Chapitre 1: Introduction

- Historique
- Structure d'un Système Informatique
- Rôle d'un Système d'Exploitation (SE)
- Services fournis par un SE
- Appels système
- Programmes système
- Structure d'un SE

Historique (1/3)

- Premiers systèmes (1945 - 1955)
  - Machines imposantes
  - Sans SE
  - Machines pilotées par un opérateur depuis une console
- Moniteurs simples (1955 - 1965)
  - SE rudimentaires
  - Moniteur (programme résident) pour enchaîner les travaux et minimiser ainsi le délai d'inactivité de l'Unité Centrale (UC)
  - Rendement du processeur amélioré mais reste limité à cause du traitement des entrées/sorties (E/S)

Historique (2/3)

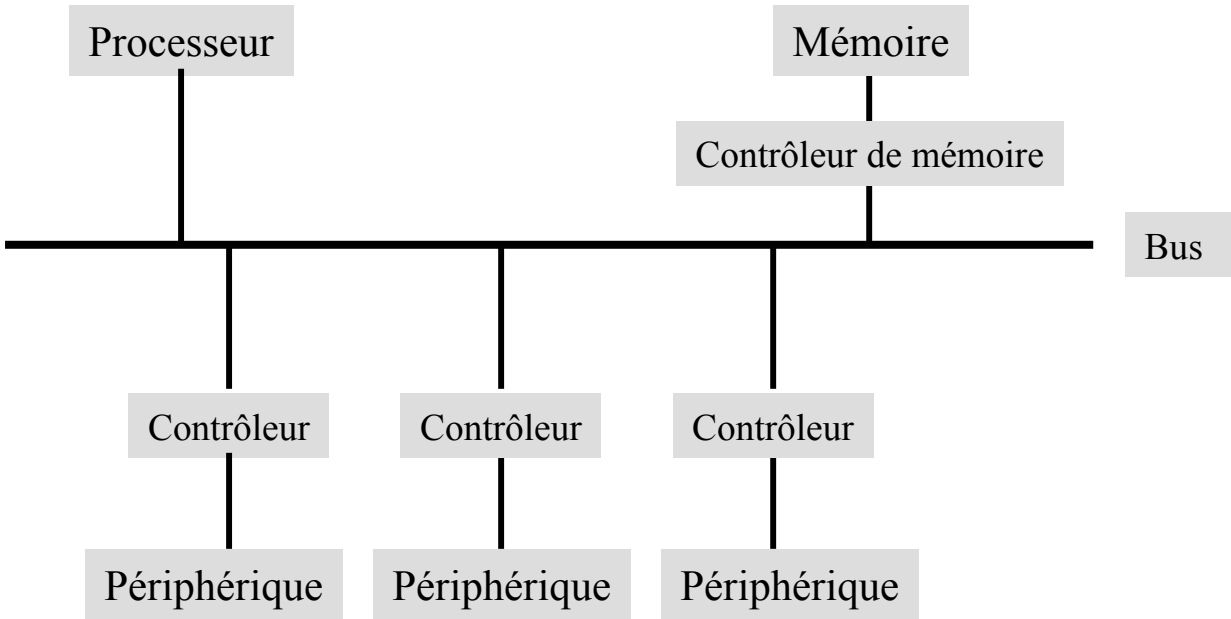
- Multiprogrammation (1965 - 1975)
  - Le SE sélectionne un programme et l'exécute. Si attente d'un événement, alors choix d'un autre programme et ainsi de suite
  - Comment garder tous les programmes simultanément en mémoire?
  - Quels algorithmes pour le choix des travaux à exécuter ?
- Partage de temps (1970 -)
  - Extension logique de la multiprogrammation.
  - Programmes en mémoire simultanément et exécution à tour de rôle avec un faible quantum de temps permettant l'intéractivité.
  - Besoin de mécanismes d'exécution concurrente des programmes.
  - Besoin d'algorithmes de gestion de la mémoire, de protection et d'ordonnancement du processeur, d'outils pour la gestion des périphériques et un système de fichiers.

Historique (3/3)

- Systèmes répartis (1980 -)
  - Systèmes fortement couplés
    - Les processeurs partagent la mémoire centrale et communiquent via cette mémoire
  - Systèmes faiblement couplés
    - Chaque processeur possède sa propre mémoire locale et communique avec les autres via un canal de communication (exemple : réseau local de machines)
- Systèmes Temps Réel
  - Contraintes temporelles strictes
  - Systèmes dédiés ne contenant pas tous les concepts avancés des SE généralistes.

Structure d'un système informatique (1/5)

1- Composants d'un ordinateur



Structure d'un système informatique (2/5)

2- Interruptions

- Une interruption est déclenchée par un événement extérieur (demande d'un périphérique, expiration d'un timer)
- Procédure de traitement d'une interruption :
  - le hardware passe le contrôle au SE
  - sauvegarde l'état du procesus interrompu dans le PCB
  - désactivation des autres interruptions
  - détermination du type de l'interruption via le vecteur d'interruptions
  - le contrôle est passé au gestionnaire approprié pour traiter l'interruption
  - l'état du processus interrompu est restauré
  - le processus interrompu (ou le processus suivant) reprend son exécution
- Avantages du concept d'interruption
  - surcharge faible pour attirer l'attention de l'UC
  - décharge l'UC du polling systématique des périphériques
  - un processsus en exécution peut initier une interruption logicielle (trappe ou signal) pour communiquer avec un autre processus.

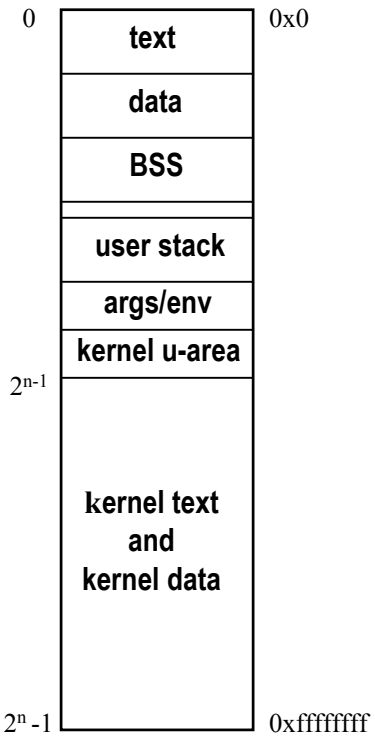
Structure d'un système informatique (3/5)

3- Modes d'exécution

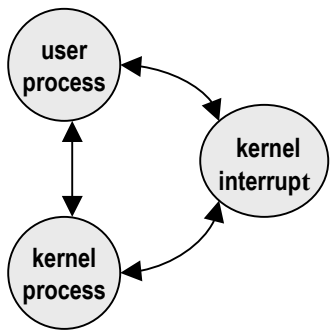
- Le partage des ressources implique la protection des utilisateurs les uns des autres. Le système utilise 2 modes :
  - mode utilisateur : instructions privilégiées inaccessibles
  - mode superviseur (noyau, moniteur, système) : accès à toutes les instructions ainsi qu'aux registres protégés du processeur
- Le mode courant est déterminé par la valeur d'un champ dans un registre spécial de status du processeur
- Le passage du mode utilisateur au mode noyau est déclenché par les interruptions et les exceptions.

Structure d'un système informatique (4/5)

- Modes d'exécution (suite)



	Process context	System context
User mode	Application	N/A
Kernel mode	Syscall or exception	Interrupt or system task



Structure d'un système informatique (5/5)

4- Protections matérielles

- Protection de la mémoire grâce à 2 registres l'un contenant la base de la mémoire du programme et l'autre sa taille
- Protection UC grâce à un timer
- Les E/S sont protégées par des instructions privilégiées (appels système)

Rôle d'un système d'exploitation

- Fourniture d'une machine virtuelle plus commode d’utilisation que la machine physique
- Gestion et partage des ressources de la machine entre utilisateurs

Services fournis par un SE (1/3)

- Gestion des processus
  - création et terminaison de processus utilisateur ou système
  - suspension et réveil de processus
  - synchronisation de processus
  - communication entre processus
  - gestion des interblocages

Services fournis par un SE (2/3)

- Gestion de la mémoire centrale
  - suivi de l'occupation de la mémoire
  - allocation et désallocation de mémoire
  - chargement de processus en mémoire
- Gestion de la mémoire secondaire
  - gestion des espaces libres sur disque
  - allocation et désallocation d'espace disque
  - ordonnancement du disque

Services fournis par un SE (3/3)

- Gestion des E/S
  - système de cache
  - gestionnaires de périphériques
  - interface d'accès uniforme aux périphériques
- Gestion des fichiers
  - création et suppression de fichiers et répertoires
  - fonctions de manipulation des fichiers et répertoires
  - stockage des fichiers sur disque
- Gestion des protections
- Gestion du réseau

Appels Système

- Interface entre les programmes en cours d'exécution et le système d'exploitation.
- 5 grandes catégories d'appels :
  - Contrôle de processus
  - Manipulation de fichiers
  - Manipulation de périphériques
  - Maintenance d'information
  - Communication

Programmes Système

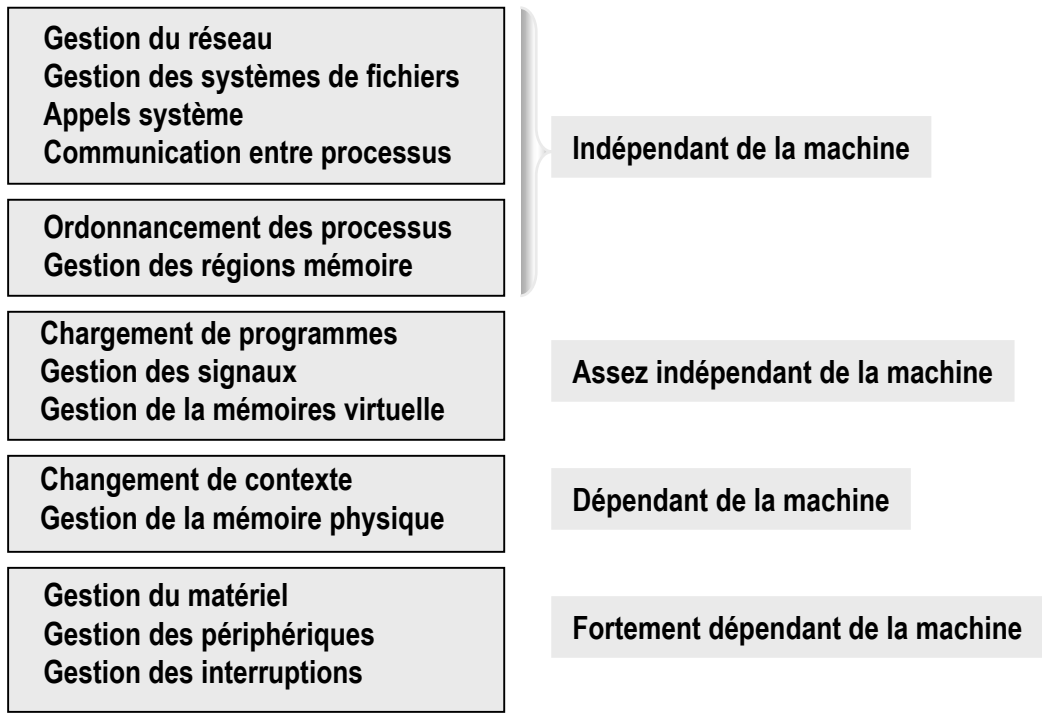
- Le SE fournit des programmes système pour un environnement de développement et d'exécution
- On peut classer ces programmes en plusieurs catégories :
  - manipulation de fichiers (création, suppression, copie, …)
  - maintenance du système
  - interpréteurs de commandes
  - éditeurs de texte
  - environnement de développement (compilateurs, assembleurs, interpréteurs)
  - exécution de programmes (chargeurs, éditeurs de liens, débogueurs)
  - programmes d'application (base de données, tableurs, jeux, …)

Structure d'un SE (1/3)

- Système monolithique
  - Constitué d'un seul gros programme
    - programme principal
    - ensemble de sous-programmes de service réalisant les appels système
    - ensemble de sous-programmes utilitaires appelés par les sous-programmes de service
    - exemple : système Unix constitué d'un seul programme, appelé le noyau, qui est structuré en plusieurs couches

Structure d'un SE (2/3)

- Système monolithique (suite)



Structure du système Unix

Structure d'un SE (3/3)

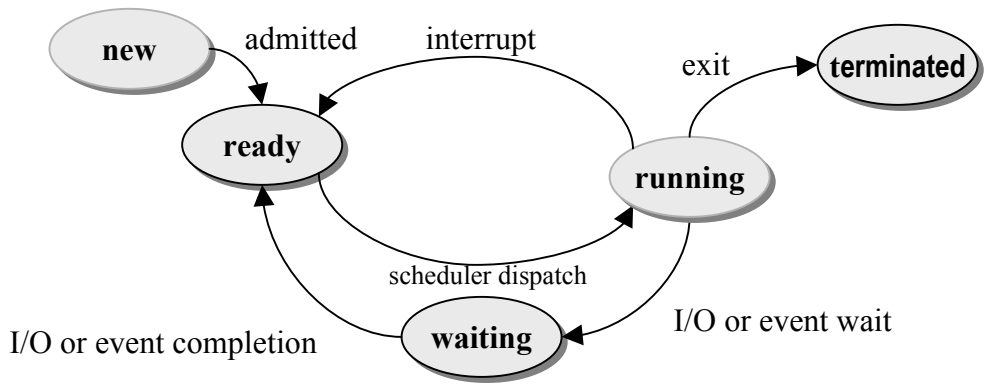
- Machine virtuelle
  - Machine abstraite plus commode à utiliser que la machine physique
  - Ce modèle cache la complexité de la machine physique
- Modèle client-serveur
  - Le SE est constitué d'un noyau minimal gérant la machine physique
  - Il offre des mécanismes de communication entre clients et serveurs
  - Exemple : le système MINIX
    - noyau minimal gérant les périphériques, les interruptions et offrant un mécanisme de transmission de messages entre programmes
    - programme gérant la mémoire appelé MM (Memory Manager)
    - programme gérant les fichiers appelé FS (File System)

Chapitre 2: Gestion de Processus

- Notions de processus et de threads
- Processus concurrents
- Ordonnancement de processus
- Algorithmes d'ordonnancement
- Synchronisation de processus
- Communication entre processus
- Gestion de l'interblocage

Notion de processus (1/2)

- Un processus est un programme en exécution
- Etats d'un processus
  - nouveau (*new*) : processus en création
  - en exécution (*running*)
  - prêt (*ready*) : en attente du processeur
  - en attente (*waiting*) : en attente d'un événement pour repasser à l'état prêt
  - terminé



Notion de processus (2/2)

- Bloc de contrôle d'un processus (*PCB: Process Control Block*)
  - structure contenant les informations nécessaires à la gestion du processus
    - état du processus
    - valeur du compteur ordinal
    - contenu des registres du processeur
    - informations nécessaires à l'ordonnancement du processeur
    - informations nécessaires à la gestion de la mémoire
    - états des E/S effectuées
  - le PCB est utilisé pour sauvegarder les informations variables entre processus

Notion de thread

- une thread est un processus léger défini par un compteur ordinal, un ensemble de registres et un espace pile

- une thread partage avec les autres threads le segment code, le segment data et les ressources du système

Processus concurrents

- Plusieurs processus peuvent s'exécuter en pseudo-parallélisme par le même processeur
  - Création de processus
    - processus parent créant un processus fils via un appel système
  - Terminaison de processus
    - appel à *exit* car exécution terminé
    - le processus parent appelle *abort* pour terminer un fils
  - Relations entre processus
    - processus indépendant : n'affecte pas et n'est pas affecté par l'exécution d'un autre processus
    - processus coopérant : peut affecter ou être affecté par l'exécution d'un autre processus (problème du Producteur-Consommateur)

Ordonnancement de processus (1/2)

- Principe - Comment gérer et conserver les informations sur tous les processus ?
  - Files d'attente
    - tout processus créé est placé dans la file d'attente des processus prêts à s'exécuter
    - tous les processus prêts et résidents dans la mémoire vive sont dans cette file qui est une liste chaînée des PCB
    - files d'attente des périphériques (une par périphérique)
  - Ordonnanceurs
      - lors de son exécution un processus est placé successivement dans différentes files d'attente
      - la sélection des processus depuis ces files est effectuée par un ordonnanceur
      - l'ordonnanceur à long terme sélectionne les processus à mettre dans la file d'attente des processus prêts
      - l'ordonnanceur à court terme alloue le processeur à un processus prêt

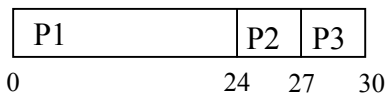
Ordonnancement de processus (2/2)

- Décision d'ordonnancement
  - passage de l'état *en exécution* à l'état *en attente*
  - passage de l'état *en exécution* à l'état *prêt*
  - passage de l'état *en attente* à l'état *prêt*
  - terminaison
- Changement de contexte
  - l'allocation du processeur à un nouveau processus implique la sauvegarde de l'état de l'ancien processus et le chargement de l'état du nouveau. Cette opération est appelée changement de contexte
  - la durée du changement de contexte dépend du support hardware

Algorithmes d'ordonnancement (1/7)

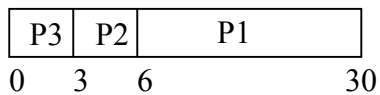
- L'ordonnanceur sélectionne un processus à exécuter suivant un algorithme d'ordonnancement. Les principaux algorithmes sont :
  - First-Come, First-Served **(FCFS)** ou **FIFO**
    - un processus qui demande le processeur en premier est exécuté le premier
    - implémentation à l'aide d'une file FIFO
    - le temps d'attente moyen peut être long

Processus	P1	P2	P3
Durée	24	3	3
Temps d'arrivée	0	0	0



Temps d'attente moyen = (0 + 24 + 27)/3 = 17

Processus	P3	P2	P1
Durée	3	3	24
Temps d'arrivée	0	0	0

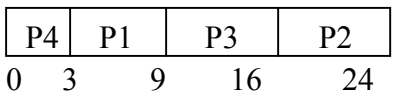


Temps d'attente moyen = (0 + 3 + 6)/3 = 3

Algorithmes d'ordonnancement (2/7)

- Shortest-Job-First (**SJF**) ou Shortest-Process-Next (**SPN**)
  - choix du processus qui a la plus petite durée d'exécution
  - 2 variantes : préemptif (appelé aussi Shortest-Remainig-Time-First) ou non préemptif
  - SJF est optimal car il fournit le temps moyen d'attente minimal pour un ensemble de processus
  - difficile à implémenter car il est impossible de prévoir exactement la durée d'exécution des processus

Processus	P1	P4
Durée		3
Temps d'arrivée		0

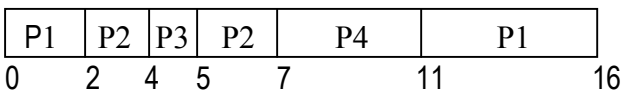


Temps d'attente moyen = (0 + 3 + 9 + 16)/4 = 7

Algorithmes d'ordonnancement (3/7)

- Shortest-Remaining-Time (SRT)
  - version préemptive de SJF
  - choix d'un autre processus qui a une durée plus petite que le temps restant pour le processus en cours d'exécution
  - exemple

Processus	P1	P2	P3	P4
Durée d'exécution	7	4	1	4
Temps d'arrivée	0	2	4	5



Temps moyen d'attente = (9 + 1 + 0 + 2)/4 = 3

Algorithmes d'ordonnancement (4/7)

- Ordonnancement avec priorités
  - à chaque processus est associée une priorité
  - priorités calculées de façon interne ou externe
    - interne : basée sur la quantité mémoire, le nombre de fichiers ouverts, % des durées d'E/S par rapport aux durées d'exécution, etc…
    - externe : priorités déterminées par des facteurs étrangers au SE (importance du travail, coût d'utilisation de la machine, etc…)
  - le processeur est alloué au processus ayant la plus haute priorité
  - c'est une extension de l'algorithme SJF (qui est un cas particulier avec la priorité égale à l'inverse de la durée de la prochaine phase d'exécution du processus)
  - risque de famine : un processus peut attendre indéfiniment le processeur qui est alloué à des processus de plus forte priorité

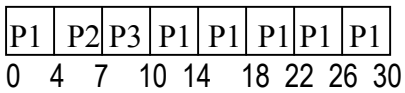
Algorithmes d'ordonnancement (5/7)

- Round-Robin (**RR**)
  - définir une tranche de temps appelée quantum de temps généralement entre 10 et 100 ms
  - algorithme préemptif
  - choisir le processus en tête de liste et l'exécuter au plus pendant un quantum
  - si le processus s'exécute en moins d'un quantum, il libère volontairement le processeur et l'ordonnanceur choisit le processus suivant dans la liste
  - si le processus consomme son quantum, le timer cause une interruption et l'ordonnanceur place le processus à la fin de la queue d'attente puis choisit le suivant en tête de liste
  - performance

Algorithmes d'ordonnancement (6/7)

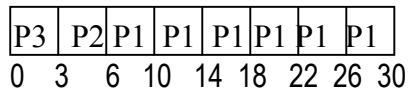
- RR (suite)
  - exemples avec quantum = 4

Processus	P1	P2	P3
Durée d'exécution	24	3	3
Temps d'arrivée	0	0	0



Temps moyen d'attente = (4 + 7 + 6)/3 = 5.66

Processus	P1	P2	P3
Durée d'exécution	3	3	24
Temps d'arrivée	0	0	0



Temps moyen d'attente = (0 + 3 + 6)/ 3 = 3

Algorithmes d'ordonnancement (7/7)

- Files d’attente multiples
  - utilisation de plusieurs queues d'attente avec une priorité associée à chaque queue, par exemple processus d'avant-plan (intéractifs) et processus d'arrière-plan (batch)
  - chaque queue a son algorithme d'ordonnancement, par exemple RR pour l'intéractif et FCFS pour le batch
  - l'ordonnancement est fait entre les queues d'attente
  - les processus restent dans les files d'attente qui leurs sont assignées
  - une extension de ces algorithmes permet de déplacer certains processus entre différentes files d'attente

Synchronisation de processus (1/22)

- Notion de section critique
  - Quand des processus coopérants partagent des données communes, des mécanismes de synchronisation doivent être fournis pour assurer l'intégrité de ces données. Le fragment de code qui manipule les données communes (ressource critique) est appelé section critique
- Exclusion mutuelle avec attente active
  - Désarmement des interruptions
    - Pour assurer l'exclusion mutuelle, désarmer les interruptions avant d'entrer en section critique et les armer en quittant la section critique
    - Solution dangereuse et peu utilisée car le SE ne peut pas laisser au processus utilisateur la possibilité de désactiver les interruptions dont il dépend

Synchronisation de processus (2/22)

- Exclusion mutuelle avec attente active (suite)
  - Variables de synchronisation
    - variables partagées entre les processus accédant à une ressource critique commune
    - tout processus doit tester ces variables avant d'entrer en section critique
    - exemple de code de section critique

```
1  while (verrou != 0)          /* Ne rien faire */;
2   verrou = 1;
3    section critique
4    verrou = 0;
```

- Le problème de cette solution est que les lignes 1 et 2 constituent elles-même une section critique, la ressource critique étant la variable *verrou*.

Synchronisation de processus (3/22)

- Exclusion mutuelle avec attente active (suite)
  - Algorithme de Peterson
    - solution logicielle au problème d'exclusion mutuelle
    - considérons 2 processus P_i et P_j , le code de P_i est le suivant :

```
1   int   flag[2];
2   int   tour;
3   flag[i] = 1;
4   tour = i;
5   while (tour == i && flag[j])        /* Ne rien faire */;
6   section critique
7   flag [i] = 0;
```

Le code du processus P_j est symétrique à celui-ci

Synchronisation de processus (4/22)

- Exclusion mutuelle avec attente active (suite)
  - Instruction *test-and-set*
    - solution hardware au problème de l'exclusion mutuelle.
    - le processeur dispose d'une instruction *test-and-set (tas)* qui lit la valeur d'une variable dans un registre et stocke une valeur non nulle dans cette variable. Ces deux opérations sont **indivisibles**.
    - l'entrée en section critique se ramène alors au fragment de code (dans un langage d'assemblage fictif) :

```
1   loop:
2         tas     R1,     lock     ; Test-And-Set de la variable lock dans R1
3         comp    R1,     0        ; Test de R1 à 0
4         jneq    loop             ; Boucle si différent de 0
```

- la sortie de la section critique consiste à mettre la variable *lock* à 0.

Synchronisation de processus (5/22)

- Exclusion mutuelle avec attente active (suite)
  - Performances
    - les solutions déjà présentées utilisent l'attente active (*busy waiting*)
    - performances généralement médiocres
- Suspension et réveil
  - Eviter l'attente active par les primitives *sleep* et *wakeup*
  - *sleep* suspend l'exécution du processus courant en le passant à l'état *en attente*
  - *wakeup* reprend l'exécution d'un processus en le passant à l'état *prêt*

Synchronisation de processus (6/22)

- Suspension et réveil (suite)
  - Exemple: 2 processus P1 et P2 cycliques partagent une ressource critique. En utilisant l'instruction *test-and-set (tas)* et les primitives *sleep* et *wakeup*, le code de ces processus peut être écrit comme suit :

P1

```
1   while (1)
2   {
3       if (tas (&lock))
4           sleep();
5       section critique
6       lock = 0;
7       wakeup(P2);
8   }
```

P2

```
1   while (1)
2   {
3       if (tas (&lock))
4           sleep();
5       section critique
6       lock = 0;
7       wakeup(P1);
8   }
```

L'exclusion mutuelle n'est pas toujours assurée car les instructions 3 et 4 ne sont pas atomiques.

Synchronisation de processus (7/22)

- Sémaphores (Dijkstra, 1965)
  - Nouveau type de variables entières protégées
  - Caractérisation par un compteur et une liste de processus en attente
  - Opérations possibles: *P* (down, wait) et *V* (up*, signal)*
    (*P* pour *proberen* en hollandais, i.e tester et *V* pour *verhogen*, i.e, incrémenter)
  - Extension des primitives *sleep* et *wakeup*
  - Protection d'une section critique
  - Outil général pour la résolution des problèmes de synchronisation
  - Exemples : ordonnancement de tâches, accès à une ressource, etc…
  - Inconvénient majeur : toutes les opérations sont à la charge du programmeur

Synchronisation de processus (8/22)

- Sémaphores (suite)
  - Exemples : ordonnancement de tâches, accès à une ressource, etc…
    - utilisation d'un sémaphore avec compteur initialisé à 1 pour protéger une section critique

```
P (mutex);
section critique;
V (mutex);
```

- 2 tâches avec T1 < T2 effectuées par 2 processus P1 et P2. On peut ordonner l'exécution de T1 et T2 avec un sémaphore S dont le compteur est initialisé à 0

```
P1
T1 ( );
V (S);
```

```
P2
P (S);
T2 ( );
```

Comme le compteur = 0, l'exécution de T2 n'est possible que lorsque l'opération V (S) a été exécutée, i.e après l'exécution de T1

Synchronisation de processus (9/22)

- Sémaphores (suite)
  - Exemples : ordonnancement de tâches, accès à une ressource, etc…
    - Accès à une ressource
      - soit une ressource présente en N exemplaires avec autant de processus en section critique que d'exemplaires. On définit un sémaphore *nr* dont le compteur est initialisé à N
      - tout processus effectue l'opération *P(nr)* avant d'accéder à un exemplaire de la ressource puis *V(nr)* après cet accès. A chaque instant, le compteur contient le nombre d'exemplaires disponibles
      - lorsque N processus sont en section critique, le compteur est < 0. Si d'autres processus cherchent à acquérir un exemplaire de la ressource par *P(nr)*, ils sont placés en attente

Synchronisation de processus (10/22)

- Sémaphores (suite)
  - Quelques critiques
    - utilisation diversifiée
      - sémaphores utilisés pour la synchronisation, l'exclusion mutuelle et l'allocation de ressources sans identification de l'usage par simple lecture du code
    - absence de sémantique
      - *wait* et *signal* sont utilisés pour différents buts sans aucune sémantique
    - interblocage facile
      - une simple inversion des opérations *wait* et *signal* peut mener à l'interblocage
    - couplage obscur
      - les opérations *wait* et *signal* d'un sémaphore peuvent être reliées aux mêmes opérations d'un autre sémaphore sans aucune visibilité dans le code
    - complexité de l'intéraction de plusieurs sémaphores

Synchronisation de processus (11/22)

- Moniteur (Hoare, 1974 ; Brinch, 1975)
  - Structure regroupant des variables partagées par plusieurs processus et les instructions qui les manipulent
  - Exclusion mutuelle assurée par le compilateur et non le programmeur
  - A tout instant, un seul processus peut accéder au moniteur
  - Tout processus qui souhaite utiliser un moniteur occupé, est suspendu et placé dans une file d'attente jusqu'à la libération du moniteur
  - Variables de condition avec opérations associées *wait* et *signal*
    - pour permettre l'attente d'un processus à l'intérieur d'un moniteur, une variable de condition doit être déclarée : **var** x : condition
    - l'opération *x.wait* signifie que le processus invoquant cette

Synchronisation de processus (12/22)

- Moniteur (suite)
  - Exemple :
    - Plusieurs processus ont accès, en exclusion mutuelle, à une ressource unique. Cette ressource peut être gérée par un moniteur. Afin de disposer d'une file d'attente des processus ayant demandé la ressource, on utilise une variable condition appelée *libre*. Une variable booléenne appelée *occupé*, indique si la ressource est disponible.

```
int             occupé = 0;
condition       libre;
void acquisition  (void)
{       if (occupé)
            wait (libre);
        occupé = 1;
 }

void libération (void)
{       occupé = 0;
        signal (libre);
}
```

Synchronisation de processus (13/22)

- Moniteur (suite)
  - Avantages
    - but unique
      - un moniteur traite uniquement l'exclusion mutuelle et il n'y a pas de confusion comme dans le cas des sémaphores
    - approche modulaire à l'exclusion mutuelle
    - simplicité d'utilisation
    - implémentation efficace

Synchronisation de processus (14/22)

- Moniteur (suite)
  - Inconvénients
    - séparation de la synchronisation
    - couplage des opérations conditionnelles
    - intéraction complexe entre de multiples variables de condition
    - moniteurs présents dans un nombre restreint de langages
    - primitive inutilisable dans un système distribué

Synchronisation de processus (15/22)

- Problèmes classiques de synchronisation
  - Producteurs-Consommateurs
    - 2 processus utilisent un tampon mémoire partagé dans lequel le producteur dépose les objets produits et le consommateur retire les objets qu'il consomme. Les contraintes sont :
      - les 2 processus s'exécutent en parallèle
      - le tampon est une ressource critique et son accès doit être protégé
      - si le tampon est vide, le consommateur doit être mis en attente jusqu'à la production d'un objet
      - lorsque le tampon est plein, le producteur doit être mis en attente jusqu'à ce qu'un objet soit consommé

Synchronisation de processus (16/22)

- Problèmes classiques de synchronisation (suite)
  - Producteurs-Consommateurs (suite)
    - solution avec sémaphores
      - on utilise 3 sémaphores - *mutex* avec compteur = 1 pour synchroniser les accès au tampon, *nbo* dont le compteur est initialisé à 0 et *nbl* dont le compteur est initialisé au nombre d'objets pouvant être stockés dans le tampon

Producteur

```
objet       o;
int         i = 0;
while (1)
{           o = produire ();
            P (nbl);
            P (mutex);
            tampon [i] = 0;
            i = (i + 1) % N;
            V (mutex);
            V (nbo);
}
```

Consommateur

```
objet       o;
int         i = 0;
while (1)
{           P (nbo);
            P (mutex);
            o = tampon [i];
            i = (i + 1) % N;
            V (mutex);
            V (nbl);
            consommer (o);
}
```

Synchronisation de processus (17/22)

- Problèmes classiques de synchronisation (suite)
  - Producteurs-Consommateurs (suite)
    - solution avec moniteur
      - il faut déterminer les variables partagées et les sections critiques. Il s'agit du tampon et des opérations *prendre* et *ajouter*
      - on utilise 2 variables de type condition
        - variable *places* pour suspendre le producteur lorsque tous les postes du tampon sont remplis
        - variable *objets* pour suspendre le consommateur lorsque tous les postes du tampon sont libres

Synchronisation de processus (18/22)

- Problèmes classiques de synchronisation (suite)
  - Producteurs-Consommateurs (suite)
    - solution avec moniteur : code du moniteur

```
int         entree  = 0;
int         sortie  = 0;
int         compte  = 0;
objet       tampon [N];
condition   places;
condition   objets;
```

```
void        ajouter (objet o)
{
            if (compte == N)
               wait (places);
            tampon [entrée] = o;
            compte ++;
            entree = (entree + 1) % N;
            signal (objets);
}
```

```
void        prendre (objet *o)
{
            if (compte == 0)
               wait (objets);
            *o = tampon [sortie];
            compte --;
            sortie = (sortie + 1) % N;
            signal (places);
}
```

Synchronisation de processus (19/22)

- Problèmes classiques de synchronisation (suite)
  - Producteurs-Consommateurs (suite)
    - solution avec moniteur (suite)
      A l'aide du moniteur précédent, les processus producteur et consommateur s'écrivent :

Producteur

```
objet   o;
while (1)
{
        o = produire ();
        ajouter (o);
}
```

Consommateur

```
objet   o;
while (1)
{
        prendre (&o);
        consommer (o);
}
```

Synchronisation de processus (20/22)

- Problèmes classiques de synchronisation **(suite)**
  - Lecteurs-Rédacteurs
    - modélisation des accès à une base de données. L'objet partagé n'est accessible que par les opérations de lecture et d'écriture. Plusieurs consultations sont possibles en même temps puisqu'elles ne modifient pas les données. En revanche, les mises à jour doivent être réalisées en exclusion mutuelle. Les écritures simultanées sont interdites et il est impossible de consulter les données pendant leur mise à jour
    - il existe plusieurs variantes à ce problème selon les priorités accordées au processus
    - chacune de ces variantes peut donner lieu à des phénomènes de famine où les lecteurs ou les rédacteurs peuvent attendre indéfiniment

Synchronisation de processus (21/22)

- Problèmes classiques de synchronisation (suite)
  - Lecteurs-Rédacteurs (suite)
    - solution avec sémaphores dans la variante priorité aux lecteurs

Lecteur

```
while (1)
{
      P (Mutex);
      nb_lec ++;
      if (nb_lect == 1)
         P (Ecrire);
      V (Mutex);
       lecture ();
       P (Mutex);
       nb_lect --;
       if (nb_lect == 0)
          V (Ecrire);
       V (Mutex);
        traiter_lecteur ();
}
```

Rédacteur

```
while (1)
{
      traiter_redacteur ();
      P (Ecrire);
      ecrire();
      V (Ecrire);
}
```

Synchronisation de processus (22/22)

- Problèmes classiques de synchronisation (suite)
  - Problème des philosophes
    - problème posé et résolu par Dijkstra. Cinq philosophes sont assis autour d'une table ronde. Pour manger, chaque philosophe a besoin de 2 fourchettes. Une fourchette sépare 2 assiettes consécutives. Les activités principales d'un philosophe sont manger et penser.
    - écrire un programme qui permet à chaque philosophe de se livrer à ses activités sans jamais être bloqué.

Communication entre processus (1/2)

- Mémoire partagée :
  - accès aux variables partagées en utilisant les outils de synchronisation
- Envoi de messages
  - Les processus peuvent communiquer sans variables partagées. Le noyau fournit au moins 2 primitives permettant à un processus d'envoyer et de recevoir des messages
- Modes de communication par messages
  - Communication directe
    - les processus doivent se nommer explicitement
      - send (P, message) - envoi de message au processus P
      - receive (Q, message) - réception de message émis par le processus Q
    - un lien bidirectionnel est automatiquement établi entre 2 processus

Communication entre processus (2/2)

- Modes de communication par messages (suite)
  - Communication indirecte
    - messages échangés via des boîtes à lettres
    - chaque boîte à lettres possède un identificateur unique
    - 2 processus peuvent communiquer s'ils partagent une boîte à lettres
    - un lien unidirectionnel ou bidirectionnel est établi entre les processus possédant une boîte à lettres commune
- Bufferisation
  - Capacité nulle : 0 message, l'émetteur doit attendre le récepteur, communication possible s'il y a synchronisation (rendez-vous)
  - Capacité bornée : n messages de longueur finie, l'émetteur doit attendre si le canal est déjà plein
  - Capacité infinie : message de longueur infinie, l'émetteur n'attend jamais

Gestion de l'interblocage (1/4)

- Interblocage (*Deadlock*)
  Deux ou plusieurs processus réclament une ressource pour progresser mais aucun ne l'obtient
  - Interblocage si 4 conditions réunies (Coffman et al., 1971)
    - exclusion mutuelle : ressource attribuée à un seul processus ou disponible
    - acquisition et attente : conservation des ressources déjà obtenues et formulation de nouvelles demandes
    - pas de réquisition : une ressource ne peut être libérée que par le processus qui la détient
    - attente circulaire : dans un cycle, chaque processus attend une ressource détenue par un autre processus du cycle

Gestion de l'interblocage (2/4)

- Interblocage
  - Exemple d'interblocage

    2 processus P1 et P2 accèdent à 2 ressources critiques A et B. Ces processus utilisent 2 sémaphores S_A et S_B initialisés à 1 :

P1
1 P (S_A);
2 P (S_B);
3 *section critique*
4 V (S_A);
5 V (S_B);

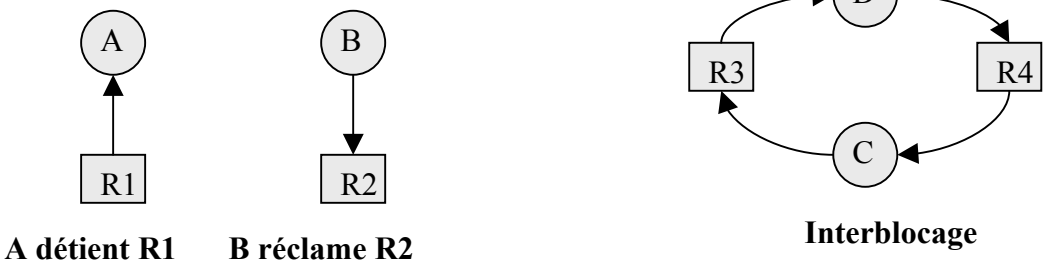
P2
1 P (S_B);
2 P (S_A);
3 *section critique*
4 V (S_B);
5 V (S_A);

L'ordonnancement suivant provoque un interblocage :
- P1 exécute la ligne 1 et réserve la ressource A
- un changement de contexte intervient
- P2 exécute la ligne 1 et réserve la ressource B, il exécute ensuite la ligne 2 et tente de réserver A. Comme A est déjà réservée par P1, P2 est suspendu
- P1 reprend son exécution, il exécute la ligne 2 et tente de réserver B. Comme B est déjà réservée par P2, P1 est suspendu

Gestion de l'interblocage (3/4)

- Graphes d'allocation des ressources (*Ressource Allocation Graphs*)

  Les interblocages sont caractérisés par des graphes d'allocation de ressources.
  Les processus sont représentés par des cercles et les ressources par des carrés :



A détient R1 **B réclame R2** **Interblocage**

  - Graphe avec cycle : interblocage si un seul exemplaire de la ressource et possibilité d'interblocage si plusieurs exemplaires
  - Graphe sans cycle : pas d'interblocage

Gestion de l'interblocage (4/4)

- Stratégies de traitement de l'interblocage
  - ignorer le problème
  - le détecter et y remédier
    - vérification après chaque allocation s'il y a interblocage et si oui appliquer un algorithme de guérison en recupérant des ressources déjà allouées
    - la guérison de l'interblocage par suppression ou retour à un état antérieur comporte un risque de famine
  - prévention à priori en supprimant l'une des 4 conditions de l'interblocage
    - la méthode de l'allocation globale vise à éliminer la condition "acquisition et attente"
    - la méthode des classes ordonnées vise à éliminer l'attente circulaire
  - l'éviter dynamiquement par une allocation prudente des ressources
    - algorithme du banquier : fondé sur la notion d'état sûr ; une ressource n'est allouée que si elle maintient le système dans un état sûr

Complément chapitre 2 : Threads

- Notion de thread
- Threads et processus
- Avantages des threads
- Exemple
- User-Level Threads (ULT)
- Avantages et inconvénients des ULT
- Kernel-Level Threads (KLT)
- Avantages et inconvénients des KLT
- Approche combinée ULT/KLT

Notion de thread (1/3)

- Thread: processus léger défini par un compteur ordinal, des registres, une pile et un état
- La thread a accès à l'espace d'adressage du processus ainsi qu'à ses ressources (les variables globales et les fichiers ouverts par une thread sont visibles aux autres threads)

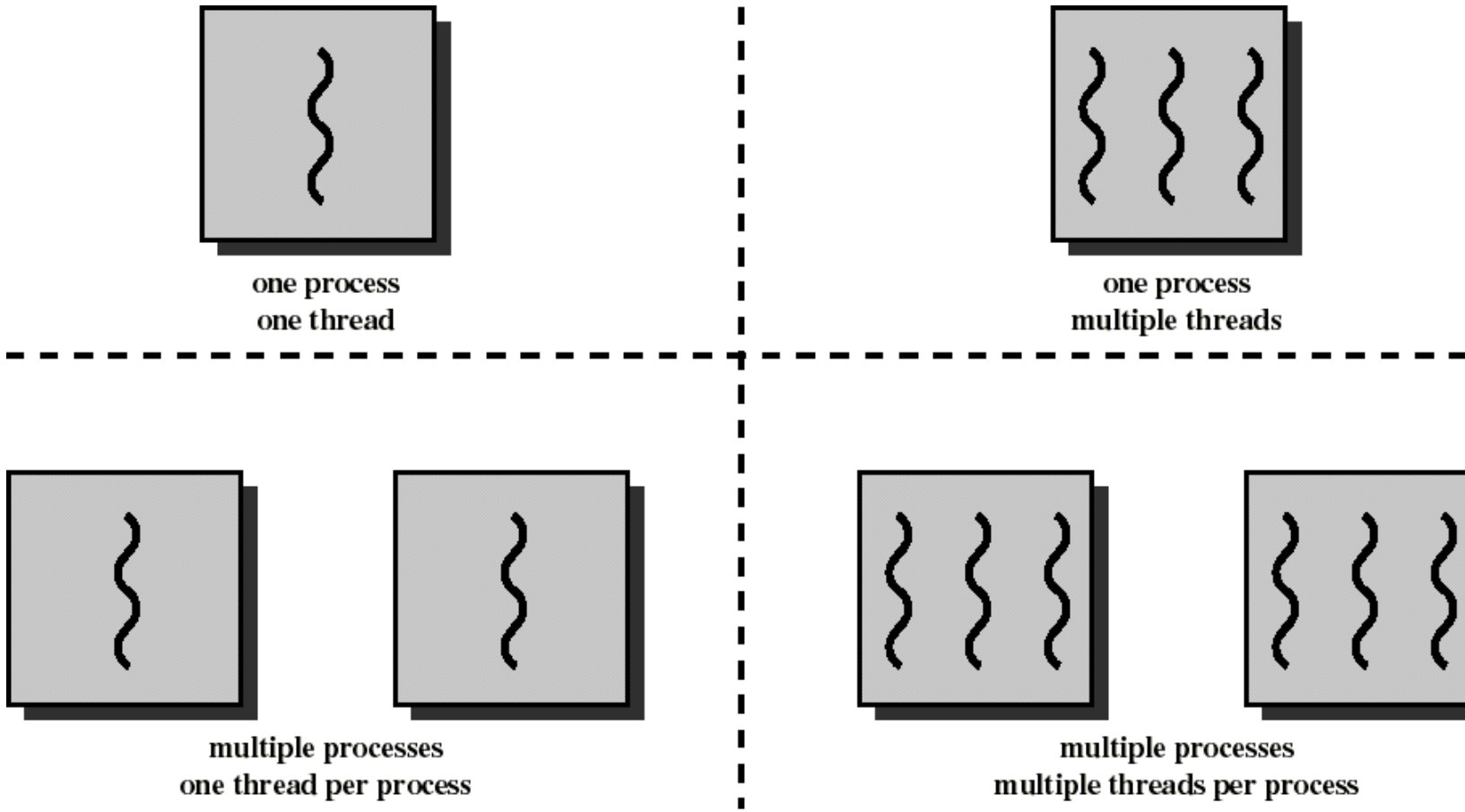
Notion de thread (2/3)

- **Multithreads: le SE supporte l'exécution de plusieurs threads au sein d'un même processus**
  - MS-DOS: mono-tâche et mono-thread
  - UNIX: multi-tâche mais une thread par processus
  - Solaris: multi-threads

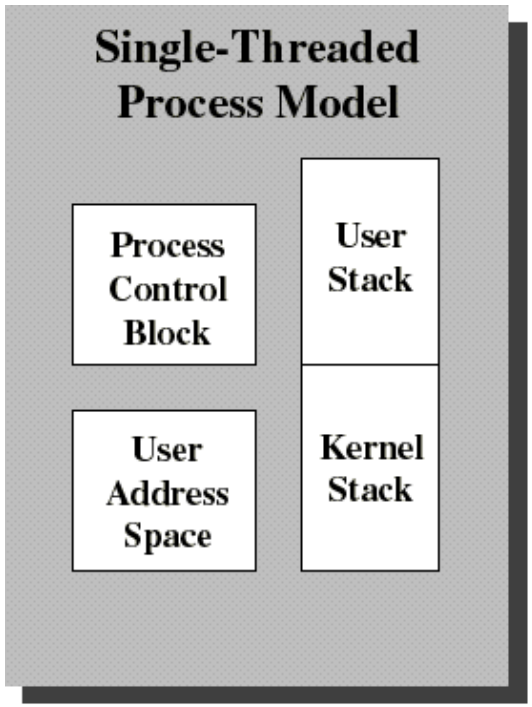
Notion de thread (3/3)

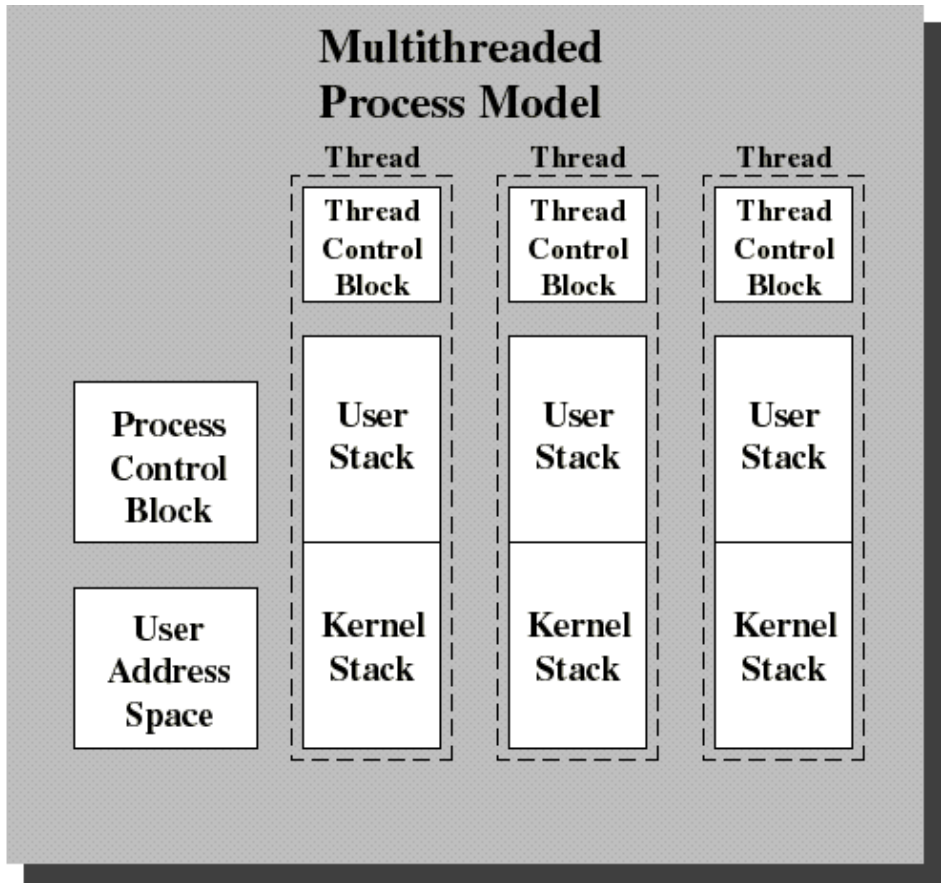
- Trois états: en exécution (*running*), prêt (*ready*), bloqué (*blocked*)
- La fin d'un processus implique la fin de toutes les threads du processus

Threads et processus (1/2)



Threads et processus (2/2)





Thread Control Block: contient une image du registre, la priorité et des informations sur l'état de la thread.

Avantages des threads (1/3)

- Création d'une thread: beaucoup plus rapide que celle d'un processus
- Exécution plus rapide
- Moins de temps pour basculer d'une thread à une autre dans le même processus

Avantages des threads (2/3)

- Si une application est composée de parties indépendantes qui n'ont pas besoin d'être exécutées en séquence, chaque élément peut être implémenté comme une thread
- Quand une thread est bloquée en attente d'une E/S, l'exécution peut continuer en passant à une autre thread du même processus.

Avantages des threads (3/3)

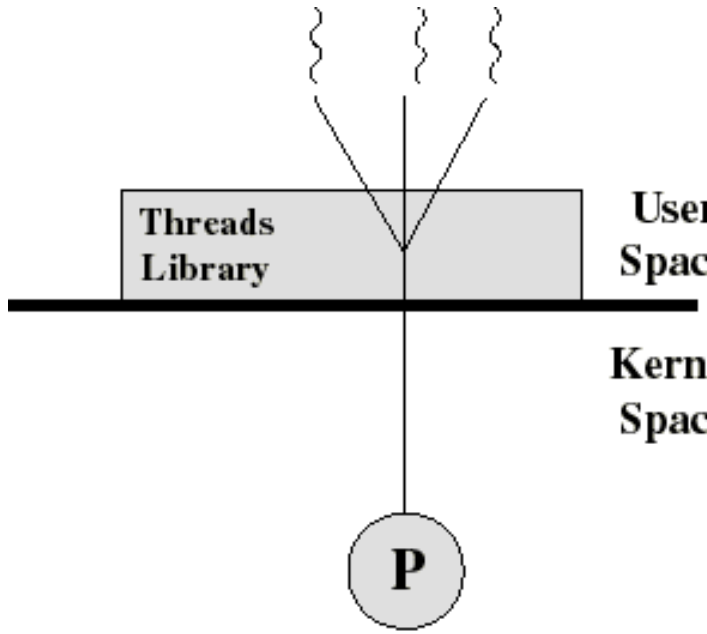
- Les threads partagent la mémoire et les fichiers et peuvent communiquer sans impliquer le noyau
- Il est nécessaire de synchroniser les threads pour ne pas avoir de résultats imprévisibles

Exemple

- **3 variables: A, B, C partagées par la thread T1 et la thread T2**
- **T1 calcule C = A+B**
- **T2 transfère une quantité X de A à B**
  - T2 doit faire: A = A -X et B = B+X (la somme A+B reste inchangée)
- **Mais si T1 calcule A+B après que T2 a fait A = A-X mais avant B = B+X**
- **Alors T1 n'obtient pas le bon résultat pour C = A + B**

User-Level Threads (ULT) (1/3)

- Les ULT sont transparents au noyau
- La gestion des threads est à la charge de l'application via une librairie de threads
- L'ordonnancement est spécifique à l'application



ULT (2/3)

- **La librairie des threads contient des fonctions pour :**
  - créer et détruire les threads
  - échanger des messages et des données entre threads
  - ordonnancer l'exécution des threads
  - sauvegarder et restaurer les contextes des threads

ULT (3/3)

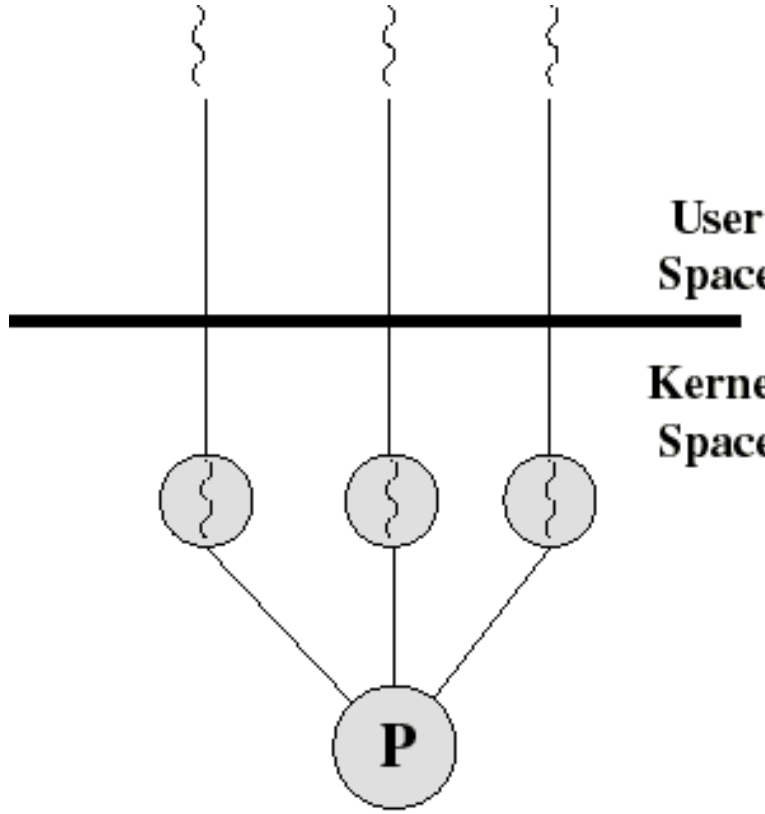
- Le noyau n'est pas au courant de l'activité des threads, mais il gère l'activité du processus
- Quand une thread fait un appel système, tout le processus est bloqué
- Mais pour la librairie threads, cette thread est toujours en état d'exécution
- On voit donc que les états des threads sont indépendants des états du processus

Avantages et inconvénients des ULT

- **Advantages**
  - Le basculement entre threads n'implique pas le noyau
  - L'ordonnancement est spécifique à l'application (On peut choisir le meilleur algorithme)
  - Les ULT peuvent s'exécuter sur n'importe quel SE. On a besoin que d'une librairie threads
- **Inconvénients**
  - Appels système bloquants, donc le noya bloque les processus. Tous les threads du même processus sont alors bloquées
  - le noyau ne sait attribue que des processus à de processeurs. 2 threads du même processus ne peuvent s'exécuter en // sur 2 processeurs

Kernel-Level Threads (KLT)

- Toute la gestion est faîte par le noyau
- Pas de librairie mais une API pour le service thread du noyau
- le noyau maintient les informations de contexte pour les processus et les threads
- L'alternance entre les threads implique le noyau
- L'ordonnancement concerne les threads
- Exemple: Windows NT

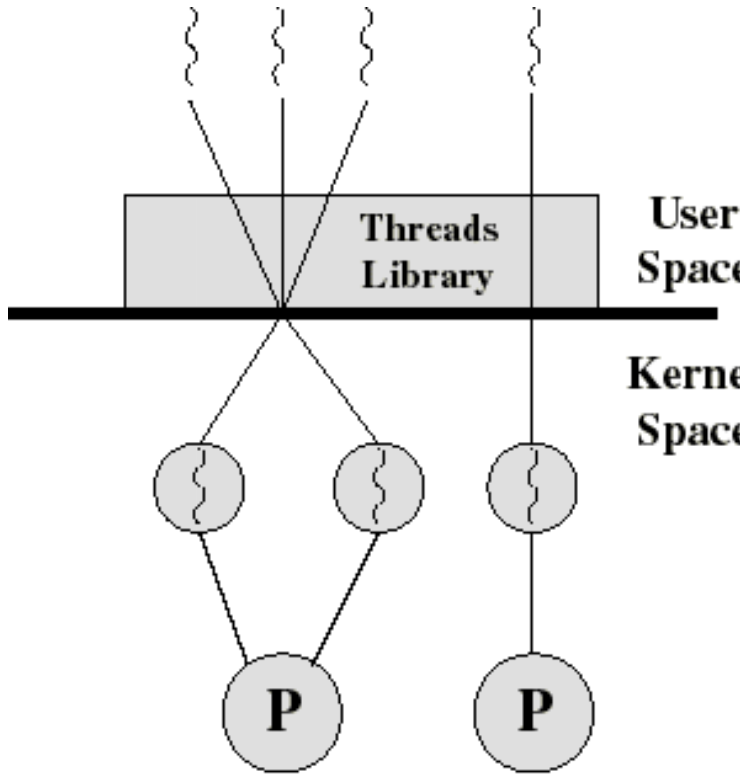


Avantages et inconvénients des KLT

- **Avantages**
  - l'ordonnancement par le noyau des threads d'un processus sur plusieurs processeurs est possible
  - le bloquage est fait au niveau thread
  - les routines du noyau peuvent être codées en multithreads
- **Inconvénients**
  - le changement de thread dans le même processus implique le noyau.
  - cela implique une lenteur importante

Approche combinée ULT/KLT

- Création de thread dans l'esapce utilisateur
- La plus grosse partie de l'ordonnancement et la synchronisation est faîte dans l'espace utilisateur
- Exemple : Solaris



Chapitre 3: Gestion de la mémoire physique

- Principe
- Allocation de mémoire
- Swapping
- Segmentation
- Pagination
- Segmentation paginée

Principe

- Un processus doit être chargé en mémoire pour son exécution
- Nécessité d'allocation de la mémoire pour le processus afin d'y placer son code, ses données et sa pile
- Le système gère la mémoire et tient à jour :
  - les zones mémoires allouées au système
  - les zones mémoires allouées à chaque processus
  - les zones mémoires disponibles

Allocation de mémoire

- Allocation de partition unique
- Allocation de partitions multiples
  - Partitions de taille fixe
  - Partitions de taille variable
  - Relogement d'adresses

Allocation de partition unique

- Allouer toute la mémoire disponible à l'utilisateur
  - avantages
    - implémentation facile
    - l'utilisateur peut décider de l'organisation de la mémoire
    - pas de composants matériels particuliers
  - inconvénients
    - le système n'offre aucun service de gestion de la mémoire
    - pas de contrôle des interruptions par le système
    - pas de multiprogrammation

Allocation de partitions multiples (1/6)

- en multiprogrammation, plusieurs processus résident en mémoire et le processeur bascule très rapidemen de l'un à l'autre
- comment allouer de la mémoire à chaque processus créé ?
- partitions de taille fixe
  - diviser la mémoire en un certain nombre de partitions de taille fixe
  - degré de multiprogrammation borné par le nombre de partitions

Allocation de partitions multiples (2/6)

- partitions de taille variable
  - mémoire constituée d'un ensemble de blocs alloués ou disponibles
    - lorqu'un processus a besoin de mémoire, un bloc disponible suffisamment grand est recherché
    - il y a 3 stratégies de recherche d'un bloc disponible
      - ***first-fit*** : le premier bloc suffisamment grand est alloué
      - ***best-fit*** : le plus petit bloc suffisamment grand est alloué
      - ***worst-fit*** : le plus grand bloc est alloué
    - *first-fit* et *best-fit* sont plus efficaces que *worst-fit* en rapidité et occupation mémoire
    - problème de fragmentation externe (espace mémoire non contigu)

Allocation de partitions multiples (3/6)

- Gestion des régions mémoire
  - comment maintenir une liste de blocs alloués et de blocs disponibles ?
    - décomposer logiquement la mémoire en régions de taille fixe et associer un bit à chaque région indiquant si elle est allouée ou disponible
      - pour un processus nécessitant N régions, il faut effectuer une recherche de N bits consécutifs à 0 ; cette recherche est une opération lente
    - maintenir une liste chaînée des blocs alloués et des blocs disponibles
      - tri par adresse de blocs --> fusion des blocs libres consécutifs
      - tri par taille de blocs --> recherche par la méthode du *best-fit*

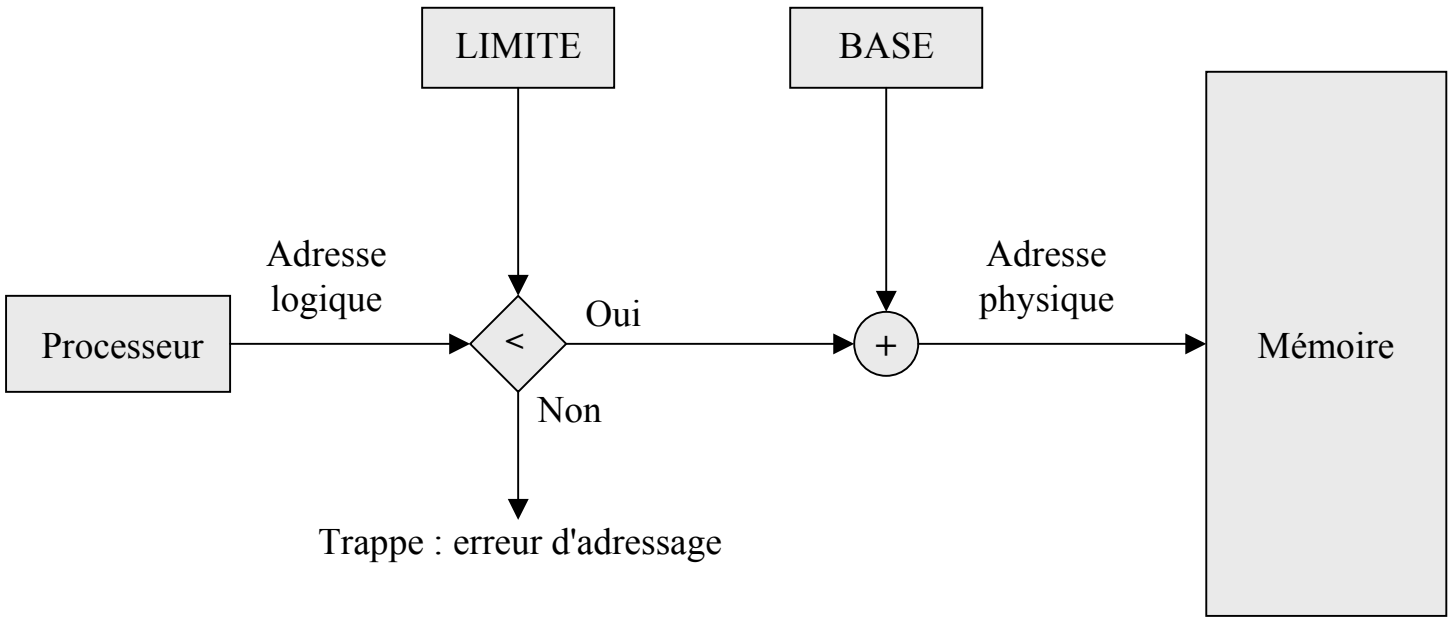
Allocation de partitions multiples (4/6)

- relogement des adresses
  - une conversion d'adresses dans un programme peut intervenir à chaque étape de son développement
    - compilation : si adresse de chargement connue alors génération du code avec des adresses absolues
    - chargement : si code relogeable alors conversion d'adresses lors du chargement
    - exécution : si possibilité de déplacement du programme pendant son exécution alors conversion des adresses à chaque déplacement en mémoire

Allocation de partitions multiples (5/6)

- relogement des adresses (suite)

  conversion en utilisant les 2 registres *base* et *limite* du processeur



Allocation de partitions multiples (6/6)

- compactage *(Compaction)*
  déplacer les blocs mémoire afin de placer toute la mémoire libre dans un seul bloc de grande taille
    - solution au problème de la fragmentation externe
    - n'est possible que si le relogement des adresses est effectué lors de l'exécution
    - quelques algorithmes de compactage :
      - déplacer tous les processus vers une extrémité de la mémoire; algorithme coûteux en temps
      - déplacer les processus vers les 2 extrémités de la mémoire en créant le bloc libre au milieu
      - choisir soigneusement les processus à déplacer

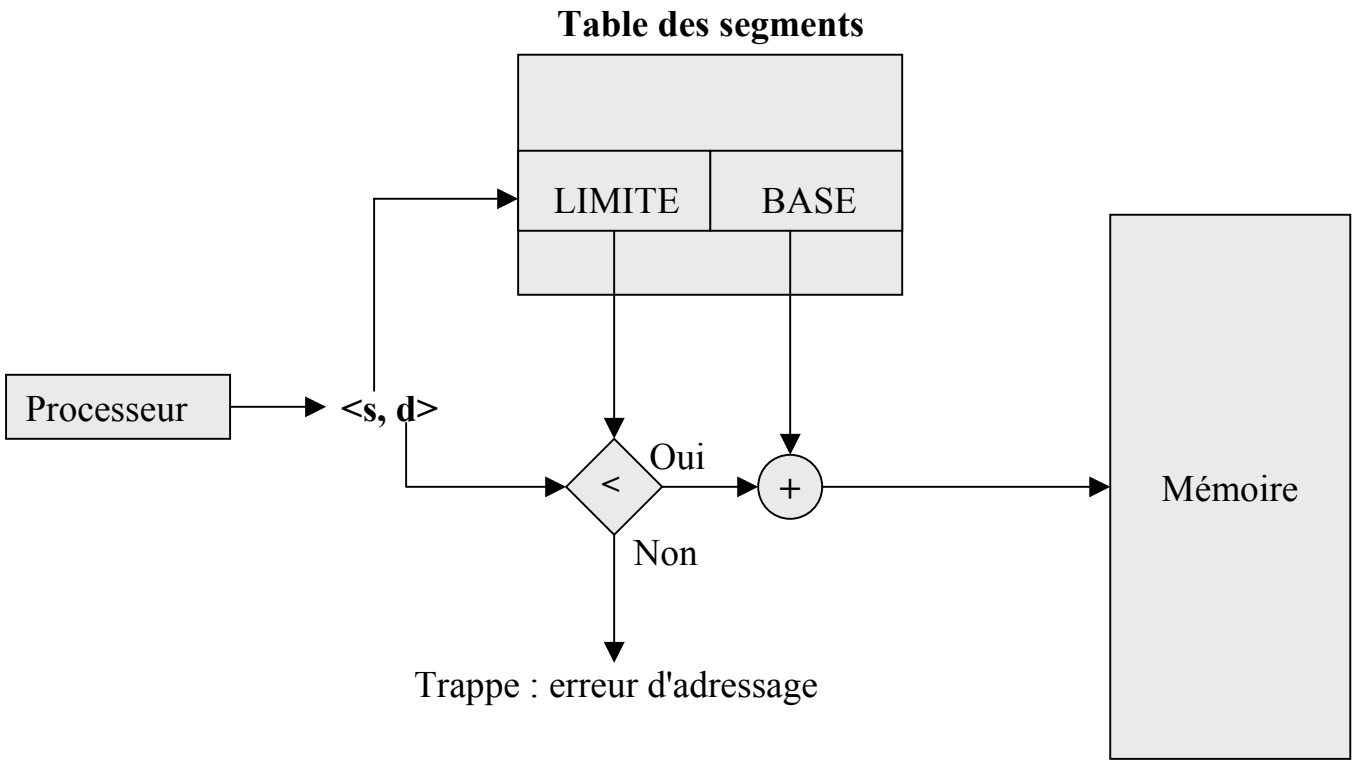
Swapping

- Un processus peut être temporairement stocké en mémoire secondaire lorsqu'il est en attente de ressources puis recharg en mémoire centrale lors de son exécution. Cette opération es appelée *va-et-vient* (swapping)
- Si le relogement d'adresses est effectué à la compilation ou a chargement, le processus doit être rechargé à la même adresse qu'il occupait précédemment en mémoire centrale
- Le swapping nécessite une mémoire secondaire (usuellement un disque) de capacité importante pour contenir les copies de images mémoire de tous les processus

Segmentation (1/6)

- Un espace d'adressage logique est composé de segments caractérisés par leur numéro et leur longueur
- Les adresses logiques sont formées d'un numéro de segment et d'un déplacement dans ce segment
- La conversion des adresses à 2 dimensions en adresses unidimensionnelles est effectuée par le processeur via une table de segments
- Le numéro de segment est utilisé comme indice dans la table des segments
- Chaque poste de cette table comporte au moins un champ *Base* et un champ *Limite*

Segmentation (2/6)



Segmentation

Segmentation (3/6)

- Implantation des tables de segments
  - la table des segments peut être placée soit dans des registres du processeur soit en mémoire
  - s'il y a un grand nombre de segments, la table de segments est conservée en mémoire et pointée par un registre de base de la table de segments (RBTS), un autre registre RTTS contient la taille de la table

Segmentation (4/6)

- Implantation des tables de segments (suite)
  - quand la table est conservée en mémoire, la conversion d'une adresse <s,d> consiste à :
    - vérifier que 0 < **s** < RTTS, sinon, une trappe est provoquée
    - additionner **s** et RBTS afin d'obtenir l'adresse du poste concerné de la table des segments
    - vérifier que 0 < **d** < LIMITE, sinon, une trappe est déclenchée
    - additionner d et le champ LIMITE du poste concerné afin d'obtenir l'adresse physique correspondante
  - pour réduire le temps de cette conversion assez coûteuse, on utilise des registres associatifs contenant les postes de la table de segments les plus récemment accédés

Segmentation (5/6)

- Protection de segments
  - le mécanisme matériel de conversion d'adresse vérifie que les bits de protection associés à chaque segment afin de prévenir les accès illégaux à la mémoire tels que les écritures dans les segments accessibles en lecture seule
  - cette protection matérielle peut également permettre de détecter des erreurs de programmation

Segmentation (6/6)

- Partage de segments
  - possibilité de partager des segments entre processus
  - un segment est partagé lorsque des postes dans les tables de segments de plusieurs processus pointent sur le même emplacement physique
  - ce mécanisme permet de ne pas dupliquer de segment de code lorsque plusieurs processus exécutent le même programme ou utilisent les mêmes routines

Pagination

- Principe
- Mécanismes matériels
- Implantation de la table des pages
- Partage de pages
- Protection

Pagination (1/8)

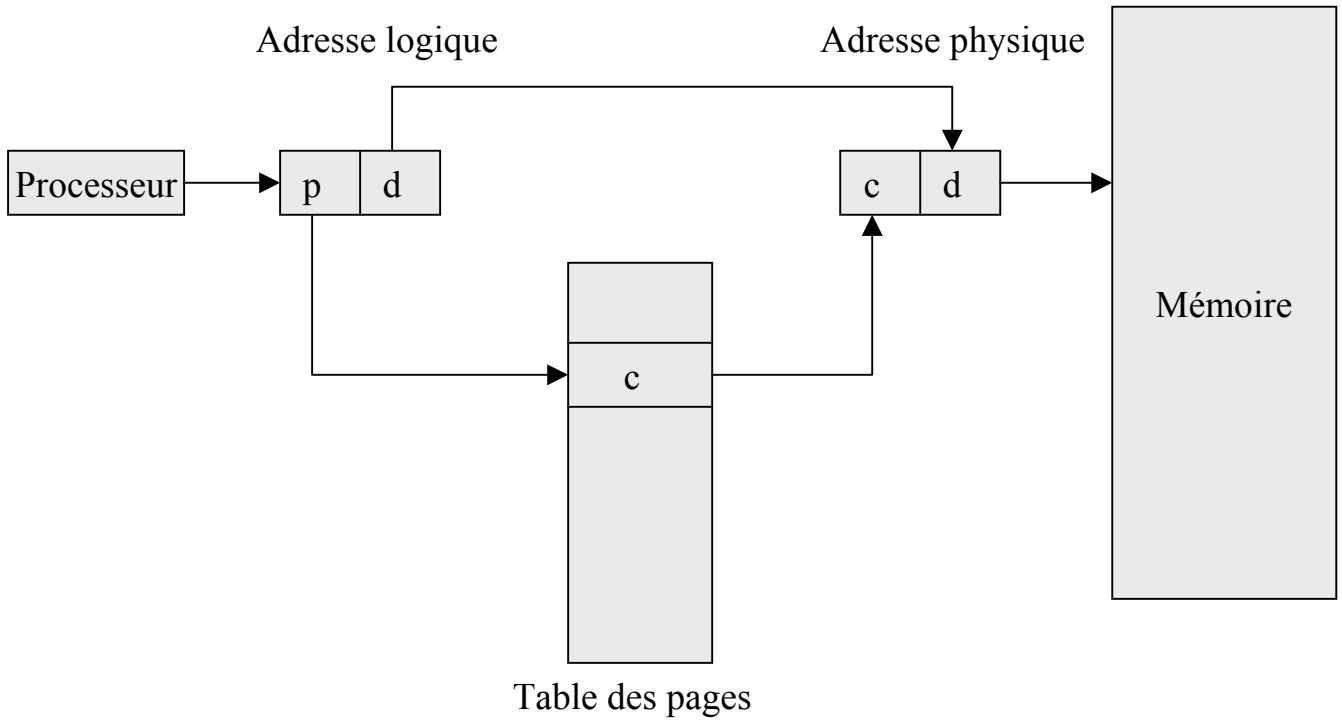
- Principe
  - comme la mémoire allouée à un processus doit être contiguë, la mémoire fragmentée ne peut être utilisée
  - la fragmentation peut être traitée par le compactage
  - il est aussi possible d'utiliser la pagination qui permet à la mémoire d'un processus d'être non contiguë
- Mécanismes matériels
  - la mémoire physique est un ensemble de blocs de taille fixe, appelés cases
  - la mémoire logique est un ensemble de blocs de taille fixe, appelés pages

Pagination (2/8)

- Mécanismes matériels (suite)
  - la pagination requiert un support matériel
  - chaque adresse générée par le processeur est composée d'un numéro de page *p* et d'un déplacement *d* dans la page. Le numéro de page est utilisé comme indice dans une table des pages qui contient l'adresse de base de chaque page en mémoire physique. Cette adresse de base est combinée avec le déplacement fourni pour obtenir l'adresse physique correspondante
  - la taille des pages est définie par le processeur et c'est généralement une puissance de 2 variant de 512 à 4096 mots mémoire, selon l'architecture de l'ordinateur

Pagination (3/8)

- Mécanismes matériels (suite)



Pagination

Pagination (4/8)

- Implantation de la table des pages
  - en général, la taille de la table des pages est importante, elle doit être placée en mémoire centrale pointée par un registre de base de la table des pages RBTP. Le changement de table de pages ne nécessite que la modification de l'adresse contenue dans ce registre
  - comme pour la segmentation, 2 accès mémoire sont nécessaires par adresse logique. Une solution à base de registres associatifs peut être utilisée pour accélérer les accès à la mémoire

Pagination (5/8)

- Implantation de la table des pages (suite)
  - si la taille de la table des pages est importante, il peut être difficile de l'implanter sous la forme d'une table classique
    - si les adresses mémoire sont sur 32 bits, l'espace d'adressage est de 4 Go, si la taille d'une page est 2048 octets, le nombre total de pages est 2097152. Si un poste de la table occupe 4 octets, la taille totale de la table est de l'ordre de 1.5 Go
  - on utilise un mécanisme de table à 2 niveaux, une case contient le catalogue de la table des pages qui est répartie dans plusieurs cases non forcément contiguës
    - dans l'exemple ci-dessus, la table serait ainsi décomposée en un catalogue de 512 pointeurs et 512 pages chacune contenant 512 pointeurs

Pagination (6/8)

- Implantation de la table des pages (suite)
  - quand une référence à une page p est effectuée, le processeur calcule *p1 = p/512* et l'utilise comme indice dans le catalogue afin d'obtenir l'adresse de la table concernée. Il calcule ensuite *p2 = p modulo 512* et l'utilise comme indice dans la table obtenue afin de trouver l'adresse physique correspondante
  - ce mécanisme permet de rendre la table des pages non contiguë et de la répartir en mémoire. Il permet aussi de réduire sa taille en ne créant des tables que pour les pages effectivement utilisées

Pagination (7/8)

- Implantation de la table des pages (suite)

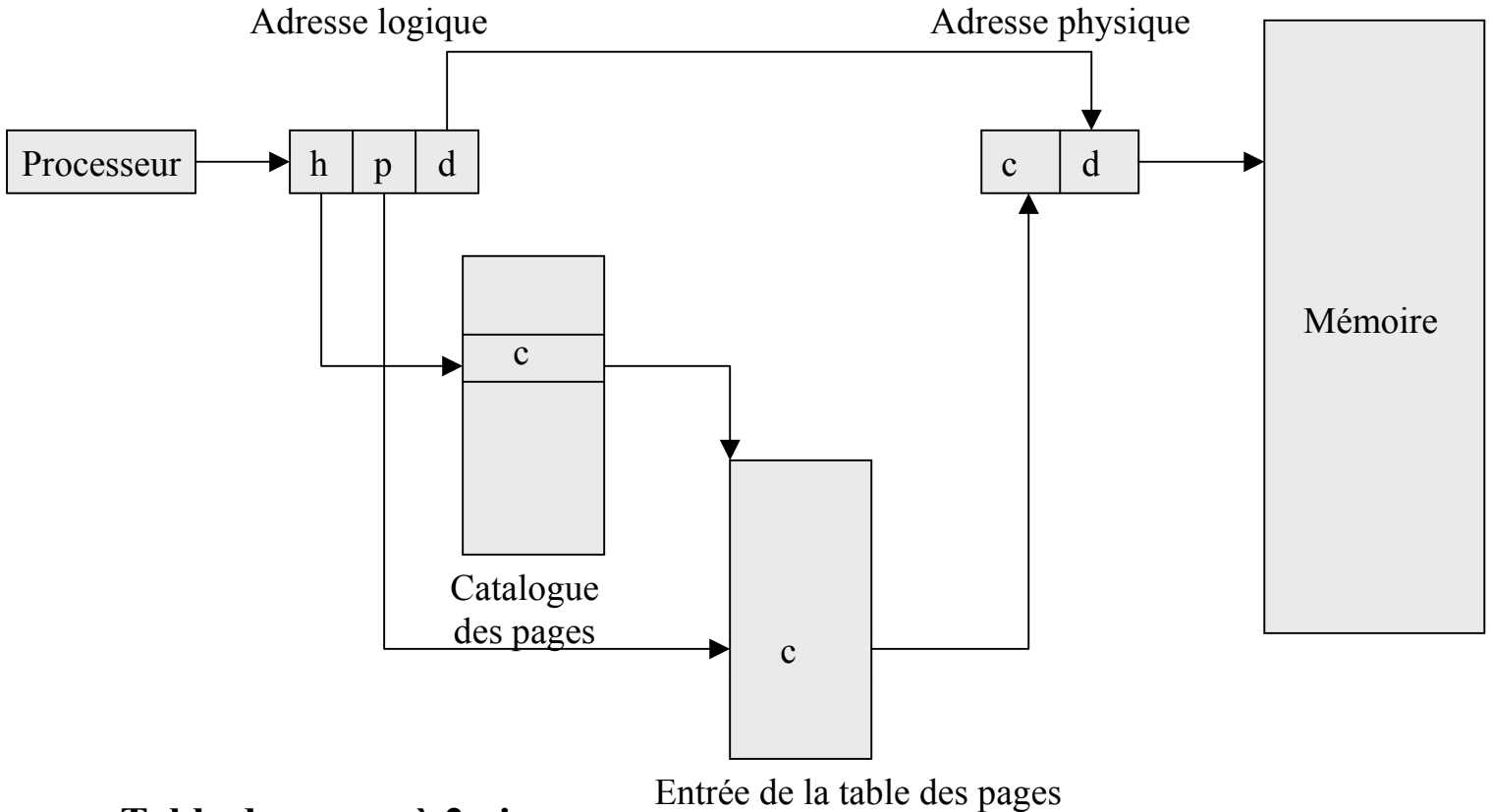


Table des pages à 2 niveaux

Pagination (8/8)

- Partage de pages
  - le partage de pages est similaire au partage de segments. Si une page est partagée entre plusieurs processus, le poste correspondant des tables de pages contient l'adresse de la même case
- Protection
  - la protection de la mémoire dans un environnement paginée est accomplie en associant des bits de protection à chaque page, ces bits sont stockés dans la table des pages
  - lorsque qu'une tentative d'accès invalide est effectuée, le processeur déclenche une trappe

Segmentation paginée

- Principe
  - la segmentation et la pagination ont des avantages et des inconvénients
  - il est possible de combiner ces 2 méthodes pour obtenir une gestion efficace de la mémoire
  - deux combinaisons sont possibles
    - une seule table de pages existe : l'adresse obtenue après application de la segmentation est convertie d'après cette tab
    - chaque segment possède une table de ses pages
  - exemples de ces 2 possibilités
    - processeur i386
    - Multics

Chapitre 4: Gestion de la mémoire virtuelle

- Objectif
- Chargement de page à la demande
- Performances
- Remplacement de page
- Algorithmes de remplacement de page
- Allocation de cases
- Ecroulement
- Autres considérations
- Chargement de segment à la demande

Objectif

- L'examen de programmes réels montre que, dans la majorité des cas, il n'est pas nécessaire de charger le programme entier en mémoire
- La possibilité d'exécuter un programme partiellement chargé en mémoire a de nombreux avantages
- La mémoire virtuelle offre une mémoire de grande taille même si la mémoire physique est très réduite
- La mémoire virtuelle est généralement implémentée grâce au chargement de page à la demande

Chargement de page à la demande

- Un système de chargement de page à la demande est similaire à un système à pagination utilisant le swapping : les processus résident en mémoire secondaire et sont chargés en mémoire lorsqu'ils sont exécutés. Plutôt que de charger le processus entier en mémoire, un tel système ne charge que les pages qui seront utilisées lors de l'exécution
- Ce schéma nécessite des dispositifs matériels. Un bit de présence est associé à chaque poste de la table des pages.
  Bit à 1 ==> page présente en mémoire
  Bit à 0 ==> page stockée sur disque
  Lors d'un accès mémoire, si le bit de présence de la page est égal à 0, une trappe, appelée *défaut de page*, est provoquée

Performances (1/3)

- Le chargement de page à la demande peut avoir un effet important sur la performance du système
- Temps d'accès effectif = $(1 - p) * t_{am} + p * t_{sdp}$

où

p : probabilité d'un défaut de page

t_{am} : temps d'accès à la mémoire physique

t_{sdp} : temps de service d'un défaut de page

Pour calculer ce temps, il faut estimer le temps de service d'un défaut de page

Performances (2/3)

- Tâches à accomplir lors d'un défaut de page
  - provoquer une trappe
  - sauvegarder l'état du processus et la valeur des registres
  - déterminer que la trappe correspond à un défaut de page
  - vérifier que la référence mémoire était valide et déterminer l'emplacement de la page sur le disque
  - lancer une lecture de la page
    - attendre que la requête soit prise en compte par le périphérique
    - attendre le positionnement des têtes de lecture-écriture du disque
    - transférer la page depuis le disque vers une case libre
  - allouer le processeur à un autre processus pendant l'attente du disq
  - recevoir une interruption du disque signalant la fin de la lecture
  - sauvegarder l'état du processus courant et la valeur des registres

Performances (3/3)

- Tâches à accomplir lors d'un défaut de page (suite)
  - déterminer que l'interruption provenait du disque
  - modifier les tables pour indiquer que la page correspondante est présente en mémoire
  - restaurer les valeurs de registres et l'état du processus puis reprendre l'instruction interrompue

Remplacement de page (1/2)

- Le chargement de page à la demande accroît le degré de multiprogrammation
- Lorsqu'un défaut de page intervient et qu'aucune case mémoire n'est libre, une politique de remplacement de page est adoptée
- Routine de gestion de défaut de page
  - trouver l'emplacement de la page demandée sur le disque
  - trouver une case libre :
    - s'il existe une case libre, l'utiliser
    - sinon, utiliser un algorithme de remplacement pour sélectionner une case victime, écrire son contenu sur disque et modifier les tables
  - charger la page désirée dans la case libre, modifier les tables
  - reprendre l'exécution du processus

Remplacement de page (2/2)

- Si aucune case mémoire n'est libre, 2 entrées-sorties sont nécessaires
  - écriture sur le disque du contenu de la case libérée
  - lecture de la page concernée
- En utilisant un bit de modification associé à chaque page, on peut éviter l'écriture sur disque de la page à remplacer si elle n'a pas été modifiée
- Pour implémenter le chargement de page à la demande, un algorithme d'allocation de cases et un algorithme de remplacement de page sont nécessaires

Algorithmes de remplacement de page

- FIFO
- Optimal
- LRU (*Least Recently Used*)
- Approximations de LRU
  - seconde chance
  - LFU (*Least Frequently Used*)
  - MFU (*Most Frequently used*)

FIFO

- associer à chaque page la date de son chargement en mémoire
- quand une page doit être évincée, la plus ancienne est choisie
- les performances ne sont pas toujours bonnes
- anomalie de Belady : les défauts de page peuvent croître avec l'augmentation de cases mémoire

OPTIMAL

- taux de défauts de page le plus bas et pas d'anomalie de Belady
- Lors d'un remplacement de page, la page qui sera choisie sera celle qui ne sera pas référencée pendant la plus grande période de temps
- difficile à implémenter
- utilisé surtout comme référence théorique

LRU (*Least Recently Used*)

- utilisation du passé récent comme approximation du futur proche
- la page qui n'a pas été accédée depuis la plus grande période de temps sera choisie
- LRU associe à chaque page la date à laquelle elle a été accédée pour la dernière fois
- la page qui possède la date la plus ancienne est choisie dans un remplacement de page
- approximation de l'*algorithme optimal* par utilisation des références passées plutôt que des références futures
- LRU assez utilisé et possède de bonnes performances
- difficulté d'implémenter cet algorithme car nécessité d'avoir certains dispositifs matériels afin de conserver la date du dernier accès pour chaque page

Approximations de LRU

- Peu de processeurs offrent de dispositifs matériels suffisants pour permettre l'implémentation de l'algorithme LRU. Généralement, le processeur gère un bit de référence associé à chaque page et positionné à 1 lors de chaque accès à la page. Après un certain temps, on peut déterminer les pages accédées mais pas l'ordre d'utilisation. Ceci permet d'élaborer des algorithmes d'approximation de LRU :
  - seconde chance
  - LFU (*Least Frequently Used*)
  - MFU (*Most Frequently Used*)

Approximations de LRU

- seconde chance
  - si le bit de référence d'une page est à 0, cette page est remplacée
  - si le bit de référence est à 1, une seconde chance est donnée à la page : son bit de référence est remis à 0, sa date de chargement est positionnée à la date courante et la page suivante est examinée
  - implémentation par une liste circulaire avec un pointeur indiquant la page suivante à évincer

Approximations de LRU

- LFU (*Least Frequently Used*)
  - LFU gère un compteur de références associé à chaque page et incrémenté lors de chaque accès à la page
  - la page possédant le plus petit compteur est choisie pour être remplacée
- MFU (*Most Frequently Used*)
  - l'algorithme MFU est basé sur l'hypothèse qu'une page possédant le plus petit nombre de références vient juste d'être chargée en mémoire et n'a pas encore été utilisé

LFU et MFU sont peu utilisés car leur implémentation est coûteuse et l'approximation de l'algorithme optimal est assez médiocre.

Allocation de cases (1/3)

- Dans un système multiprogrammé, le SE doit allouer équitablement des cases mémoire à chaque processus
- Le nombre minimal de cases allouées à un processus est défini par l'architecture matérielle, le nombre maximal dépend de la taille de la mémoire disponible
- 2 stratégies d'allocation
  - allocation fixe
  - allocation à priorité

Allocation de cases (2/3)

- Allocation fixe
  - allocation équitable : si M cases et N processus, allouer M/N cases pour chaque processus
  - allocation proportionnelle : allocation selon la taille du processus
    - $\mathbf{s_i}$: taille de la mémoire virtuelle du processus $\mathbf{p_i}$
    - $\mathbf{S} = \sum \mathbf{s_i}$
    - **M** : nombre total de cases
    - $\mathbf{a_i}$: allocation pour $\mathbf{p_i}$
    - $\mathbf{a_i} = (\mathbf{s_i} / \mathbf{S}) * \mathbf{M}$

Allocation de cases (3/3)

- Allocation à priorité
  - utiliser une stratégie d'allocation proportionnelle à la priorité des processus et non plus à leur taille
  - si un processus génère un défaut de page
    - sélectionner une de ses cases pour le remplacement
    - sélectionner pour le remplacement une case d'un processus moins prioritaire

Ecroulement (1/3)

- Si un processus n'a pas suffisamment de pages, le taux de défauts de page est assez élevé ==>
  - diminution de l'utilisation de l'UC
  - le système tente alors d'augmenter le degré de multiprogrammation
  - un nouveau processus est admis dans le système
- Un processus est en état d'écroulement *(Thrashing)* s'il passe plus de temps à provoquer des défauts de page qu'à s'exécuter
- Afin d'éviter l'écroulement, on utilise la notion d'ensemble de travail et le contrôle de la fréquence de défauts de page

Ecroulement (2/3)

- Ensemble de travail (*Working Set*)
  - ensemble de travail (Δ) : ensemble de pages qu'un processus utilis activement
  - Δ est basé sur le principe de localité
  - Δ est calculé par l'examen des références mémoire effectuées par un processus pendant un nombre de références fixé, l'ensemble des pages comprises dans ces références constituent Δ
  - allocation d'autant de cases que la taille de Δ pour chaque processus
  - si $\sum\Delta_i > M$ (nombre total de cases) ==> un processus est suspendu et ses cases mémoire sont libérées pour les autres processus
  - le maintien des Δ est difficile et l'implémentation est coûteuse en temps

Ecroulement (3/3)

- Fréquence de défaut de page
  - quand le taux de défaut de page est trop élevé, cela signifie que les processus qui s'exécutent ont besoin de plus de cases
  - quand le taux de défaut de page est trop bas, cela signifie que trop de cases sont allouées aux processus.
  - établir une limite inférieure et une limite supérieure à ce taux
  - quand ce taux dépasse la limite supérieure, une case de plus est allouée au processus concerné
  - quand ce taux descend sous la limite inférieure, une case est prise à ce processus
  - le système peut suspendre un processus si le taux de défaut de page augmente et si aucune case n'est disponible

Autres considérations

- Remplacement global ou local
- Préchargement
- Taille de page
- Structure d'un programme
- Verrouillage de page

Remplacement global ou local

- remplacement global : un processus choisit une case parmi l'ensemble de toutes les cases, même si elle est allouée à un autre processus
- dans un remplacement global, aucun processus ne contrôle son propre taux de défauts de page et le temps d'exécution dépend du comportement des autres processus
- remplacement local : un processus choisit une case parmi celles qui lui sont déjà allouées.

Pré-chargement

- taux de défauts de page élevé au début de l'exécution d'un processus
- pré-chargement : tentative d'éliminer ce taux initial élevé de défauts de page en chargeant toutes les pages qui seront nécessaires
- Taille de page
  - le choix de la taille d'une page obéit à des contraintes contradictoire
  - une grande taille pour réduire la taille de la table des pages
  - une petite taille pour réduire la fragmentation interne
  - une grande taille pour minimiser le nombre de défauts de page

  La tendance actuelle consiste à utiliser des pages de taille importante même si cela augmente la fragmentation interne.

Taille de page

- Le choix de la taille d'une page obéit à des contraintes contradictoires
- une grande taille pour réduire la taille de la table des pages
- une petite taille pour réduire la fragmentation interne
- une grande taille pour minimiser le nombre de défauts de page

La tendance actuelle consiste à utiliser des pages de taille importante même si cela augmente la fragmentation interne

Structure d'un programme (1/2)

- Normalement, l'utilisateur ne se préoccupe pas des mécanismes de la mémoire virtuelle. Dans certains cas, toutefois, la performance du système peut être améliorée par la connaissance des stratégies sous-jacentes de chargement à la demande
- un processus, dont les pages ont une taille de 128 mots, exécute le programme suivant :

```
int  a[128][128];
for (j=0; j<128); j++)
  for (i = 0; i<128; i++)
            a[i][j] = 0;
```

Structure d'un programme (2/2)

- si le tableau est rangé en mémoire ligne par ligne, chaque ligne occupe une page. Si le système alloue moins de 128 cases au processus, son exécution provoquera 128*128 =16384 défauts de page
- on modifie le programme précédent :

```
int   a[128][128];
for (i=0; i < 128; i++)
    for (j=0; j < 128; j++)
      a[i][j]=0;
```

Le programme met à 0 tous les mots d'une page avant de passer à la suivante. Le nombre de défauts de page est maintenant de 128.

Verrouillage de page

- quand la mémoire virtuelle est utilisée, il est parfois nécessaire de verrouiller des pages en mémoire physique pour empêcher qu'elles soient choisies lors de remplacements
- un tel verrouillage peut intervenir lors des entrées-sorties
- verrouillage réalisé par un bit associé à chaque case
- ce bit de verrouillage peut être utilisé lors d'un remplacement de page

Chargement de segment à la demande

- Quand le processeur ne fournit pas de dispositifs matériels pour le chargement de pages à la demande, le chargement de segment à la demande peut être utilisé
- Exemple : OS/2 sur i286
- le chargement de segment à la demande est beaucoup plus coûteux que le chargement de page à la demande
- pour l'efficacité d'un système n'implémentant pas la notion de page, il est préférable de gérer une mémoire physique de taille importante sans mémoire virtuelle.

Chapitre 5: Gestion des entrées-sorties

- Module d'entrées-sorties (E/S)
- Types de périphériques
- Pilotes de périphériques
- Gestion des disques

Module d'E/S (1/2)

- Le module d'E/S du noyau optimise le traitement des E/S afin d'améliorer les performances globales du système
- Services du module d'E/S
  - ordonnancement des entrées-sorties
  - bufferisation
  - utilisation de caches
  - spooling
  - traitement des erreurs

Module d'E/S (2/2)

- Techniques de communication des E/S
  - entrées-sorties programmées (*Programmed I/O*)
  - entrées-sorties commandées par les interruptions (*Interrupt-driven I/O*)
  - accès direct à la mémoire (*DMA*)

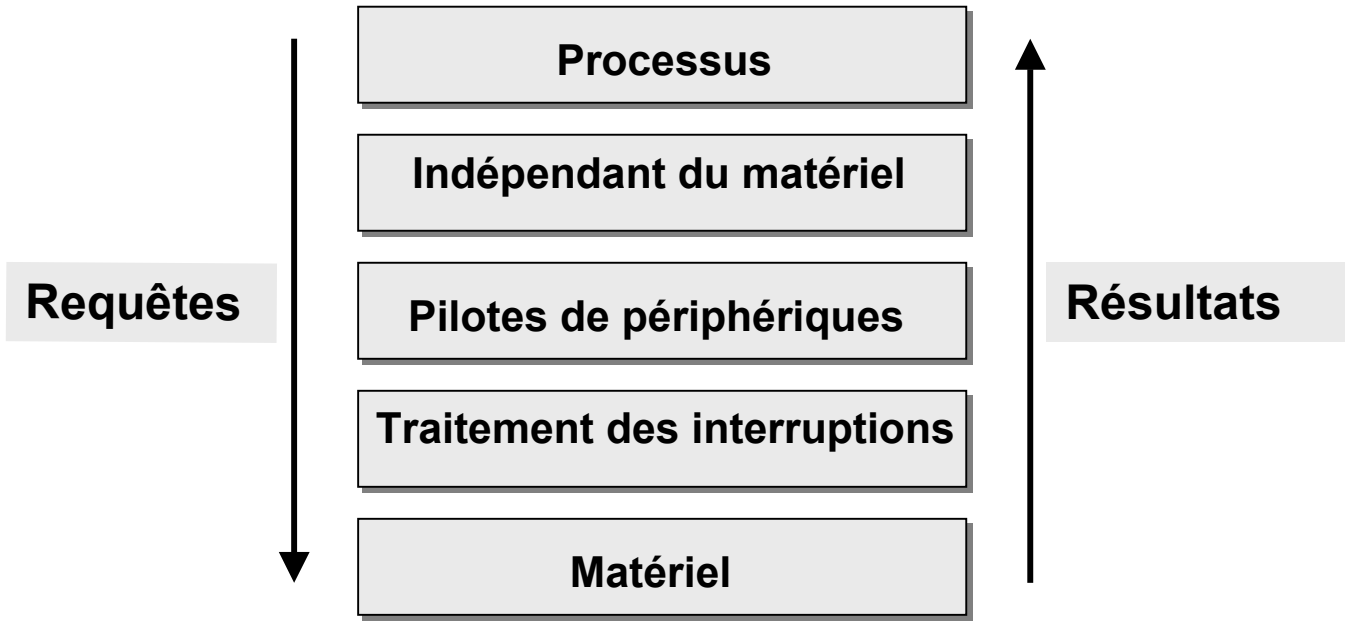
Types de périphériques

- Périphérique caractères
  - flots de caractères sans structure de blocs
  - caractères non adressables
  - pas d'opération de positionnement
  - exemples : terminaux, imprimantes, etc…
- Périphérique blocs
  - bloc de taille fixe (128 - 1024 octets)
  - bloc adressable
  - exemple : disques

Pilotes de périphériques (1/2)

- Le pilote de périphérique (*Device Driver*) est un programme qui commande le fonctionnement élémentaire d'un périphérique
- Le pilote gère directement l'interface matérielle du périphérique, traite les interruptions émises par celui-ci, détecte et traite les cas d'erreur
- Le pilote est transparent aux processus

Pilotes de périphériques (2/2)



Structure du logiciel de gestion des entrées-sorties

Gestion des disques

- Structure d'un disque
- Principaux algorithmes d'ordonnancement
  - FCFS (*First Come First Served*)
  - SSTF (*Shortest Seek Time First*)
  - SCAN

Structure d'un disque

- Un disque a généralement plusieurs **plateaux** (*platters*)
- Chaque face d'un plateau contient des **pistes** (*tracks*) concentriques
- Chaque piste est décomposée en **secteurs** (*sectors*)
- A chaque face correspond une tête de lecture-écriture
- Pour accéder à une donnée sur le disque, le processeur fournit une adresse au contrôleur sous la forme <*t*, *p*, *s*>
  où *t* : numéro de la tête (ou la face)
  p : numéro de la piste sur la face *t*
  s : numéro de secteur dans la piste *p* sur la face *t*

Algorithmes d'ordonnancement (1/2)

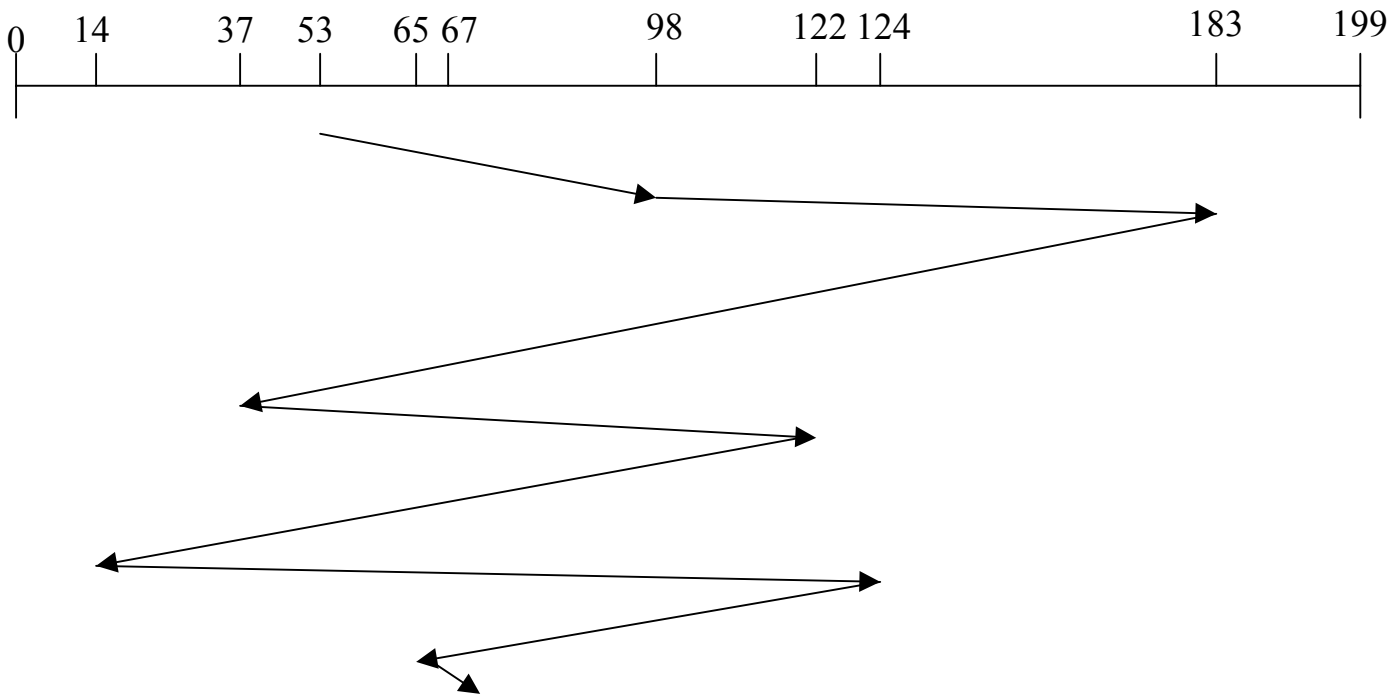
- Le temps d'accès à une donnée sur un disque est la somme de 3 temps :
  - temps de positionnement de la tête de lecture-écriture sur le cylindre approprié (*Seek Time*)
  - temps de latence (*Rotational Latency*) : temps d'attente du passage du bloc désiré sous la tête
  - temps de transfert des données entre le disque et la mémoire centrale

Algorithmes d'ordonnancement (2/2)

- Chaque requête d'E/S peut être mise dans une file d'attente correspondant au périphérique concerné
- Quand un processus effectue une requête d'E/S, il fait appel au système d'exploitation et spécifie :
  - le sens de l'entrée-sortie (lecture ou écriture)
  - l'adresse disque concernée (disque, cylindre, plateau, bloc)
  - l'adresse mémoire d'un tampon
  - la taille des données à transférer
- Divers algorithmes d'ordonnancement permettent de choisir la prochaine requête à satisfaire.
- Illustration par un exemple sur les requêtes intervenant sur les pistes : 98, 183, 37, 122, 14, 124, 65, 67

FCFS

- Le plus simple et le plus facile à programmer, mais il ne fournit pas le meilleur temps moyen d'E/S



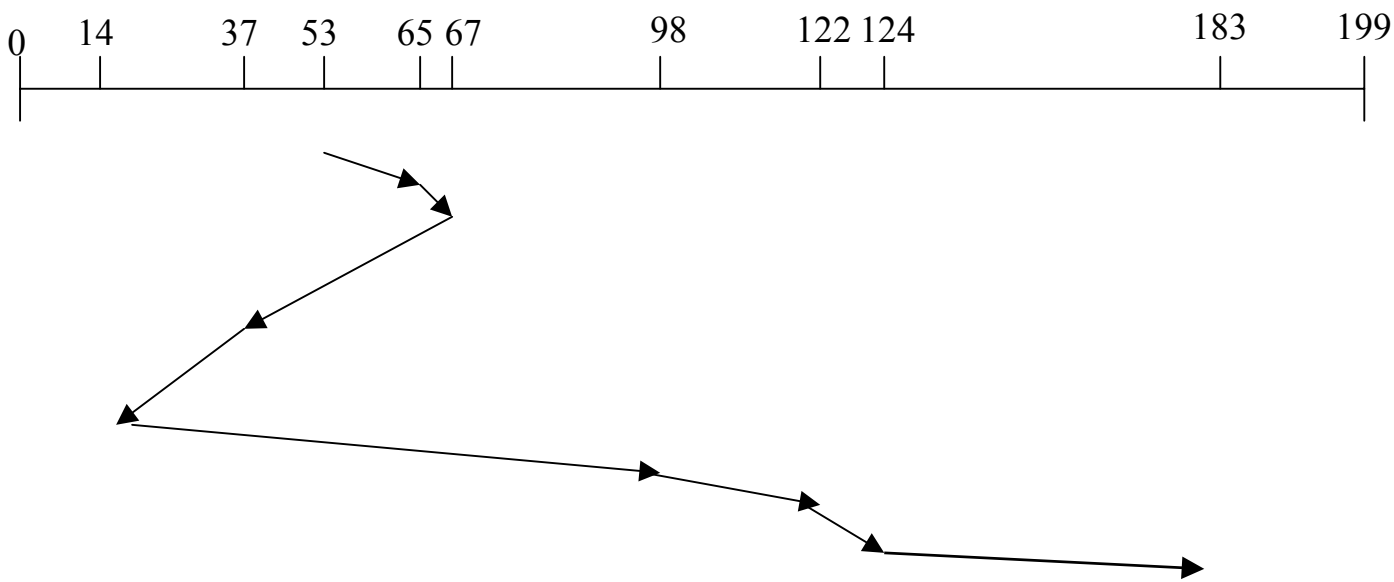
Algorithme FCFS

SSTF (***S**hortest **S**eek **T**ime **F**irst*)(1/2)

- choix de la requête impliquant un temps de déplacement minimal de la tête, i.e. la requête intervenant sur la piste la plus proche de la piste courante
- amélioration des performances par rapport à FCF
- n'est pas optimal
- peut donner lieu à des famines

SSTF (2/2)

•

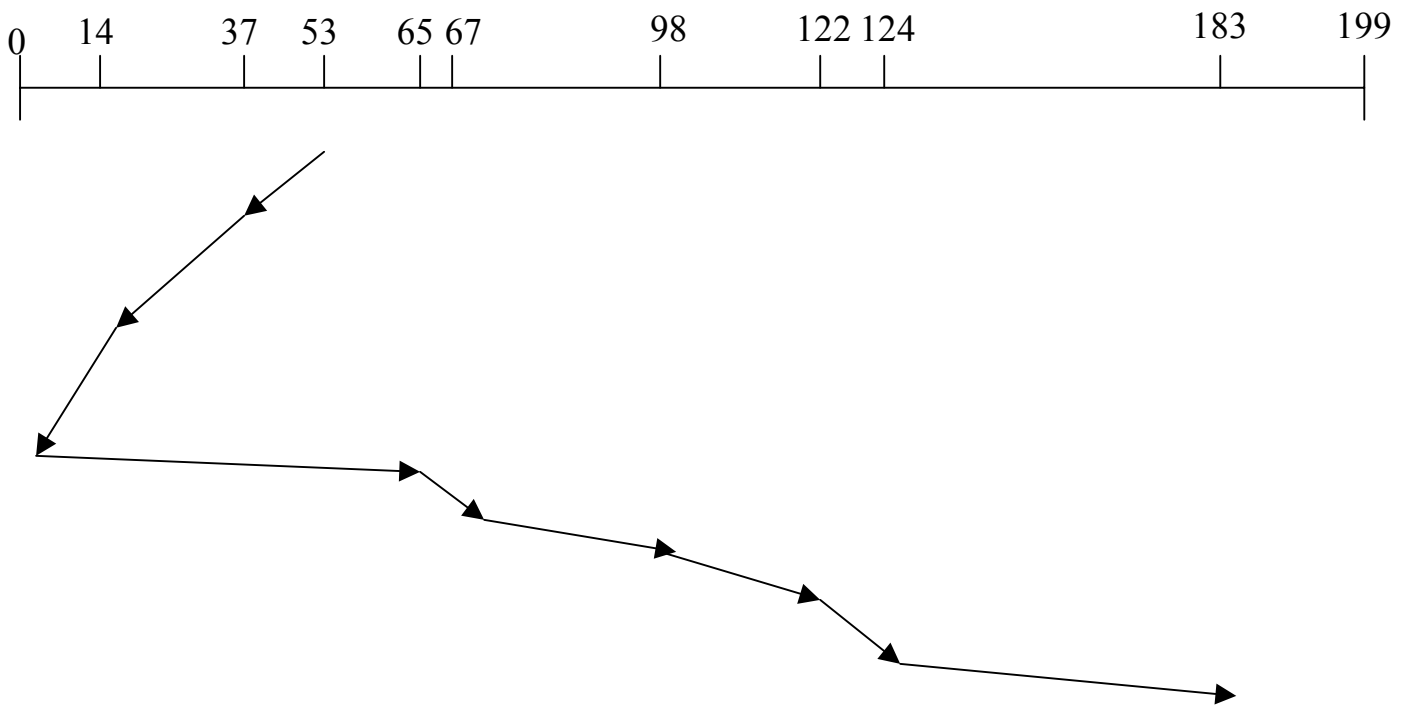


Algorithme SSTF

SCAN (ascenseur) **(1/2)**

- la tête démarre depuis une extrémité et se déplace vers l'autre en traitant les requêtes concernant la piste courante jusqu'à ce qu'elle atteigne l'autre extrémité. A ce point, son déplacement s'inverse et elle revient vers la première extrémité en traitant les requêtes suivantes
- cet algorithme privilégie les requêtes concernant le milieu du disque

SCAN (2/2)

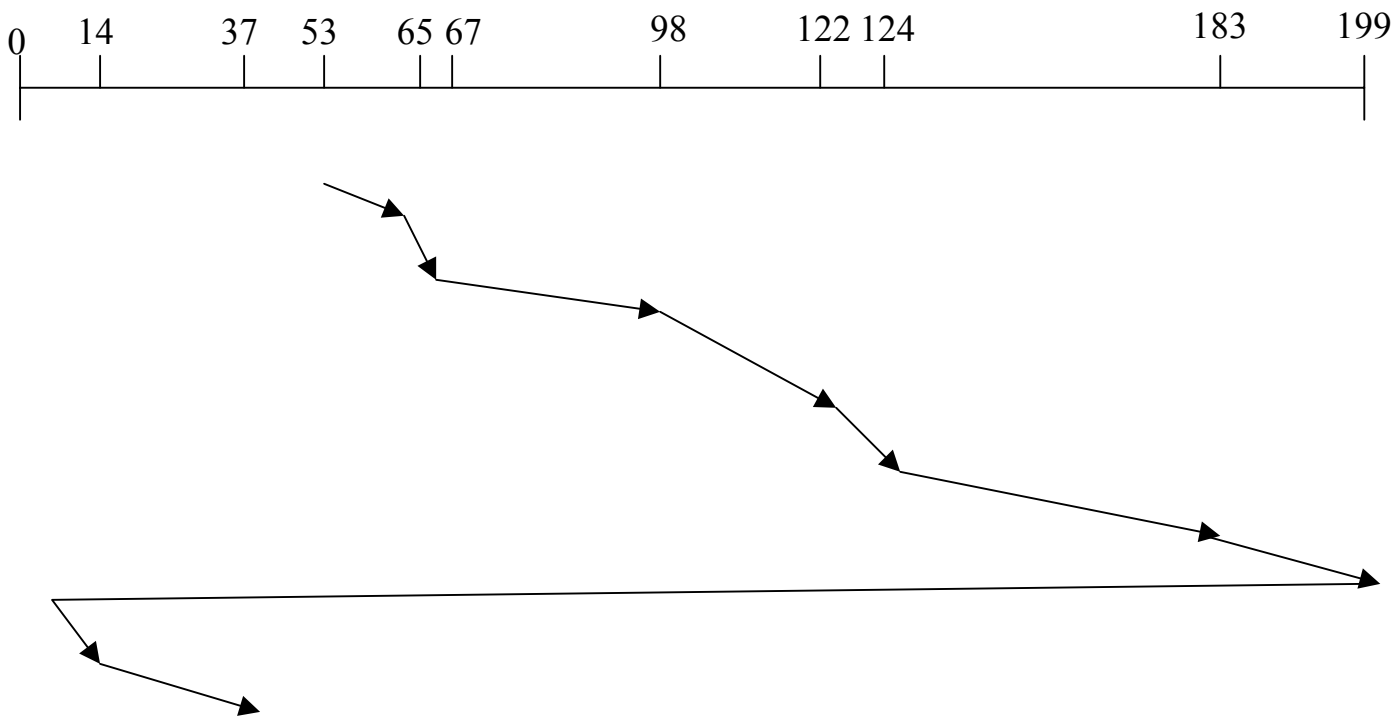


Algorithme SCAN

C-SCAN (Circular SCAN) (1/2)

- une variante de SCAN destinée à fournir un temps d'attente uniforme.
- la tête se déplace d'une extrémité à l'autre en traitant les requêtes, quand elle atteint l'autre extrémité, elle revient immédiatement au début du disque sans traiter de requête sur son parcours puis reprend son déplacement en traitant les autres requêtes.

C-SCAN (2/2)

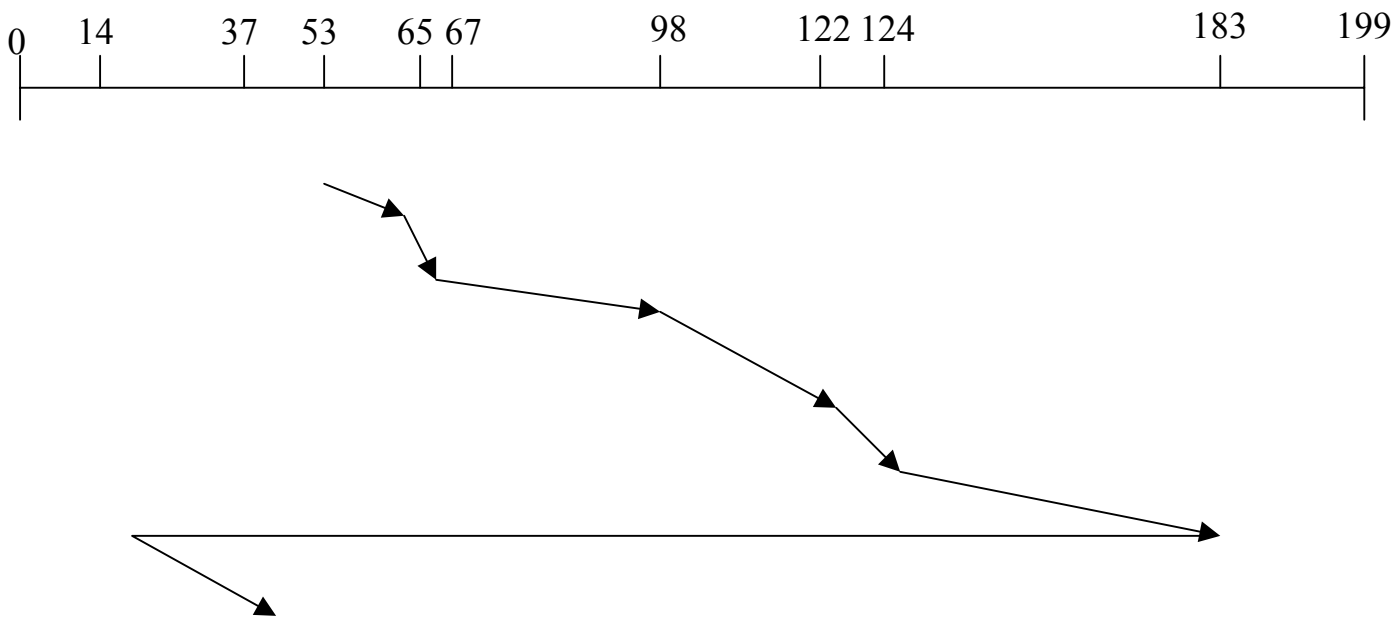


Algorithme C-SCAN

LOOK et **C-LOOK** **(1/2)**

- les algorithmes SCAN et C-SCAN déplacent la tête d'une extrémité vers l'autre. En pratique, ils ne sont pas implémentés ainsi. En général, la tête est déplacée aussi longtemps qu'il existe une requête dans la direction du déplacement. Dès qu'il n'y a plus de requêtes à traiter dans la direction courante, le déplacement est inversé
- ces versions sont appelées LOOK et C-LOOK

LOOK et C-LOOK (2/2)



Algorithme C-LOOK

Chapitre 6: Gestion des fichiers

- **Fonctions d'un système de fichiers**
- **Organisation logique des fichiers**
- **Protection des fichiers**
- **Structure logicielle du gestionnaire du système de fichiers**

Fonctions d'un système de fichiers (1/3)

- **Notion de fichier**
  - un fichier est une suite de bits, d'octets, de lignes ou d'enregistrements
  - la signification est définie par le créateur
  - un fichier possède un nom, un type, une date de création, le nom de son créateur, sa longueur, etc…
  - un fichier est généralement stocké sur disque et composé de blocs de longueur fixe
  - le système d'exploitation convertit les enregistrements logiques gérés par les applications en blocs physiques gérés par le disque

Fonctions d'un système de fichiers (2/3)

- **Opérations de gestion de fichiers**
  Le système d'exploitation fournit des appels système permettant de :
  - créer un fichier
  - écrire dans un fichier
  - lire depuis un fichier
  - supprimer un fichier

Fonctions d'un système de fichiers (3/3)

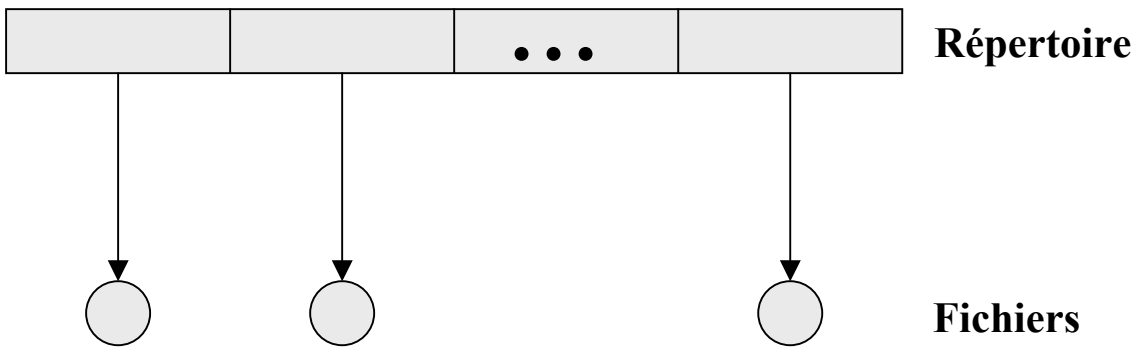
- **Méthodes d'accès**
  - accès séquentiel
  - accès direct
  - autres accès

Organisation logique des fichiers

- **Structure des répertoires**
  Un répertoire est essentiellement une table de symboles.
  Le système d'exploitation utilise le nom symbolique du fichier stocké dans un répertoire puis accède aux blocs composant le fichier
  - Répertoire à un niveau
  - Répertoire à deux niveaux
  - Répertoires arborescents
  - Répertoires sous forme de graphes sans cycle
  - Répertoires sous forme de graphes avec cycles

Structures des répertoires (1/9)

- **Répertoire à un niveau** (*Single-Level Directory*)
  - un seul répertoire pour tous les utilisateurs
  - problème de nommage
  - problème de groupage

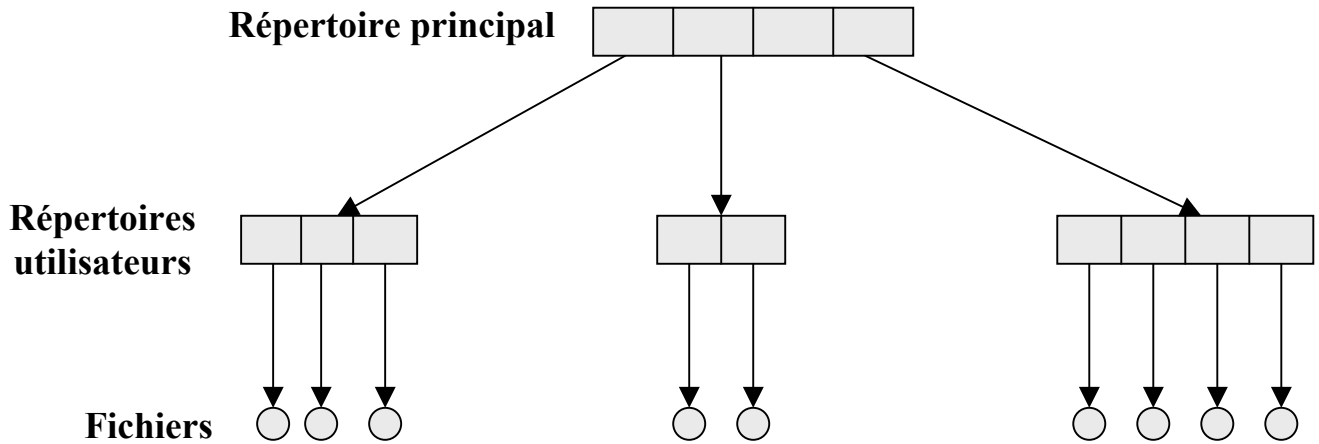


Répertoire à un niveau

Structures des répertoires (2/9)

- **Répertoire à deux niveaux** (*Two-Level Directory*)
  - un répertoire distinct pour chaque utilisateur
  - différents fichiers peuvent avoir le même nom à condition d'être dans des répertoires différents
  - recherche efficace
  - pas de possibilité de groupage

Structures des répertoires (3/9)

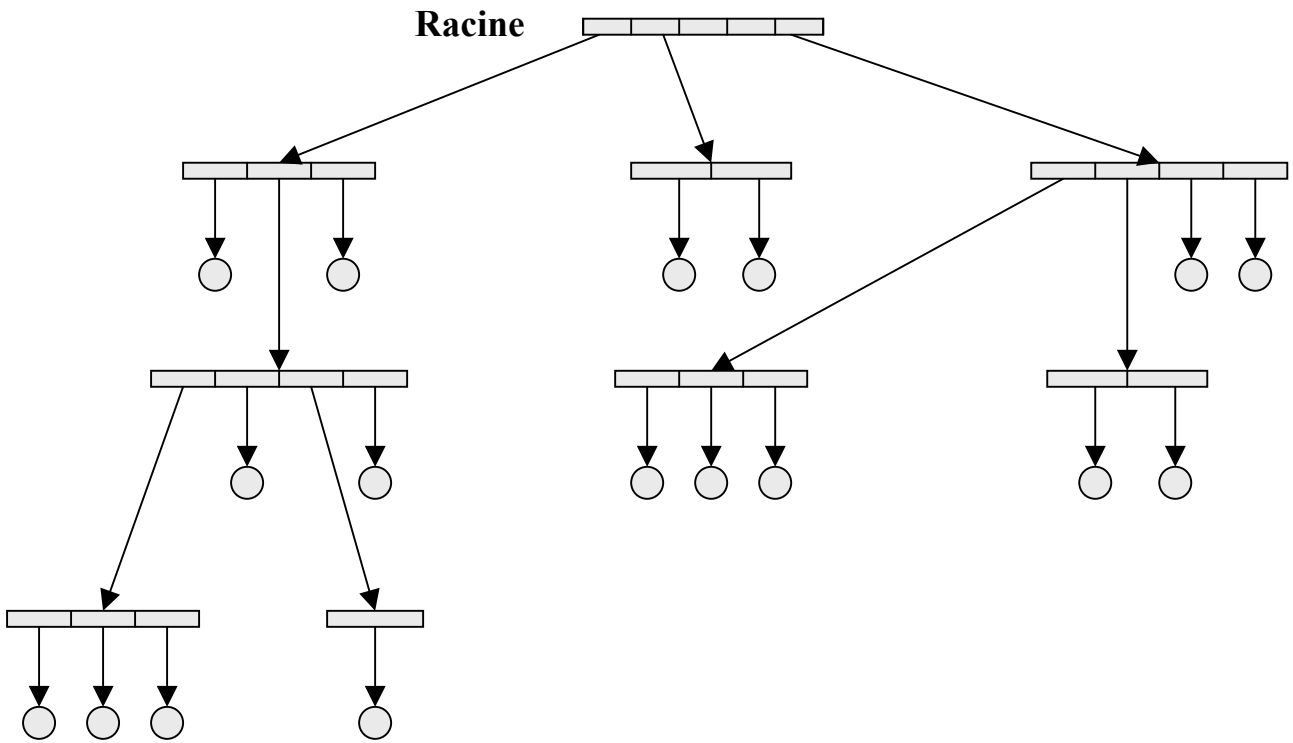


Répertoire à deux niveaux

Structures des répertoires (4/9)

- **Répertoires arborescents**
  - recherche efficace
  - possibilté de groupage (*Grouping capability*)
  - nom de fichier absolu ou relatif
  - la création d'un nouveau fichier est réalisée dans le répertoire courant
  - la création d'un nouveau répertoire est réalisée dans le répertoire courant

Structures des répertoires (5/9)

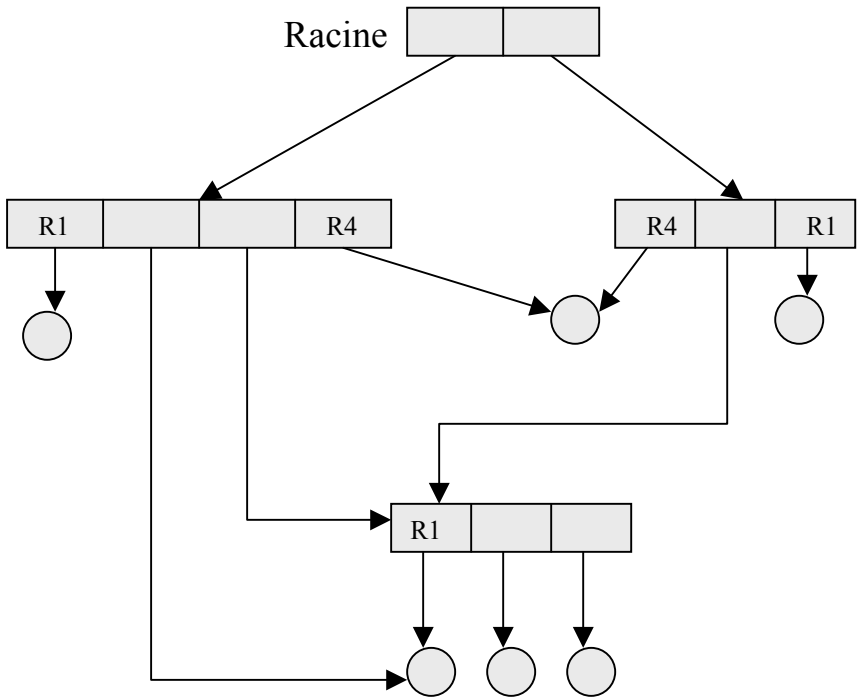


Structure arborescente de répertoires

Structures des répertoires (6/9)

- **Répertoires sous forme de graphes sans cycle** (*Acyclic-Graph Directories*)
  - fichier ou répertoire partagé présent plusieurs fois dans le système de fichiers
  - pas de duplication mais utilisation de liens
  - problème de modification d'un fichier référencé par plusieurs noms
  - problème de suppression d'un fichier référencé par plusieurs noms

Structures des répertoires (7/9)

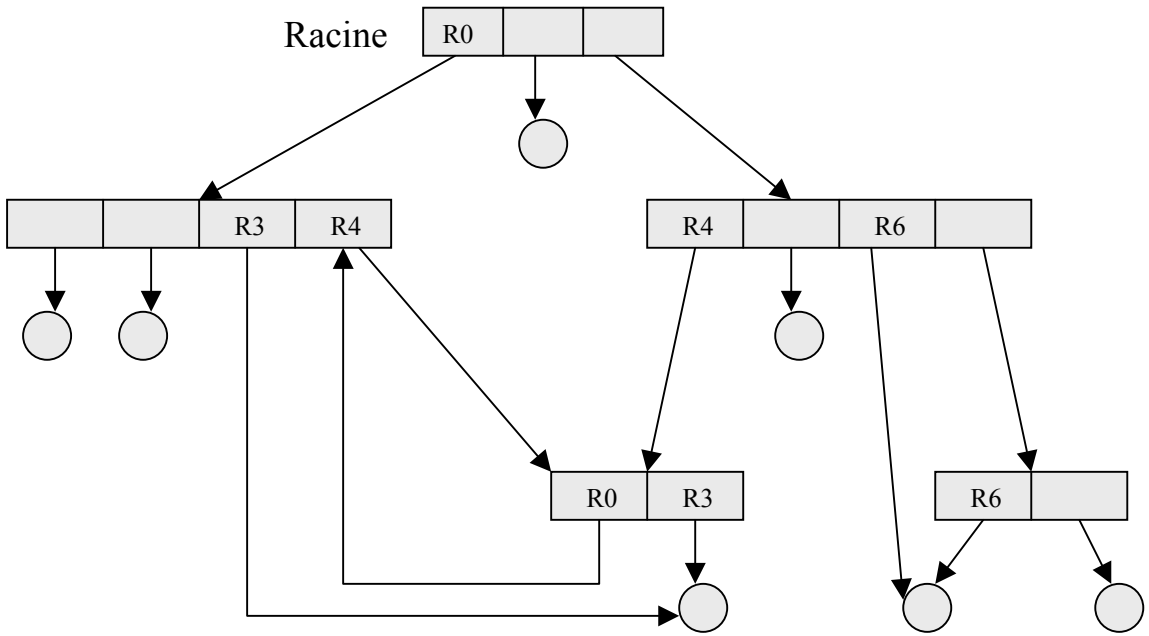


Répertoire sous forme de graphe sans cycle

Structures des répertoires (8/9)

- **Répertoires sous forme de graphes avec cycles**
  - ajout de liens ==> possibilité de structure de graphe avec cycles
  - exploration mal conçue du système de fichier ==> boucle infinie
  - éviter ce problème ==> mémoriser les différents répertoires explorés afin de détecter un cycle

Structures des répertoires (9/9)



Répertoire sous forme de graphe avec cycle

Gestion des espaces libres (1/2)

- Le système maintient une liste des blocs libres du disque
- moyens utilisés
  - vecteurs de bits
  - liste chaînée
  - groupage

Gestion des espaces libres (2/2)

- **Vecteurs de bits**
  - chaque bloc est représenté par un bit
  - bit = 0 ==> bloc libre, bit = 1 ==> bloc alloué
  - vecteur de bits conservé en mémoire pour une gestion efficace
- **Liste chaînée**
  - méthode peu efficace car temps d'E/S important
- **Groupage**
  - premier bloc libre contient les adresses des N blocs libres suivants
  - obtention rapide d'un grand nombre de blocs libres

Méthodes d'allocation (1/3)

Comment allouer de l'espace disque aux fichiers d'une manière optimale ?

- allocation contiguë
- allocation chaînée
- table d'allocation des fichiers
- allocation indexée

Méthodes d'allocation (2/3)

- **Allocation contiguë**
  - définie par l'adresse de son premier bloc et par son nombre de blocs
  - comment trouver n blocs libres consécutifs ?
  - combien d'espace nécessaire pour un fichier ?
- **Allocation chaînée**
  - fichier constitué de blocs non consécutifs
  - pas de fragmentation externe ==> pas de compactage
  - efficace pour les accès séquentiels et inefficace pour les accès directs
  - espace nécessaire pour stocker les pointeurs

Méthodes d'allocation (3/3)

- **Table d'allocation des fichiers** (*File Allocation Table - FAT*)
  - FAT située au début du disque, contient une entrée par bloc et indexée par le numéro de bloc
  - quelques avantages par rapport au chaînage des blocs
  - mais aussi quelques inconvénients
- Allocation indexée
  - pointeurs regroupés dans le bloc d'index (tableau de pointeurs)
  - chaque fichier possède son bloc d'index
  - gestion efficace de l'accès direct sans fragmentation externe
  - comment gérer de gros fichiers en utilisant des blocs d'index de taille réduite ?

Protection des fichiers

- Sûreté
- sécurité
  - nommage
  - mots de passe
  - liste d'accès

Gestionnaire d'un système de fichiers

Exemple de structure d'un gestionnaire de fichiers