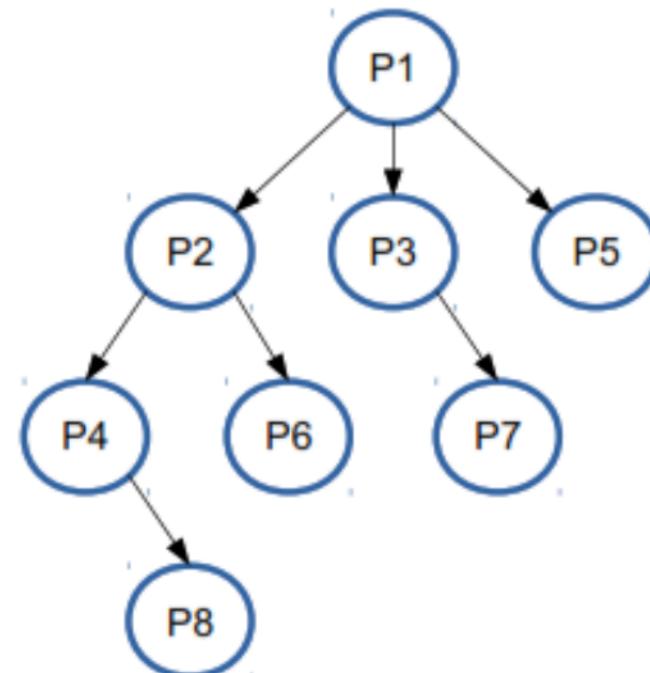
– A la lecture de la première instruction fork(), P1 se duplique et crée alors P2.

Les deux processus continuent l'exécution à partir de la ligne incluse ;

– A la lecture de la seconde instruction fork(), P1 se duplique et crée P3 tandis que P2 crée P4.

– Les quatre processus continuent l'exécution à partir de la ligne incluse ;

– A la lecture de la troisième instruction fork(), P1 se duplique et crée P5, P2 crée P6, P3 crée P7 et P4 crée P8



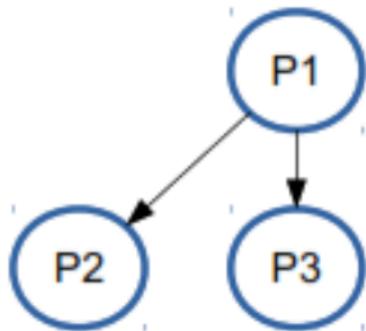


## Exercices

### Correction de l'exercice 2 –Code2

3 processus sont créés :

- L'exécution du programme crée un processus P1.
- A la lecture de la première instruction fork(),  
P1 se duplique et crée alors P2. P1 est le processus parent, P2 le processus enfant.
- Les deux processus continuent l'exécution à partir de la ligne incluse ; Le résultat de l'appel précédent est supérieur à 0 pour P1. Ce dernier rentre donc dans la suite d'instructions conditionnée et exécute l'instruction fork().
- P1 se duplique et crée donc P3.
- En revanche, le résultat de l'appel précédent est égal à 0 pour P2, qui ne rentre donc pas dans la suite d'instructions conditionnée.

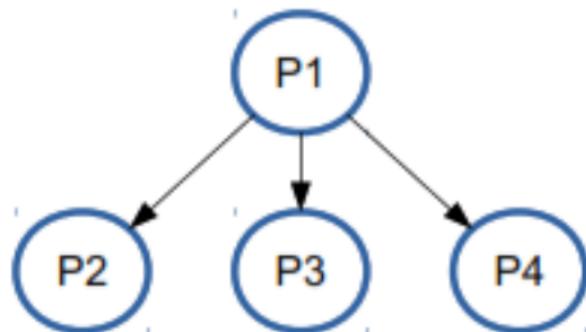


## Exercices

### Correction de l'exercice 2 –Code3

4 processus sont créés :

- L'exécution du programme crée un processus P1, qui initialise la variable cpt à 0.
- P1 rentre dans la boucle while() et se duplique lors de l'exécution de fork(). Il crée alors P2.
- Le résultat de l'appel précédent est supérieur à 0 pour P1. Ce dernier rentre donc dans la suite d'instructions conditionné par "if" et incrémente son compteur cpt qui passe à 1.
- En revanche, le résultat de l'appel précédent est égal à 0 pour P2, qui rentre donc dans la suite d'instructions conditionnée par else et affecte cpt à 3. Dès lors P2 sort de la boucle et n'exécutera plus d'instruction.
- Seul P1 ré-exécute la séquence d'instruction de la boucle while(), et le même schéma ce reproduit : à chaque entrée dans la boucle, P1 se duplique, tandis que le processus dupliqué n'exécute aucune instruction.
- P1 aura ainsi exécuté 3 fois l'instruction fork() jusqu'à ce que sa variable cpt atteigne 3.
- Il aura donc engendré P2, P3 et P4



## Exercices

### Exercice3

Soit le code suivant :

1- Lancer le programme ci-dessous avec les arguments 10 et 20. Que constatez-vous ?

Donnez une explication.

2- Lancer le programme ci-dessous avec les arguments 10 et 0. Que constatez-vous ? Donnez une explication.

```
int main(int argc, char * argv[]) {
    pid_t pid;
    int attente_fils,attente_pere;

    if(argc != 3)
    {
        printf("usage: ex1 'attente_pere' 'attente_fils'\n");
        exit(-1);
    }

    attente_pere = atoi(argv[1]);
    attente_fils = atoi(argv[2]);

    switch(pid=fork()) {
        case -1:
            perror("fork error");
            break;
        case 0:
            sleep(attente_fils);
            printf("fils attente finie\n");
            break;
        default:
            sleep(attente_pere);
            printf("pere attente finie\n");
            break;
    }
    return 0;
}
```



## Exercices

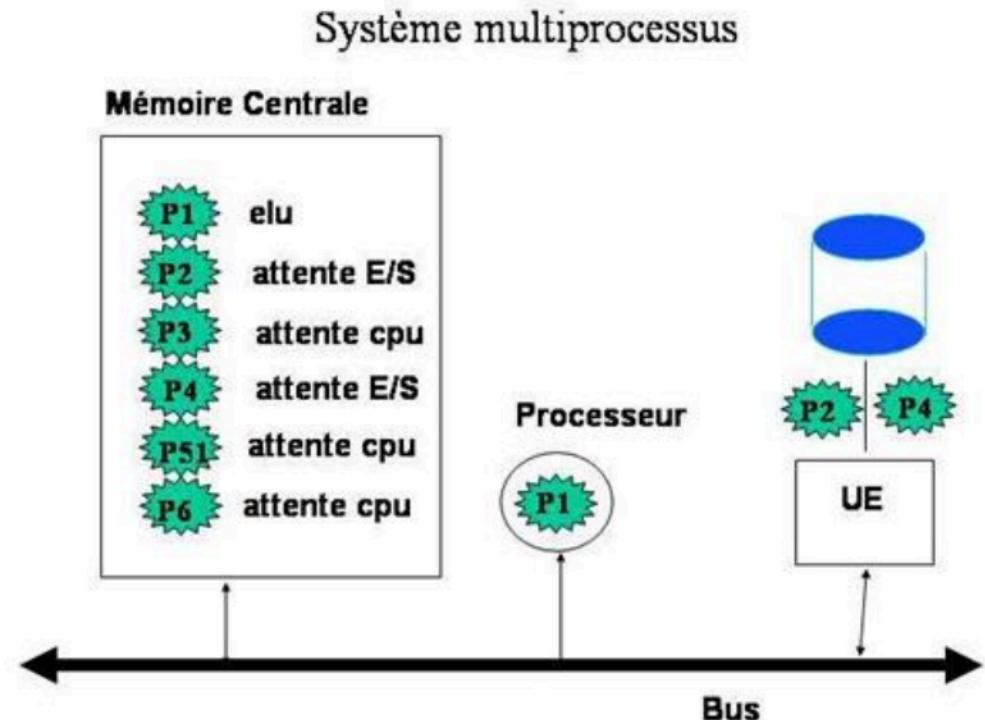
### Exercice3

- 1- Le père meurt avant son fils, le fils devient orphelin
  
- 2- Le fils meurt avant son père, le père est en sommeil, il ne lit pas le code de retour de son fils. Le fils devient zombie.

## L'ordonnancement des processus

### Qu'est-ce que l'ordonnancement de processus ?

La fonction d'ordonnancement gère le partage du processeur entre les différents processus en attente pour s'exécuter, c'est-à-dire entre les différents processus qui sont dans l'état prêt. L'opération d'élection consiste à allouer le processeur à un processus.



## L'ordonnancement des processus

### Ordonnancement préemptif ou non préemptif

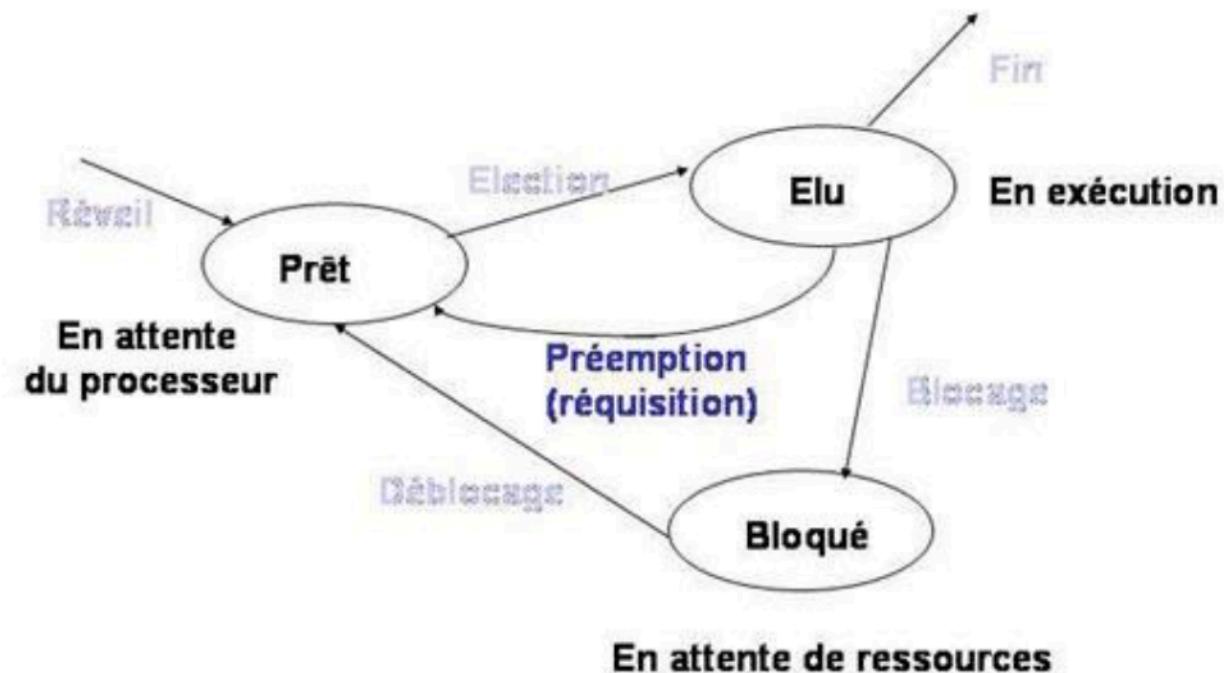
Selon si l'opération de réquisition du processeur est autorisée ou non, l'ordonnancement sera qualifié d'ordonnancement préemptif ou non préemptif :

- si l'ordonnancement est non préemptif, la transition de l'état élu vers l'état prêt est interdite : un processus quitte le processeur si il a terminé son exécution ou si il se bloque.
- si l'ordonnancement est préemptif, la transition de l'état élu vers l'état prêt est autorisée : un processus quitte le processeur si il a terminé son exécution , si il se bloque ou si le processeur est réquisitionné.

# L'ordonnancement des processus

## Ordonnancement préemptif ou non préemptif

Système multiprocessus  
Etats des processus



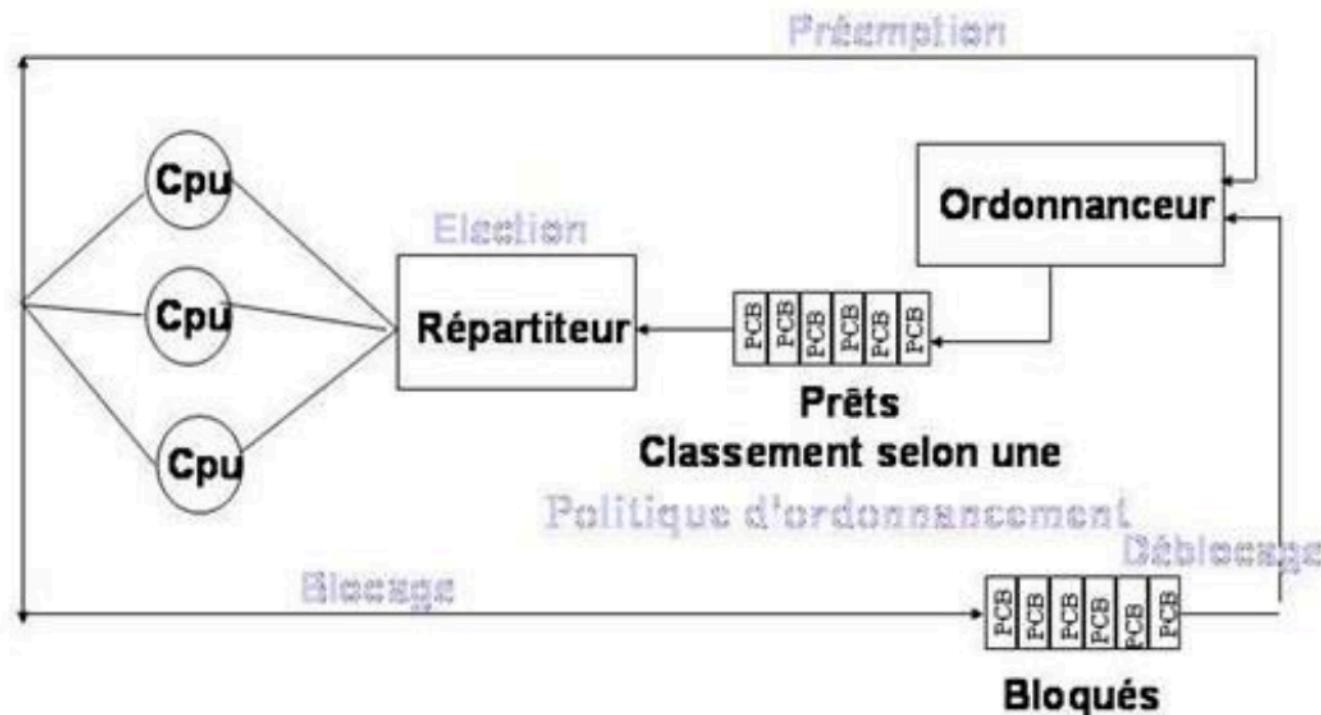


## L'ordonnancement des processus

### L'ordonnanceur

L'ordonnanceur est un programme système dont le rôle est d'allouer le processeur à un processus prêt.

Système multiprocessus  
Ordonnanceur et répartiteur





## L'ordonnancement des processus

### Ordonnancement : quels objectifs ?

L'ordonnancement consiste à choisir le processus à exécuter à un instant ‘t’ et à déterminer le temps durant lequel le processeur lui sera alloué

L'objectif de l'ordonnanceur est d'optimiser certains aspects des performances du système.

#### Compromis entre :

Temps de traitement moyen du système

Utilisation efficace du processeur

Temps de réponse moyen/max du système

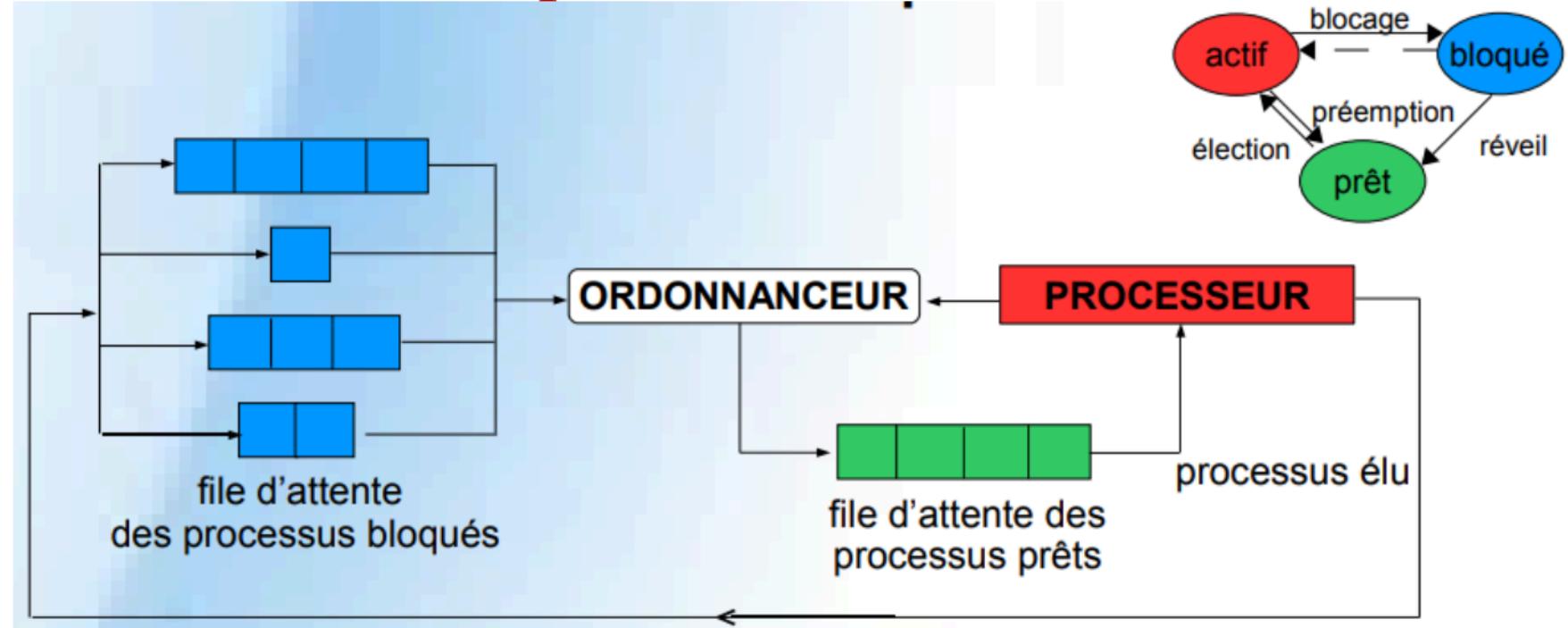
Satisfaction des conditions d'échéance pour les processus

Bonne utilisation des autres ressources du système



## L'ordonnancement des processus

### Ordonnancement des processus



**ORDONNANCEUR** : alloue le processeur aux différents processus selon un algorithme d'ordonnancement donné



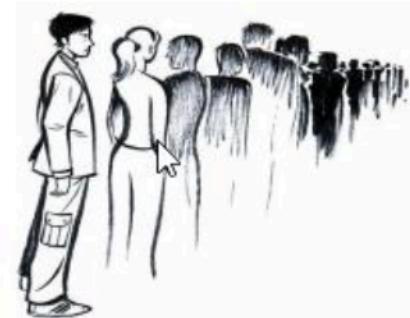
## L'ordonnancement des processus

### Ordonnancement non préemptif

Ordonnancement selon l'ordre d'arrivée :

- ✓ premier arrivé, premier servi

*(First Come First Serve (FCFS))*



Ordonnancement selon la durée de calcul :

- ✓ travail le plus court d'abord

*(Shortest Job First (SJF))*





## L'ordonnancement des processus

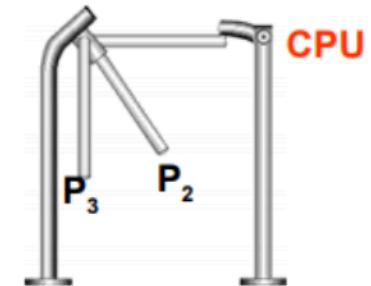
### Ordonnancement préemptif

Ordonnancement selon la durée de calcul restante :  
temps restant le plus court d'abord

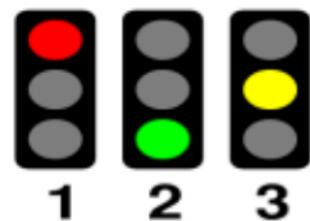
*Shortest Remaining Time (SRT)*



Ordonnancement sans notion de priorité :  
temps-partagé avec politique du tourniquet  
*Round-Robin (RR)*



Ordonnancement à priorités (statiques ou dynamiques) :  
*la tâche la plus prioritaire obtient le processeur*



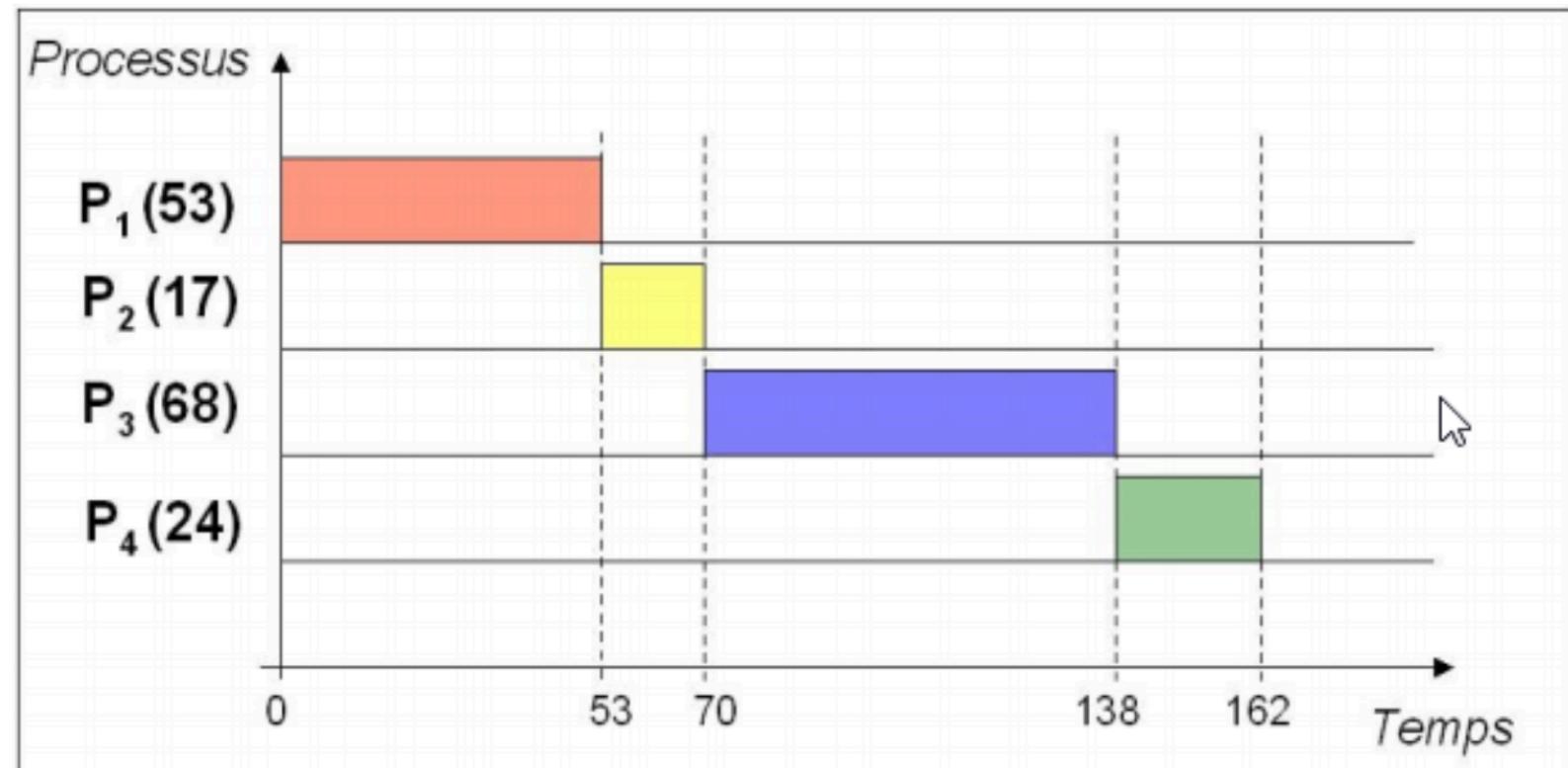


## L'ordonnancement des processus

### Ordonnancement First Come First Served (FCFS)

**Principe:** Les processus sont ordonnancés selon leur ordre d'arrivée

**Exemple:** ( $P_1, P_2, P_3$  et  $P_4$  arrivent dans cet ordre à  $t = 0$ ) :





## L'ordonnancement des processus

### Ordonnancement First Come First Served (FCFS)

#### Intérêts :

- Algorithme facile à comprendre
- Faible complexité d'implémentation (une seule liste chaînée)

#### Inconvénients :

- Pas de prise en compte de l'importance relative des processus
- Temps d'attente du processeur généralement important

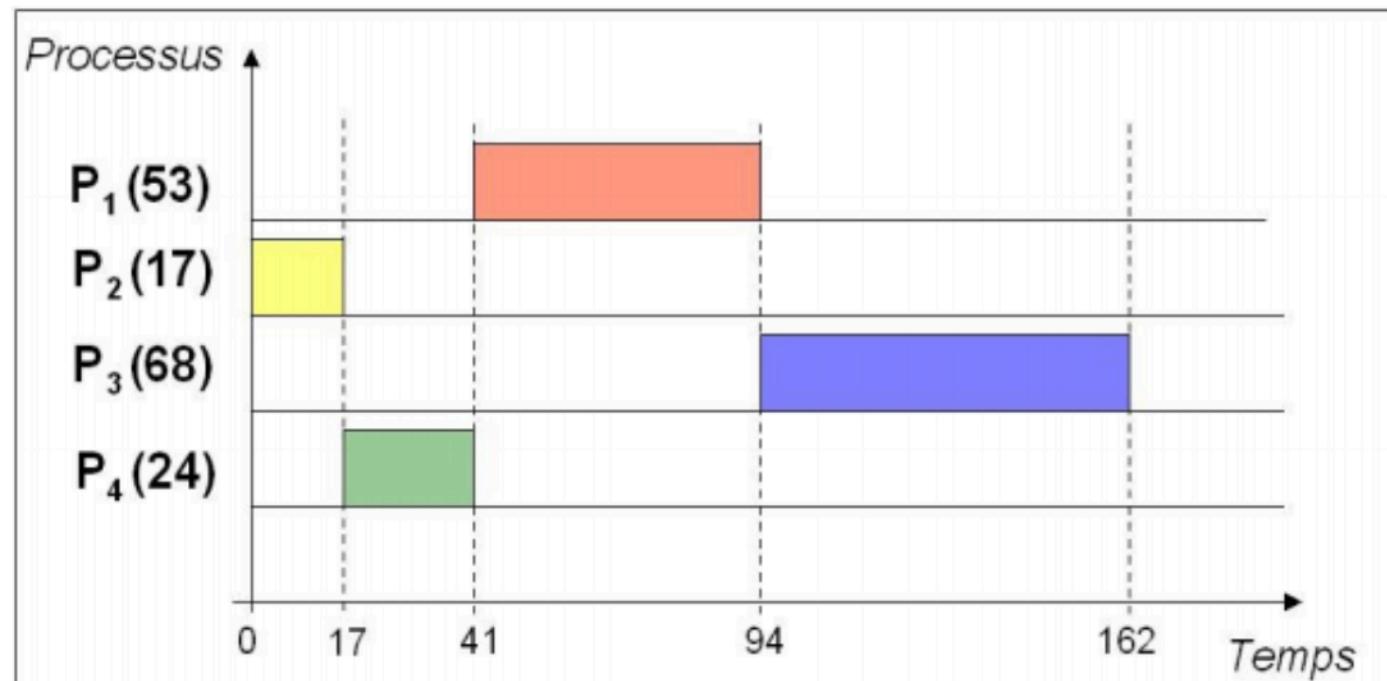


## L'ordonnancement des processus

### Ordonnancement Shortest Job First (SJF)

**Principe :** Le processus dont le temps d'exécution est le plus court est ordonné en priorité

**Exemple :** ( $P_1, P_2, P_3$  et  $P_4$  arrivent à  $t = 0$ ) :





## L'ordonnancement des processus

### Ordonnancement Shortest Job First (SJF)

#### Intérêts :

- SJF réduit le temps d'attente des processus
- Utilisation limitée à des environnements et à des applications spécifiques

#### Inconvénients :

- Pas de prise en compte de l'importance relative des processus
- Algorithme optimal uniquement dans le cas où tous les processus sont disponibles simultanément

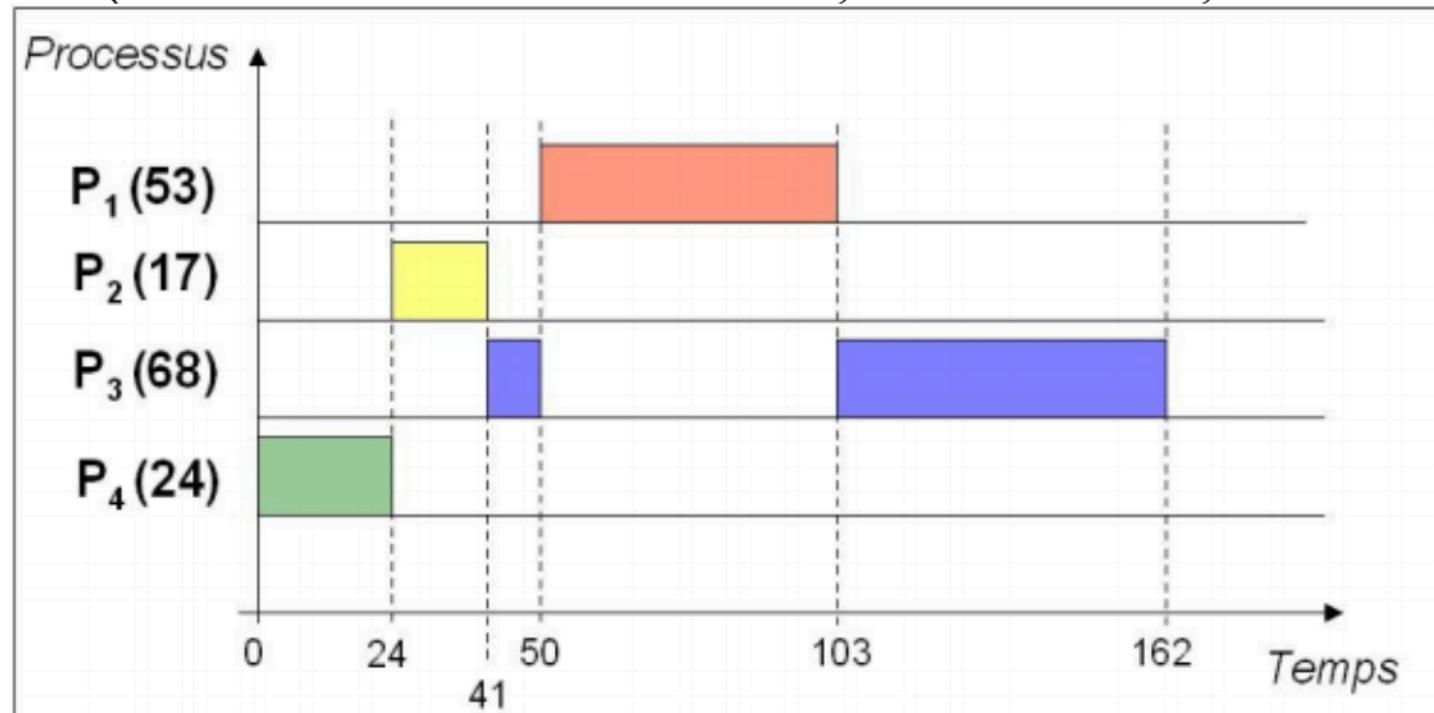


## L'ordonnancement des processus

### Ordonnancement Shortest Remaining Time (SRT)

**Principe :** Le processus dont le temps d'exécution restant est le plus court parmi ceux qui restent à exécuter, est ordonné en premier

**Exemple :** (P<sub>3</sub> et P<sub>4</sub> arrivent à t = 0; P<sub>2</sub> à t = 20; P<sub>1</sub> à t = 50)





## L'ordonnancement des processus

### Ordonnancement Shortest Remaining Time (SRT)

#### Intérêts :

- SRT minimise le temps d'attente moyen des processus les plus courts
- Utilisation limitée à des environnements et à des applications spécifiques

#### Inconvénients :

- Pas de prise en compte de l'importance relative des processus
- Non équité de service : SRT pénalise les processus longs
- Possibilité de famine pour les processus longs

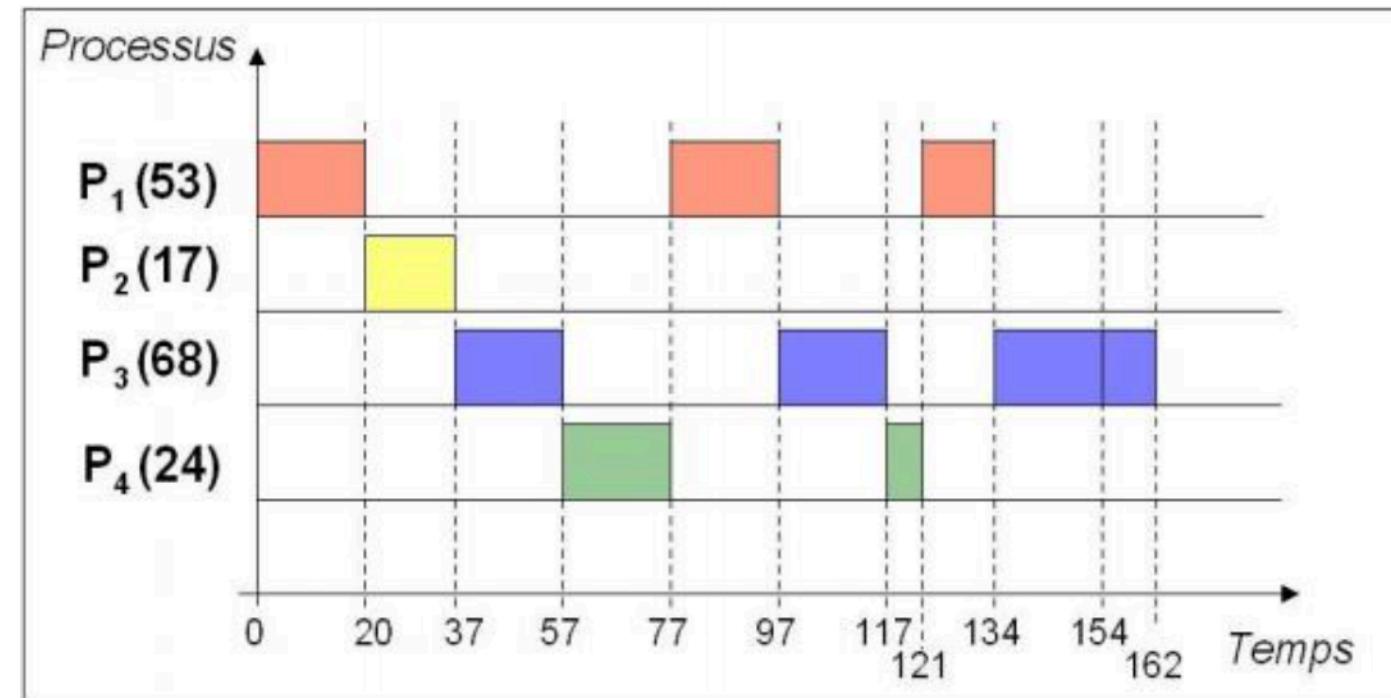


## L'ordonnancement des processus

### Ordonnancement temps-partagé (Round-Robin)

**Principe :** allocation du processeur par tranche (quantum) de temps

**Exemple :** ( $q=20$ ,  $n=4$ ) :





## L'ordonnancement des processus

### Ordonnancement Shortest Job First (SJF)

#### Intérêts :

- Equité de l'attribution du processeur entre toutes les tâches
- Mise en œuvre simple

#### Inconvénients :

- Pas de prise en compte de l'importance relative des tâches
- Difficulté du choix de la tranche de temps
  - ✓ Si  $q$  est trop grand, Round-Robin devient équivalent à FIFO
  - ✓ Si  $q$  est trop petit, il y a augmentation du nombre de changements de contexte !

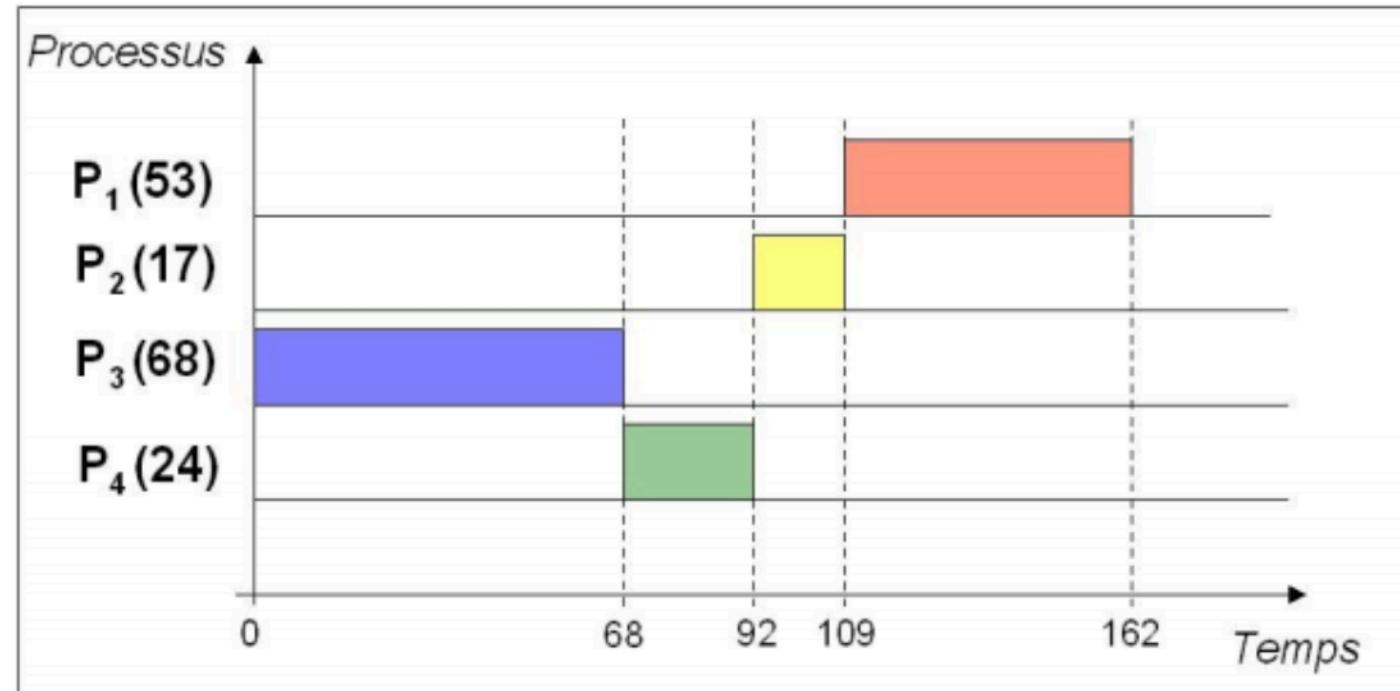


## L'ordonnancement des processus

### Ordonnancement à priorités statiques

**Principe :** allocation du processeur selon des priorités statiques (numéros affectés aux processus pour toute la vie de l'application)

**Exemple :** ( priorités( $P_1, P_2, P_3, P_4$ )=(3,2,0,1) ) :





## L'ordonnancement des processus

### Ordonnancement Shortest Job First (SJF)

#### Intérêts :

- Prise en compte de l'importance relative des processus
- Mise en œuvre simple

#### Inconvénients :

- Problèmes de blocages durant des périodes de temps illimitées
- Problèmes de famines des processus de moindres priorités



## La Synchronisation

### Ressource critique

Lorsque deux tâches (processus ou threads) accèdent à une même zone mémoire, elles peuvent interagir de manière indésirable

- même si chaque tâche, prise individuellement, se comporte correctement.



# La Synchronisation

## Concurrence sans problème

tâche 1 (compte  $\pm= 1000$ )      tâche 2 (compte  $-= 500$ )

charger (r1, compte)

charger (r2, 1000)

ajouter (r1, r2)

écrire (r1, compte)

*sauvegarde contexte*

*restauration contexte*

charger (r1, compte)

charger (r2, 500)

soustraire (r1, r2)

écrire (r1, compte)

ressources  
*compte*      *r1*      *r2*

2000

2000

2000

**3000**

3000

3000

3000

3000

**2000**      0

2000      **1000**

**3000**      1000

3000      1000

3000      1000

0      0

**3000**      0

3000      **500**

**2500**      500

2500      500



# La Synchronisation

## Concurrence problématique

tâche 1 (compte  $\pm= 1000$ )      tâche 2 (compte  $\pm= 500$ )

charger (r1, compte)  
 charger (r2, 1000)  
 ajouter (r1, r2)

*commutation de contexte →*

charger (r1, compte)  
 charger (r2, 500)  
 soustraire (r1, r2)  
 écrire (r1, compte)

*← commutation de contexte*

écrire (r1, compte)

ressources  
*compte      r1      r2*

2000	2000	0
2000	2000	1000
2000	3000	1000
	0	0
2000	2000	0
2000	2000	500
2000	1500	500
1500	1500	500
	3000	1000
3000	3000	1000



## La Synchronisation

### Définitions

- Une ressource est dite ressource critique lorsque des accès concurrents à cette ressources peuvent résulter dans un état incohérent.
- On parle aussi de situation de compétition (race condition) pour décrire une situation dont l'issue dépend de l'ordre dans lequel les opérations sont effectuées.
- Une section critique est une section de programme manipulant une ressource critique.



## La Synchronisation

### Exclusion mutuelle

- Une section de programme est dite atomique lorsqu'elle ne peut pas être interrompue par un autre processus manipulant les mêmes ressources critiques.
- → c'est donc une atomicité relative à la ressource
- Un mécanisme d'exclusion mutuelle sert à assurer l'atomicité des sections critiques relatives à une ressource critique.
- → en anglais : mutual exclusion, ou mutex

# La Synchronisation

## Mise en œuvre abstraite

```
int compte;                                /* commun */  
void entrer_SC_compte();  
void sortir_SC_compte();  
printf("ajout de 1000€\n");                  /* tâche 1 */  
entrer_SC_compte();  
compte += 1000;  
sortir_SC_compte();  
printf("ajout effectué\n");  
  
printf("retrait de 500€\n");                 /* tâche 2 */  
entrer_SC_compte();  
compte -= 500;  
sortir_SC_compte();  
printf("retrait effectué\n");
```



## La Synchronisation

### Mutex: Critères d'évaluation

- **Exclusion** : deux tâches ne doivent pas se trouver en même temps en SC
- **Progression** : une tâche doit pouvoir entrer en SC si aucune autre ne s'y trouve
- **Équité** : une tâche ne devrait pas attendre indéfiniment pour entrer SC
- L'exclusion mutuelle doit fonctionner dans un contexte multi-cœurs

## La Synchronisation

### Mauvaise solution : attente active

On utilise un booléen partagé comme « verrou »

```
int verrou = 0;  
void entrer_SC() {  
    while (verrou) {}  
    verrou = 1;  
}  
  
void sortir_SC() {  
    verrou = 0;  
}
```



## La Synchronisation

### Défauts de l'attente active

Le processus qui attend consomme du temps processeur (d'où le nom d'attente active).

→ NB: on peut améliorer cela en mettant un sleep ou un yield dans la boucle.

Cette approche n'assure pas l'exclusion!

→ verrou est lui même une ressource critique.



## La Synchronisation

### Bonne solution

On associe à la ressource un jeton, que les processus peuvent prendre et reposer

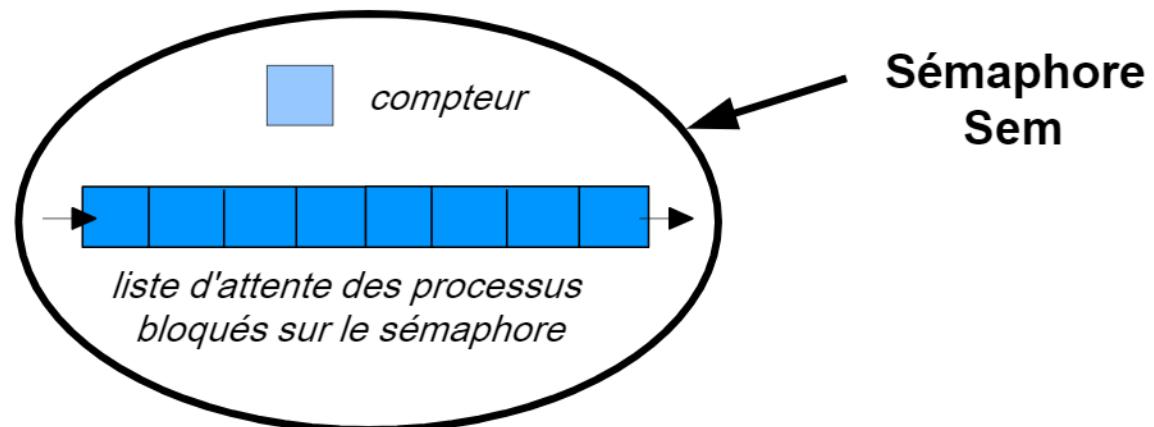
Seul le processus possédant le jeton devrait manipuler la ressource  
Donc, si un processus souhaite manipuler la ressource et que le jeton est pris, il doit d'abord attendre que le jeton redevienne disponible

Utilisation d'un sémaphore initialisé à 1

## La Synchronisation

### Synchronisation par sémaphore

- Les sémaphores sont utilisés dans le **contrôle d'accès à une ressource**
- Un sémaphore est une **structure de données**
  - contenant un compteur (valeur entière non négative)
  - gérant une file d'attente de processus attendant qu'advienne une condition particulière propre au sémaphore

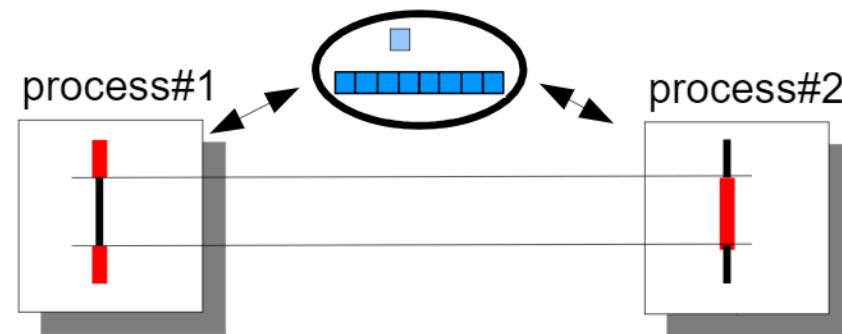




## La Synchronisation

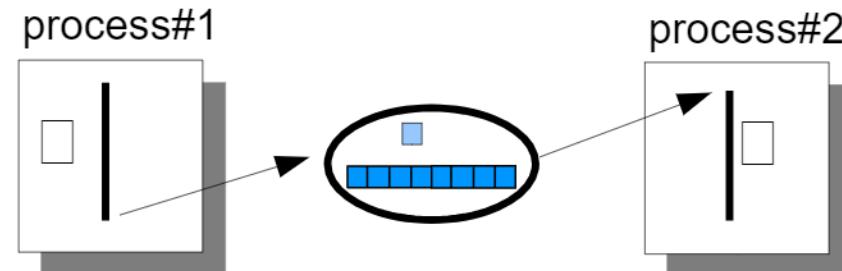
### Synchronisation par sémaphore

- Types de synchronisation possibles
  - Exclusion mutuelle



■ Section critique

- Barrière de synchronisation

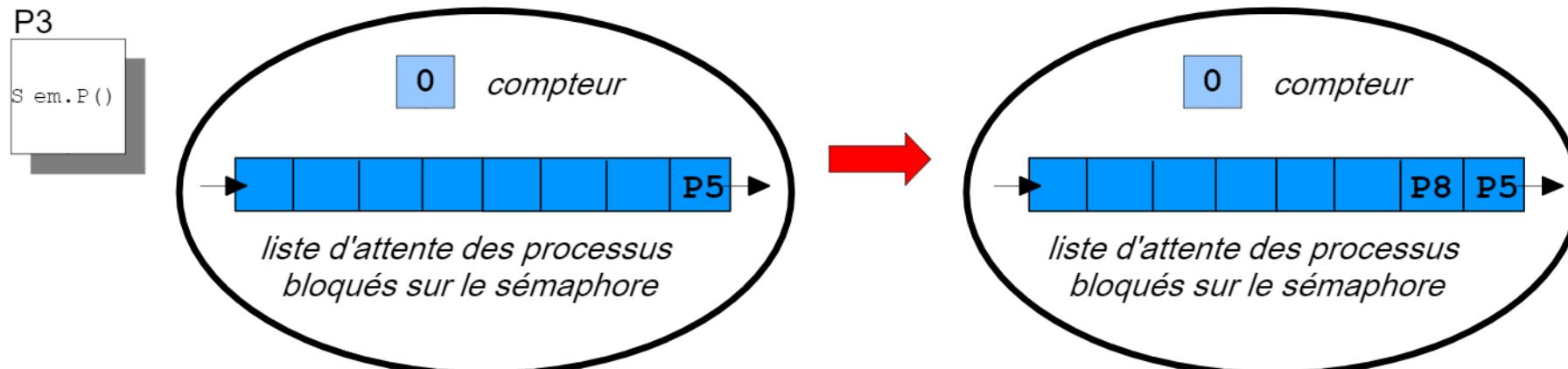


# La Synchronisation

## Synchronisation par sémaphore

- 1ère opération : **test de prise du sémaphore** (« Puis-je ?)

```
Sem.P() : si (Sem.compteur>0)
            alors Sem.compteur = Sem.compteur-1
            sinon insère_ce_processus(Sem.file)
finsi
```

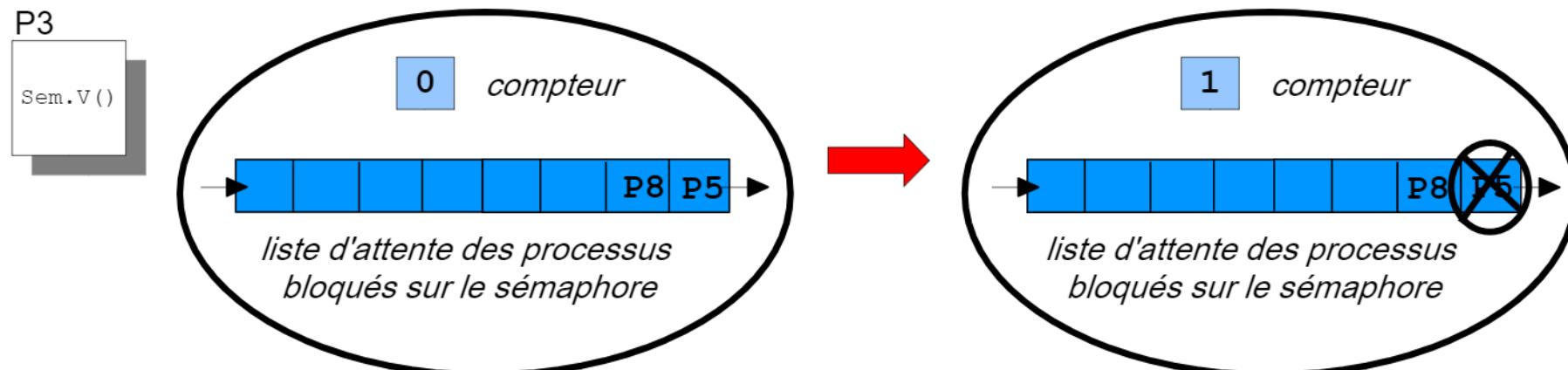


# La Synchronisation

## Synchronisation par séaphore

- 2ème opération : **libération** du séaphore (« Vas-y ! »)

```
Sem.V() : Sem.compteur = Sem.compteur+1  
           si (Sem.compteur > 0)  
           alors extrait_un_processus(Sem.file)  
finsi
```



## La Synchronisation

### Synchronisation par sémaphore

- Types de sémaphores :

- **Sémaphore binaire** : implémentation de l'exclusion mutuelle

- ↳ le compteur ne peut admettre que les valeurs 0 ou 1
  - le compteur doit être initialisé à 1



- **Sémaphore compteur** : implémentation de la barrière de synchronisation

- ↳ Le compteur doit être initialisé à 0



**Fin du chapitre 2**

# **Chapitre 3**

# **La gestion de la mémoire**

## Les concepts de base

- On appelle « **Mémoire** » tout composant électronique capable de stocker temporairement des données
- Caractéristiques principales d'une mémoire :
  - **la capacité**
  - **le temps d'accès**
  - **le temps de cycle**
  - **le débit**
  - **la non volatilité**
- Mémoire idéale (dans l'absolu) ?



## Les concepts de base

- 3 types de mémoires :

- ➡ **Mémoire morte** (appelée également *mémoire non volatile*)
  - ◆ mémoire ROM (Read-Only Memory)
  - ◆ mémoire ne s'effaçant pas en absence de courant électrique
  - ◆ mémoire conservant les données nécessaires au démarrage de l'ordinateur
  - ◆ temps d'accès de l'ordre de 150ns
- ➡ **Mémoire vive** (appelée également *mémoire volatile*)
  - ◆ mémoire RAM (Random Access Module)
  - ◆ données ne perdant pas en l'absence de courant électrique
  - ◆ 2 types de mémoire RAM : DRAM et SRAM
  - ◆ temps d'accès pour la DRAM de l'ordre de 50ns
  - ◆ temps d'accès pour la SRAM de l'ordre de 10ns
- ➡ **Mémoire flash**
  - ◆ compromis entre la mémoire RAM et la mémoire ROM
  - ◆ non volatilité de la mémoire morte
  - ◆ accès en lecture/écriture de la mémoire vive

## Les concepts de base

- 2 niveaux de gestion de la mémoire :

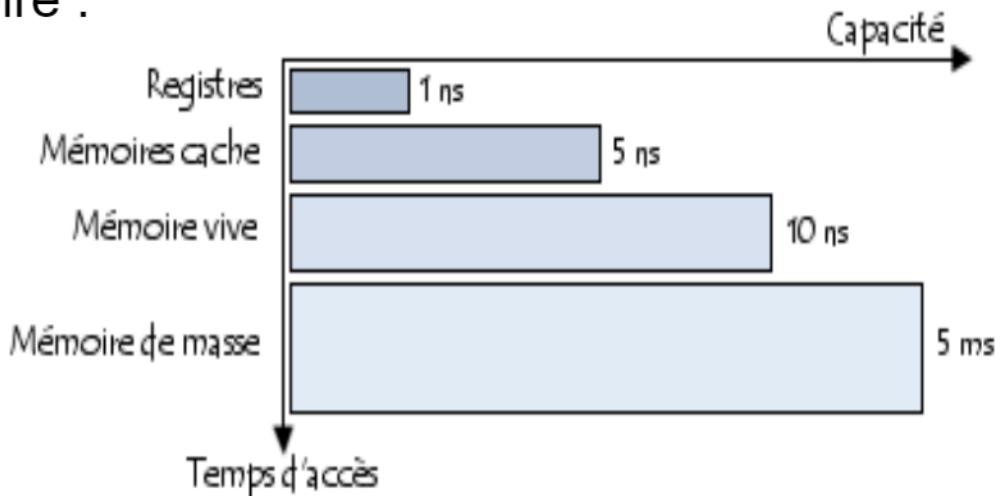
- **Niveau matériel**

- ♦ les registres du processeur
  - ♦ la mémoire cache

- **Niveau système d'exploitation**

- ♦ la **mémoire principale** (appelée également *mémoire centrale ou interne*)
  - ♦ la **mémoire secondaire** (appelée également *mémoire de masse ou physique*)

→ **Rôle du gestionnaire de mémoire du SE**



## Le gestionnaire de mémoire du SE

- Objectifs du gestionnaire de mémoire du système d'exploitation :

Partager la mémoire (système multi-tâche)

Allouer des blocs de mémoire aux différents processus

Protéger les espaces mémoire utilisés

Optimiser la quantité de mémoire disponible

Mécanisme  
de mémoire virtuelle

+

Mécanismes  
de découpage de la mémoire

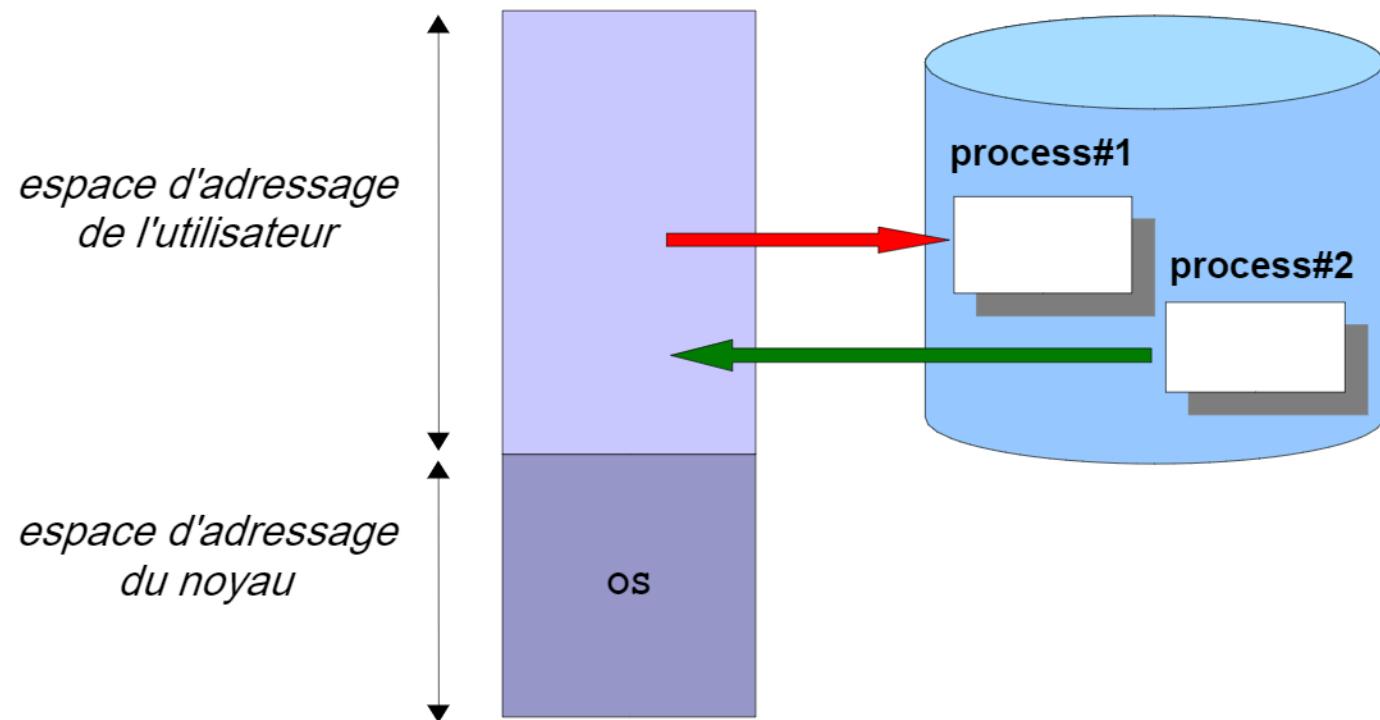


## La mémoire virtuelle

- **La mémoire virtuelle** est une technique permettant d'exécuter des programmes dont la taille excède la taille de la mémoire réelle
- **La mémoire virtuelle** permet :
  - d'augmenter le nombre de processus présents simultanément en mémoire centrale
  - de mettre en place des mécanismes de protection mémoire
  - de partager la mémoire entre processus
- Mécanisme mis au point dans les années 60

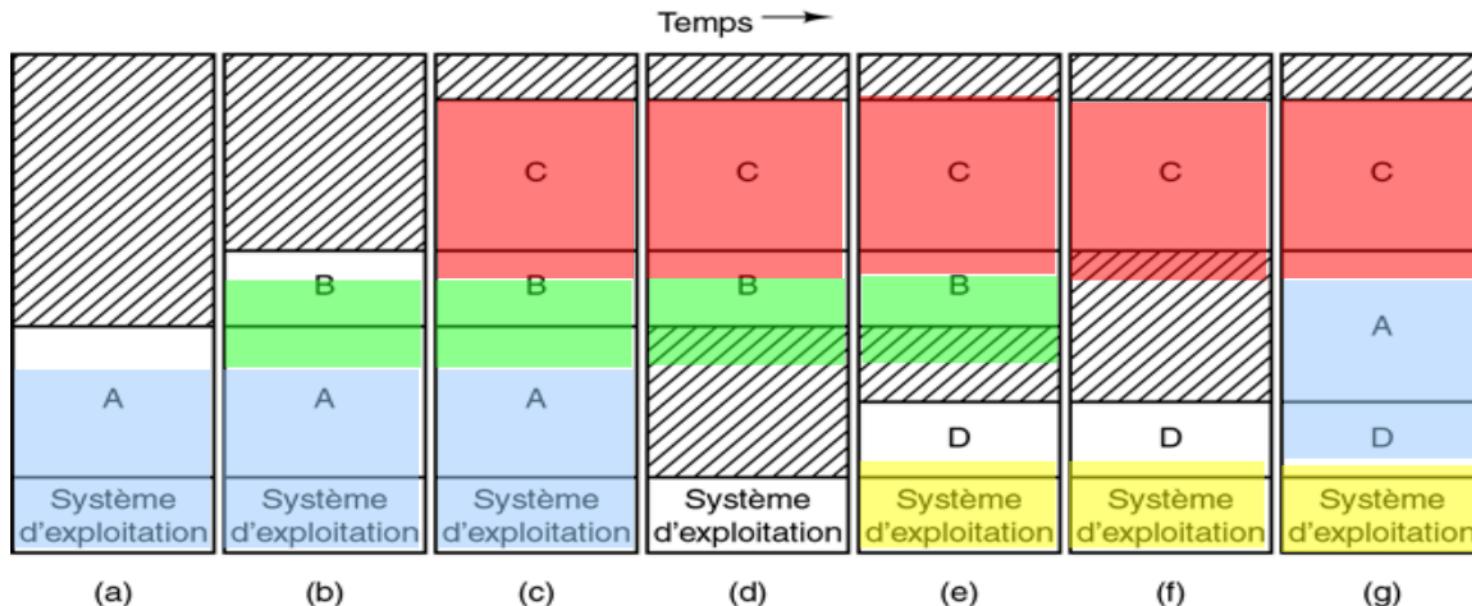
## La mémoire virtuelle

- Une partie de l'espace d'adressage d'un processus peut être enlevé temporairement de la mémoire centrale au profit d'un autre (**swapping**)



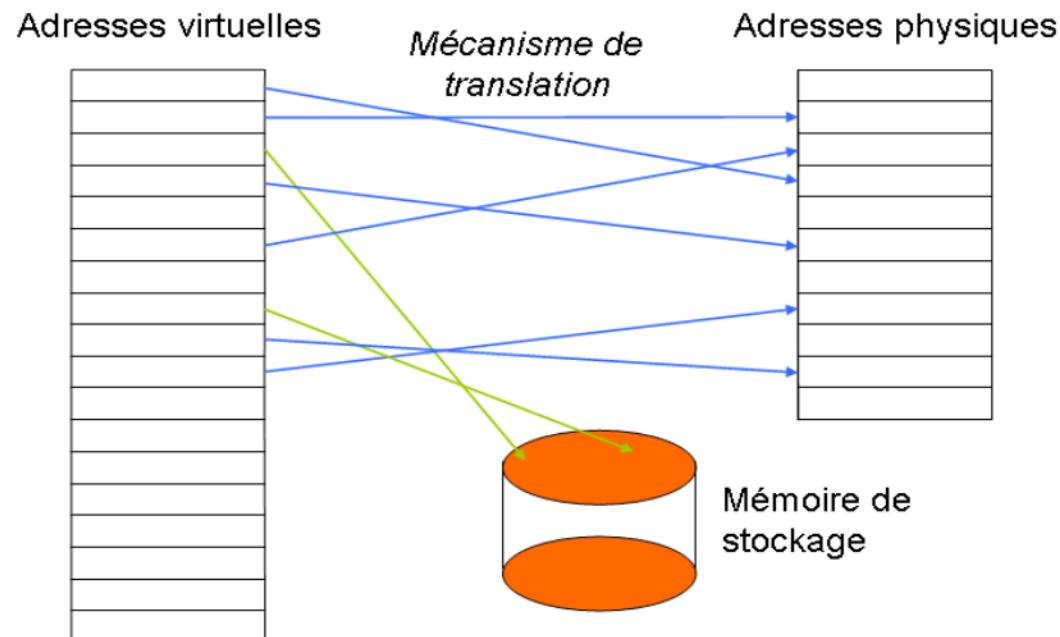
# La mémoire virtuelle

- Illustration du mécanisme de **va-et-vient** (swapping)



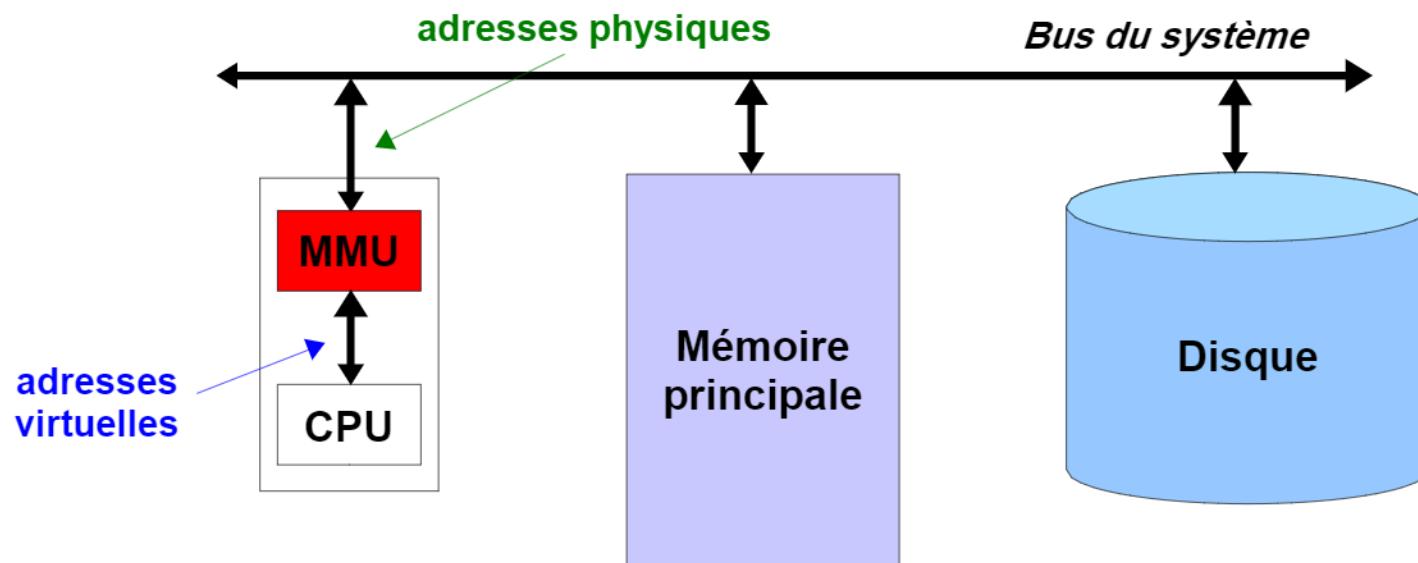
## La mémoire virtuelle

- Au lancement d'un processus, son espace d'adressage est majoritairement stocké en mémoire secondaire
- Au fur et à mesure de l'exécution du processus, des parties de son espace d'adressage sont chargées en mémoire principale



## La mémoire virtuelle

- Les **adresses virtuelles** doivent être traduites en **adresses physiques**
- Cette traduction est assurée par un circuit matériel spécifique pour la gestion de la mémoire :
  - la **MMU** (*Memory Management Unit*)



## Les mécanismes de découpage de la mémoire

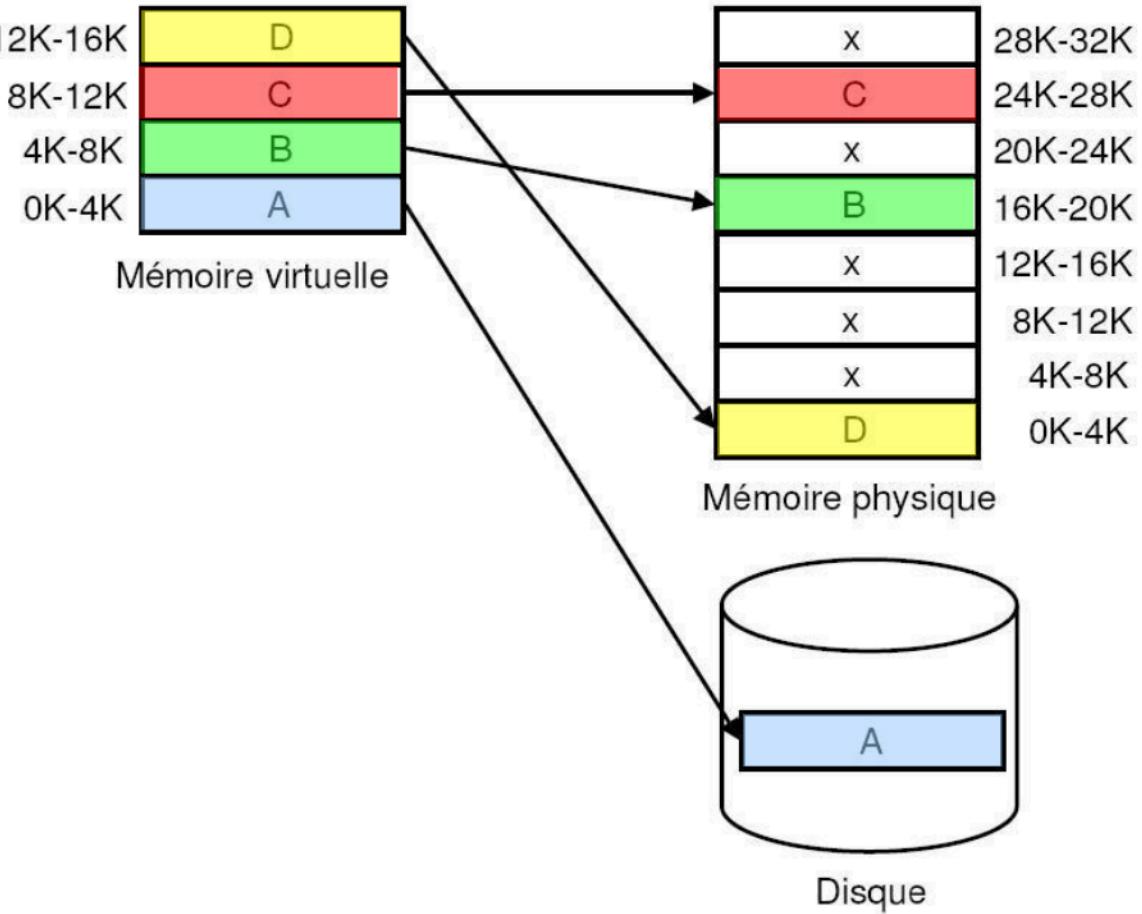
- La mémoire principale peut être découpée de 3 façons :
  - **par pagination**
  - elle consiste à diviser la mémoire en blocs, et les programmes en pages de **longueur fixe**.
  
- **par segmentation**
- les programmes sont découpés en parcelles ayant des **longueurs variables** appelées *segments*.
  
- **par segmentation paginée**
- certaines parties de la mémoire sont segmentées, d'autres paginées

## La pagination

- La mémoire virtuelle et la mémoire physique sont structurées en unités d'allocation
  - ➡ L'espace d'adressage **virtuel** est découpé en **pages**
  - ➡ L'espace d'adressage **physique** est découpé en **cadres**
- Taille d'une page = taille d'un cadre
- La taille d'une page est **fixe** (de 2Ko à 16Ko)
- Toutes les pages sont de la même taille
- Il peut y avoir plus de pages que de cadres (c'est là tout l'intérêt)

# La pagination

- Exemple :



# La pagination

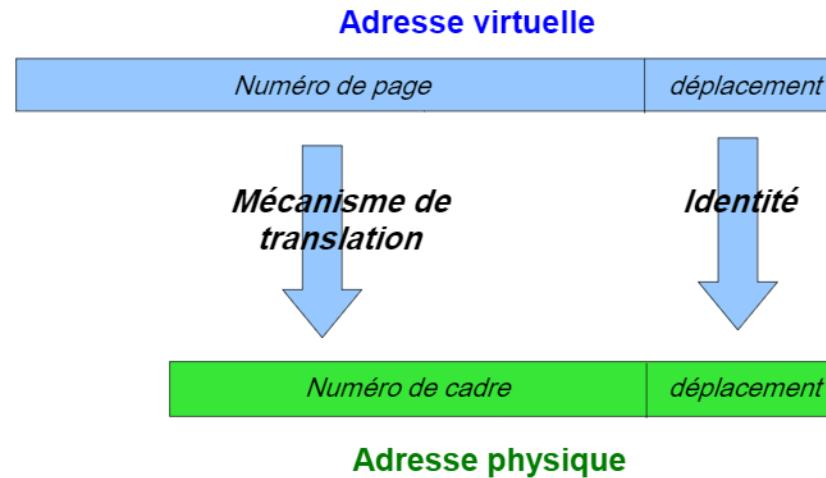
- Une **adresse virtuelle** est définie par :

- ➡ un numéro de page
  - ➡ un déplacement dans la page

- Une **adresse physique** est définie par :

- ➡ un numéro de cadre
  - ➡ un déplacement dans le cadre

- Un mécanisme de traduction assure la conversion des adresses virtuelles en adresses physiques, en consultant une **table des pages**, pour connaître le numéro du cadre qui contient la page recherchée



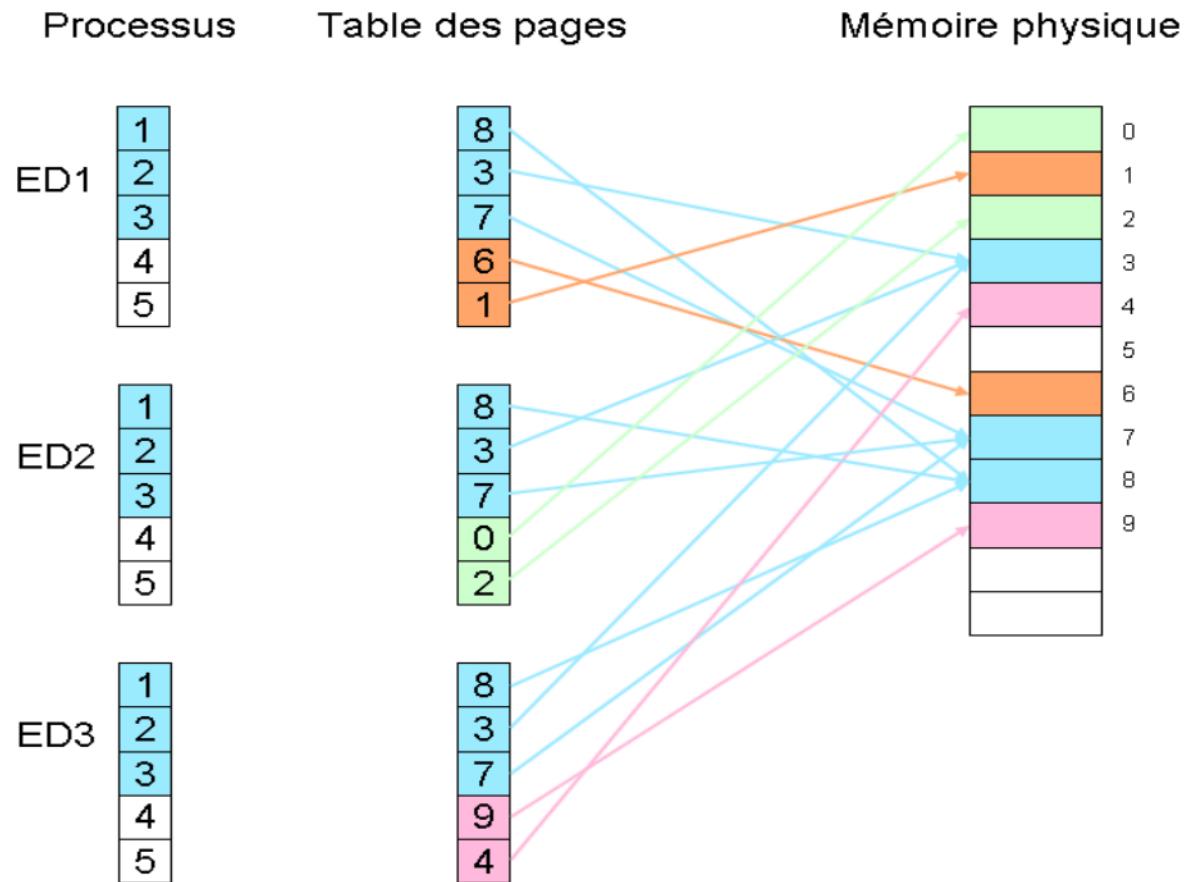
# La pagination

- La **table des pages virtuelles** (TPV) est composée de plusieurs champs :
  - le **numéro de cadre**
  - un bit de **présence**
  - un bit de **modification** (M)
  - un bit de **référence** (R)
  - un bit de **protection**



# La pagination

- Exemple de partage de code entre processus
- 3 pages de code  
- 2 pages de données*



# La pagination

- **Verrouillage** de pages

- Les processus privilégiés (root) peuvent spécifier des zones de leur espace virtuel non « swappables »

- **Applications**

- Processus temps réel
  - Applications multimédia
  - Traitement de données confidentielles

## La segmentation

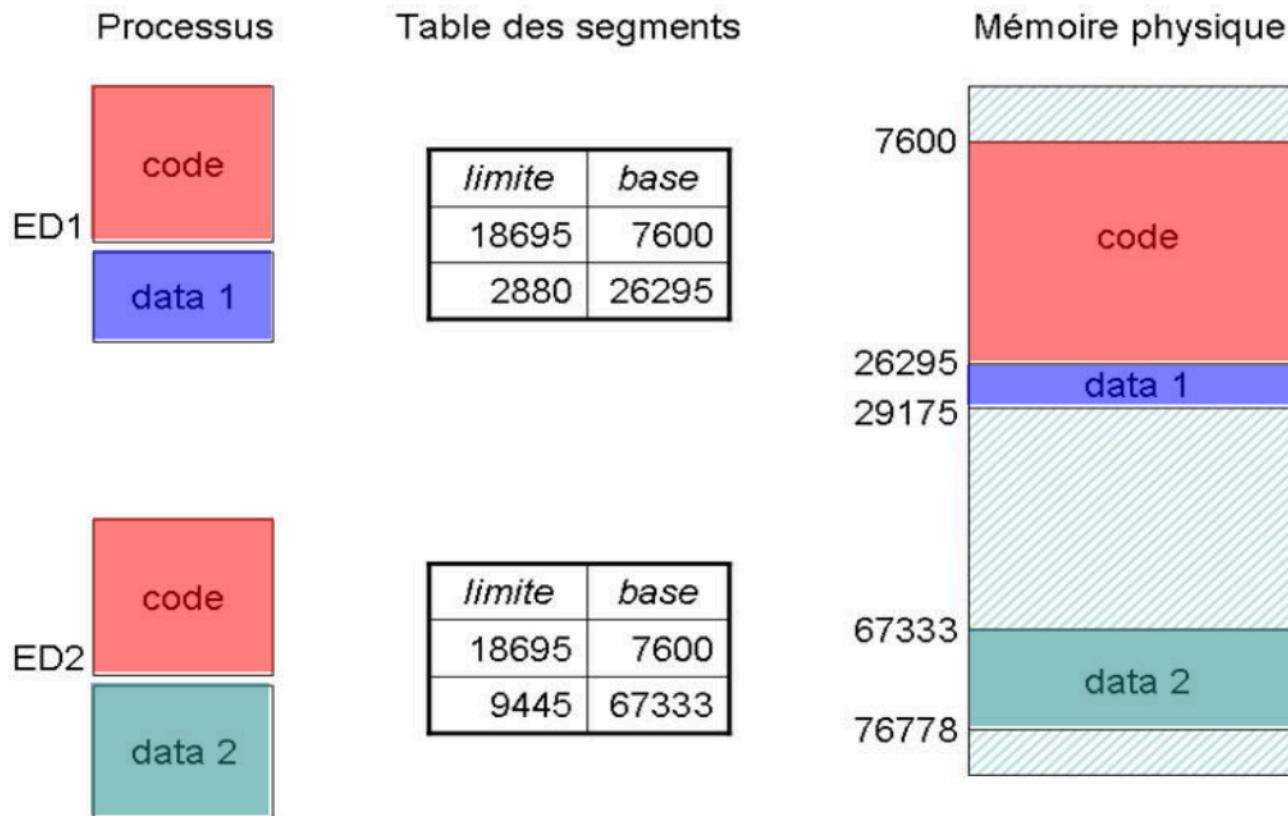
- Un **segment mémoire** est un espace d'adressage indépendant défini par 2 valeurs :
  - une **adresse** où il commence (aussi appelée *base* ou *adresse de base*)
  - une **taille** ou un décalage (aussi appelé *limite* ou *offset*)
- L'**adresse virtuelle** d'une donnée est donc exprimée sous la forme :
  - $\text{adr\_virtuelle} = \text{base} : \text{limite}$
- Cette adresse virtuelle est traduite en adresse physique par le biais d'une **table des segments**

## La segmentation

- Il existe différents **types de segments** :
  - les segments de **données statiques**
  - les segments de **données globales**
  - les segments de **code**
  - les segments d'**états de tâche**

# La segmentation

- Exemple de partage de segments entre processus :



## Pagination vs. segmentation

- **Pagination :**

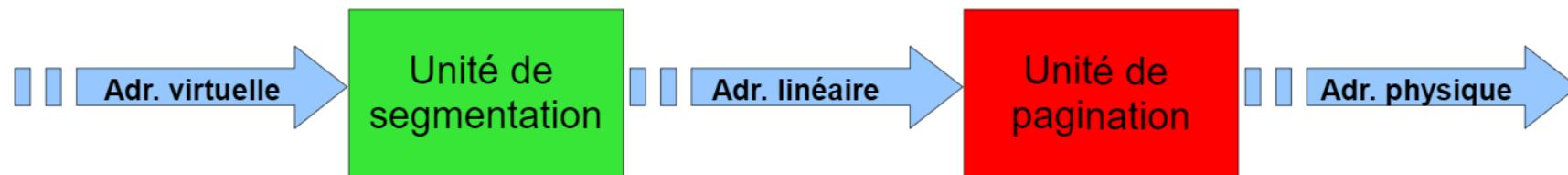
- sert à obtenir un grand espace d'adressage linéaire sans avoir à acheter de la mémoire physique

- **Segmentation :**

- permet la séparation des programmes et des données dans des espaces d'adressage logiquement indépendants
  - facilite le partage et la protection

## La segmentation paginée

- Chaque segment est composé d'un ensemble de pages
- Une adresse virtuelle est définie par :
  - un numéro de segment
  - un numéro de page
  - un déplacement dans la page
- La traduction des adresses virtuelles en adresses physiques est réalisée grâce à une **table des segments** et une **table des pages**



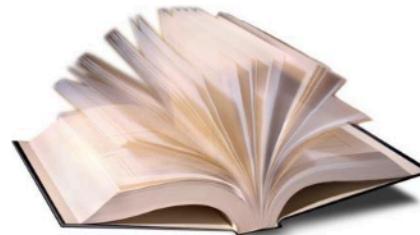


## Problèmes communément rencontrés

- La fragmentation mémoire



- Les défauts de pages

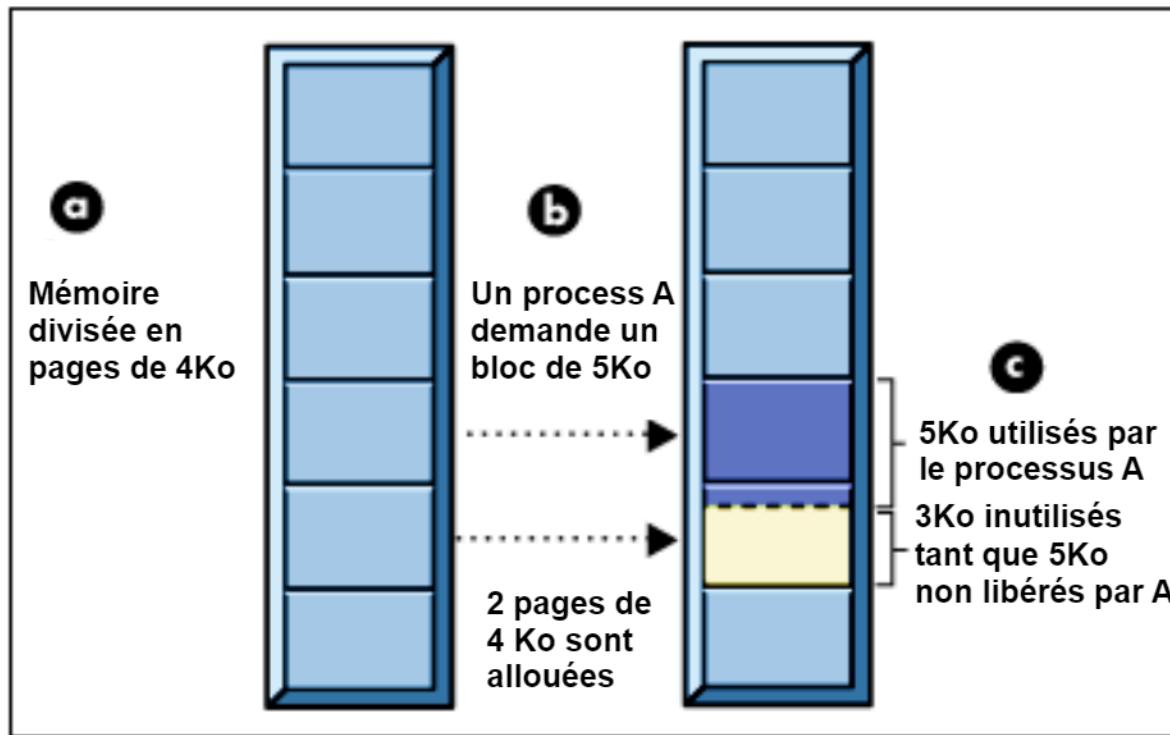


## La fragmentation

- Une **mémoire fragmentée** est une mémoire dans laquelle plusieurs blocs de mémoire non contigus sont libres
- La **fragmentation interne** dans les **systèmes paginés**
  - entre chaque partition de taille fixe, un peu de mémoire est perdue
- La **fragmentation externe** dans les **systèmes segmentés**
  - des espaces entre les segments existent suite au retrait de programmes
- Il existe des **méthodes de compaction** mais généralement **coûteuses**

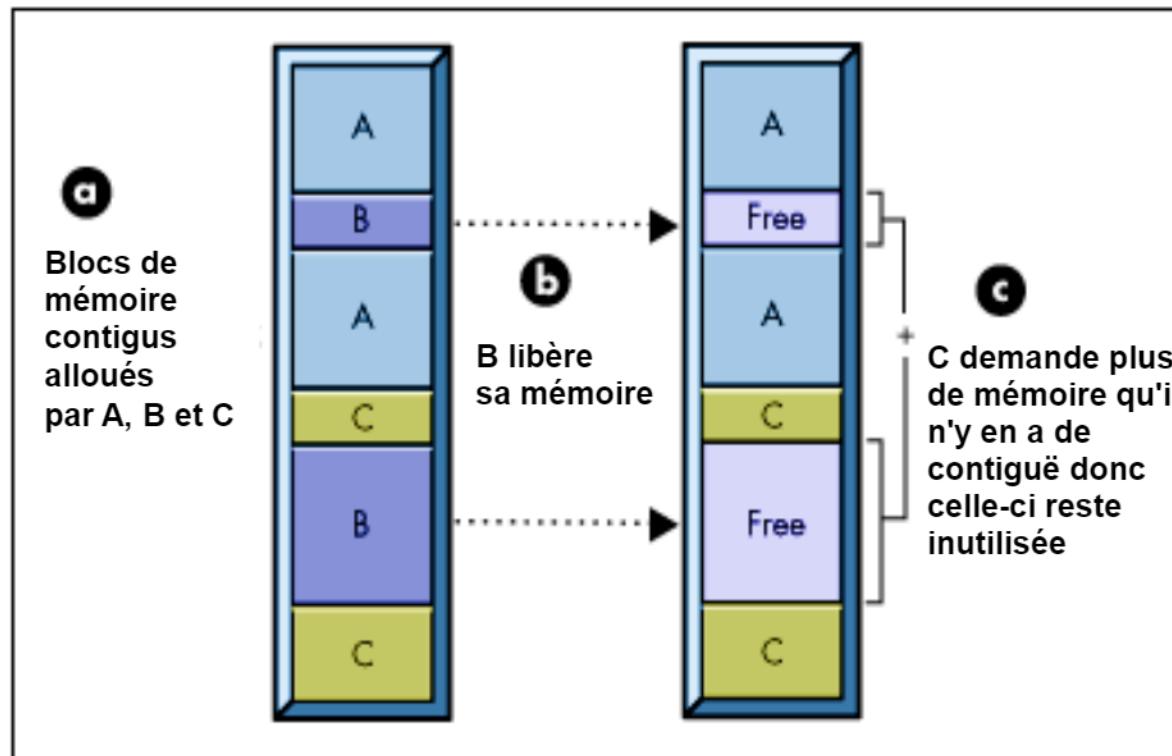
# La fragmentation

- Illustration de la **fragmentation interne** :



# La fragmentation

- Illustration de la **fragmentation externe** :

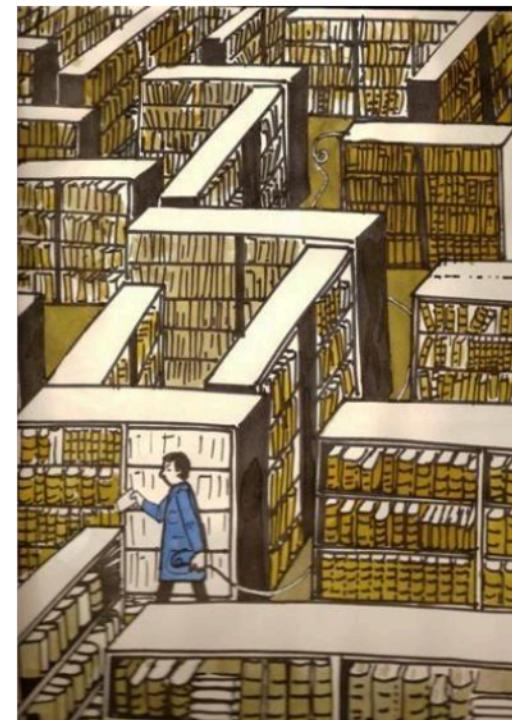


## Les défauts de pages

- Si une page virtuelle n'est pas présente en mémoire physique, il se produit un **défaut de page**
- Le système d'exploitation doit alors recharger la page manquante depuis le disque en plaçant éventuellement une autre page physique en mémoire secondaire
- Si la mémoire physique est pleine :
  - il faut retirer de la mémoire physique une page (**remplacement**)
    - ♦ choisir une page « victime »
    - ♦ si elle a été modifiée, la réécrire sur le disque
    - ♦ modifier les indicateurs de présence dans la TPV
- Puis, dans tous les cas :
  - charger la page référencée en mémoire physique (**placement**)
  - modifier les indicateurs de présence dans la TPV

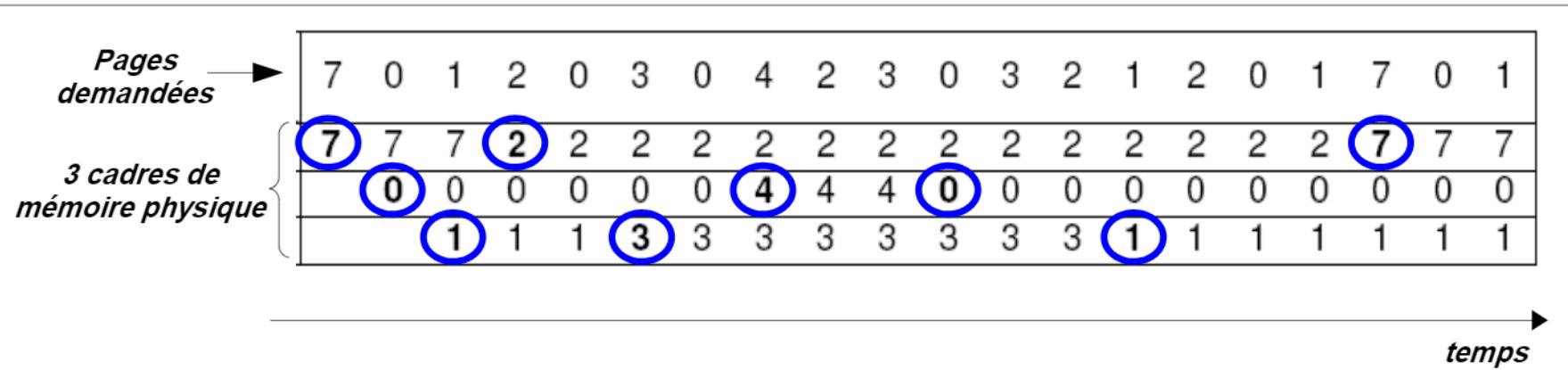
## Les algorithmes de remplacement de page

- Objectif : choisir la page à retirer de manière à **minimiser** le nombre de défauts de page
- **Algorithmes existants :**
  - algorithme de remplacement aléatoire
  - algorithme de Belady (optimal)
  - algorithme FIFO
  - algorithme de l'Horloge
  - algorithme LRU
  - algorithme NRU



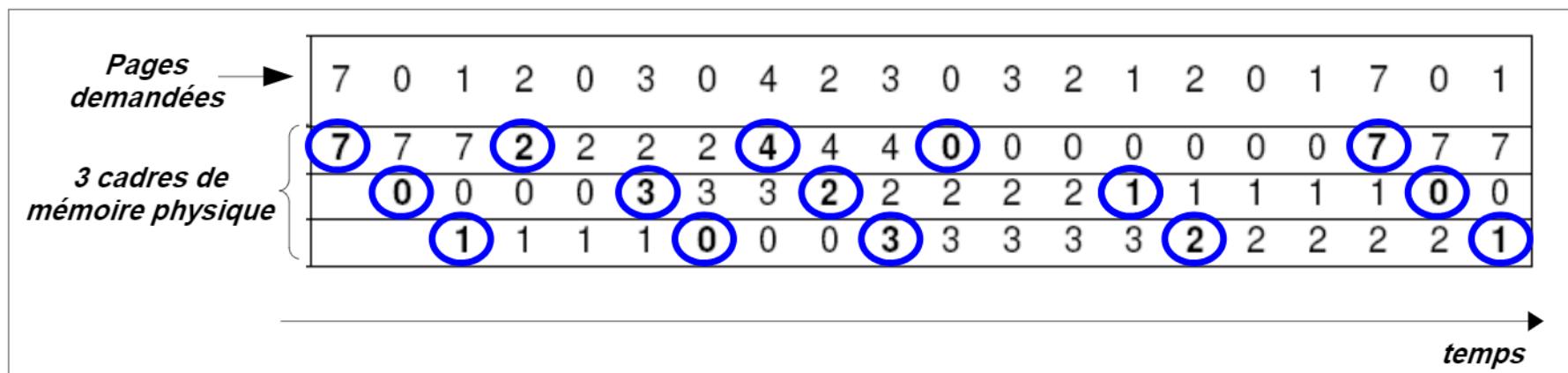
## L'algorithme de Belady

- **Principe** : choisir la page qui sera référencée le plus tard possible dans le futur
- Stratégie théorique et impossible à mettre en oeuvre dans la réalité
- Algorithme **optimal**
- Algorithme servant de **base de référence** pour les autres stratégies



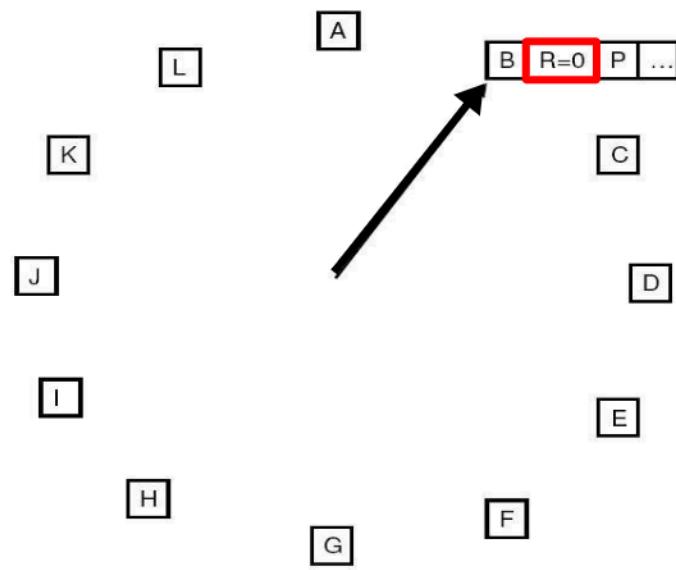
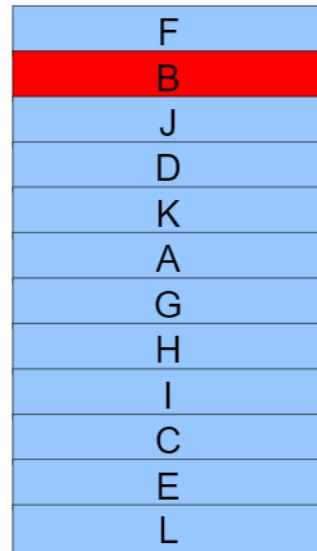
## L'algorithme FIFO

- **Principe** : choisir la page la plus ancienne en mémoire
- Stratégie qui ne tient pas compte de l'utilisation de chaque page
- Algorithme **rarement utilisé** car il génère beaucoup de défauts de page



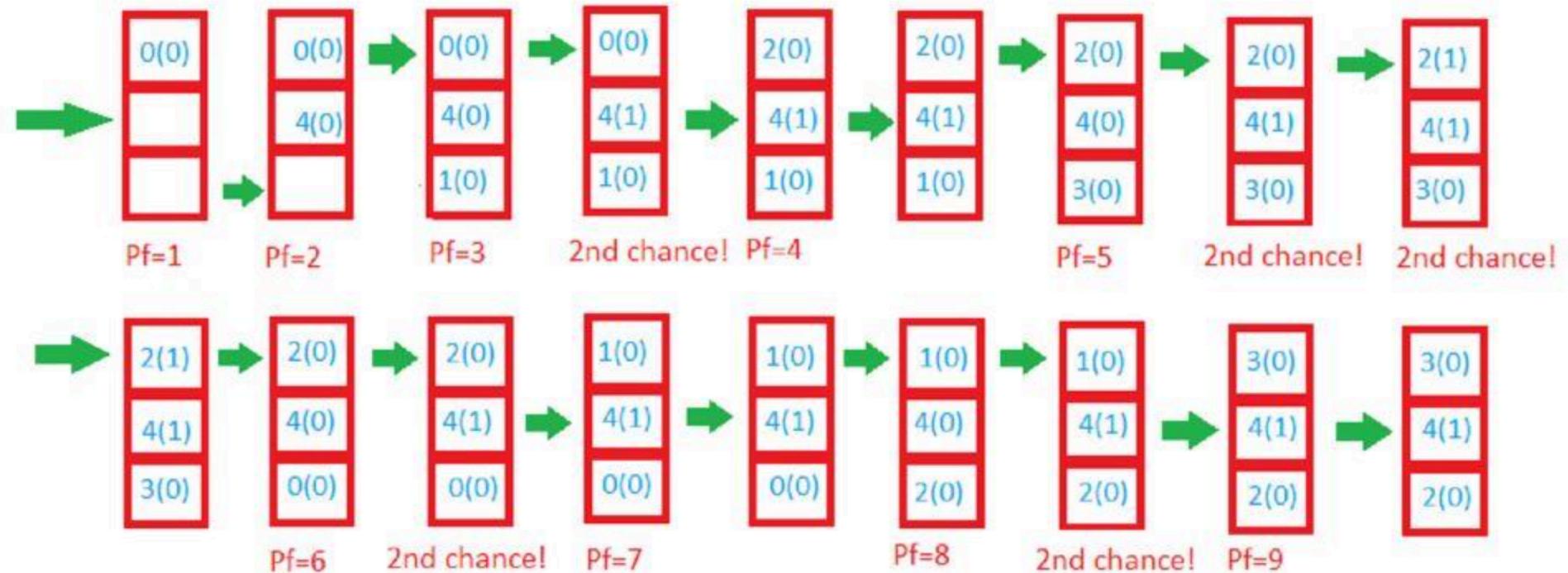
# L'algorithme d'Horloge

- **Principe :** algorithme FIFO + examen du bit de référence
  - bit de référence à 0 : choix de la page
  - bit de référence à 1 : on donne une seconde chance à la page
- Les pages en mémoire sont mémorisées dans une liste circulaire en forme d'horloge



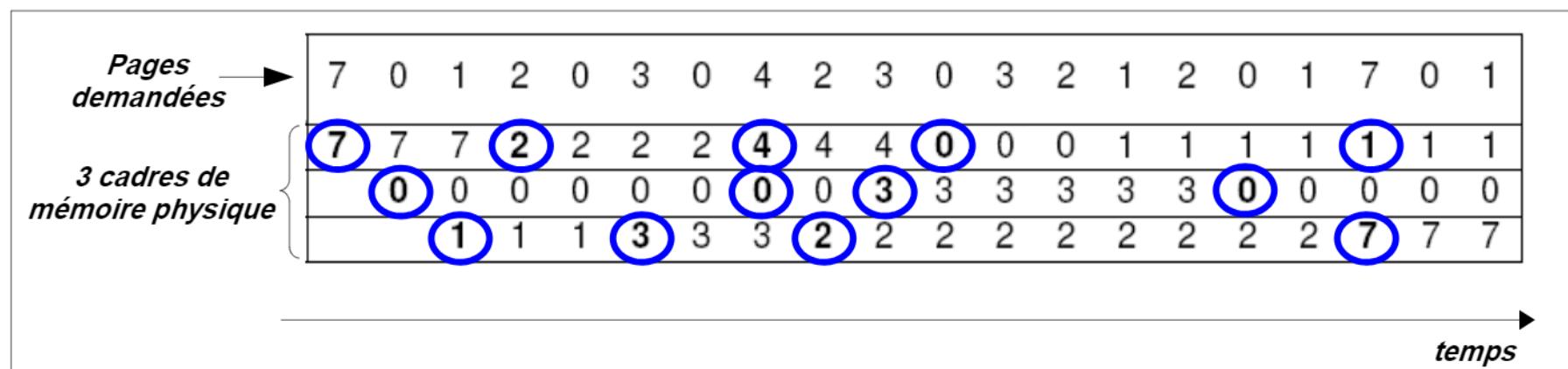
# L'algorithme d'Horloge

Page sequence: 0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3 4



## L'algorithme LRU

- **Principe** : choisir la page la moins récemment utilisée
- Stratégie plus complexe à implémenter
- Algorithme **coûteux** mais efficace



## L'algorithme LRU

- **Principe** : examiner les bits de référence (R) et de modification (M)

$(R,M)=(0,0)$



$(R,M)=(0,1)$



$(R,M)=(1,0)$



$(R,M)=(1,1)$

- Algorithme utilisé dans Mac OS

**Fin du chapitre 3**