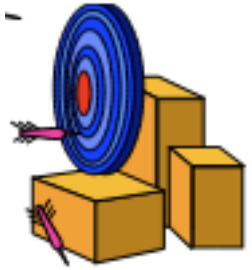
# Cours de principes des systèmes d'exploitation



LGI S4 – 2017/2018

Basé sur le livre d'Andrew Tanembaum

Pr. Thiam

Chapitre 1

# INTRODUCTION

# Définition

- *Un système d'exploitation est une machine abstraite conçue pour faciliter l'exploitation du matériel (pilotes de périphériques) ou pour arbitrer l'accès au matériel par les utilisateurs.*

# Rappels – 1

- Étude des SE transversale à
  - Algorithmique
  - Électronique
  - Statistique
  - Auxquelles elle emprunte des outils.
- Pas une discipline fondamentale mais
- Une application de l'informatique
- Comme l'informatique de gestion

# Rappels – 2

- Omniprésence
- Pression continue des utilisateurs
  - Extension des fonctionnalités
  - Efficacité maximale
  - Meilleur compromis entre
    - Fonctionnalité
    - Performance
    - Maintenabilité

# Rappels – 3

- Nécessite
  - Rigueur dans l'analyse des problèmes
  - Rigueur dans l'inventivité
  - Astuce dans l'implémentation
  - Lisibilité du code produit
  - Maintenabilité du code

# Rappels – 4

- Tout programme est concerné, écrire ses programmes en fonction du SE.
- Tout programmeur est susceptible de rencontrer les mêmes problèmes (ne pas inventer la roue)
- C'est un sujet intéressant dont le but est la recherche de l'efficacité.

# Historique (1)

- Première génération (1936-1955)
  - Ordinateurs à relais ou à tube à vides
  - Programmés par tableaux de connecteurs
  - Puis par cartes perforées (début des années 50)
  - Pas de SE
  - Grande lenteur
  - Très fragile

# Historique (2)

- Deuxième génération (1955-1965)
  - Apparition du transistor, ordinateur plus fiable
  - Distinction entre constructeur, opérateur, user programmeur
  - Calcul scientifique et ingénierie (équations diff.)
  - Apparition de langages FORTRAN et assembleur
  - Interprète de commande permettant
    - Chargement et Exécution des programmes
  - Apparition de Système d'Exploitation FMS (Fortran Monitor System) et IBSYS (IBM7094).

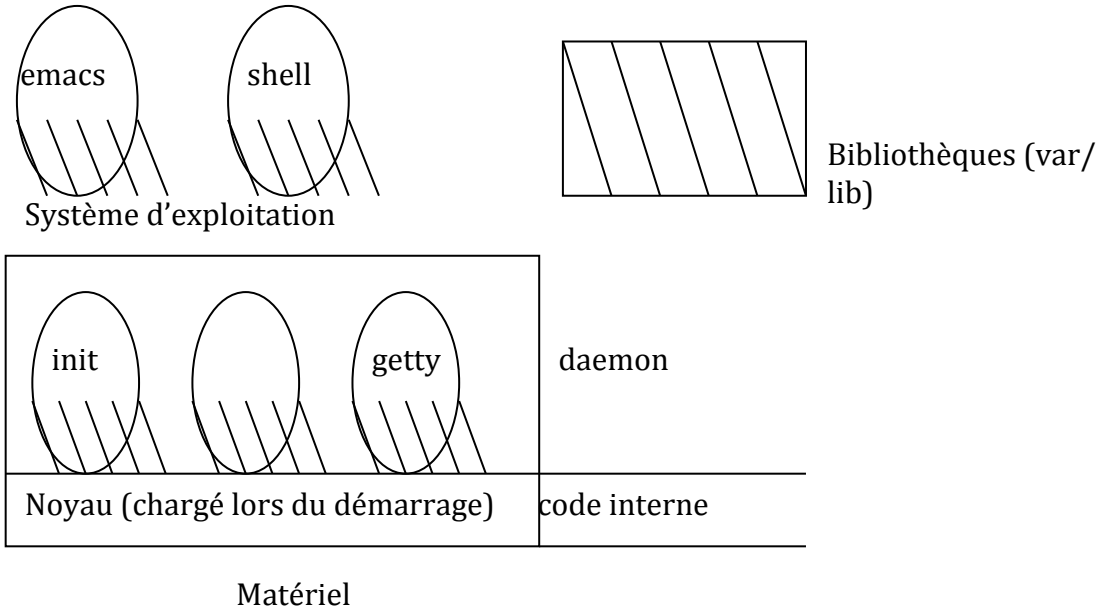
# Historique (3)

- Troisième génération (1965-1980)
  - Apparition des circuits intégrés et multiprogrammation
  - Diminution du rapport coût/performance
  - Standardisation
  - Familles de machines partageant
    - Langage machine
    - SE
    - Avec puissances & applications différentes

# Historique (4)

- Quatrième génération (1980-????)
  - Développement des circuits LSI (Large Scale Integration), puis VLSI
  - Avènement des ordinateurs personnels
  - Développement des réseaux de communication
  - Développement d'Internet
  - Systèmes d'exploitation en réseaux.

# Place du SE

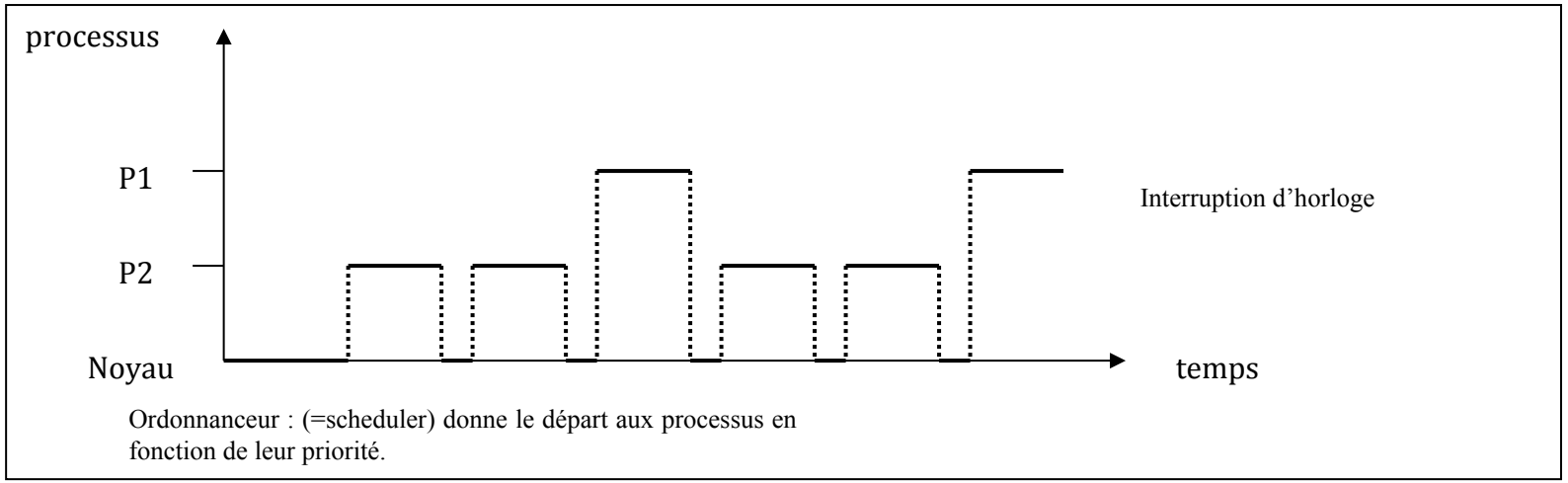



*Interface entre le noyau et les processus = appels – systèmes = interruption particulière*

# Fonctionnement – 1

- Lors de l'initialisation du noyau, le vecteur d'interruption de la machine est initialisée.
- Lors d'un appel système (ou toute autre interruption), le processeur bascule en mode « noyau » (réel).
- Lors du retour, le processeur rebascule en mode « utilisateur » (partagé).

# Fonctionnement – 2




Lorsque l'Ordonnanceur décide de changer le processus en cours, il effectue un changement de contexte entre P1 et P2. Au sein du noyau, chaque processus est décrit par une structure (Process Control Block – PCB).

# Process Control Block

- Espace pour sauver une copie des registres
- Priorité du processus
- Utilisateur réel/effectif
- Statistiques d'utilisation du système (temps système, temps utilisateur)
- Table des fichiers ouverts
- Gestion des signaux.

Chapitre 2

# THREADS / PROCESSUS

# Processus – utilités

- Sert à faire plusieurs activités en même temps
- **Exemple** : Faire travailler plusieurs utilisateurs sur la même machine. Chaque utilisateur a l'impression d'avoir la machine à lui tout seul

# Processus – utilités

- **Exemple** : Compiler tout en lisant son mail.
- Problème : Un processeur ne peut exécuter qu'une seule instruction à la fois
- **But** : Partager un (ou plusieurs) processeur(s) entre différents programmes (les processus).

# Processus – définition

- **Un processus est l'activité résultant de l'exécution d'un programme séquentiel, avec ses données, par un processeur**.
- Deux stratégies d'allocation du processeur
  - Avec ou sans réquisition du processeur ;
  - Fixées à priori en utilisant des informations sur le processus et l'activité du système.

# Processus – caractéristiques

- Dans la diapositive suivante le résultat possible de la commande *ps* sous Unix.
- Chaque ligne concerne un processus et chaque colonne donne une caractéristique des processus.

# Processus – caractéristiques

~ taha$ ps

| User | PID | %cpu | %mem | Stat | Start | Time | command |
|---|---|---|---|---|---|---|---|
| taha | 2345 | 12 | 1.5 | S | 19:54 | 0:00 | Tch(tch) |
| root | 5678 | 2.5 | 32 | I | Jan 4 | 19:23 | lemacs |
| taha | 7345 | 4.5 | 13 | W | 19:54 | 1:0 | swapper |
| root | 8678 | 19 | 22 | D | Dec 4 | 22:23 | syslog |

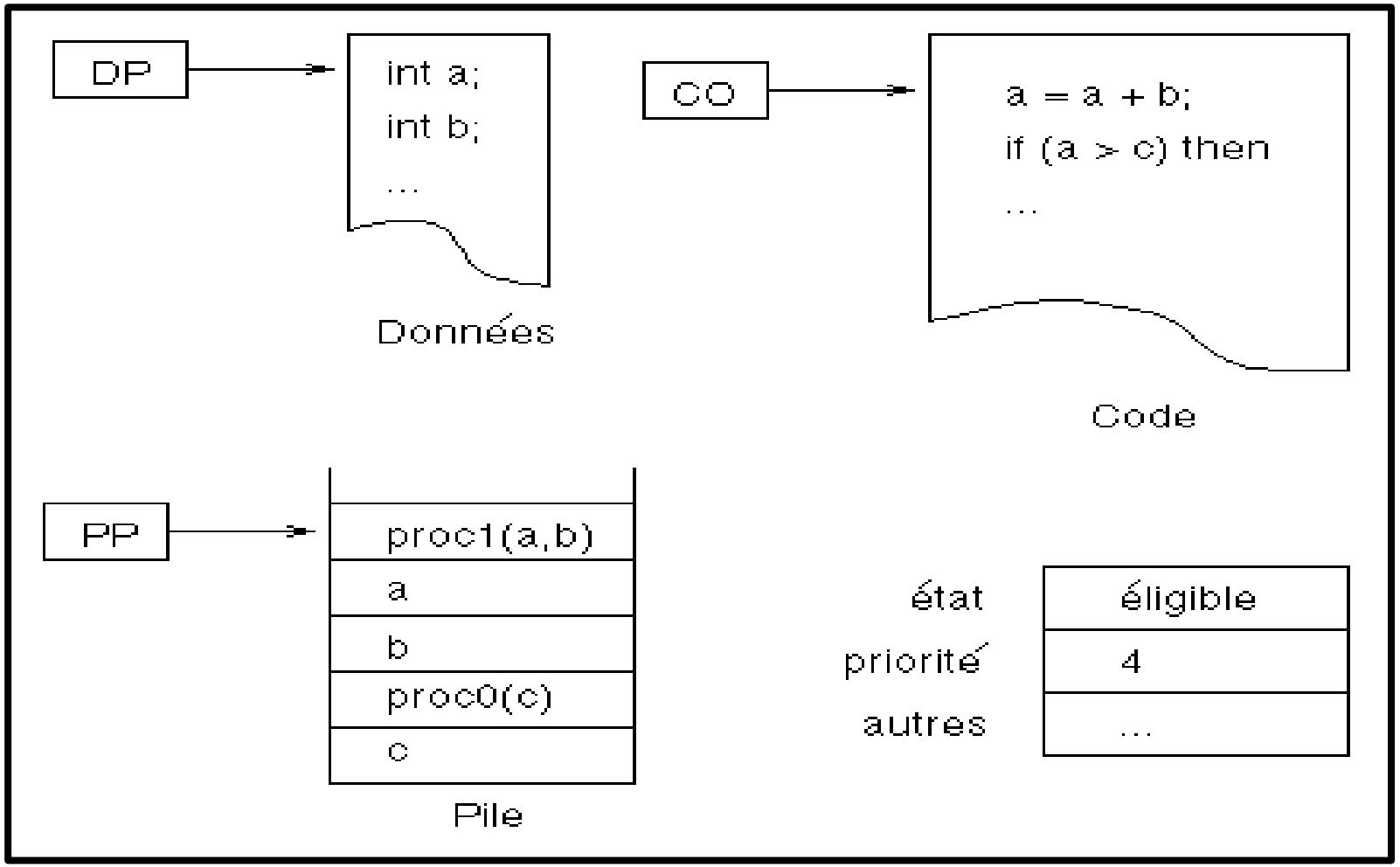
# Processus – caractéristiques

- Par exemple PID est l'IDentificateur du Processus.
- Pour plus d'information, reportez-vous aux man pages en tapant la commande man ps.

# Processus – contexte

- Le contexte d'un processus est l'ensemble des informations dynamiques qui représente l'état d'exécution d'un processus.

# Processus – contexte



# Processus légers

- Les processus légers ont été introduit dans les langages puis dans les systèmes d'exploitations.
- Un processus classique dispose de son propre contexte dans lequel s'exécute un seul flot d'exécution
- *Le multithreading permet, dans un seul espace d'adressage, d'effectuer plusieurs flots d'exécutions en parallèle.*

# Processus légers – exemple

- #include ”pthreads.h”
- void main(){
  pthread_t pid;
  pthread.create(&pid, NULL, F, NULL);
  …
}

- void F(){
  …
  printf( “thread %d\n”, pthread_set(F));
}

- La durée de vie du thread correspond à la durée de vie de la fonction.

# Threads – partage mémoire

- Le partage de la mémoire entre les threads est efficace dans la mesure ou *les échanges d'informations sont performants* (par exemple les threads peuvent simplement se passer des pointeurs). Cependant, cela implique quelques précautions.

# Threads – exemple de partage mémoire (1/3)

- Imaginons deux threads qui effectuent des opérations sur une liste chaînée. Le premier s'occupent des ajouts dans la liste. Le second, des suppressions.

# Threads – exemple de partage mémoire (2/3)

- Que ce passe-t-il si le 1$^{er}$ insére un élément **E** à la suite d'un élément $\alpha$ de la liste alors qu'en même temps, le 2$^{ème}$ est en train de supprimer le même élément $\alpha$ ?

# Threads – exemple de partage mémoire (3/3)

- Cela peut causer un dysfonctionnement. Pour empêcher cela, il faut faire en sorte que les 2 processus n'accèdent pas aux variables globales (ici : la liste chaînée) en même temps. (cf. sémaphores).

# Threads – remarques

- **R1** : Lorsque le multithreading a été implémenté sur les machines, il a fallu par exemple modifier l'implémentation de malloc/free pour empêcher leur exécution simultanément.

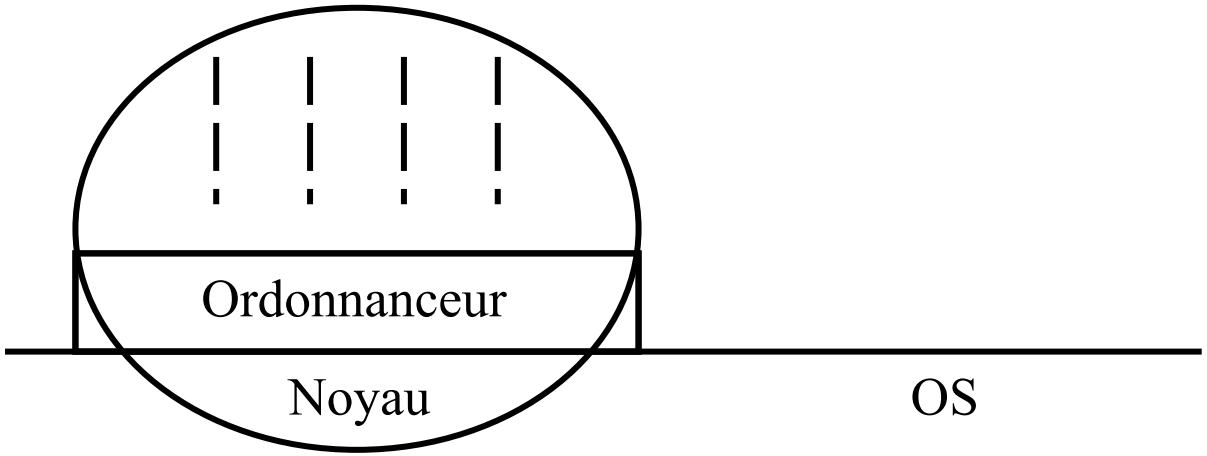
# Threads – remarques

- **R2** : Pour utiliser les versions sécurisées des threads, il faut utiliser l'option
- –D_REENTRANT à la compilation.

- **R3** : *Avec les threads, toutes les variables globales sont partagées.*

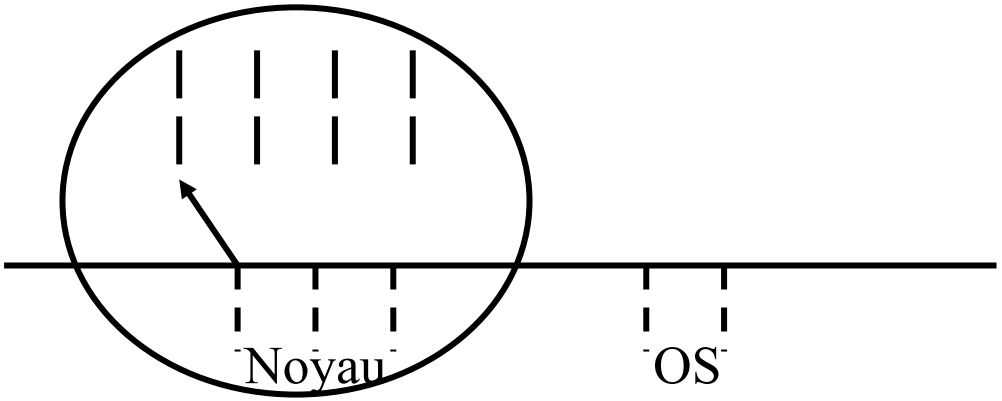
# Threads – implémentation niveau utilisateur

- Le système donne un temps d'exécution. C'est l'ordonnanceur qui fait avancer le plus de threads possibles.



# Threads – implémentation niveau noyau

- Création de plusieurs processus légers, codés dans le noyau. Ces processus ne sortent pas du noyau et permettent d'effectuer des opérations bas-niveaux nécessaires mais invisibles pour l'utilisateur. Cependant les threads noyau peuvent effectuer du code au niveau utilisateur.
- **Remarque** : sur une machine multiprocesseur, les threads d'un même processus sont partagés entre les différents processeurs par le noyau.



Chapitre 3

# ORDONNANCEMENT DE PROCESSUS

# Ordonancement des processus

- Objectifs
  - Chaque processus reçoit « sa partie » de temps processeur (équité)
  - Utilise le processeur à 100%
  - Mémorise le temps de réponse et/ou le temps de réutilisation des processus interactifs

# Ordonancement des processus

- Objectifs
  - Minimise l'attente des utilisateurs qui exécutent des « traitements par lots » (processus consommateur de temps processus)
  - Maximise le nombre travaux effectués en un temps donné

# Ordonancement des processus

- Quelques types
  - Systèmes interactifs (Linux, Windows XP)
  - Systèmes temps réels : échéancier des taches à exécuter (Linux RT)
  - Systèmes destinés aux « traitements par lots » (sont implémentés en partie en dehors du système)

# Ordonancement des processus
# Critères de performance

- Taux d'utilisation de l'unité centrale
- Débit
- Temps réel d'exécution
- Temps d'attente
- Temps de réponse

Ces cinq critères sont plus ou moins mutuellement exclusifs. Les comparaisons des différents algorithmes se font donc sur une sélection de ces critères.

# Algorithmes d'ordonancement

1. FIFO
2. FIFO+Horloge : Round Robin (tourniquet)
3. RR Avec priorité
4. Plus court d'abord
5. Systèmes interactifs

# Algorithmes d'ordonancement

1. **FIFO**
2. FIFO+Horloge : Round Robin (tourniquet)
3. RR Avec priorité
4. Plus court d'abord
5. Systèmes interactifs

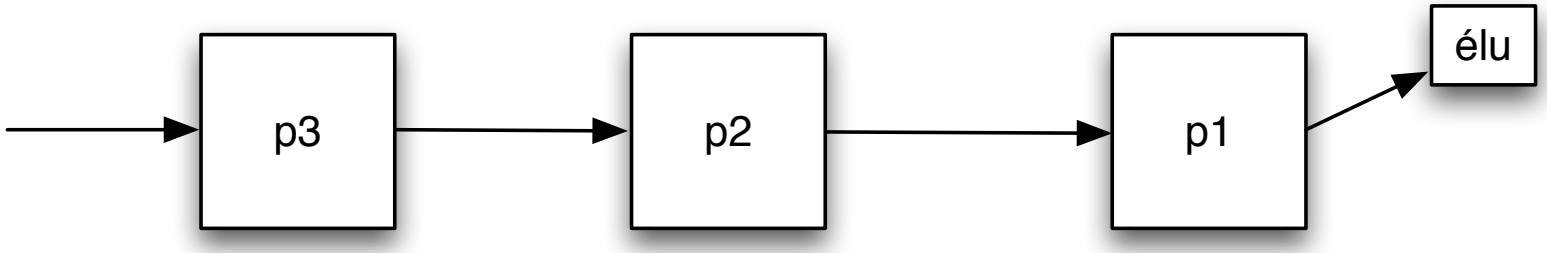
# FIFO (First In First Out)

- On exécute toujours le premier processus de la file.
- Lorsqu'un processus se bloque (ou se termine), on passe au suivant.
- A sa création ou à son réveil, le processus se place en queue de liste.

# FIFO (First In First Out)

- Appelé traitement par ”train”, par ”lot” ou ”batch”.
- Processus  accèdent au processeur,
  - chacun à leur tour,
  - dans l’ordre d’arrivée,
  - et monopolisent le processeur jusqu’`a leur terminaison.

# FIFO (First In First Out)



# FIFO (First In First Out)

- Travaux courts pénalisés
- Temps de réponse fonction de la charge du système
- Stratégie indépendante du temps d'exécution des processus
- Cette méthode est non-préemptive, c'est-à-dire qu'un processus monopolise le processeur jusqu'à sa terminaison.

# FIFO (First In First Out)

- Une amélioration de la stratégie FIFO est d'ordonner la file en fonction du temps estimé d'exécution des processus.
- Dans ce cas, le temps de réponse des travaux courts est diminué,
- et celui des travaux long est augmenté.

# Algorithmes d'ordonancement

1. FIFO
2. **FIFO+Horloge : Round Robin (tourniquet)**
3. RR Avec priorité
4. Plus court d'abord
5. Systèmes interactifs

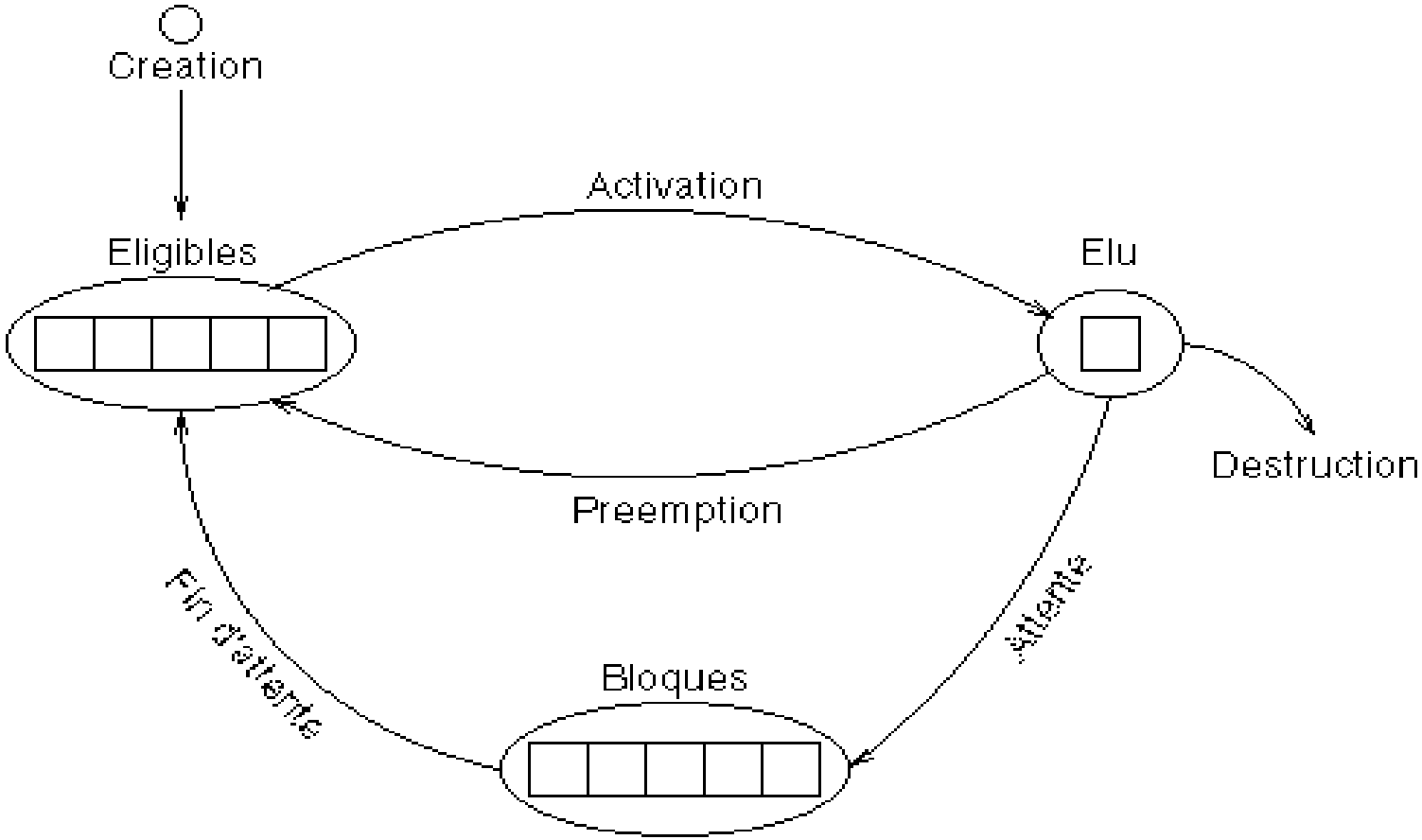
# Round Robin (tourniquet)

- Appelé ”balayage cyclique”
- les processus accèdent au processeur, chacun à leur tour,
- pour un temps déterminé à l’avance (le quantum).
- Un processus en attente d’une entrée-sortie sera placé dans une file des bloqués.

# Round Robin (tourniquet)

- Le principe reste le même que pour l'ordonnancement FIFO mais on ajoute une horloge qui permet, à chaque quantum de temps, d'effectuer un changement de contexte (toute les 100ms environ) pour faire tourner la liste de processus

# Round Robin (tourniquet)



# Round Robin (tourniquet)

- Plus court d'abord (SJF : Shortest Job First)
- Priorité = Ordonnancement de la file
- Privation des travaux longs
- Cette méthode est préemptive, c'est-`a-dire que le processeur est retiré à un processus au bout d'un certain quantum.

# Round Robin (tourniquet)

- Le quantum est calculé par tâtonements
- si le quantum est très grand, on obtient une file d'attente simple
- si le quantum est très petit, les processus n'auront pas le temps d'être exécutés en partie avant d'être réinsérés dans la file.

# Algorithmes d'ordonancement

1. FIFO
2. FIFO+Horloge : Round Robin (tourniquet)
3. **RR Avec priorité**
4. Plus court d'abord
5. Systèmes interactifs

# RR Avec priorité

- Le tourniquet présume que tous les processus sont d'égale importance.
- L'idée de base de cet algorithme est simple:
  - chaque processus possède une priorité et on lance le processus prêt dont la priorité est la plus élevée.
  - L'algorithme du tourniquet est appliqué aux processus de la même classe.

# RR Avec priorité

- Les priorités peuvent être statiques ou dynamiques.
- Pour empêcher les processus de priorité élevée de s'exécuter indéfiniment, l'ordonnanceur diminue la priorité du processus élu à chaque interruption de l'horloge.

# RR Avec priorité

- Les priorités peuvent être aussi assignées par le système.
- Certains processus font beaucoup d'E/S.
- Il faut donc leur allouer le processeur dès qu'ils le demandent.

# RR Avec priorité

- Un bon algorithme consiste à donner à ces processus la priorité 1/f où f est la fraction du quantum que le processus a utilisé à sa dernière activation.
- Si un processus n'a utilisé que 2ms de son quantum de 100ms, sa priorité est de 50.

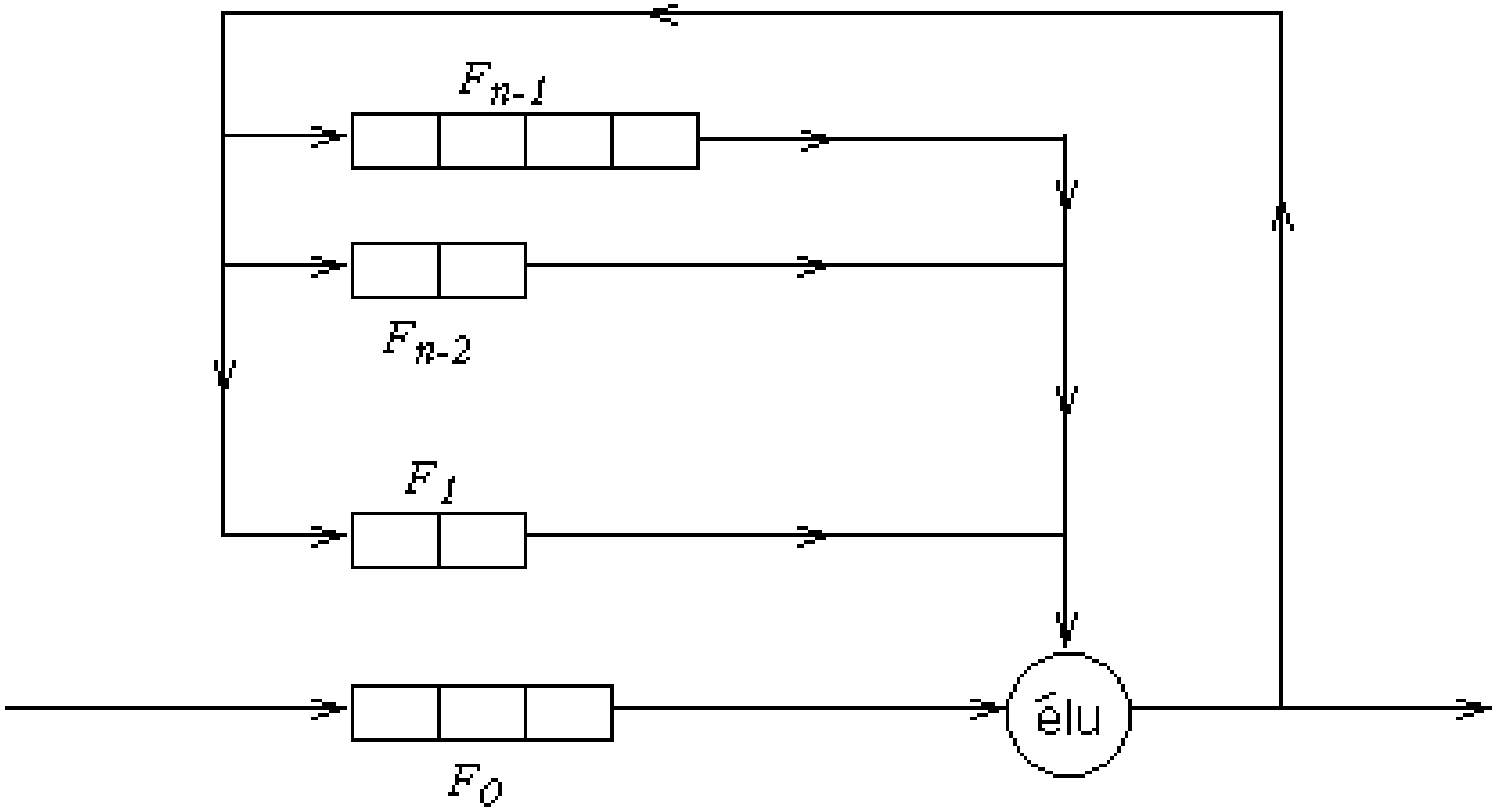
# Round Robin : multiniveaux

- Avant d'accéder au processeur, les processus sont rangés dans les files correspondant à leur niveau de priorité.
- Un processus ne peut accéder au processeur que s'il n'existe plus de processus dans les files de plus haute priorité.

# Round Robin : multiniveaux

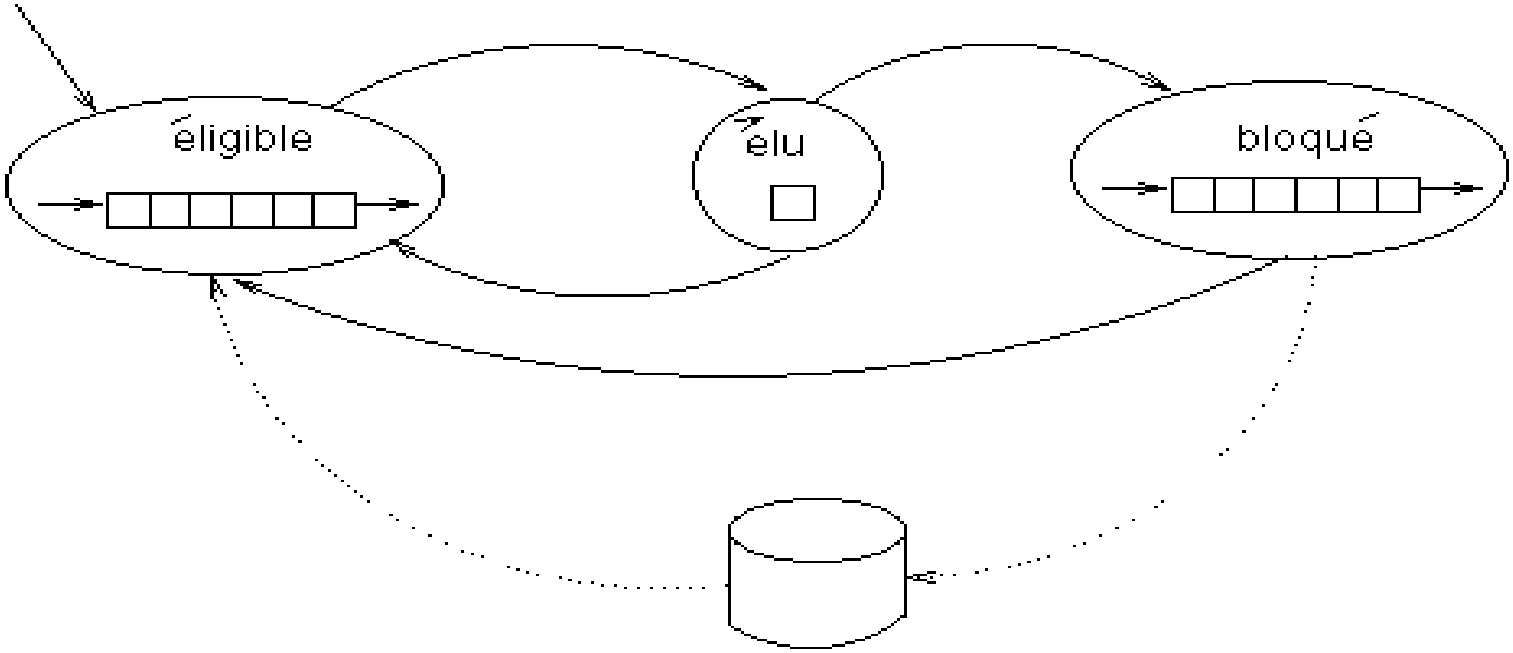
- Amélioration du tourniquet simple
- Priorités différentes par niveau peuvent être
  - externes, fixées avant la prise en compte du processus
  - internes, ajustées par le système
  - mixtes.

# Round Robin : multiniveaux



# Amélioration du système par le SWAP

Un processus peut être rangé (swapped) sur disque s'il reste trop longtemps dans la file des bloqués.



# Algorithmes d'ordonancement

1. FIFO
2. FIFO+Horloge : Round Robin (tourniquet)
3. RR Avec priorité
4. **Plus court d'abord**
5. Systèmes interactifs

# Plus court d'abord

- La plupart des algorithmes présentés sont conçus pour les systèmes interactifs.
- Il existe des algorithmes plus adaptés au traitement par lots pour lesquels le temps d'exécution par le processeur est connu à l'avance.
- Soient 4 taches prêtes dès le départ.

# Plus court d'abord

- Le but consiste à diminuer le temps moyen de restitution.

- Si leurs durées sont respectivement a, b, c, d, le temps moyen de restitution est : a + (a+b) + (a+b+c) + (a+b+c+d)

- $$Temps = \frac{4a + 3b + 2c + d}{4}$$

# Plus court d'abord

- Ce temps est donc minimal si $a<b<c<d$.

- On choisira donc de commencer par la tache dont la durée est la plus courte pour finir par celle dont la durée est la plus importante.

# Plus court d'abord

- Intéressant d'appliquer le même principe aux processus interactifs.
- Ces processus attendent une commande, l'exécutent et attendent la suivante, ...
- On pourrait minimiser le temps de réponse en considérant la commande la plus courte.
- Il reste à déterminer le plus court des processus restants en faisant une estimation basée sur le comportement passé des processus.

# Algorithmes d'ordonancement

1. FIFO
2. FIFO+Horloge : Round Robin (tourniquet)
3. RR Avec priorité
4. Plus court d'abord
5. **Systèmes interactifs**

# Systèmes interactifs

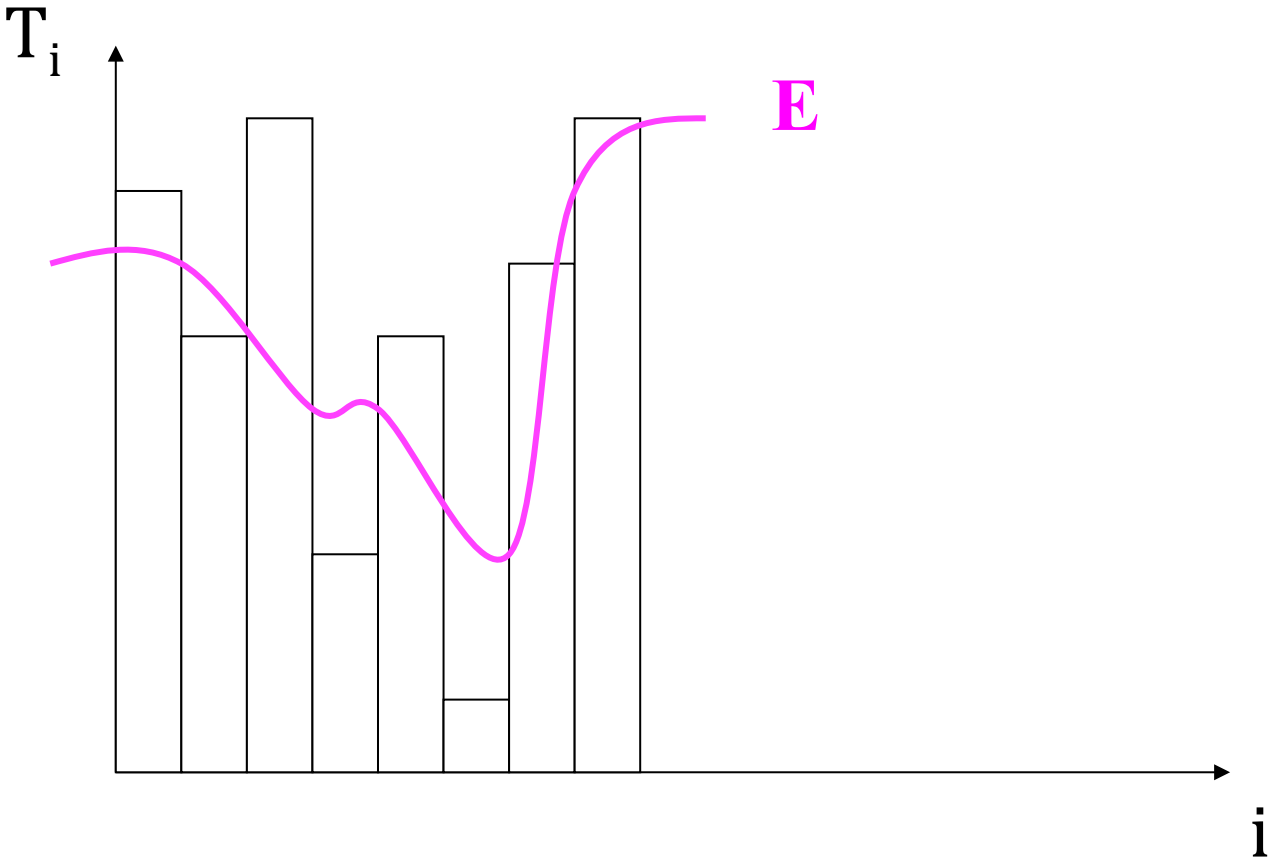
- Tenter d'estimer la durée du prochain cycle d'UC pour un processeur.
- La dernière fois qu'un processus s'est exécuté, il a utilisé $T_n$ secondes.
  L'estimation E pour le moment n+1 est :

$$E_{n+1} = \alpha . E_n + (1 - \alpha) T_n$$

# Systèmes interactifs

- Si $\alpha=0$ : on considère uniquement le dernier quantum
- Si $\alpha=1$ : jamais utilisé car l'estimation ne peut alors pas évoluer.
- Si $\alpha=1/2$ : cela permet d'obtenir une courbe d'estimation comme le montre le schéma de la diapositive suivante

# Systèmes interactifs ($\alpha$=1/2)



# Idée de la méthode

- Faire ordonnancer le processus qui à la valeur de E la plus petite.
- Attention, il faut faire vieillir les estimations même lorsque les processus ne s'exécutent pas.

# Exemple : noyau Linux 2.4

- *Dans le noyau 2.4 de Linux, chaque processus possède une priorité (que l'on peut changer par la commande nice).*

- *Cette priorité est converti en « nombre de quantum de temps » auxquels on aura pendant « une époque ».*

- Chaque processus à un compteur de priorité

# Exemple : noyau Linux 2.4

- Soit 4 processus $T_1$, $T_2$, $T_3$, $T_4$ ayant les priorités respectives 10, 15, 5, 10.

- Lorsqu'une époque commence, $T_2$ commence à s'exécuter (car c'est celui pour lequel le compteur est le plus élevé) et son compteur est amené à 14.

# Exemple : noyau Linux 2.4

- Puis l'ordonnanceur regarde à nouveau les priorités.

- $T_2$ est toujours le plus élevé, donc il s'exécute, et ce jusqu'à ce que son compteur descende à 9.

# Exemple : noyau Linux 2.4

- Puis $T_1$ et $T_4$ vont s'exécuter, puis une alternance $T_1$, $T_2$, $T_4$ jusqu'à l'arrivée de $T_3$ lorsque les compteurs sont tous à 4.

- *Lorsqu'on arrive à zéro pour tous, c'est la fin de l'époque*.

# Exemple : noyau Linux 2.4

- *Cependant, un processus peut se bloquer et l'époque se termine*.
- Imaginons que $T_2$ se soit bloqué à 4.
- Au début d'une nouvelle époque les compteurs de $T_1$, $T_3$ et $T_4$ sont réinitialisés à 10, 5, 10.

# Exemple : noyau Linux 2.4

- Le compteur de $T_2$ reçoit
  - *15+1/2(reste)*.
- Ainsi, le nombre compteur ne peut pas devenir infini (au pire : 2 fois la valeur initiale).

Chapitre 4

# SYNCHRONISATION DE PROCESSUS

# Pourquoi les outils de synchronisation

- Pour protéger du code
  - qui accède à des portions communes de la mémoire
- qui doit être exécuté par exemple en exclusion mutuelle
  - pas plus d'un processus qui exécute le code simultanément
- Bloquer des processus en attente d'un évènement.

# Exemple de code

- Soit le code exécuté en exclusion mutuelle.
- Est-ce que cela concerne aussi des codes très simples ?

```
Int glob=0 ;
Thread 1 :
For (int i=0,I<100, I++)
    Glob++;
```

```
Thread 2 :
For (int i=0, I<100, I++)
    Glob++;
```

# Exemple de code

- Les deux thread s'exécute en même temps et accèdent à une variable globale Glob.
- L'incrémentation glob++ n'étant pas atomique

# Exemple de code

- Possibilité de se retrouver dans des cas où
  - le thread 1 charge en mémoire une valeur i,
  - le thread 2 charge la même valeur i, l'incrémente et la place en mémoire
  - le thread 1 reprend la main et incrémente.
  - On se retrouve avec une nouvelle valeur i+1 alors qu'il y a eu 2 incrémentations.

# ALGORITHMES DE SYNCHRONISATION

# Idée des alogorithmes

- Permettre à un processus de
- rentrer en section critique c'est-à-dire
  - d'accéder à une variable globale si elle est libre,
  - attendre si la variable est occupée par un autre processus.
  - et accéder une fois libre

# Méthode 1

- Bool occupé
- Entrer_sc() {

  While (occupé)

  Occupé=TRUE

}

- Sortir_sc() {

  Occupé= FALSE

}

# Méthode 2 (2 processus )

- Int tour=0 ;
- Bool je_veux_entrer [2]
- Entrer_sc () {

  Je_veux_entrer[i] = TRUE

  Tour = i

  Attendre (je_veux_entrer[1 – i] == FALSE et tour == 1 – i )

}
- Sortir_sc () {

  Je_veux_entrer[i]=FALSE

}

- **Mauvaise méthode car l'attente est active et occupe donc 100% de CPU**

# Méthode 3 : Support matériel pour instructions atomiques

- Exemple de l'instruction matérielle test_and_set qui peut se traduire par l'algorithme suivant
  - cette fonction étant une instruction machine, elle est atomique
- Int test_and_set (int* verrou){

```
    Int old = *verrou;
    *verrou=1;
    return old;
}
```

# Utilisation de test_and_set

- On peut utiliser l'instruction pour entrer en section critique
- Entrer_sc() {

```
    While (test_and_set(&verrou))
    While (verrou==1) {}
}
```

- Sortir_sc() {

```
    Verrou=0 ;
}
```

# Problèmes de ces solutions

- Dans toutes les solutions précédentes, l'attente est active et utilise donc le processeur.
- Comment rendre la primitive d'entrée en section critique réellement bloquante ?

# DES SOLUTIONS

# SÉMAPHORES

# Sémaphores de Dijkstra

- Un sémaphore contient
  - Une valeur entière
    - initialement positive ou nulle
  - Une liste de processus en attente
- On utilise sur les sémaphores deux primitives atomiques
    - du point de vue processus
  - P(Sémaphore Sem)
  - V(Sémaphore Sem)

# Définitions

- C'est un mécanisme de synchronisation entre processus.
- Un sémaphore S est une variable à valeurs entières positives ou nulle, manipulable par l'intermédiaire de deux opérations
  - P (proberen) et
  - V (verhogen)

# P et V

- P (S) : si $S \leq 0$, alors mettre le processus en attente ; sinon : $S := (S - 1)$.

- V (S) : $S := (S + 1)$ ; réveil d'un processus en attente.

# P et V

- P (S) : Cette opération correspond à une tentative de franchissement. S'il n'y a pas de jeton pour la section critique alors attendre, sinon prendre un jeton et entrer dans la section.

- V (S) : Rendre son jeton à la sortie de la section critique. Chaque dépôt de jeton V (S) autorise le passage d'une personne. Il est possible de déposer des jetons à l'avance.

# Primitive P : bloquer un processus

**P**(Semaphore Sem) {

Sem.value -- ;

If (Sem.value <0 )

Se bloquer sur la liste interne au sémaphore

}

# Primitive V : libérer un processus

**V**(Semaphore Sem) {

Sem.value ++ ;

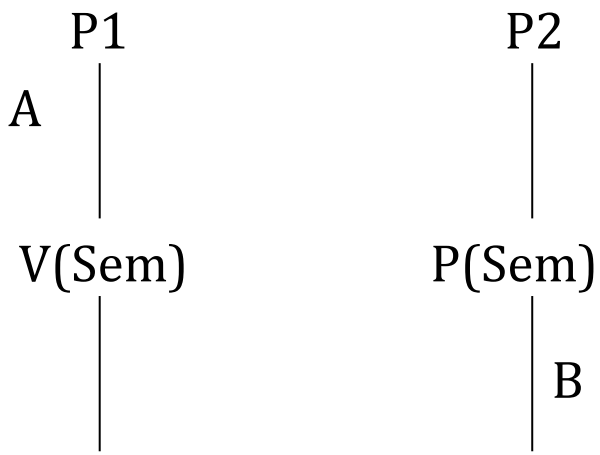
If (Sem.value ≥ 0)

Réveiller un processus bloquer de la liste d'attente

}

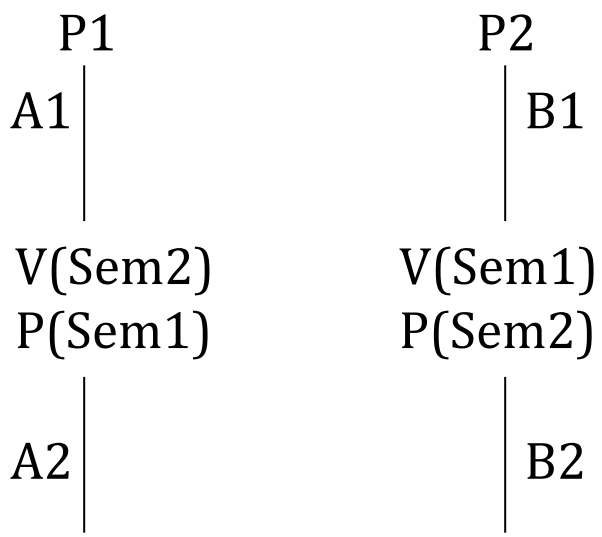
# Exemple 1

- Permettre au processus $P_2$ d'attendre la fin de l'exécution du code A de $P_1$ avant de continuer son exécution B



# Exemple 1

- On souhaite que le code $B_2$ soit exécuté après $A_1$ et $B_1$ après $A_2$
  - c'est-à-dire qu'à un moment donné les deux processus s'attendent mutuellement



# Sémaphore pour N processus

- Si l'on se table sur l'exemple précédent, il semble nécessaire d'avoir N sémaphores pour permettre à N processus de s'attendre mutuellement.

- But : utiliser un *seul sémaphore* pour bloquer tous les processus.

# Sémaphore pour N processus

```
Int nb=0 ; // pour compter le nombre de processus
Sem wait(0)
Sem mutex(1) //Pour protéger la variable globale nb
Barriere () {
    P(mutex)
    Nb++ ;
    If (nb<N) {
            V(mutex)
            P(wait)
    }
    else {
            V(mutex)
            Nb=0;
            For (int i=0; i<N-1; i++)
                    V(wait)
    }
}
```

# Remarques

- Ne pas mettre d'opérations bloquantes (P) dans une section critique d'exclusion mutuelle
  - c'est à dire entre un P(mutex) et V(mutex).
- Si l'on utilise plusieurs barrières dans les processus, il se pourrait qu'on libère un processus attendant la synchronisation à la barrière 2 avec le processus libérant de la barrière 1.

# Remarques

- D'où erreur, car le processus libéré en 2 n'a pas attendu les autres.
- Une solution consiste à doubler les barrières : on utilise alternativement la barrière 1 puis la barrière 2.

# PRODUCTEUR CONSOMMATEUR

# Producteur consommateur

- Soit une file FIFO de taille MAX avec 2 primitives :
  - Put() : insérer un élément dans la file.
  - Get() : récupérer un élément de la file.

# Un producteur, un consommateur

- Dans un 1er temps comment utiliser cette file lorsque
  - un processus est producteur dans la file
    - insère des éléments dans la liste
  - un autre est consommateur
    - récupère les éléments de la liste
- Il s'agit de vérifier de + que la liste n'est pas vide lorsqu'on veut récupérer un élément et non pleine lorsqu'on veut en ajouter un.

# Un producteur, un consommateur

- //Variables globales :
- Sem cons(0) ; Sem prod(MAX) ;
- **Produire** (){
  P(prod) //pas d'insertion si la file est pleine
  Put()
  V(cons) //prévient qu'il y a un élément dans la liste
  }
- **Consommer**(){
  P(cons) //Pas de récupération si la file est vide
  Get()
  V(prod) //prévient qu'il y a une place vide
  }

# Remarques

- Le sémaphore « cons » compte le nombre de place occupées
- Le sémaphore « prod » compte le nombre de places vides

# Plusieurs producteurs consommateurs

- Il faut rajouter un $mutex_1$ pour mettre en section critique « produire » et un $mutex_2$ pour « consomme ».
- **//Variables globales :**
  - Sem cons(0) ;
  - Sem prod(MAX) ;
  - Sem $mutex_1$(1) ;
  - Sem $mutex_2$(1) ;

# Plusieurs producteurs consommateurs

- Produire (){
  P(prod) //pas d'insertion si la file est pleine
  P($mutex_1$)
  Put()
  V($mutex_1$)
  V(cons) //prévient qu'il y a un élément dans la liste
  }
- Consommer (){
  P(cons) //Pas de récupération si la file est vide
  P($mutex_2$)
  Get()
  V($mutex_2$)
  V(prod) //prévient qu'il y a une place vide
  }

# LECTEURS – RÉDACTEURS

# Lecteurs – rédacteurs

- Processus lecteur un processus qui consulte une donnée en lecture seule
- Processus rédacteur qui accède à cette donnée en lecture et écriture.
- Un seul rédacteur en même temps qui exécute écrire().

# Lecteurs – rédacteurs

- On n'a jamais un rédacteur et un lecteur à la fois qui exécutent lire()
- On peut avoir plusieurs lecteurs à la fois s'il n'y a pas de rédacteurs.

# Lecteurs – rédacteurs

- Sem write(1) ;
- Sem mutex(1) ; *// Pour Nbl*
- Int Nbl = 0 ;

- Redacteur (){
  P(write) ;
  Ecrire() ;
  V(write) ;
  }

# Lecteurs – rédacteurs

- Lecteur(){

```
        P(mutex)
        Nbl++ ;
        If (Nbl==1) P(write) ;
        V(mutex) ;
        Lire() ;
        P(mutex) ;
        Nbl-- ;
        If (Nbl==0) V(write) ;
        V(mutex) ;
}
```

# Remarques

- Ce fonctionnement n'est pas équitable en processus.
- Les lecteurs sont fortement prioritaires.
- Pour éviter cela, il faut utiliser un sémaphore entonnoir supplémentaire initialisé à 1.
- Les sémaphores ne sont pas strictement FIFO dans la vraie vie.
- Dans ce cours nous considèrons que si.

# HOARE : MONITEURS

# Moniteurs

- Begin monitor m ;
- Condition cond ;
- Procedure $P_1$ () {
  P(mutex) ;
  Wait (cond) ;
  V(mutex) ;
  }
- Procédure $P_2$ () {
  …
  signal (cond)
  }
- end monitor.

# Moniteurs

- La fonction signal ***réveille*** un processus bloqué sur une condition.
- La main passe au processus débloqué.
- Dans le langage Mesa, un sémaphore différent (de celle de Hoare) a été proposé.
- Le processus réveillé par un signal n'est pas assuré d'obtenir l'entrée dans le moniteur en priorité

# Utilisation des moniteurs

- Utilisation des primitives de bibliothèque
  - Monitor mutex_t ;
  - Mutex_lock (mutex_t *m) ;
  - Mutex_unlock (mutex_t *m) ;

# Utilisation des moniteurs

- On travaille au travers de conditions via les primitives suivantes
  - Cond_t ;
  - Cond_wait(cond_t *c, mutex_t *m) ;
  - Cond_signal(cond_t *c) ;
  - Cond_beast(cond_t *c) ;

# Exemple : barrière de synchronisation

- Mutex_t m ;
- Cond_t wait ;
- Int nb=0 ;
- Barrière() {

```
    Mutex_lock(&m) ;
    Nb++ ;
    If (nb < N)
        Cond_wait(&wait, &m) ;
    Else{
        Cond_beast(&wait) ;
        Nb=0 ;
    }
    mutex_unlock(&m) ;
}
```

# Exemple : producteur consommateur

- mutex m ;
- cond_t cond ;
- int nb_elt=0 ;

- Consommateur(){
```
    Mutex_lock(&m) ;
    While (nb_elt==0)
      Wait(cond) ;
    Get() ;
    Nb_elt-- ;
    Mutex_unlock(&m) ;
}
```
- Producteur(){
```
    Put() ;
    Nb_elt++ ;
    Signal(cond) ;
}
```

# Remarques

- Les moniteurs sont davantage utilisés que les sémaphores (car plus simple à utiliser)

```
F() {                          G(){
    Mutex_lock(&m) ;               Mutex_lock(&m)
    …                              …
    Mutex_unlock(&m) ;             Mutex_unlock(&m)
}                              }
```

# Exemple : inversion de priorité

- Le robot PathFinder (**1997**) utilisé pour l'exploration sur Mars fonctionnait avec 3 taches dans le système :
  - Une tache faiblement prioritaire (*mutex_lock*, *mutex_unlock*) de traitement de données
  - Une tache fortement prioritaire (*mutex_lock*, *mutex_unlock*) de création des données
  - Une tache moyennement prioritaire de communication des données et résultats

# Exemple : inversion de priorité

- Le scénario de bug vécu sur ce robot est le suivant
  - La tache 1 (**traitement**) détient le moniteur
  - La tache 2 (**création**) essaie d'acquérir le moniteur (mais se retrouve bloquée par 1)
  - La tache 3 (**communication, traitée sans le moniteur**) se réveille à cause d'une interruption et prend la main.

# Exemple : inversion de priorité

- Cela implique le déroulement suivant
  - La tache 1 est interrompu par la 3.
  - La tache 2 attends toujours
  - La tache 3 étant très longue, le système pense qu'il y a un problème et reboote

# Exemple : inversion de priorité

- Ce déroulement montre qu'il y a eu une inversion de priorité sur les processus.
- Une solution empirique à ce problème consiste en un héritage de priorité.
- Lorsqu'une tache est bloquée sur un mutex_lock, on s'assure que la priorité de la tache qui détient le verrou est au moins aussi forte que celle qui le demande.

# Problèmes connus

- Diner des philosophes
- Paradoxe de l'autriche
- Le coiffeur endormi

Chapitre 5

# GESTION DE LA MEMOIRE

# Gestion de la mémoire

## Plan

1. Introduction
2. Gestion élémentaire de la mémoire
3. Le va-et vient (swaping)
4. La mémoire virtuelle : pagination
5. Algorithmes de remplacement de pages
6. La segmentation

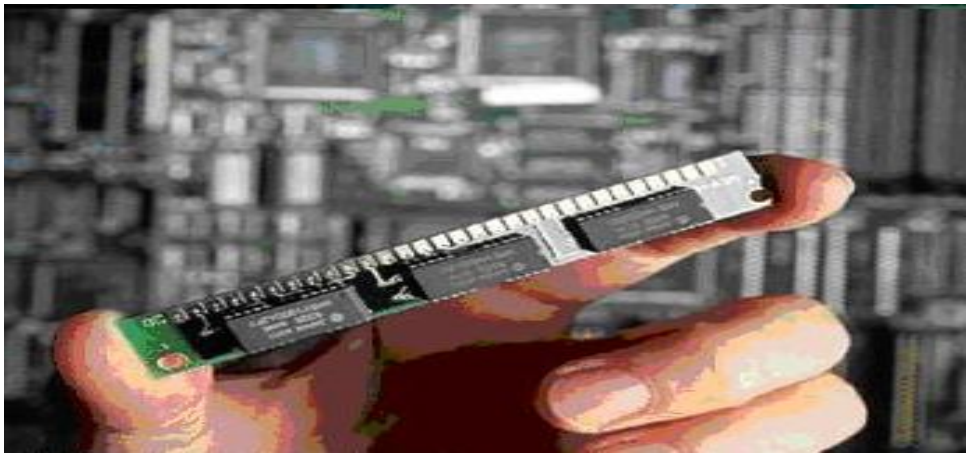
# Gestion de la mémoire

La gestion de la mémoire est un difficile compromis entre les performances et la quantité d'espace disponible.

Elle s'occupe de l'allocation de l'espace mémoire au système et aux différentes processus en cours d'exécution.

# OUPS!!!

## Souvenez – vous…La Mémoire !!!



**Barrette SIMM**

Ce module SIMM (single in-line memory module) est un circuit électronique qu'on installe sur la carte mère d'un ordinateur pour recevoir des microprocesseurs de mémoire. Les SIMM exigent moins d'espace que les barrettes de mémoire conventionnelles.

**Microsoft ® Encarta ® 2008.** © 1993-2007 Microsoft Corporation. Tous droits réservés.

# Introduction

1. La mémoire est un élément important de l'ordinateur. D'elle, dépend une bonne partie des performances du système.

2. Les mécanismes de sa gestion sont assurés grâce à des techniques dont la segmentation.

3. Pour mieux comprendre ce mécanisme, Il serait plus judicieux de parler d'abord de la mémoire et de ses caractéristiques.

# Définition de la mémoire

- On appelle « **mémoire** » tout composant électronique capable de stocker temporairement des données.

- Elle est aussi une juxtaposition de cellules ou mots-mémoires ou cases-mémoires à m bits.

- Il existe plusieurs types de mémoires. Pour pouvoir les classer, il faudra étudier certaines caractéristiques.

# Caractéristiques d'une mémoire

Les caractéristiques d'une mémoire sont généralement :

1. Sa capacité
2. Son temps d'accès
3. Son cycle mémoire
4. Son débit
5. Sa volatilité.

Ainsi, suivant leurs caractéristiques, nous avons plusieurs types de mémoires.

# Typologies de mémoires (I)

Les mémoires peuvent être classées en trois Catégories:

La **mémoire centrale** (*mémoire interne*) permettant de mémoriser temporairement les données lors de l'exécution des programmes.

# Typologies de mémoires (II)

La **mémoire de masse** (*mémoire externe*) permettant de stocker des informations à long terme, y compris lors de l'arrêt de l'ordinateur.

La **mémoire cache** qui est une zone de mémoire tampon (accès rapide) où des données sont emmagasinées temporairement.

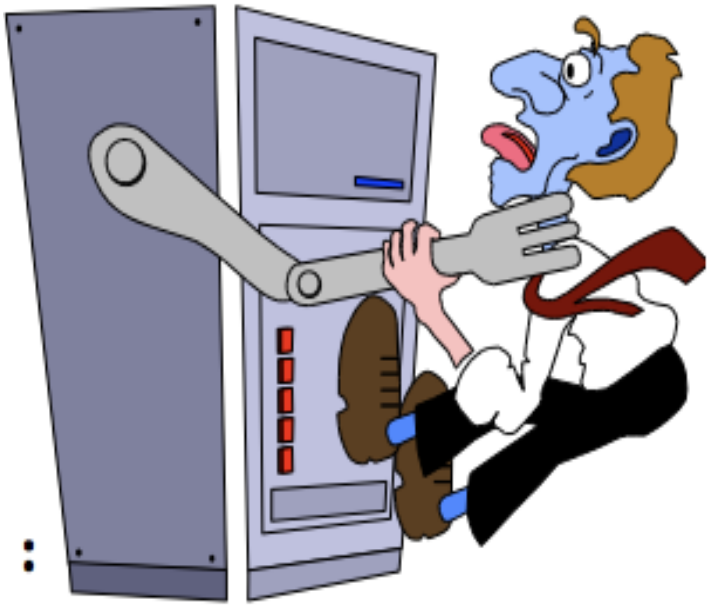
# Typologies de mémoires (III)

- Idéalement, on voudrait que la mémoire soit
  - grande
  - rapide
  - non volatile
  - bon marché
- Hiérarchie de la mémoire
  - Petite quantité de mémoire rapide et volatile mais chère (cache)
  - Quelques mégaoctets de mémoire volatile de vitesse et prix moyen (mémoire principale)
  - Quelques gigaoctets de mémoire plus lente, non volatile et bon marché (disque)
- Cette hiérarchie est gérée par le **gestionnaire de la mémoire**

# Exigences

Les mécanismes de gestions de la mémoire doivent être capable de satisfaire à 5 exigences :

- la relocalisation (réallocation)
- la protection
- le partage
- l'organisation logique
- l'organisation physique.



# Notions Générales

La mémoire centrale peut-être découpée de trois façons:

- la **segmentation** : les programmes sont découpés en parcelles ayant des longueurs variables appelées «*segments*».

- la **pagination**: elle consiste à diviser la mémoire en blocs, et les programmes en pages de longueur fixe. Elle est transparente aux programmes qui n'utilisent que des adresses logiques sans se soucier de leur emplacement physique.

- une combinaison de **segmentation** et de **pagination**: certaines parties de la mémoires sont segmentées, les autres sont paginées.

# Organisation de la mémoire physique

Sur une architecture 32 bits, la mémoire physique peut être considérée comme une suite d'octets. A chaque octet est associé un numéro de 0 à 232 − 1.

Remarquez que l' on peut adresser environ 4 giga-octets (Go) de mémoire ce qui est plus que n'en possèdent la plupart des machines actuelles.
Une partie de cette mémoire est virtuelle (cf. suite).
La gestion de la mémoire vive présente des analogies avec la gestion de la mémoire persistante.

Mais le lien avec le matériel est plus étroit car la gestion de la mémoire vive se doit d'être très efficace.
Ainsi, des codes correcteurs d'erreurs sont utilisés afin de permettre des manipulations matérielles provoquant des erreurs tolérables.

# Répartition de la mémoire entre système et processus

La mémoire principale se divise en deux parties :

– l'une est destinée au système d'exploitation et contient
  - une pile d'interruption système ;
  - la table des processus ;
  - code du système.

Cette partie est toujours résidente en mémoire

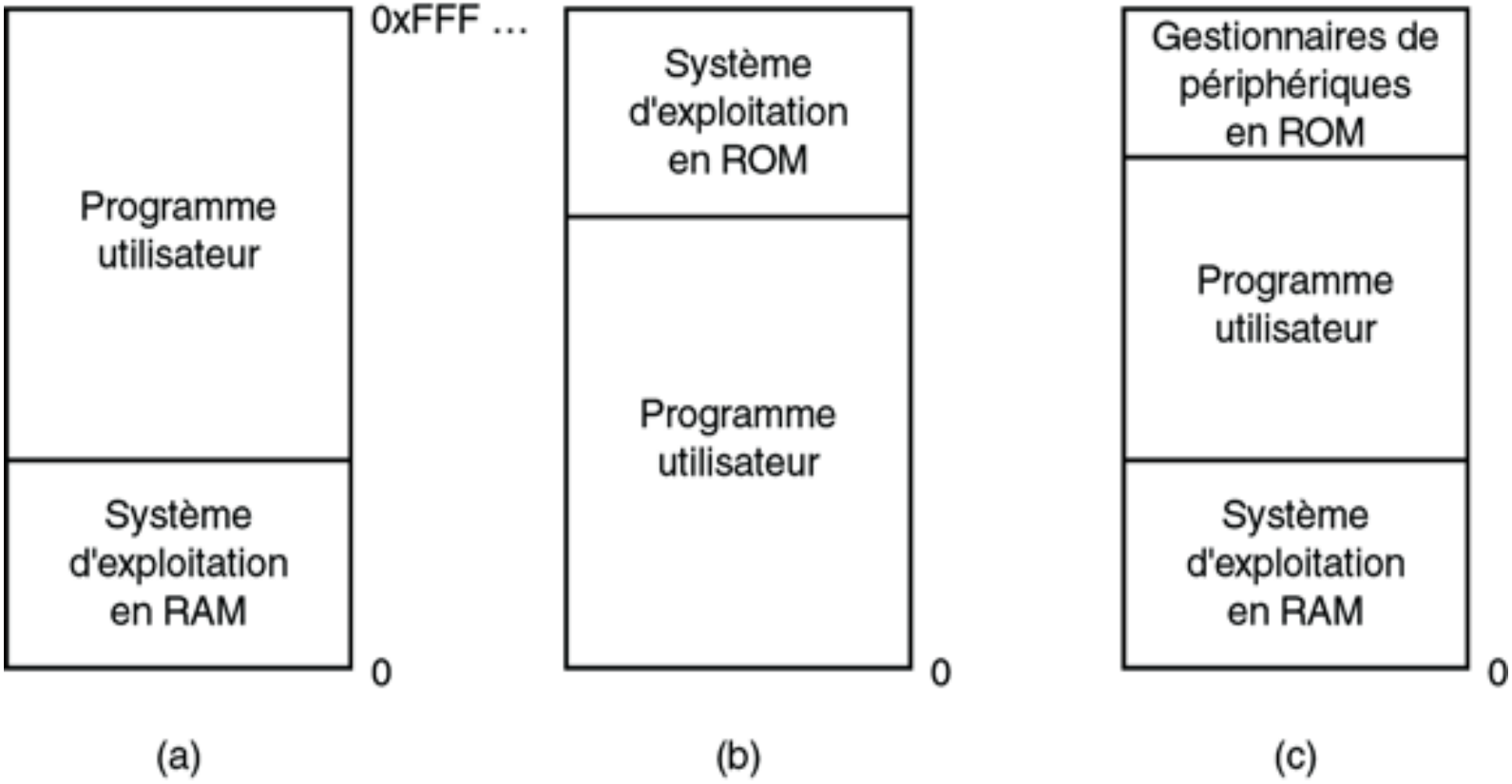
– l'autre est destinée aux processus en cours d'exécution et doit contenir de la mémoire dévolu aux processus :

| Image d'un processus en mémoire |
|---|
| Pile d'exécution (pass. param.) |
| tas (zone malloc) |
| données non initialisées (à 0) |
| données initialisées |
| code utilisateur |

| un autre processus |
|---|
| Pile d'exécution (pass. param.) |
| tas (zone malloc) |
| données non initialisées (à 0) |
| données initialisées |
| code utilisateur |

# Gestion élémentaire de la mémoire

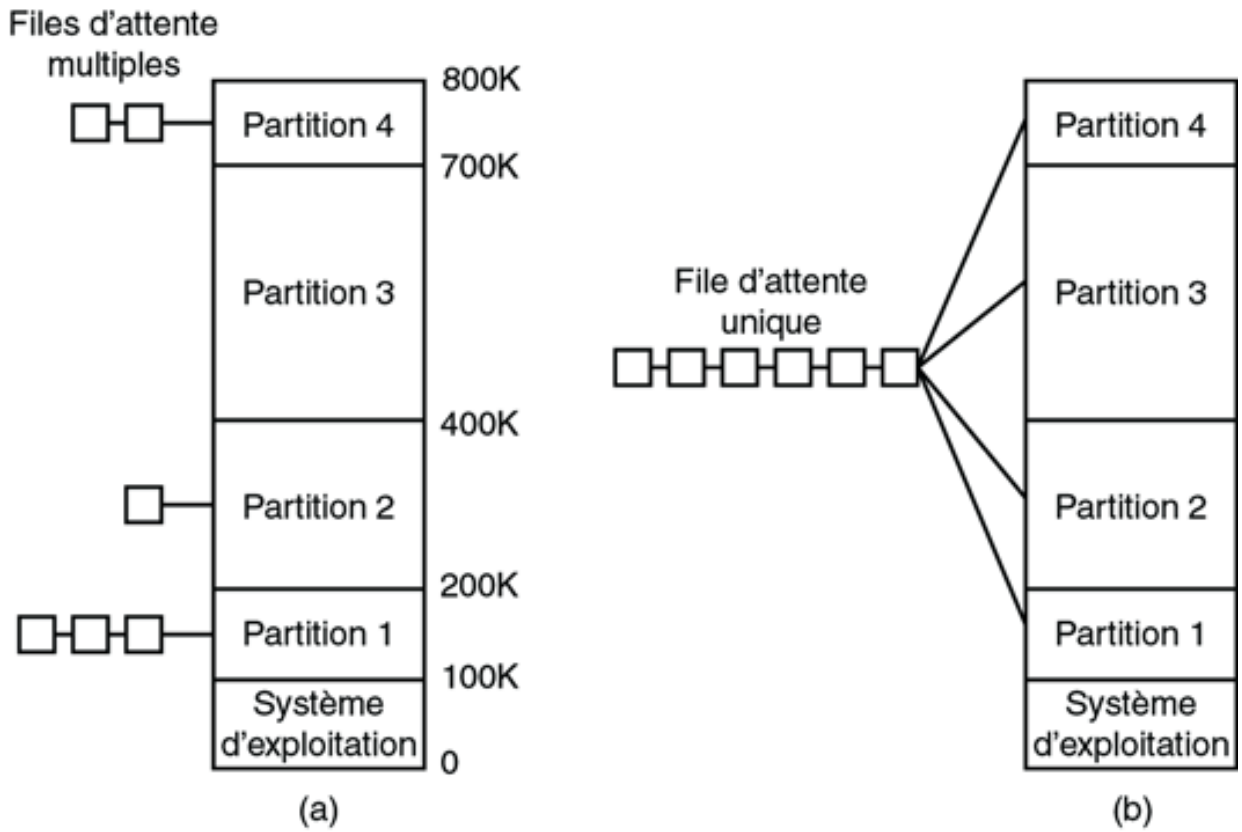
## Monoprogrammation sans va-et-vient ni pagination




Trois façons simples d'organiser la mémoire

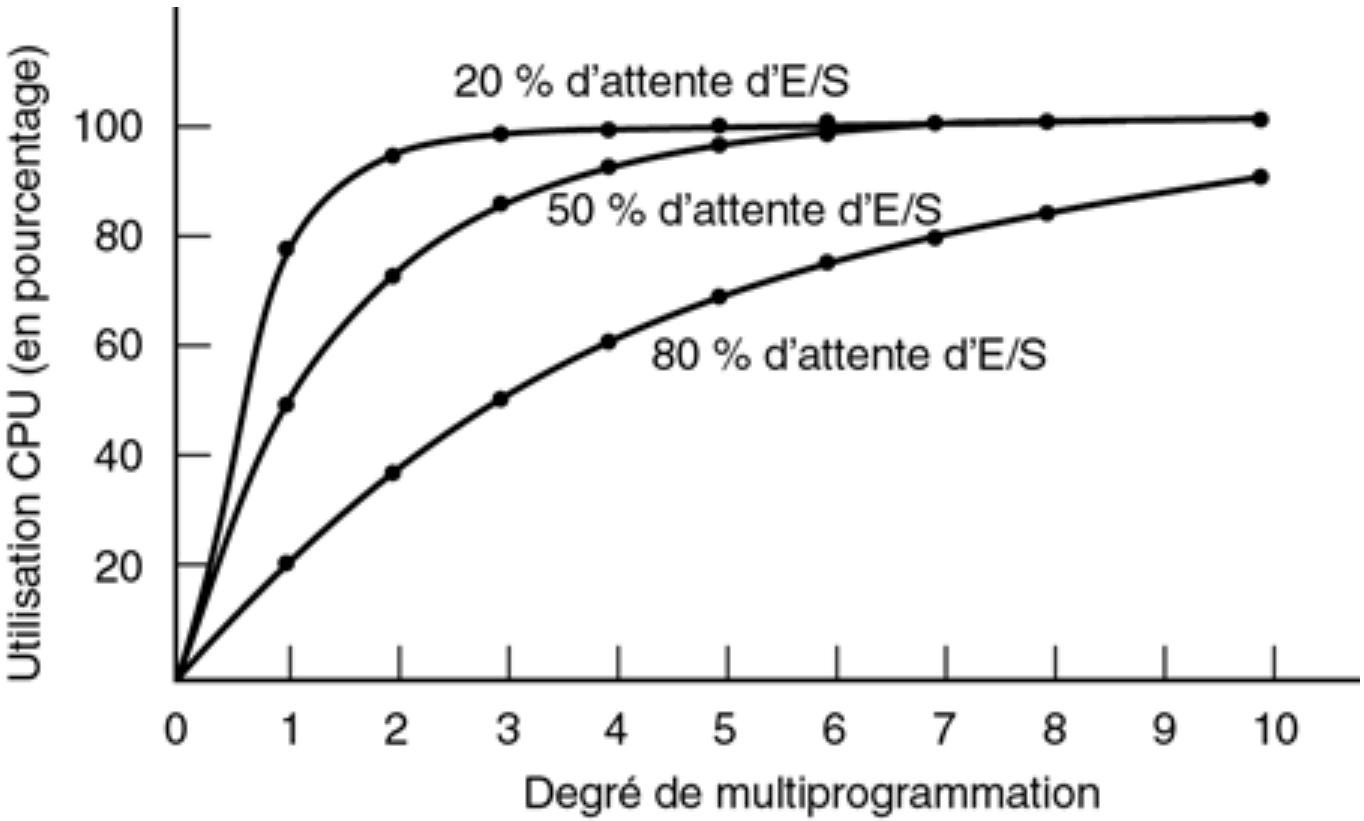
- avec un système d'exploitation et un seul processus

# Multiprogrammation avec partitions




- Partitions de taille fixe
  - files d'attente différentes pour chaque partition
  - une seule file d'attente

# Modélisation de la multiprogrammation




Taux d'utilisation du CPU ( $1-p^n$ ) en fonction du nombre de processus en mémoire.

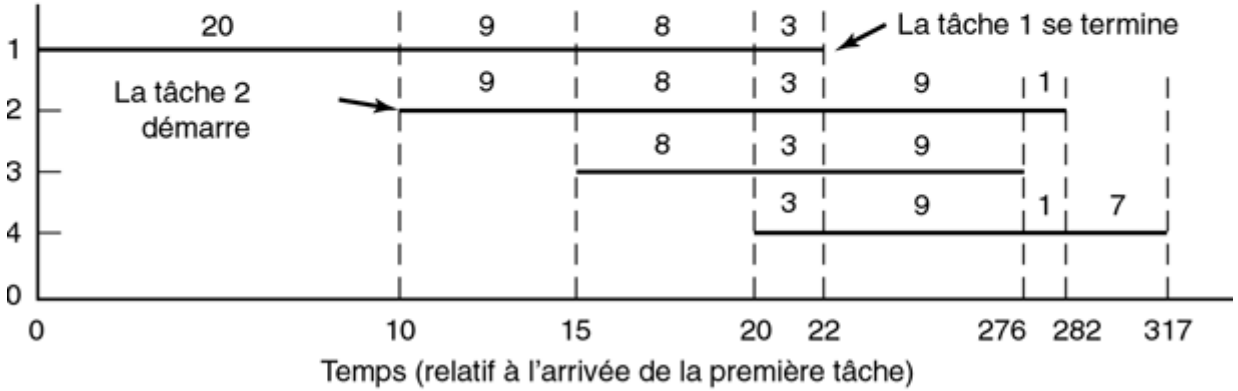
# Analyse des performances des systèmes multiprogrammés

| | Temps d'arrivée | Besoins CPU en minutes |
|---|---|---|
| 1 | 10:00 | 4 |
| 2 | 10:10 | 3 |
| 3 | 10:15 | 2 |
| 4 | 10:20 | 2 |

(a)

| | Nombre de processus 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Temps CPU inactif | 80 % | 64 % | 51 % | 41 % |
| Temps CPU occupé | 20 % | 36 % | 49 % | 59 % |
| Temps CPU par processus | 20 % | 18 % | 16 % | 15 % |

(b)




(c)

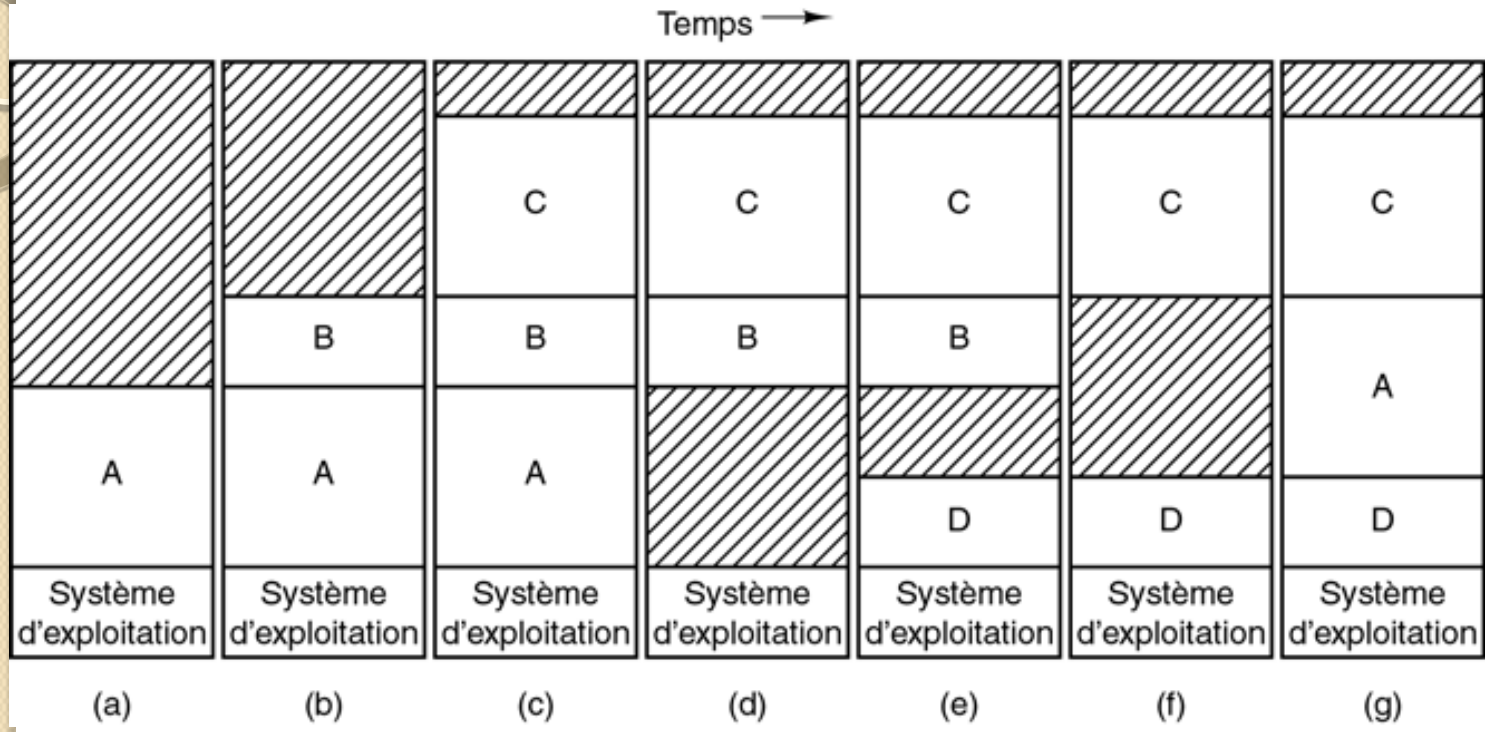
- Temps d'arrivé et travail pour 4 tâches
- Taux d'utilisation du CPU avec 80% d'entrée/sortie
- Succession des événements alors que les tâches arrivent et se terminent.

# Réallocation et protection

- On ne peut pas prévoir où en mémoire un programme sera chargé
  - Les adresses ne peuvent pas être absolues
  - Un programme ne doit pas adresser l'espace d'un autre

- Registre de base et de limite
  - Chaque adresse est additionnée au registre de base pour déterminer l'adresse physique
  - Il y a erreur si l'adresse est plus grande que la valeur du regisre de limite.

# Le va-et vient (swapping)

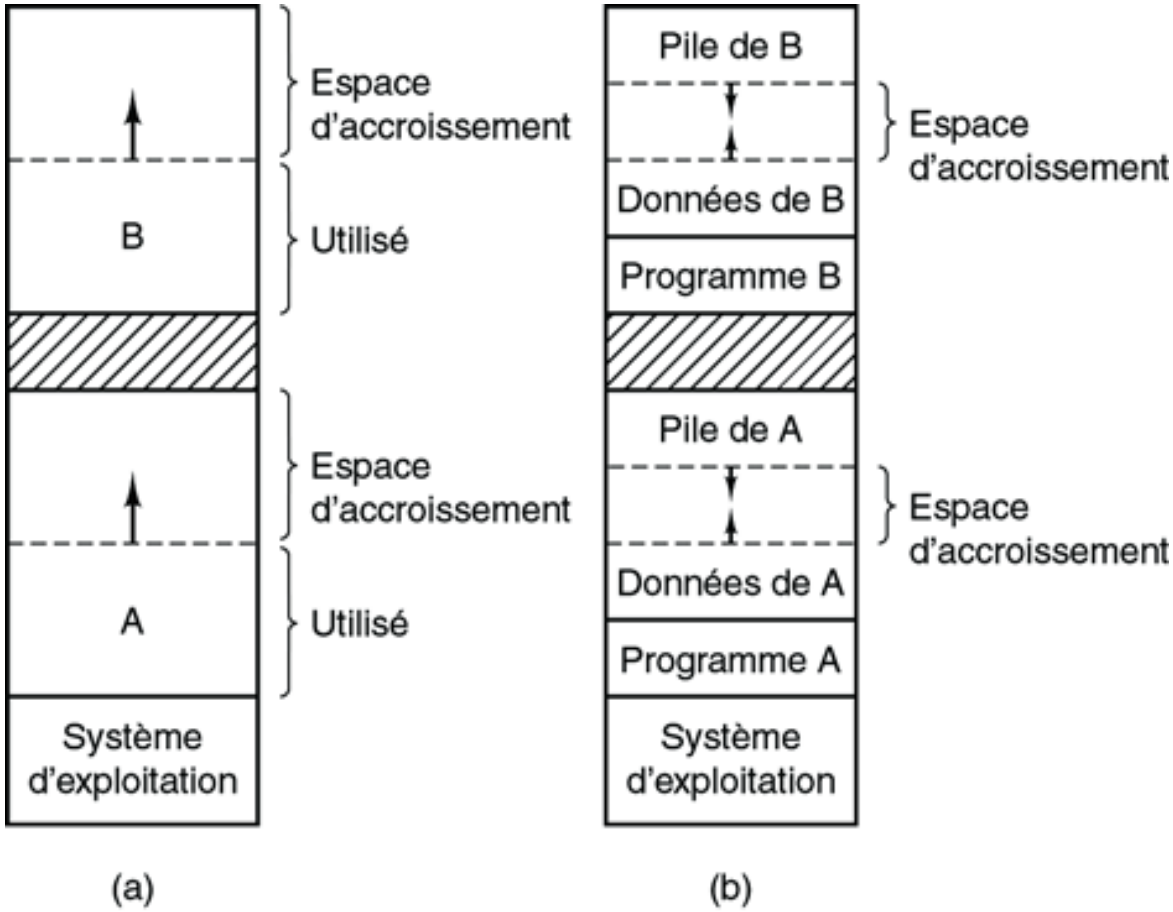



L'allocation de la mémoire change

- au gré des processus qui viennent en mémoire
- et quittent la mémoire

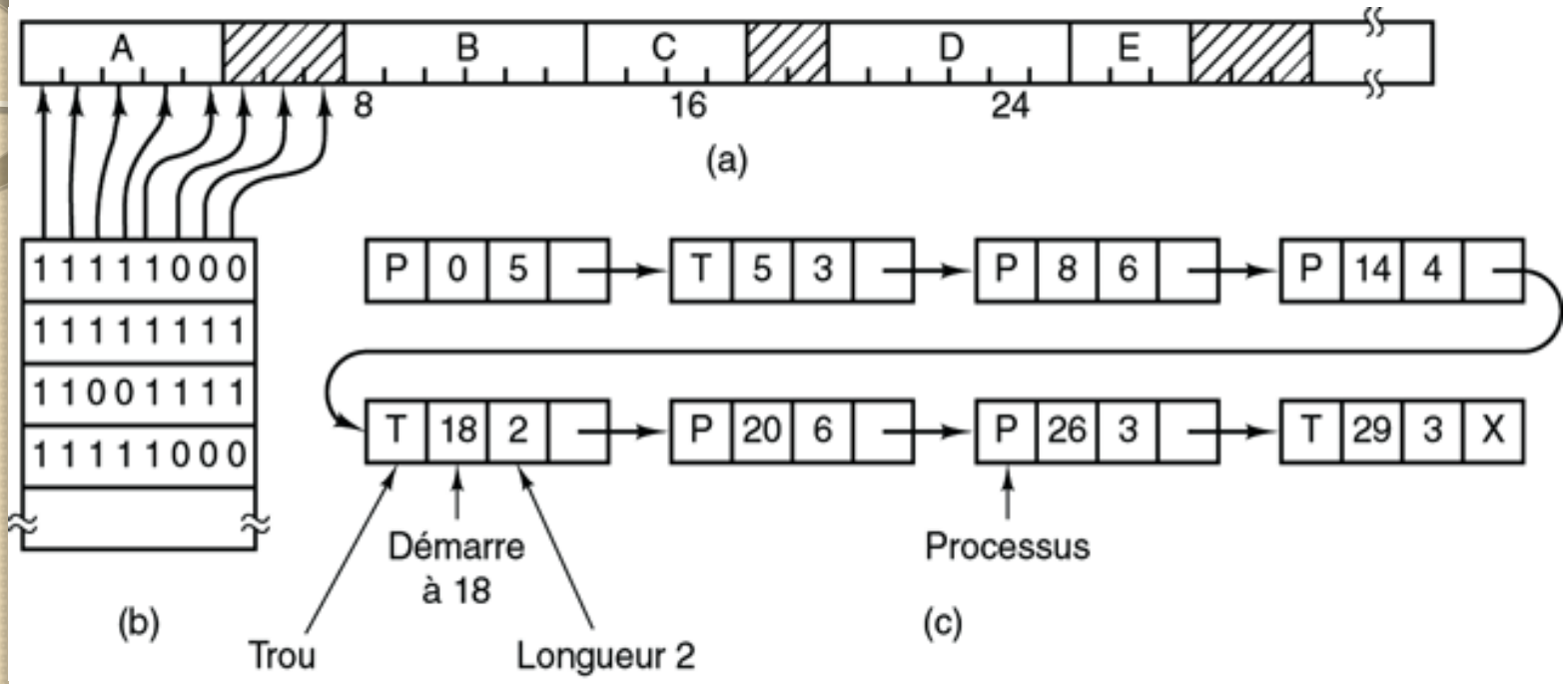
Les zones grises indiquent la mémoire non utilisée

# Le va-et-vient (2)



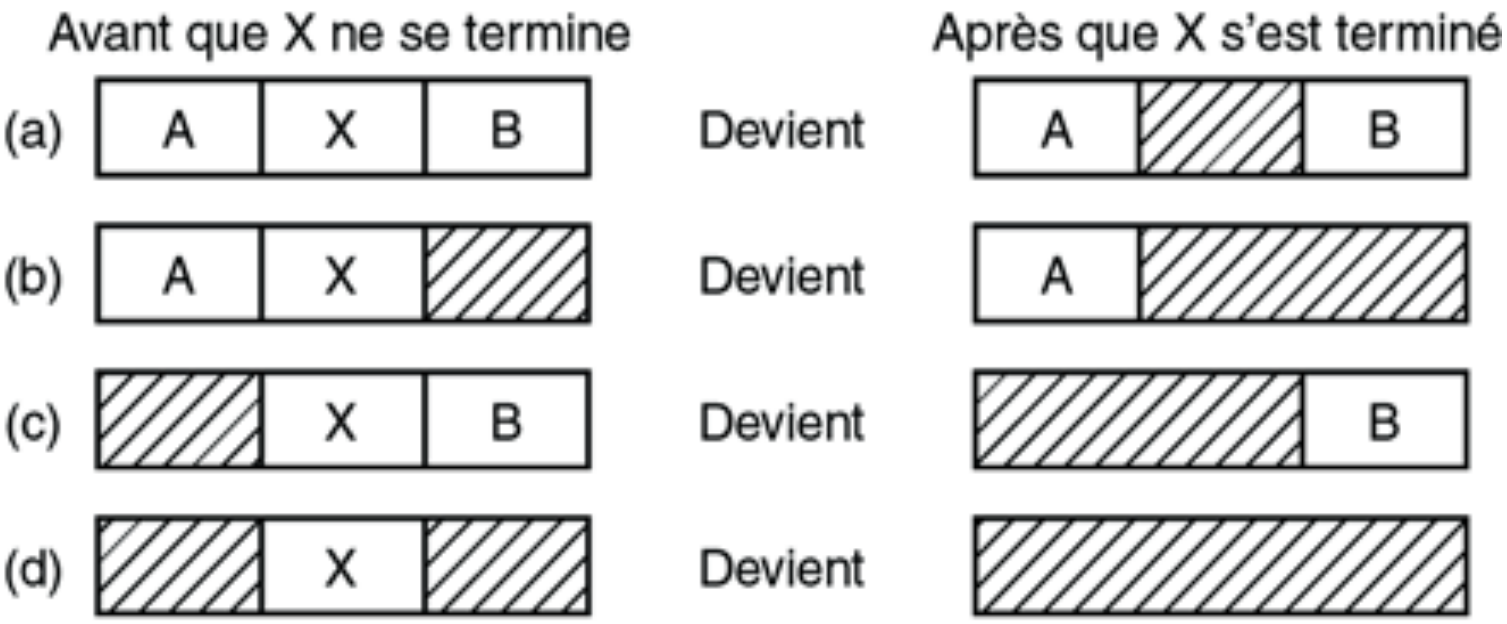

- Allocation d'espace pour l'accroissement du segment de données
- Allocation d'espace pour l'accroissement de la pile et du segment de données

# Gérer la mémoire avec les tableaux de bits




- Une partie de la mémoire avec 5 processus, 3 trous
  - Les petites marques verticales indiquent les **unités d'allocation**
  - Les zones grises sont libres
- Le tableau de bits correspondant
- Mêmes informations sous la forme d'une liste chaînée

# Gestion de la mémoire avec listes chaînées




Quatre combinaisons de voisinage pour un processus X qui se termine

# Gestion de la mémoire avec listes chaînées

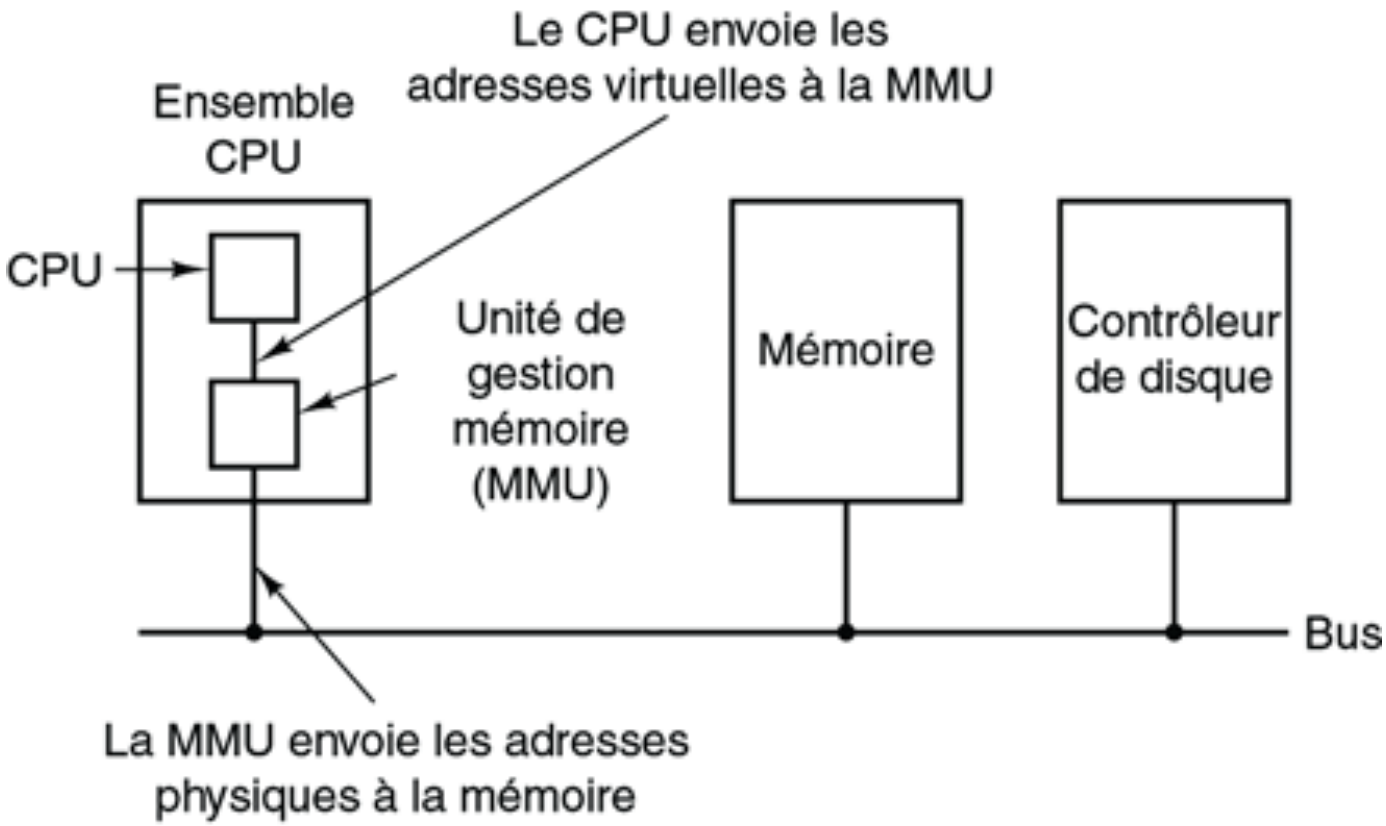
Algorithmes d'allocation de la mémoire:

1. Première zone libre (first fit)
2. Zone libre suivante (next fit)
3. Meilleur ajustement (best fit)
4. Plus grand résidu (worst fit)

Deux listes distinctes (zones libres et occupées)

- On peut utiliser les zones libres elle même pour la liste des zones libres.
- Algorithme de placement rapide: listes séparées pour certaines des tailles les plus communément demandées.

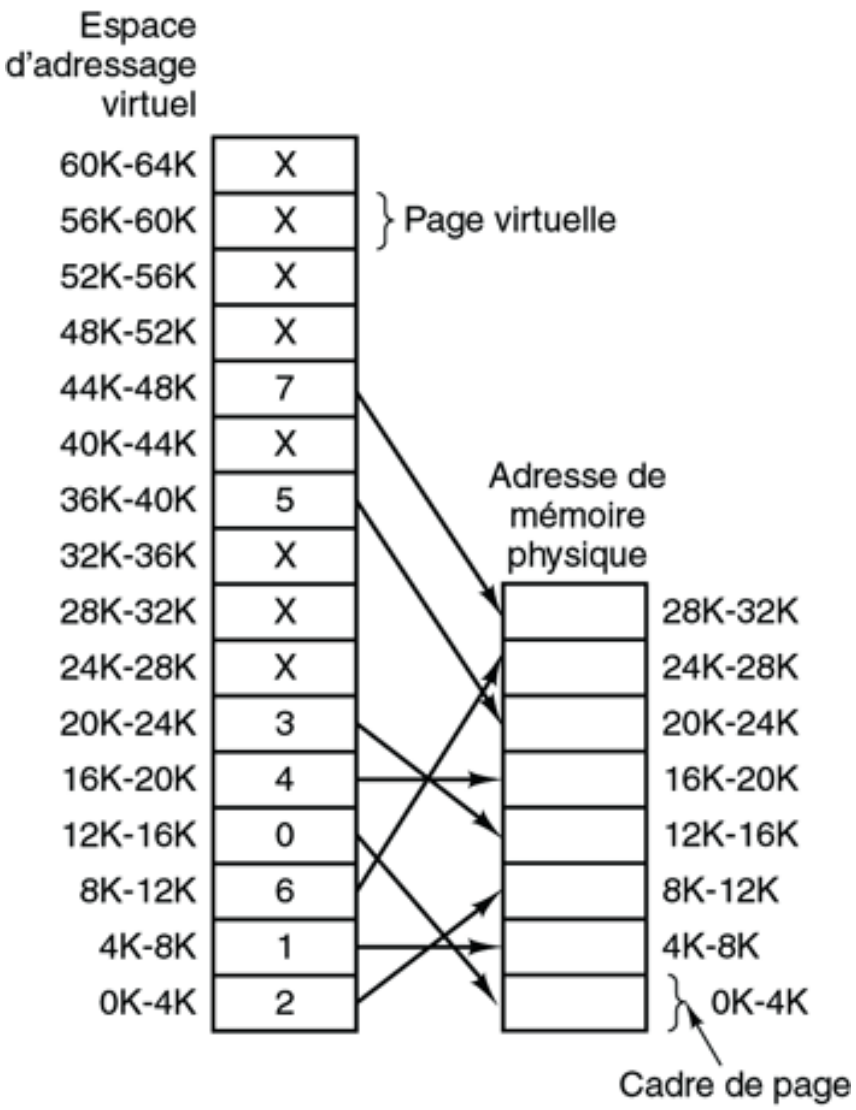
# Mémoire virtuelle
## La pagination (1)




Localisation et fonction du MMU

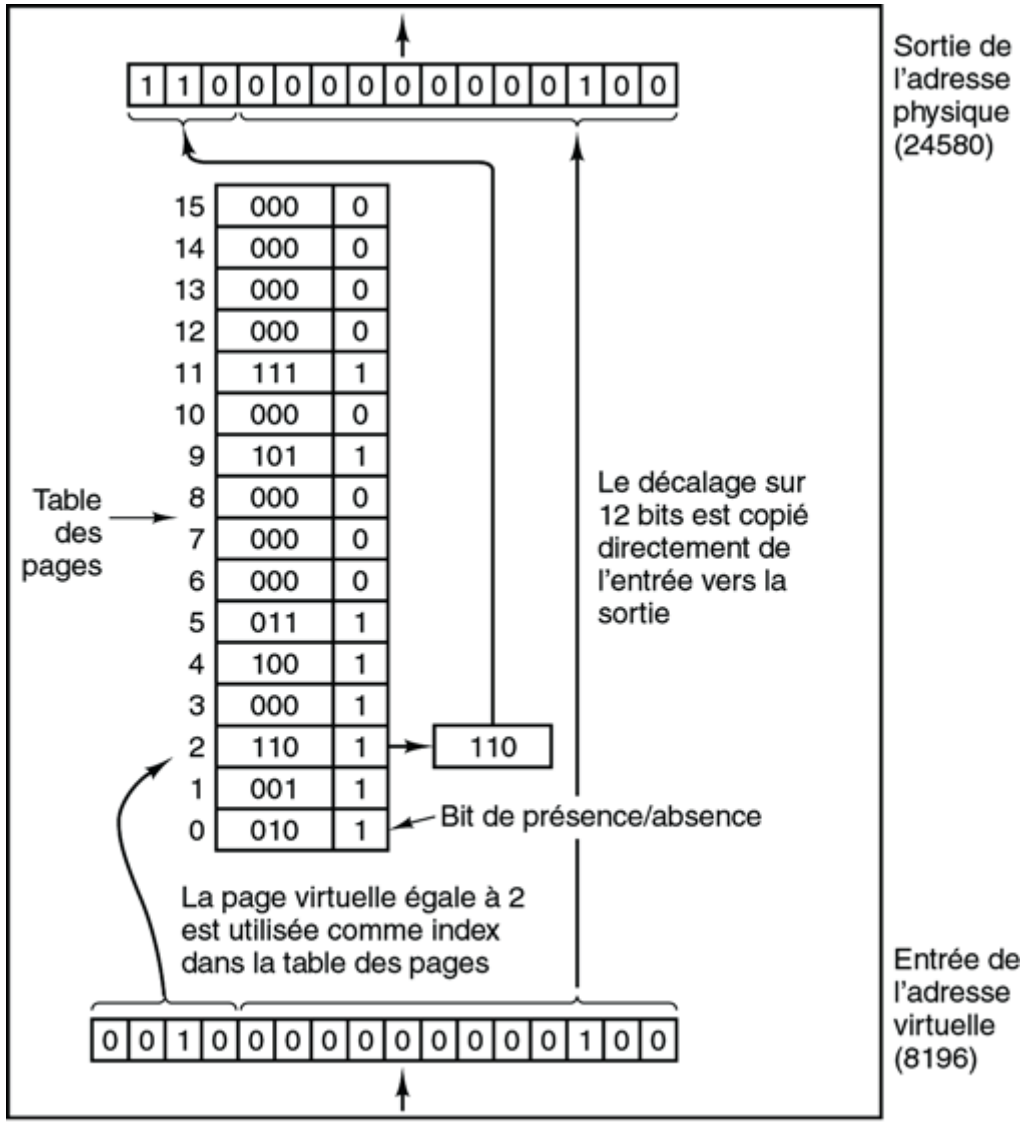
# La pagination (2)

La relation entre les adresses virtuelles et physiques est indiquée dans la table des pages



# Table des pages (1)

Fonctionnement interne du MMU avec 16 pages de 4Ko



# Table des pages (2)

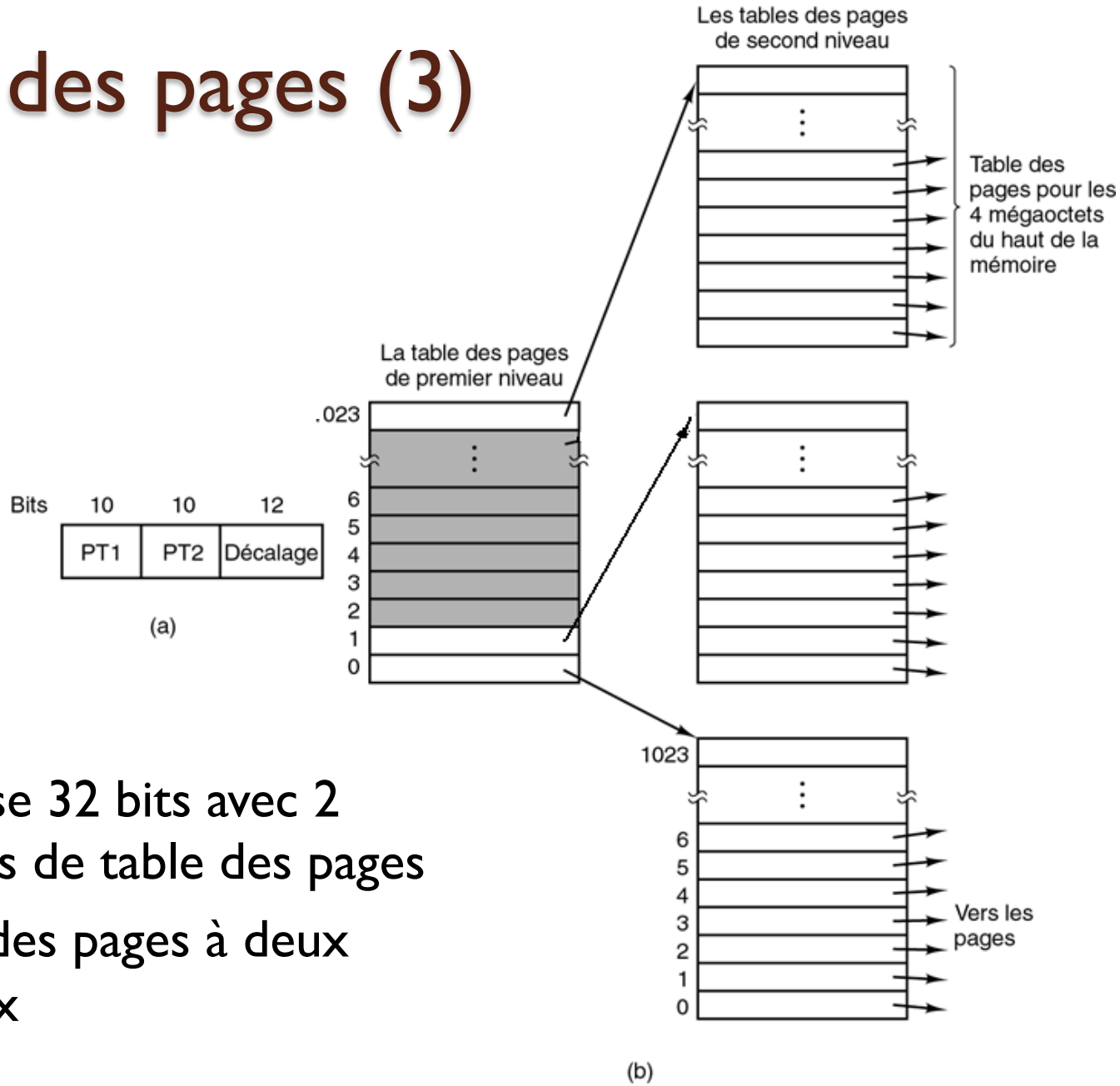
Deux problèmes:

1. La table des pages peut être très grande
2. Trouver l'adresse réelle à partir de l'adresse virtuelle doit se faire très rapidement

Deux cas extrèmes (et non applicables):

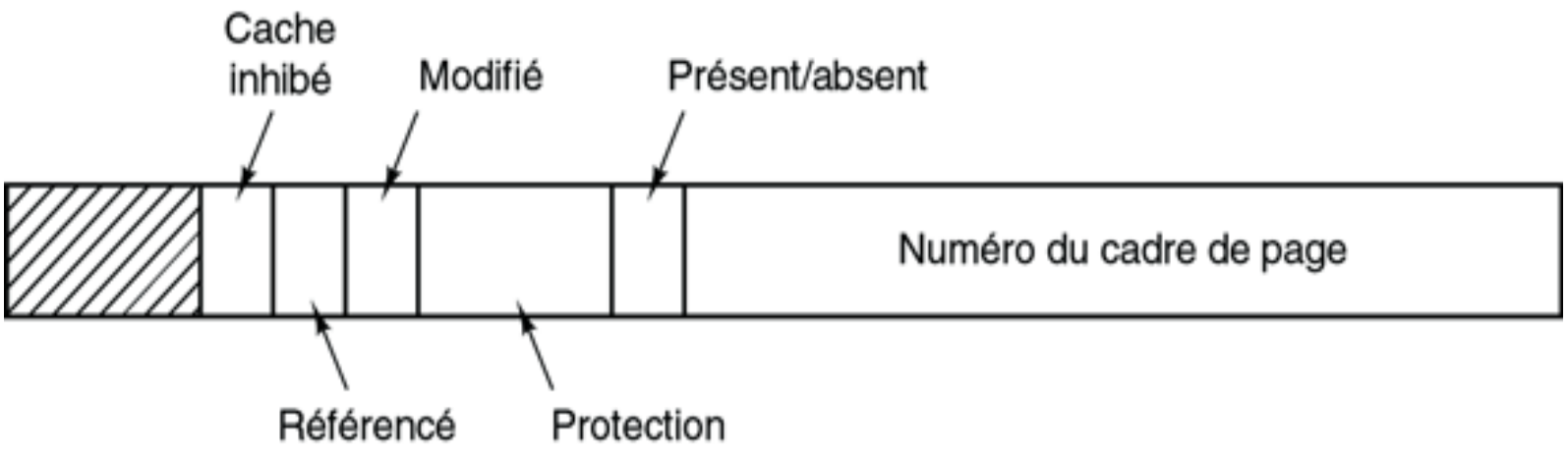
- La table est entièrement implémentée par le matériel à l'aide d'un tableau de registres.
- Un seul registre pointant sur le début de la table qui est entièrement chargée en mémoire.

# Table des pages (3)




- Adresse 32 bits avec 2 champs de table des pages
- Table des pages à deux niveaux

# Table des pages (4)



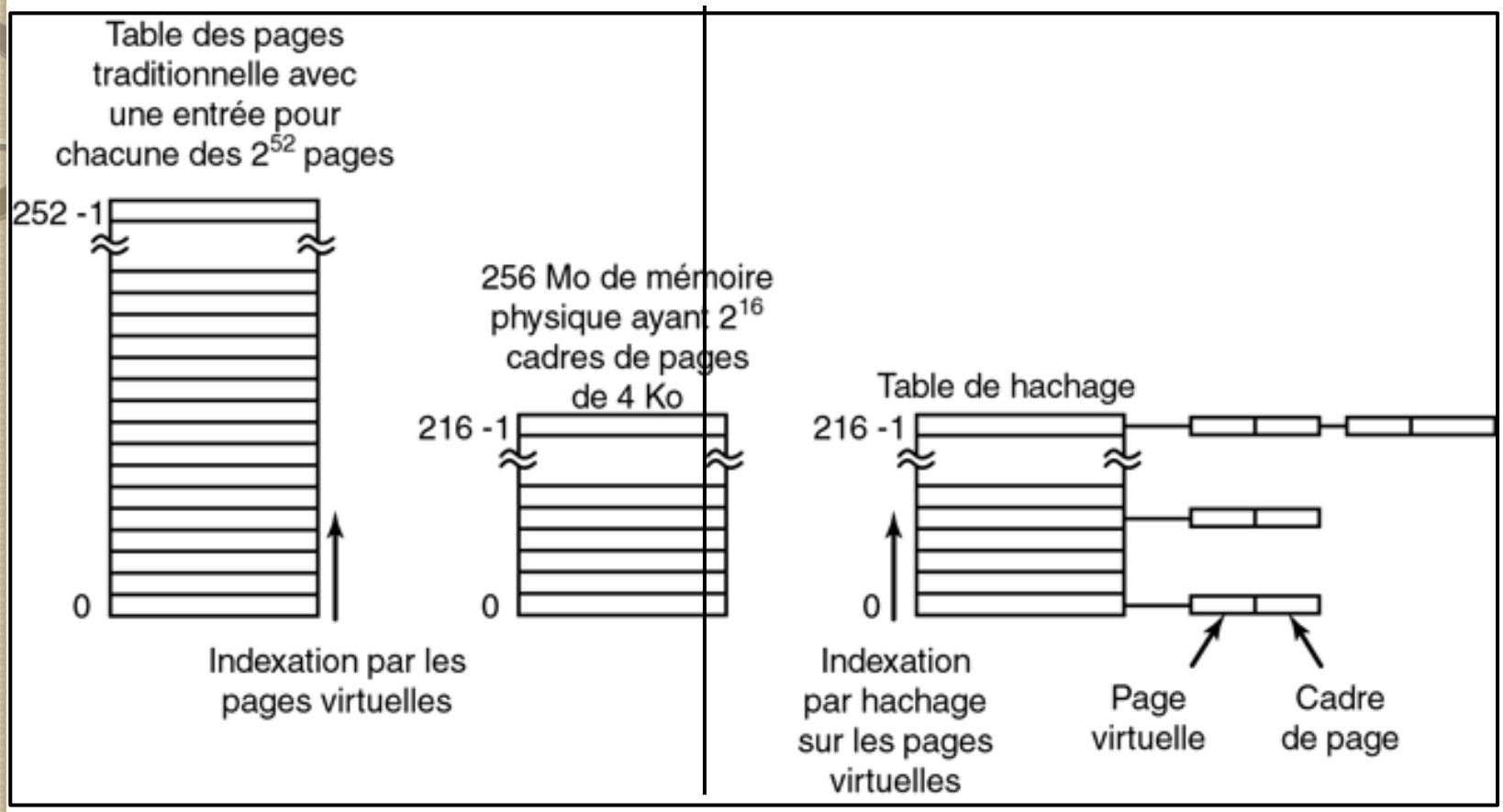

Entrée typique d'une table des pages

# TLBs – Translation Lookaside Buffers (Mémoire associative)

| Valide | Page virtuelle | Modifié | Protection | Cadre de page |
|---|---|---|---|---|
| 1 | 140 | 1 | RW | 31 |
| 1 | 20 | 0 | R X | 38 |
| 1 | 130 | 1 | RW | 29 |
| 1 | 129 | 1 | RW | 62 |
| 1 | 19 | 0 | R X | 50 |
| 1 | 21 | 0 | R X | 45 |
| 1 | 860 | 1 | RW | 14 |
| 1 | 861 | 1 | RW | 75 |

Une TLB pour accélérer la pagination

# Tables des pages inversées




- Comparaison entre une table des pages traditionnelle et une table des pages inversée.
- De plus en plus utilisé (surtout avec le développement des machines 64 bits)

# Algorithmes de remplacement de pages

- Une faute de page force le SE à choisir:
  - quelle page doit être enlevée
  - faire de la place pour la nouvelle page
- Les pages modifiées doivent d'abord être sauvegardées
  - sinon on peut simplement les écraser
- Il est préférable de ne pas choisir une page souvent utilisée
  - sinon il faudra la recharger bientôt

# L'algorithme optimal

- Remplacer la page qui sera utilisée après toutes les autres
  - Optimal mais irréalisable

- On exécute deux fois un programme
  - La seconde fois on a l'information nécessaire pour implémenter l'algorithme de remplacement de page optimal
  - Permet de comparer les performances d'algorithmes réalisable avec l'algorithme optimal.

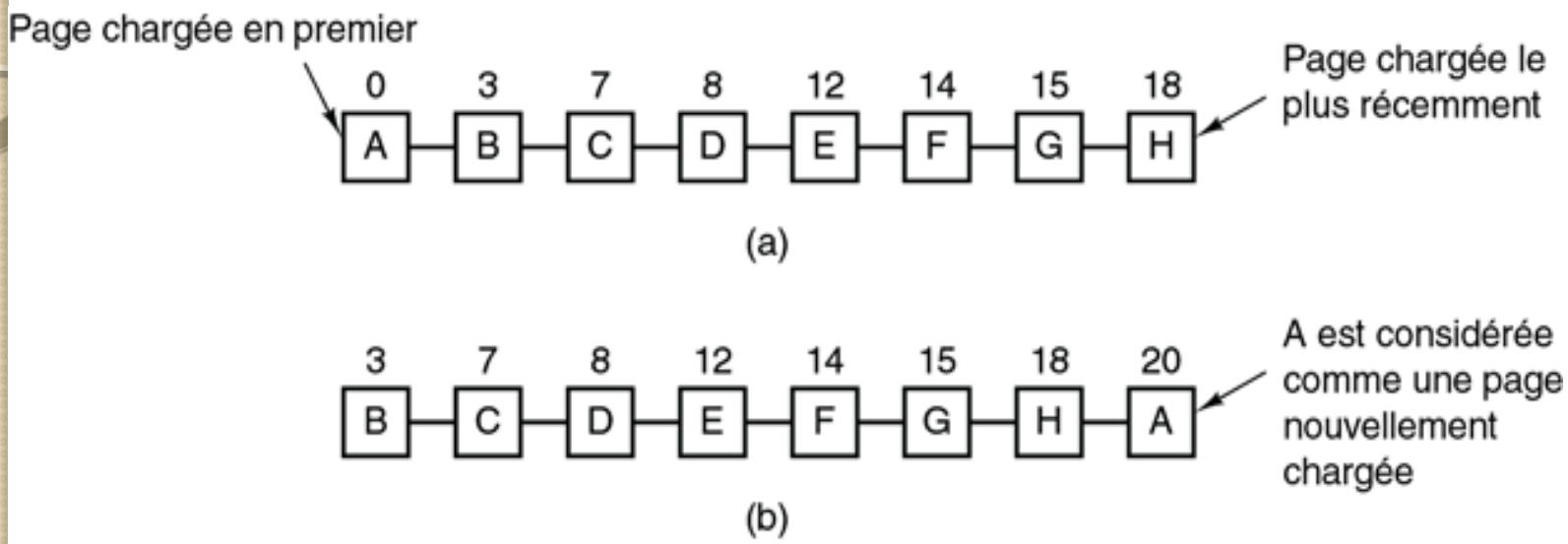
# Algorithme de remplacement de la page non récemment utilisée (NRU)

- Chaque page possède deux bits: R et M
  - ces bits sont mis à 1 lorsque la page est référencée ou modifiée (le bit R est remis à 0 périodiquement).
- 4 classes de pages
  1. non référencée, non modifiée
  2. non référencée, modifiée
  3. référencée, non modifiée
  4. référencée, modifiée
- NRU enlève une page au hazard
  - en partant du plus petit numéro de classe

# Algorithme premier arrivé, premier sorti

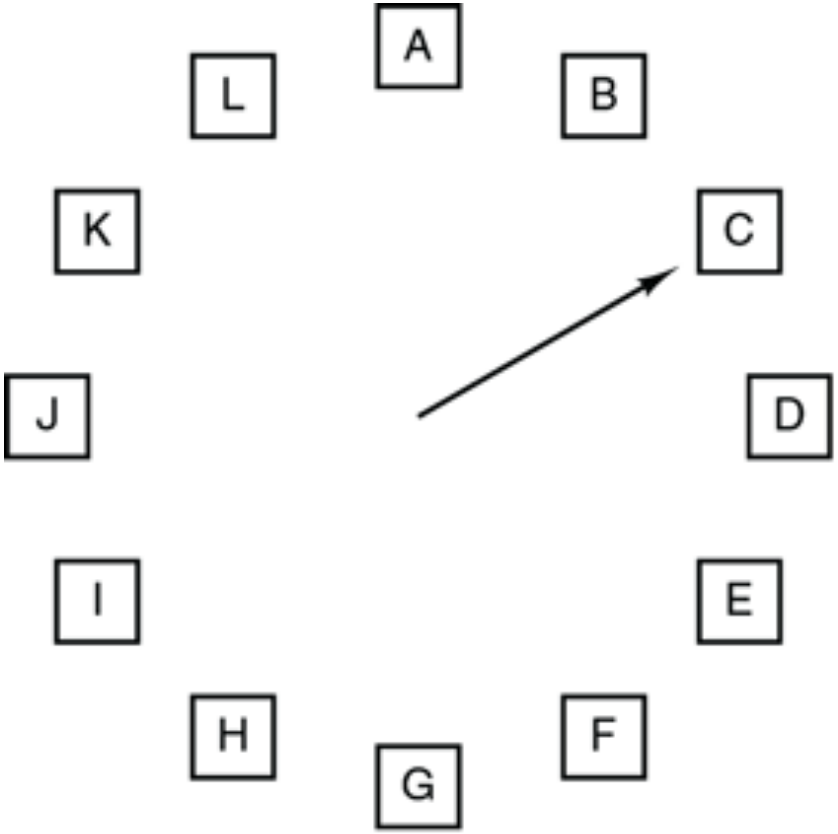
- On maintient un liste des pages
  - dans l'ordre de leur chargement en mémoire
- La page au début de la liste (la plus ancienne) est remplacée
- Désavantage
  - les vielle pages peuvent aussi être celles qui sont le plus utilisées.

# Algorithme de la seconde chance




- Opération *seconde chance:*
  - pages triées selon l'ordre d'arrivé
  - Si le bit R de la page la plus ancienne est 1 alors on la met à la fin de la liste comme une nouvelle page (son bit R à 0)

# Algorithme de l'horloge




Quand un défaut de page se produit, la page pointée est testée. L'action entreprise dépend de la valeur du bit R :
R = 0 : retirer la page
R = 1 : mettre R à 0 et avancer le pointeur

Comme l'algorithme précédent mais l'utilisation d'une liste circulaire évite de déplacer les pages.

# Algorithme de la page la moins récemment utilisée (LRU)

- On suppose que les pages qui ont été utilisées récemment le seront à nouveau prochainement
  - on enlève les pages qui n'on pas été utilisées depuis longtemps

- Tenir à jour une liste chaînée de pages
  - page la plus récente en premier
  - mettre à jour cette liste à <u>chaque référence mémoire</u> !!

- On utilise un compteur C incrémenté après chaque instruction
  - on copie C dans la page des tables pour l'entrée de la page référencées
  - choisir la page avec la plus petite valeur de compteur
  - on remet le compteur à 0 périodiquement

# Algorithme de la page la moins récemment utilisée (LRU)

- Autre solution matérielle
  - Pour une machine à n cadre
  - Matrice n×n
  - Quand une page k est référencée
    - mettre à 1 tous les bits de la rangée k
    - mettre à 0 tous les bits de la colonnne k

# Exemple – référence des pages: 0,1,2,3,2,1,0,3,2,3

Page

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |

(a)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

(b)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(c)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(d)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(e)

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 |

(f)

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(g)

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(h)

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(i)

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

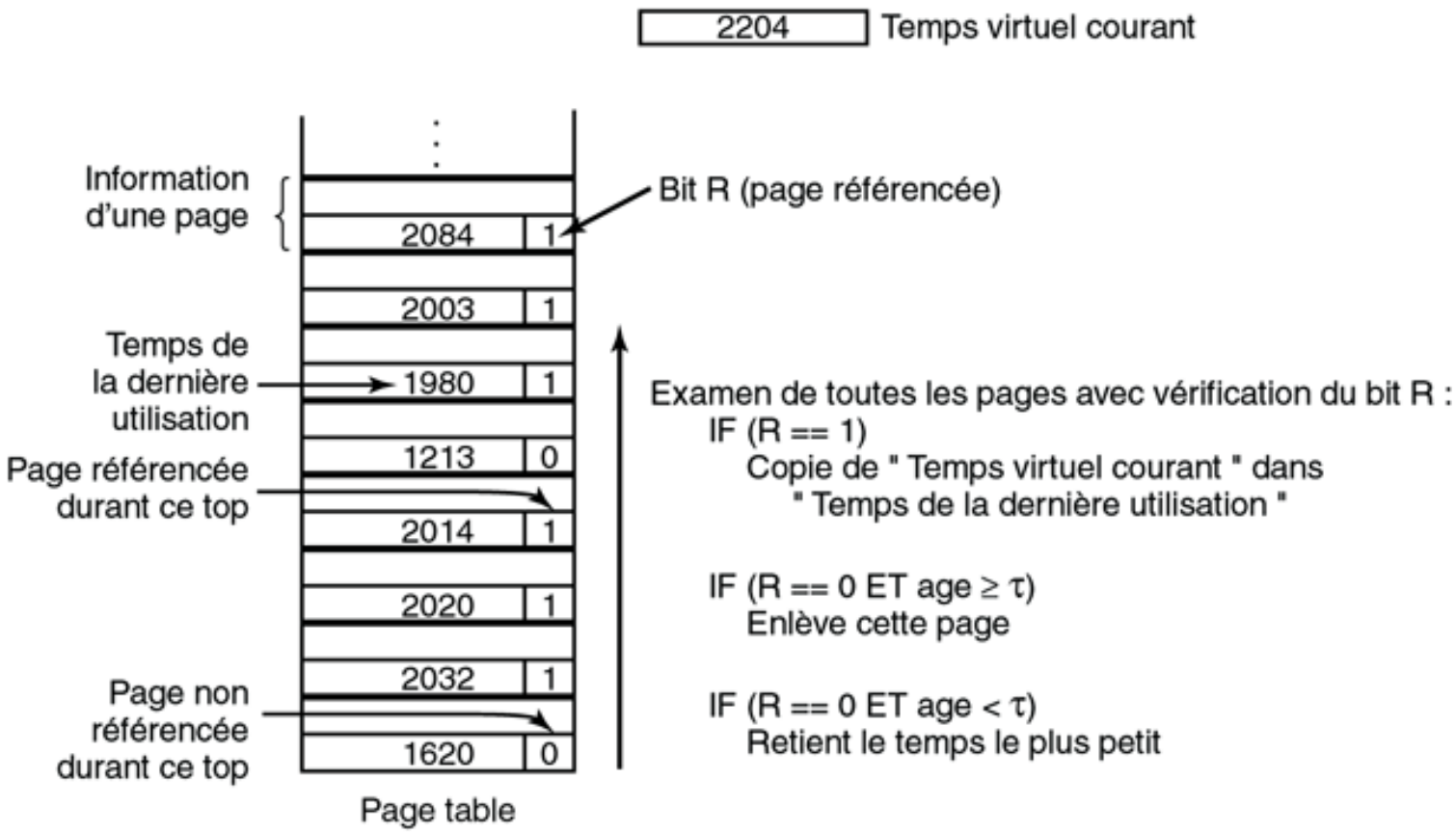
(j)

# Algorithme du vieillissement

| | Bits R des pages 0 à 5 au top d'horloge 0 | Bits R des pages 0 à 5 au top d'horloge 1 | Bits R des pages 0 à 5 au top d'horloge 2 | Bits R des pages 0 à 5 au top d'horloge 3 | Bits R des pages 0 à 5 au top d'horloge 4 |
|---|---|---|---|---|---|
| | 1 0 1 0 1 1 | 1 1 0 0 1 0 | 1 1 0 1 0 1 | 1 0 0 0 1 0 | 0 1 1 0 0 0 |
| Page | | | | | |
| 0 | 10000000 | 11000000 | 11100000 | 11110000 | 01111000 |
| 1 | 00000000 | 10000000 | 11000000 | 01100000 | 10110000 |
| 2 | 10000000 | 01000000 | 00100000 | 00100000 | 10001000 |
| 3 | 00000000 | 00000000 | 10000000 | 01000000 | 00100000 |
| 4 | 10000000 | 11000000 | 01100000 | 10110000 | 01011000 |
| 5 | 10000000 | 01000000 | 10100000 | 01010000 | 00101000 |
| | (a) | (b) | (c) | (d) | (e) |

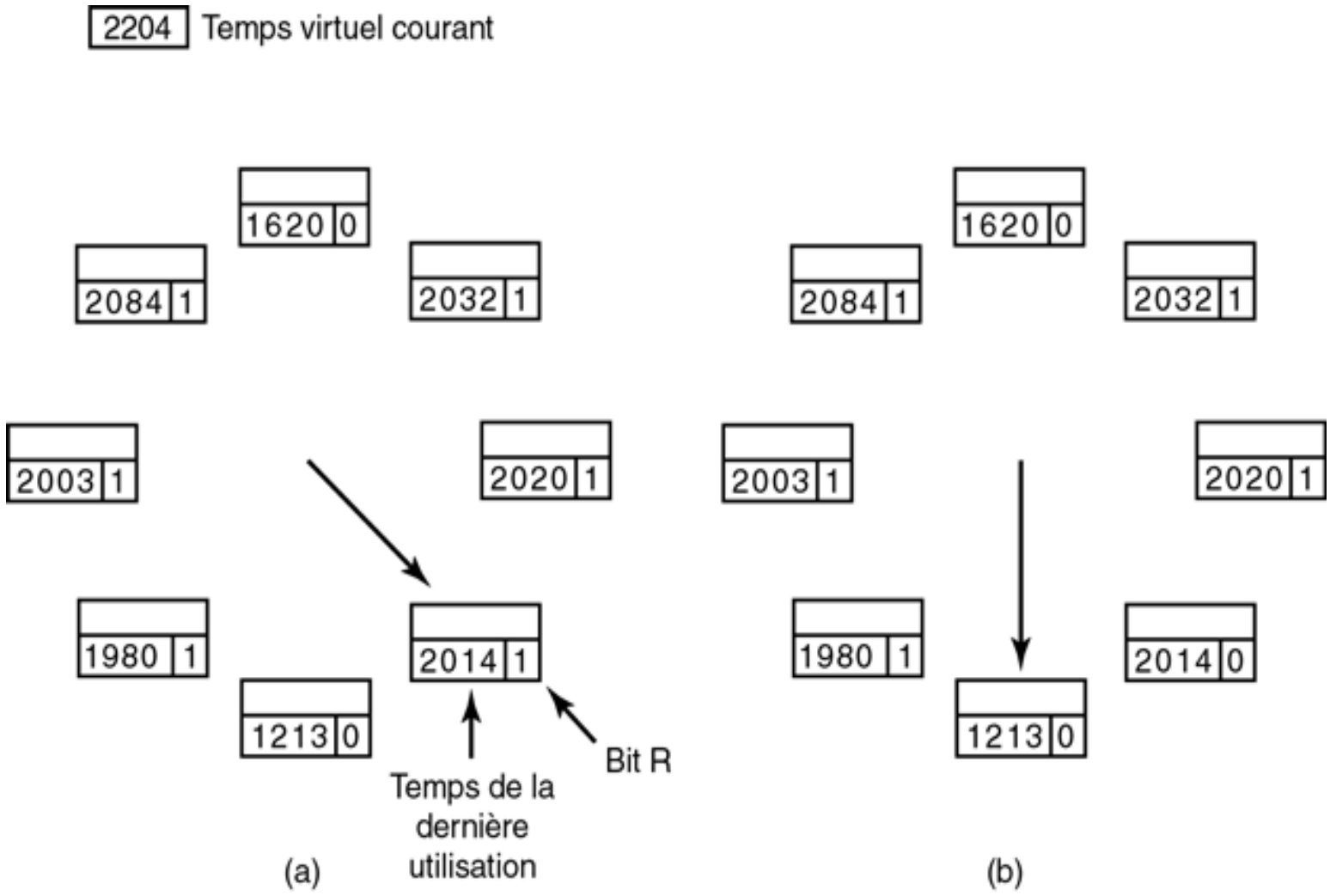
- Un compteur logiciel par page (initialisé à 0)
- À chaque interruption d'horloge, on examine toutes les entrées de la table des pages
- On décale à droite le compteur et on ajoute à gauche le bit R

# L'algorithme de l' *ensemble de travail*



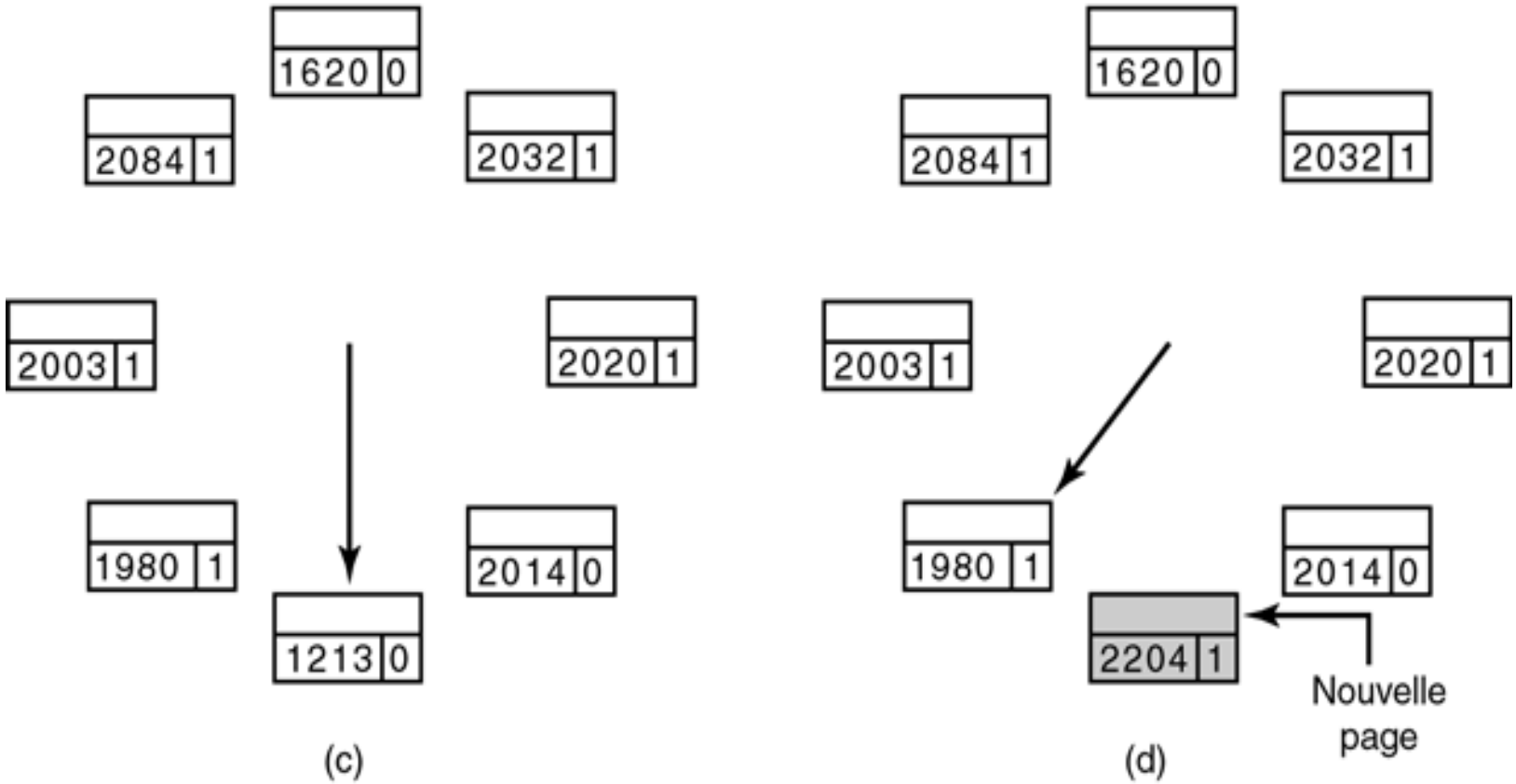

- Pour un entier k (ex. 10 millions), on considère les pages référencées au cours des k dernières références mémoire.
- Il est souvent plus simple de considérer les pages référencées au cours des t (ex. 100) dernières millisecondes.

# L'algorithme WSClock
## (Working Set Clock)




Se qui se passe lorsque R=1

# L'algorithme WSClock




Se qui se passe lorsque R=0

# Résumé des algorithmes de remplacement de pages

| Algorithme | Commentaires |
|---|---|
| Optimal | *Utilisé comme banc d'essai* |
| NRU | *Très grossier* |
| FIFO | *Peut écarter des pages importantes* |
| Seconde chance | *Grande amélioration de FIFO* |
| Horloge | *Réaliste* |
| LRU | *Excellent mais difficile à implanter* |
| NFU | *Approximation grossière de LRU* |
| Vieillissement | *Approximation efficace de LRU* |
| Ensemble de travail | *Coûteux* |
| WSClock | *Bonne efficacité* |

# Anomalie de Belady

Tous les cadres sont initialement vides

| | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page la plus récente | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 4 | 4 | 2 | 3 | 3 |
| | | | 0 | 1 | 2 | 3 | 0 | 1 | 1 | 1 | 4 | 2 | 2 |
| Page la plus ancienne | | | | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 1 | 4 | 4 |
| | | | P | P | P | P | P | P | P | | P | P | |

9 défauts de pages

(a)

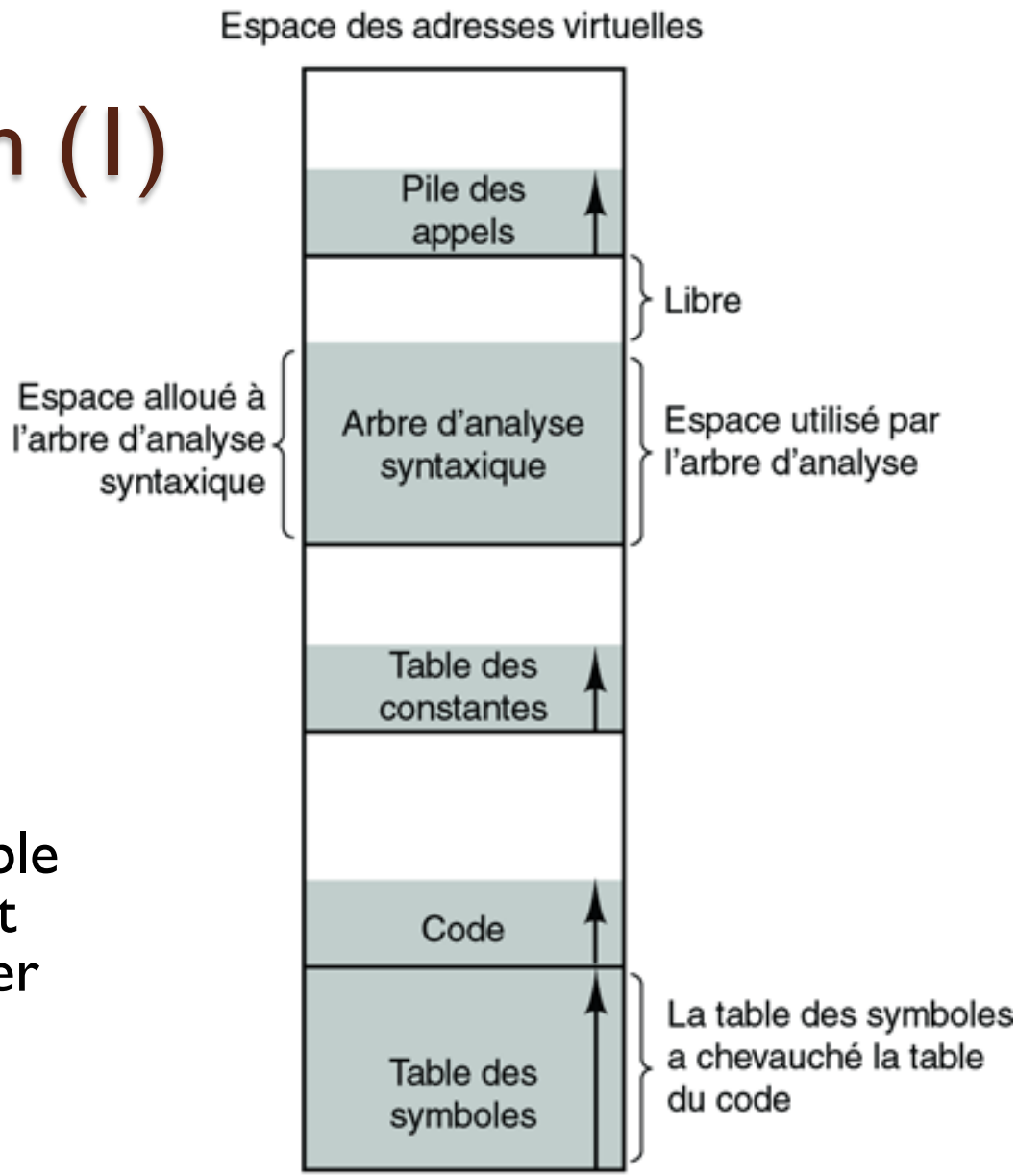
| | | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page la plus récente | | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |
| | | | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 0 | 1 | 2 | 3 |
| Page la plus ancienne | | | | 0 | 1 | 1 | 1 | 2 | 3 | 4 | 0 | 1 | 2 |
| | | | | | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 |
| | | | P | P | P | P | | P | P | P | P | P | P |

10 défauts de pages

(b)

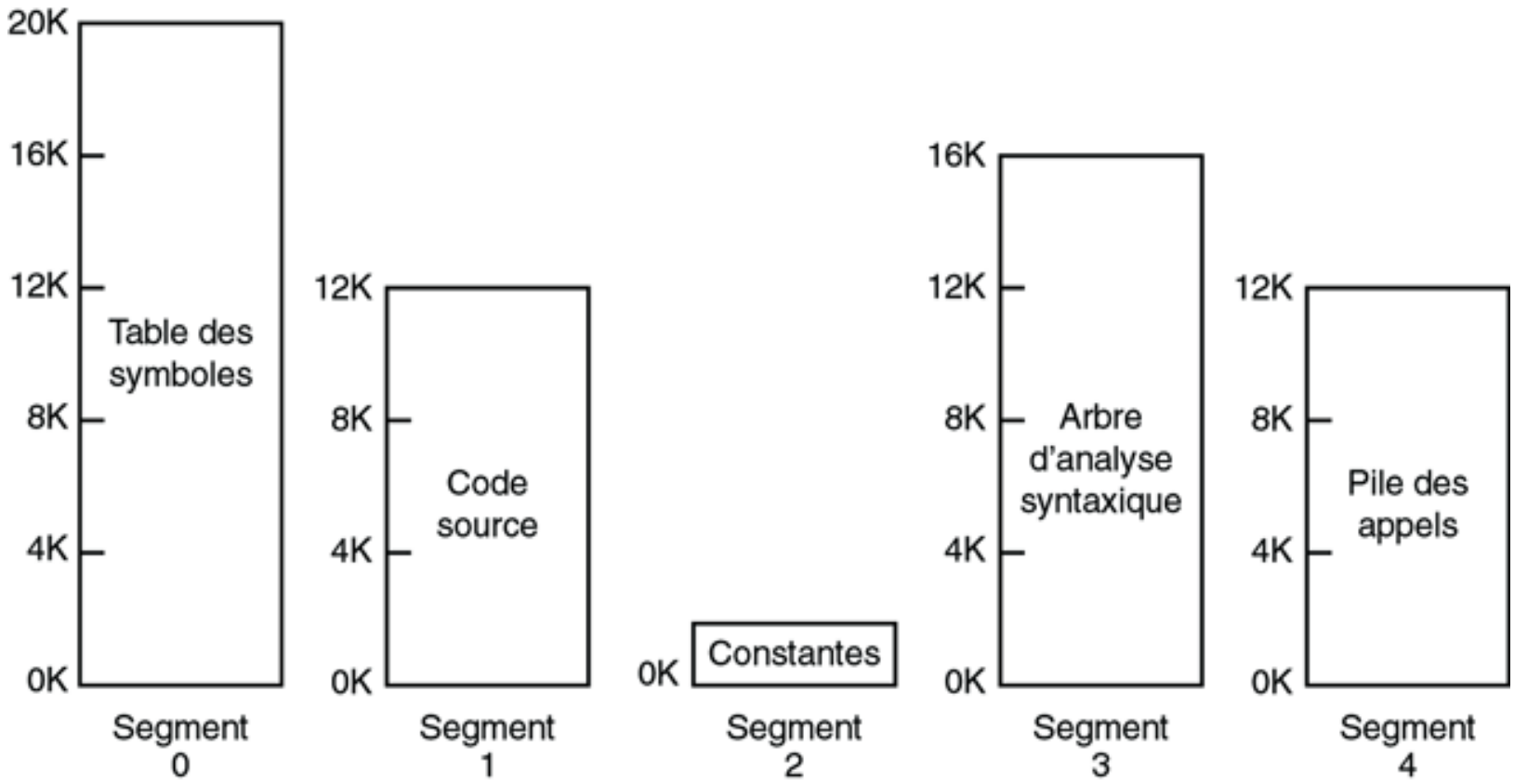
- Algorithme FIFO avec 3 cadres de pages
- Algorithme FIFO avec 4 cadres de pages
- La lettre P indique les fautes de pages
- L'algorithme LRU n'a pas fce problème

# Segmentation (1)




Dans un espace d'adressage à une dimension, une table des pages qui croît peut en chevaucher une autre.

# Segmentation (2)




Une mémoire segmentée permet à chaque table de croître ou de diminuer indépendamment des autres.

# Segmentation (3)

| Considérations | Pagination | Segmentation |
|---|---|---|
| Le programmeur doit-il connaître la techique utilisée? | Non | Oui |
| Combien y a-t-il d'espaces d'adressage linéaires? | 1 | Plusieurs |
| L'espace total d'adressage peut-il dépasser la taille de la mémoire physique? | Oui | Oui |
| Les procédures et les données peuvent-elles être séparées et protégées séparément? | Non | Oui |
| Peut-on traiter facilement des tables dont les tailles varient? | Non | Oui |
| Le partage de procédures entre utilisateurs est-il simplifié? | Non | Oui |

Comparaison entre la segmentation et la pagination

# Segmentation (4)

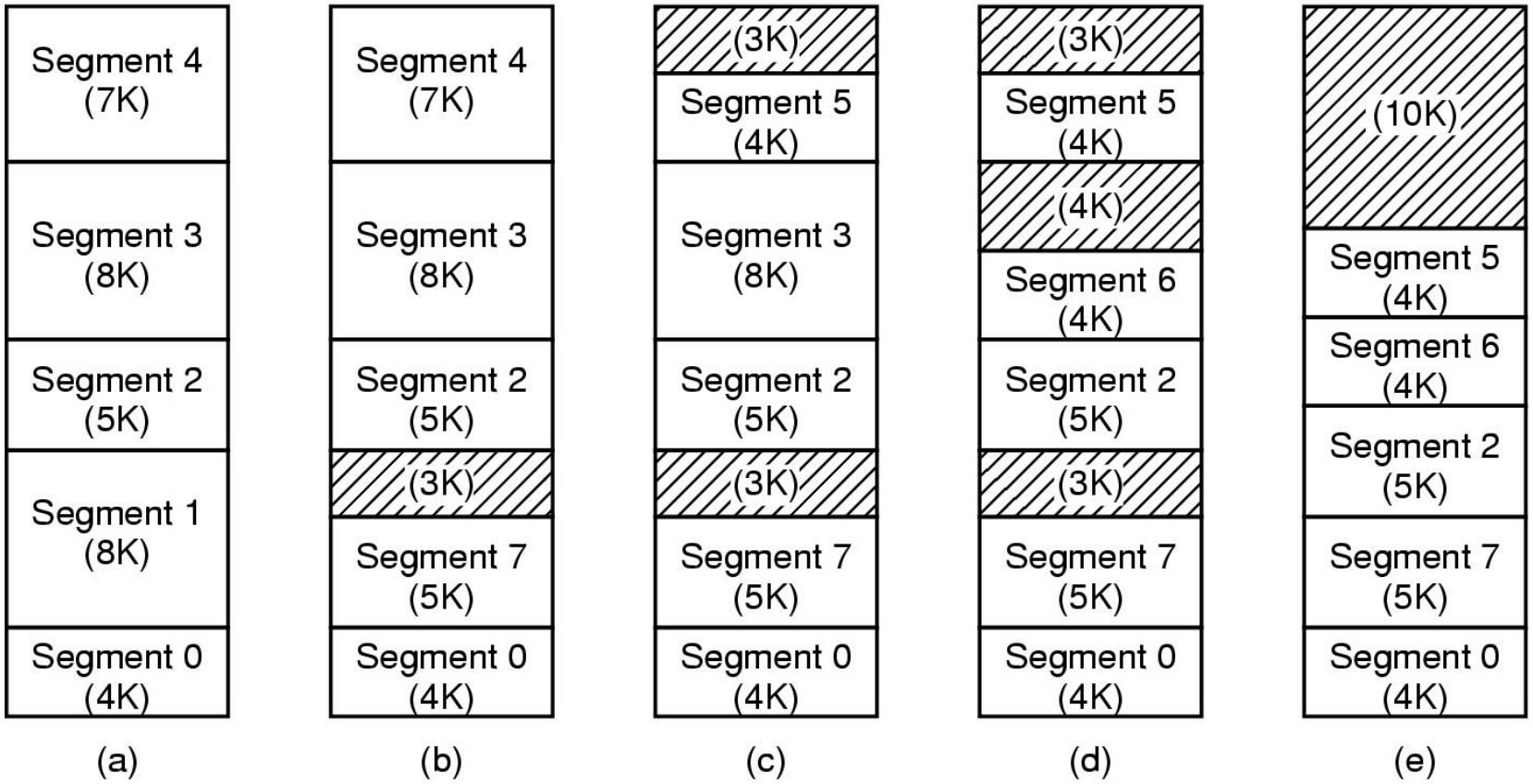
**Pagination:**

- 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.

# Implémentation de la segmentation pure




(a)-(d) Dévelopement de la fragmentation externe
(e) Compactage

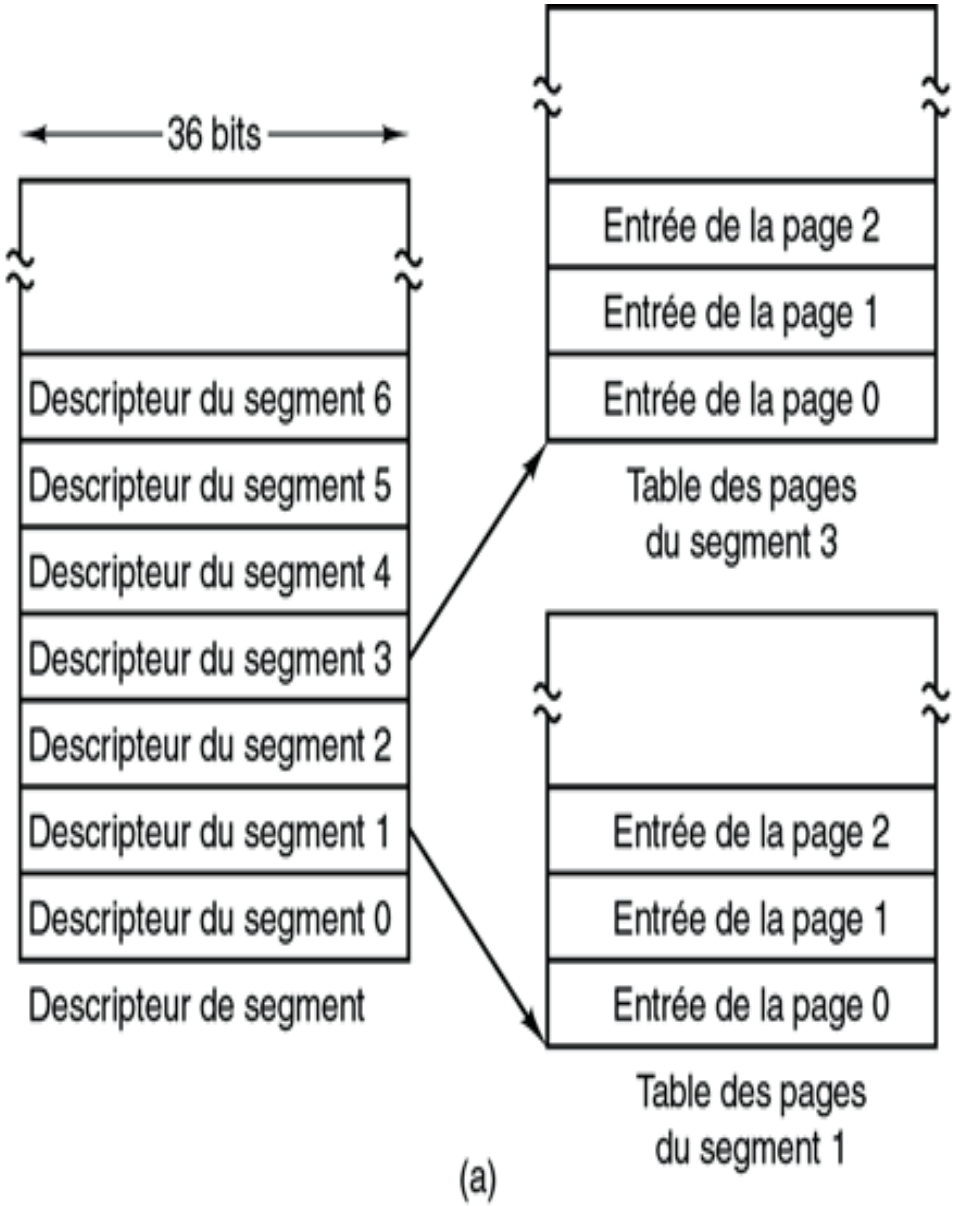
# Segmentation avec pagination: MULTICS (1)

Le descripteur de segment pointe sur les tables de pages:

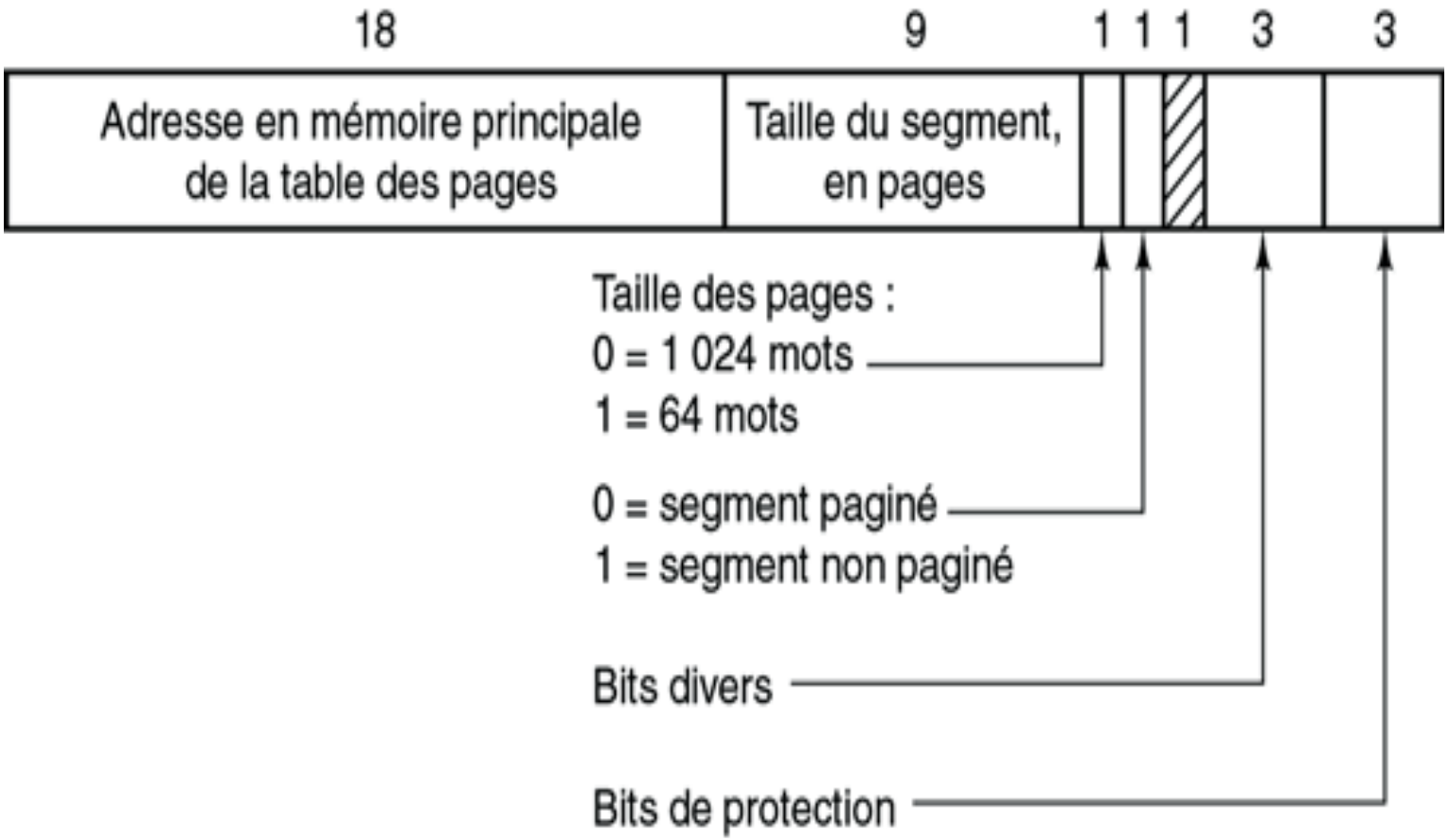
256K = 262144 segments

64K mots de 36 bits par segment

Adresse physique sur 24 bits

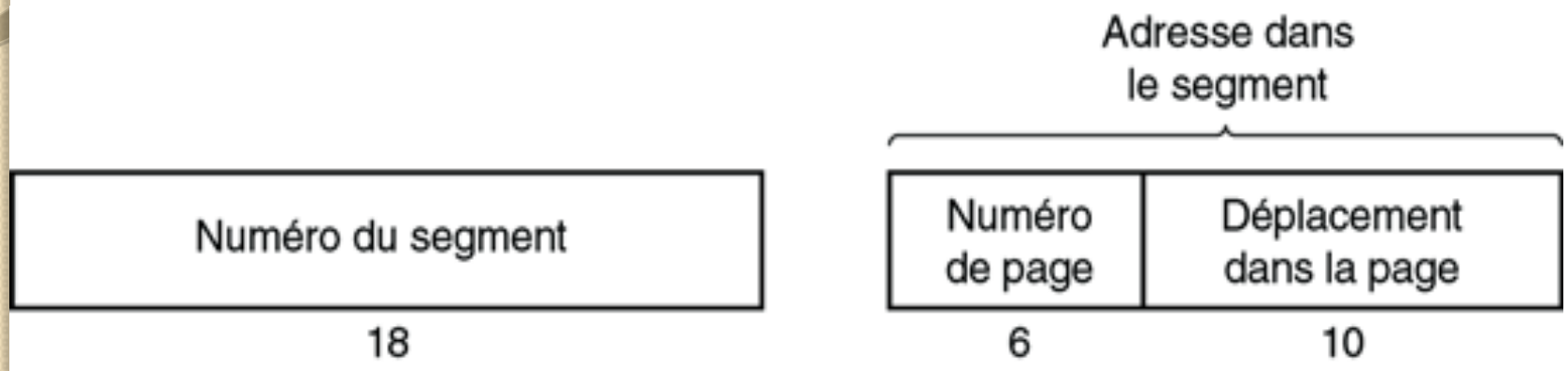


# Segmentation avec pagination: MULTICS (2)



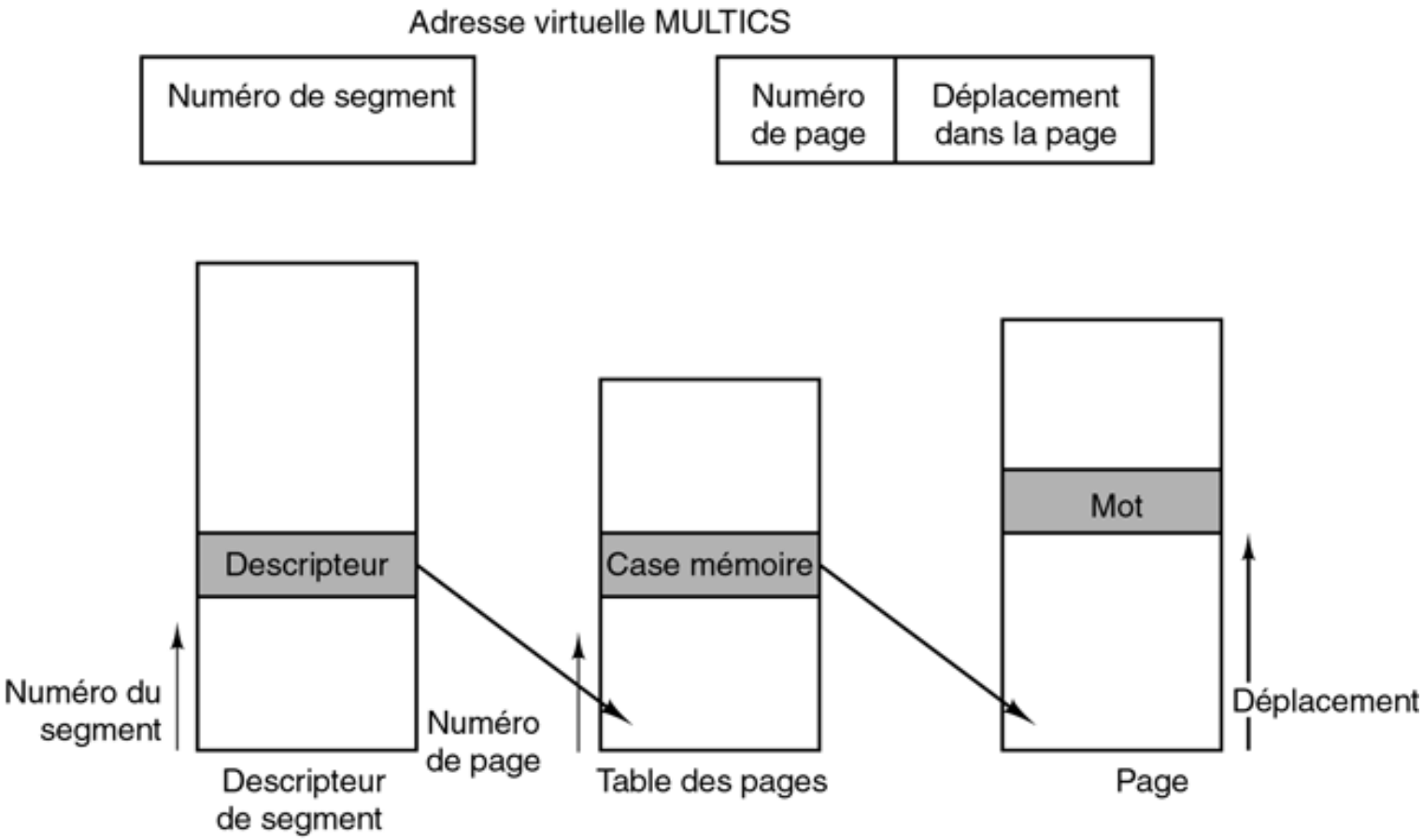

Un descripteur de segment.
L'adresse est complétée 6 zéros

# Segmentation avec pagination: MULTICS (3)




Une adresse virtuelle MULTICS sur 34 bits

# Segmentation avec pagination: MULTICS (4)




Conversion d'une adresse virtuelle MULTICS en une adresse réelle

# Segmentation avec pagination: MULTICS (5)

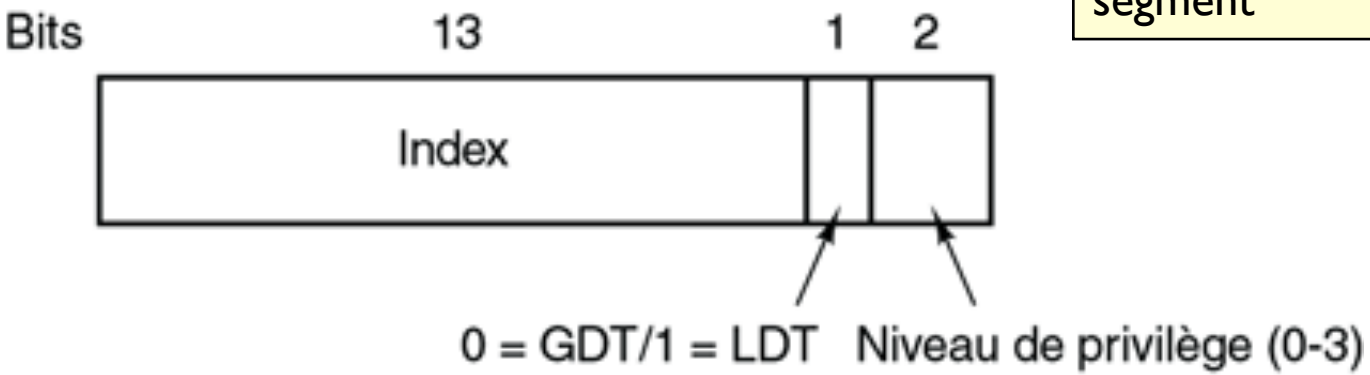
| Champ de comparaison | | | | | Entrée utilisée ? |
|---|---|---|---|---|---|
| Numéro de segment | Page virtuelle | Case mémoire | Protection | Age | |
| 4 | 1 | 7 | Lecture/écriture | 13 | 1 |
| 6 | 0 | 2 | Lecture seule | 10 | 1 |
| 12 | 3 | 1 | Lecture/écriture | 2 | 1 |
| | | | | | 0 |
| 2 | 1 | 0 | Exécution seule | 7 | 1 |
| 2 | 2 | 12 | Exécution seule | 9 | 1 |
| | | | | | |

Version simplifiée de la TLB de MULTICS

# Segmentation avec pagination
# Le Pentium d'Intel (1)

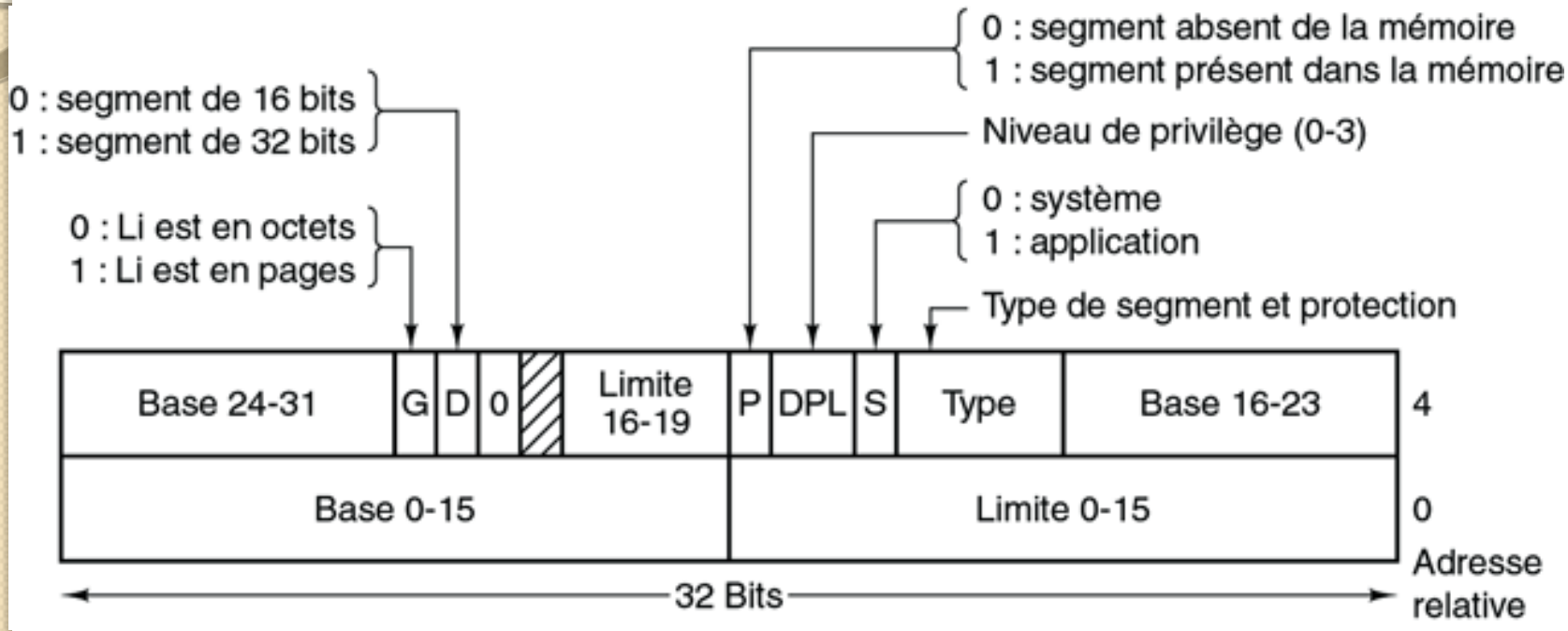
16K segments

2M mots de 32 bits par segment



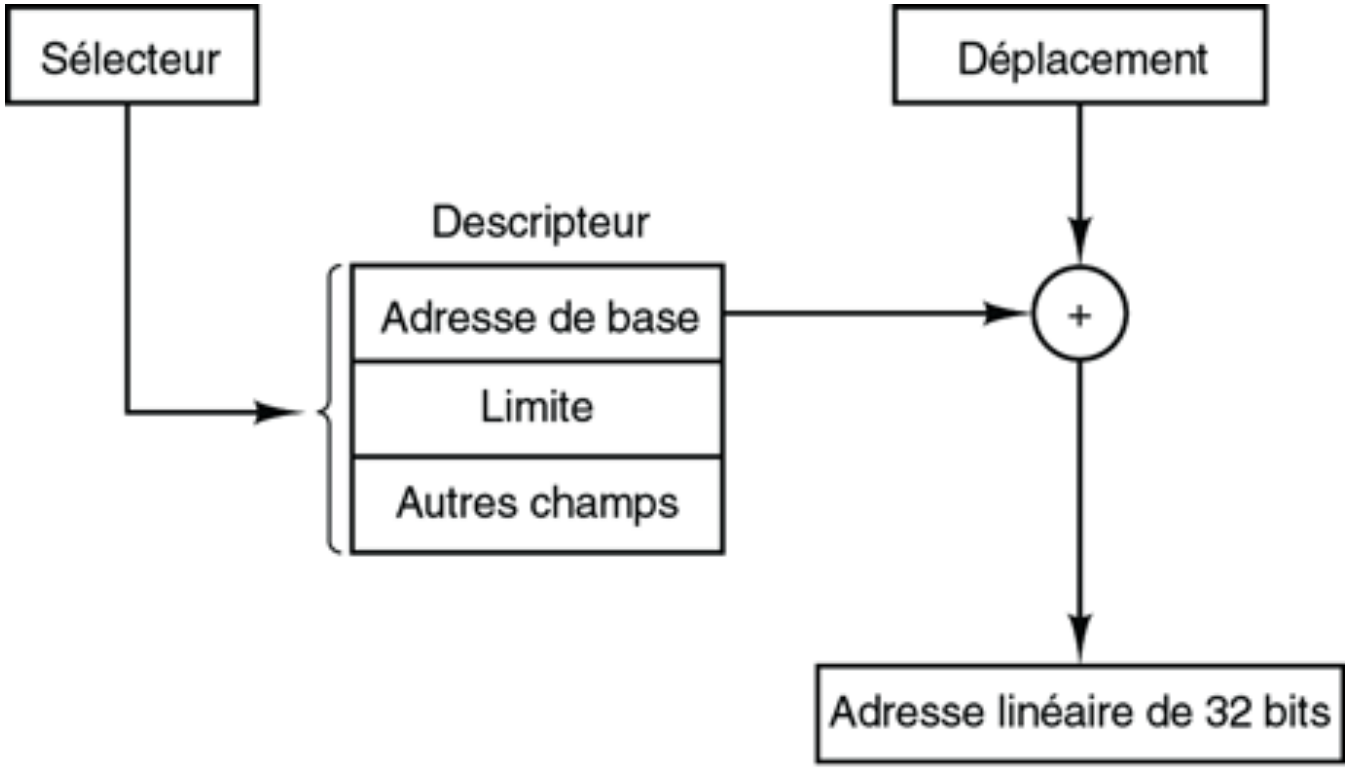

- Sur un Pentium une entrée de la table de segments est appelée *sélecteur*.
- Utilise 2 tables (en mémoire):
  - LDT (Local Descriptor Table)
  - GDT (Global Descriptor Table)
- 6 registres de segments (CS, DS, etc.)

# Le Pentium d'Intel (2)



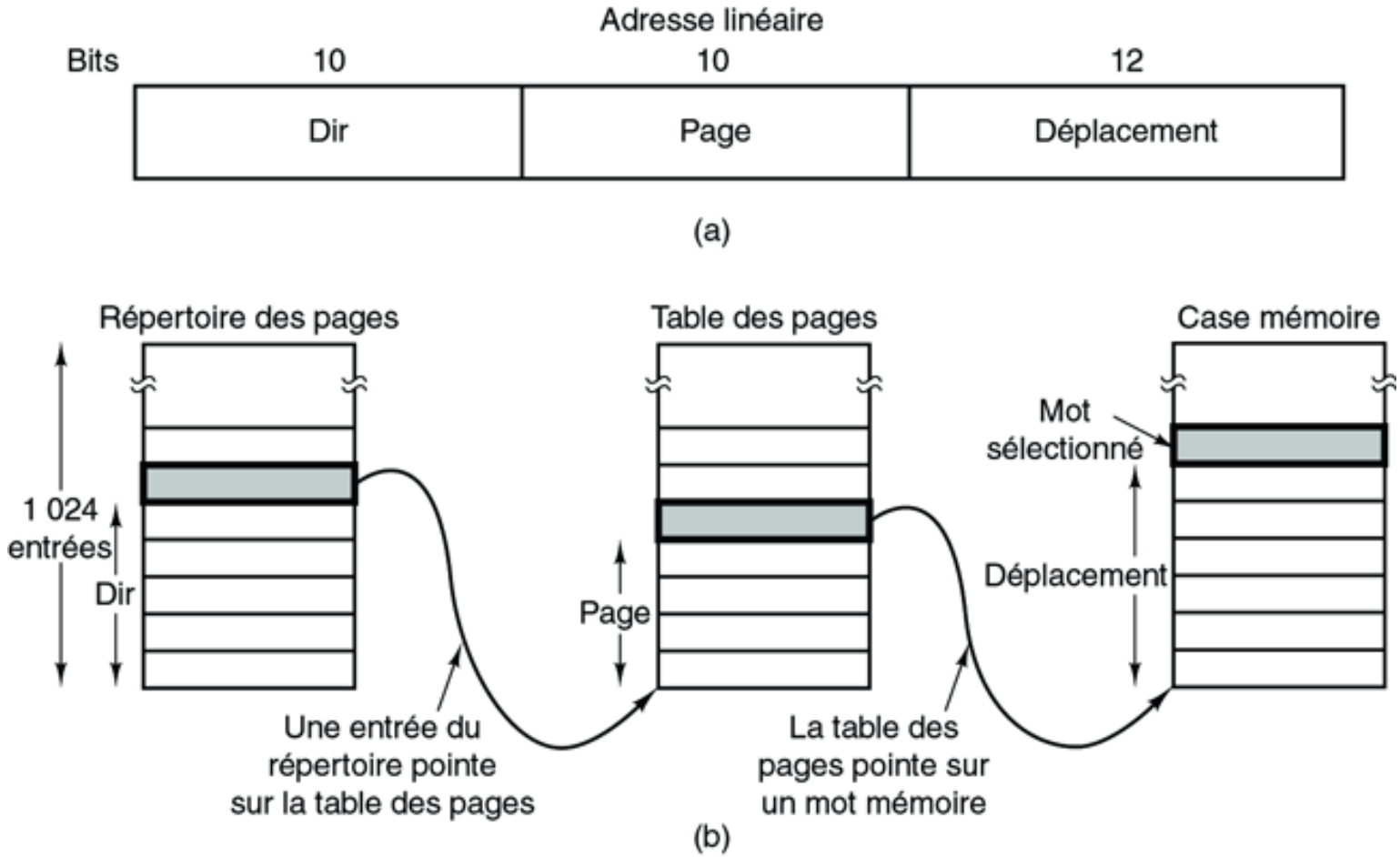

- Un descripteur du segment de code d'un Pentium
- Les segments de données diffèrent légèrement

# Le Pentium d'Intel (3)



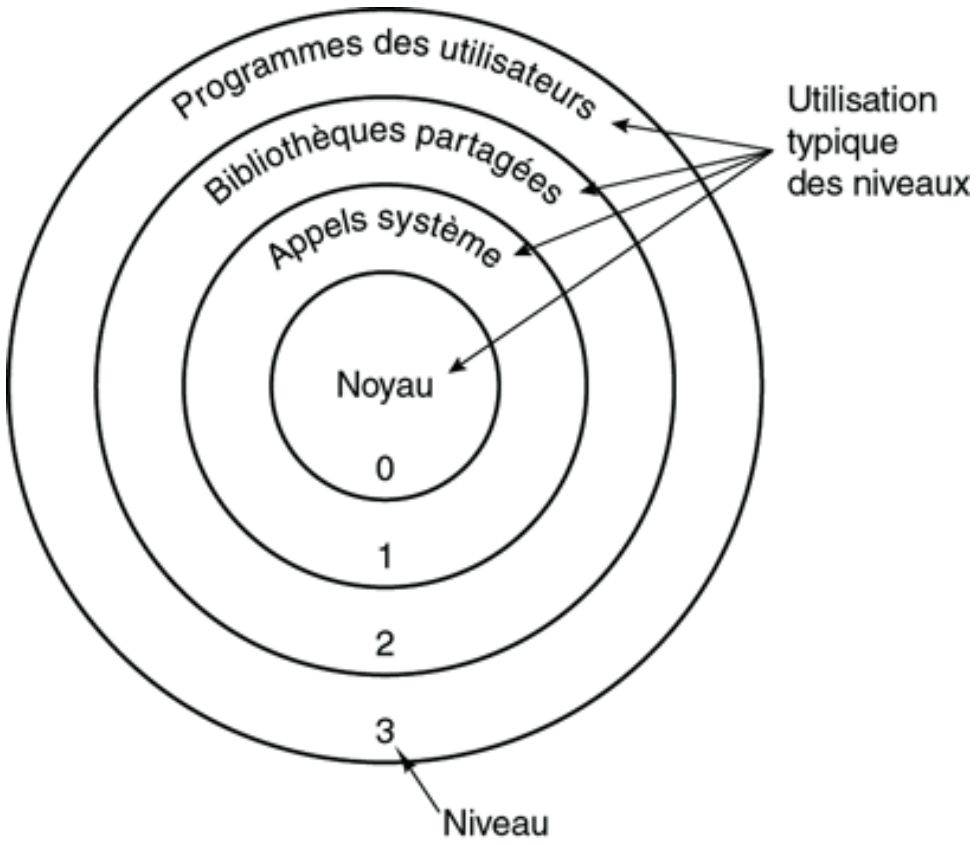

- Conversion d'une paire (selecteur, déplacement) en une adresse linéaire.
- Si la pagination est désactivée alors l'adresse linéaire correspond à une adresse physique.

# Le Pentium d'Intel (4)




- Mappage d'une adresse linéaire en une adresse physique
- Table des pages à 2 niveaux

# Le Pentium d'Intel (5)




Le mécanisme de protection du Pentium