Bienvenue!

INF3723:
Systèmes d'exploitation

Luigi Logrippo

luigi@uqo.ca
http://w3.uqo.ca/luigi/

CES NOTES DE COURS SONT DISPONIBLES A PARTIR DE MA PAGE WEB

Évaluation

- **Examen Intra (2h): 25% (le *25 octobre)***
- **Examen Final (3h) 45% (le *13 décembre*)**
  - **Examens à livre fermé – aucun doc permis**
  - **Le final sera sur toute la matière**
- **Devs. théoriques et de programm. 30%**
  - 2-3 devs de programmation: 20% tous
  - 3 devs théoriques: 10% tous

Contenu du cours

- **PARTIE 0: INTRODUCTION**
  - Revue des principales composantes d'un ordinateur.
  - Structure générale des systèmes d'exploitation.
- **PARTIE 1: PROCESSUS et GESTION d'UNITÉ CENTRALE**
  - Description et contrôle des processus.
  - Fils (threads).
  - Parallélisme: exclusion mutuelle et synchronisation.
  - Ordonnancement des processus sur un uniprocesseur.
  - Parallélisme: interblocage et famine.
- **PARTIE 2: GESTION DE MÉMOIRE**
  - Adressage et gestion de la mémoire.
  - Mémoire virtuelle.
- **PARTIE 3: FICHIERS, E/S ET PROTECTION**
  - Systèmes de fichiers, systèmes d'E/S
  - Protection

- **Accent en classe sur les concepts théoriques de *longue durée de vie***
- **La programmation, l'application seront dans les sessions exercices**

Manuel:

- **Silberschatz, Galvin, Gagne.**
  - Principes appliqués des systèmes d'exploitation, Vuibert
- **Avantages:**
  - très clair dans la présentation
  - utilise Java
  - beaucoup de notions intéressantes et utiles
- **Désavantages:**
  - beaucoup plus gros que nécessaire
- **Je le suivrai, mas pas toujours**
- **Cependant la lecture du manuel est indispensable pour une bonne compréhension de la matière.**
  - Les examens et les devoirs contiendront des questions prises du manuel
- **Livres de Stallings et Tanenbaum en réserve a la bibliothèque**

Les labos et Java

- **1er labo la semaine prochaine sur Unix**
- **Les devoirs de programmation se feront en Java**
  - Pas un langage utilisé dans les SE d 'aujourd 'hui
  - Mais un langage qui facilite la programmation parallèle
  - Il commence à être utilisé utilisé dans les SE répartis expérimentaux
- **Les labos fourniront une introduction a Java, et de l'aide dans la programmation Java**
- **Java ne sera pas utilisé en classe: pseudocode sera utilisé au lieu**
- **Le test et l 'examen contiendront des questions sur Unix et Java**

Labos et devoirs

- **Les devoirs seront constitués de:**
  - 20%: 3 travaux de programmation (Unix et Java) sur:
    - Parallélisme
    - Synchronisation de processus
  - 10%: Exercices écrits sur la théorie vue en classe (probabl. 3 ensembles d'exercices)

Introduction

Chapitre 1

Que c’est qu’un SE

Développement historique des SE

http://w3.uqo.ca/luigi/

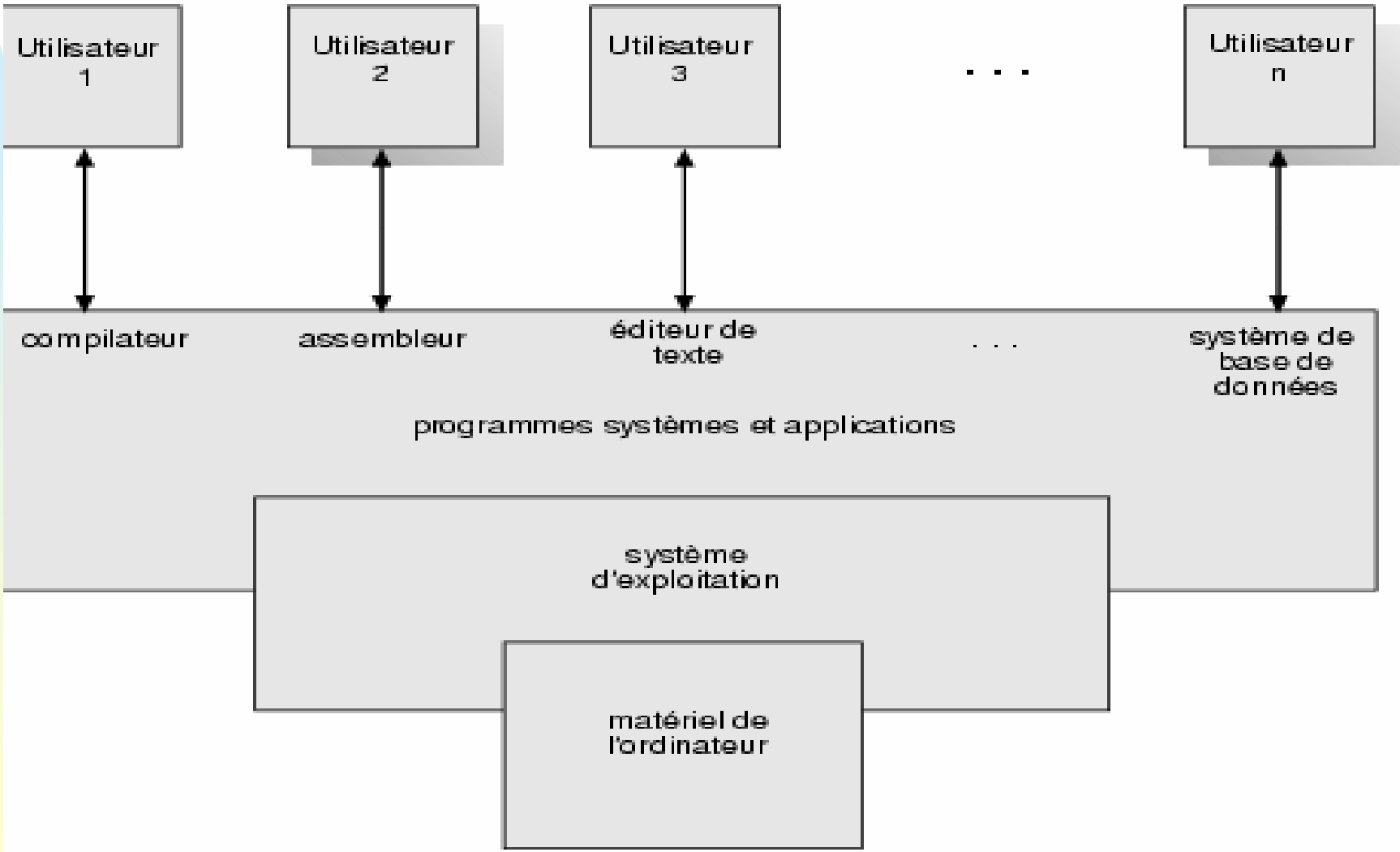
Concepts importants du Chapitre 1

- **Que c'est que un SE**
- **Évolution historique**
  - Par lots
  - Multiprogrammés – balance de travaux
  - À partage de temps (time-sharing)
  - Parallèles:
    - Fortement couplés
      - Symétriques,
      - Asymétriques: maître-esclave
    - Faiblement couplés:
      - Répartis
      - Réseaux
- **Caractéristiques de matériel et logiciel requises pour cette évolution**
- **Systèmes à temps réel: durs, souples**

Système d'exploitation (SE)

- **Fournit l'interface usager/machine:**
  - Masque les détails du matériel aux applications
  - Le SE doit donc traiter ces détails
- **Contrôle l'exécution des applications**
  - Le fait en reprenant périodiquement le contrôle de l'UCT
  - Dit à l'UCT quand exécuter tel programme
- **Il doit optimiser l`utilisation des ressources pour maximiser la performance du système**

Vue abstraite d'un SE



Ressources et leur gestion

- **Ressources:**
  - physiques: mémoire, unités E/S, UCT...
  - Logiques = virtuelles: fichiers et bases de données partagés, canaux de communication logiques, virtuels...
  - les ressources logiques sont bâties par le logiciel sur les ressources physiques
- **Allocation de ressources: gestion de ressources, leur affectation aux usagers qui les demandent, suivant certains critères**

Pourquoi étudier les SE?

- **Logiciel très important…**
  - tout programme roule sur un SE
  - interface usager-ordinateur
- **Les SE utilisent beaucoup d ’algorithmes et structures de données intéressants**
  - Les techniques utilisées dans les SE sont aussi utilisées dans nombreuses autres applications informatiques
    - il faut les connaître

Développement de la théorie des SE

- **La théorie des SE a été développée surtout dans les années 1960 (!!)**
- **A cette époque, il y avait des machines très peu puissantes avec lesquelles on cherchait à faire des applications comparables à celles d'aujourd'hui (mémoire typique: 100-500K!)**
- **Ces machines devaient parfois desservir des dizaines d'usagers!**
- **Dont le besoin de développer des principes pour optimiser l'utilisation d 'un ordinateur.**
- **Principes qui sont encore utilisés**

Évolution historique des SE

- **Le début: routines d`E/S, amorçage système**
- **Systèmes par lots simples**
- **Systèmes par lots multiprogrammés**
- **Systèmes à partage de temps**
- **Ordinateurs personnels**
- **SE en réseau**
- **SE répartis**

❖ ***Les fonctionnalités des systèmes simples se retrouvent dans les systèmes complexes.***

❖ ***Les problèmes et solutions qui sont utilisés dans les systèmes simples se retrouvent souvent dans les systèmes complexes.***

Phase 1: Les débuts

- **Au début, on a observé qu`il y avait des fonctionnalités communes à tous les programmes**
- **il fallait les pré-programmer et les fournir au programmeur à moyen d`instructions d` appel:**
  - amorçage du système
  - entrée/sortie

Phase 2: Systèmes de traitement par lots *(batch)* simples

- **Sont les premiers SE (mi-50)**
- **L'usager soumet une job à un opérateur**
  - Programme suivi par données
- **L'opérateur place un lot de plusieurs jobs sur le dispositif de lecture**
- **Un programme, le moniteur, gère l'exécution de chaque programme du lot**
- **Le moniteur est toujours en mémoire et prêt à être exécuté**
- **Les utilitaires du moniteur sont chargés au besoin**
- **Un seul programme à la fois en mémoire, programmes sont exécutés en séquence**
- **La sortie est normalement sur un fichier, imprimante, ruban magnétique…**

Un ordinateur principal (mainframe)
du milieu des annnées ‘60

disques

rubans

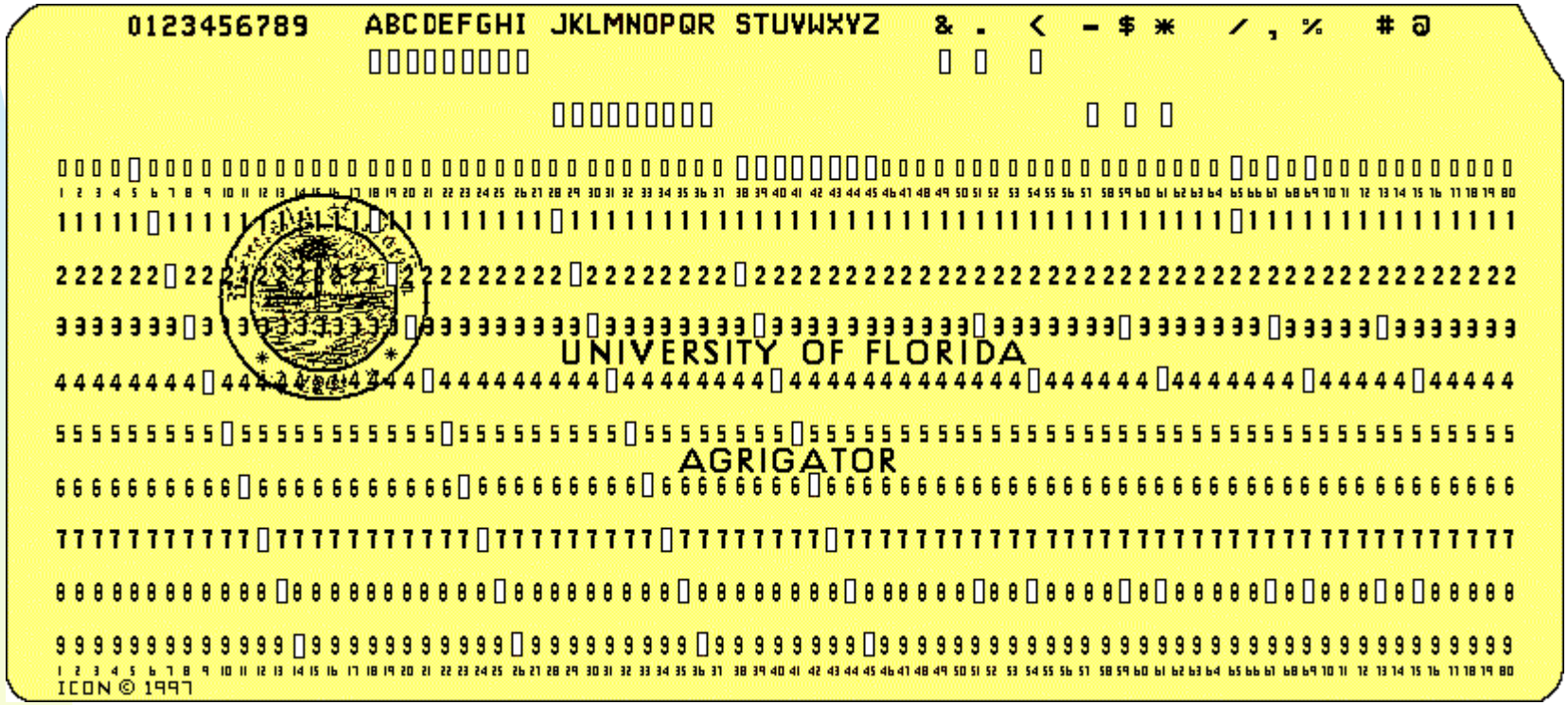
lecteur de cartes

UCT
(mémoire probablem.
autour de 250-500K)

console opérateur

Musée de l’histoire de l’informatique http://www.computerhistory.org/

Oui, cartes perforées…



Une ligne de données ou de programme était codée dans des trous qui pouvaient être lus par la machine

Opérateur lisant un paquet de cartes perforées

Source: http://www.tietokonemuseo.saunalahti.fi/eng/kuva_32_eng.htm
Finnish Data Processing Museum Association

Langage de contrôle des travaux (JCL)

- **Utilisé pour contrôler l 'exec d 'une job**
  - le compilateur à utiliser
  - indiquer où sont les données
- **Exemple d'une job:**
  - paquet de cartes comme suit:
- **$JOB début**
- **$FTN charge le compilateur FORTRAN et initie son exécution**
- **$LOAD charge le pgm objet (à la place du compilateur)**
- **$RUN transfère le contrôle au programme usager**
- **les données sont lues par le moniteur et passées au progr. usager**

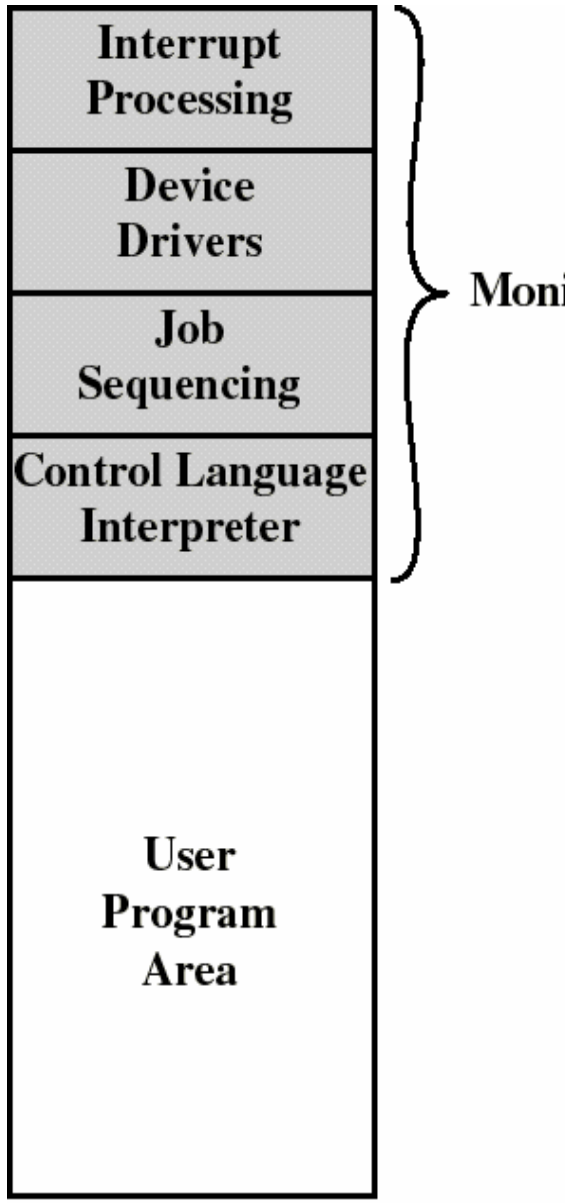
```
$JOB
$FTN
...
Programme
FORTRAN
...
$LOAD
$RUN
...
Données
...
$END
$JOB
...
(job suivant)
```

Langage de contrôle des travaux (JCL)

- **L'E/S est déléguée au moniteur**
- **Chaque instruction d'E/S dans pgm usager invoque une routine d'E/S dans le moniteur:**
  - s'assure de ne pas lire une ligne JCL
  - un usager ne peu pas interférer avec les E/S d`un autre usager…
- **Quand le programme usager se termine, la prochaine ligne de JCL est lue et exécutée par le moniteur.**

Le moniteur *par lots*

- **Lecture de cartes perforées**
- **Interprétation de commandes JCL**
- **Lecture (load) d'une job (du lecteur de cartes)**
- **Chargement en mémoire (dans la région de l'usager) de cette job**
- **Transfère le contrôle au programme usager (job sequencing)**
- **Exécution du programme usager jusqu'à:**
  - fin du programme
  - E/S
  - erreur
- **À ce point, le moniteur reprend le contrôle**
  - Pour le redonner plus tard au même programme ou à un autre programme



Memory Layout of Resident Monitor

Stallings

Caractéristiques désirables du matériel (1)

- **Protection de la mémoire**
  - ne pas permettre aux pgms usager d'altérer la région de la mémoire où se trouve le moniteur
- **Minuterie**
  - limite le temps qu`une job peut exécuter
  - produit une interruption lorsque le temps est écoulé

Caractéristiques désirables du matériel (2)

- **Instructions privilégiées**
  - exécutables seulement par le moniteur
  - une interruption se produit lorsqu'un programme usager tente de les exécuter
    - UCT peut exécuter en mode *moniteur* ou mode *usager*
    - Les instructions privilégiées ne peuvent être exécutées que en mode *moniteur*
    - l 'usager ne peut exécuter que en mode *usager*
    - seulement le SE ou une interruption peuvent changer de mode
- **Interruptions**
  - facilitent le transfert de contrôle entre le système d 'exploitation, les opérations d`E/S et les programmes usagers
  - Le mode moniteur sera plus souvent appelé mode superviseur

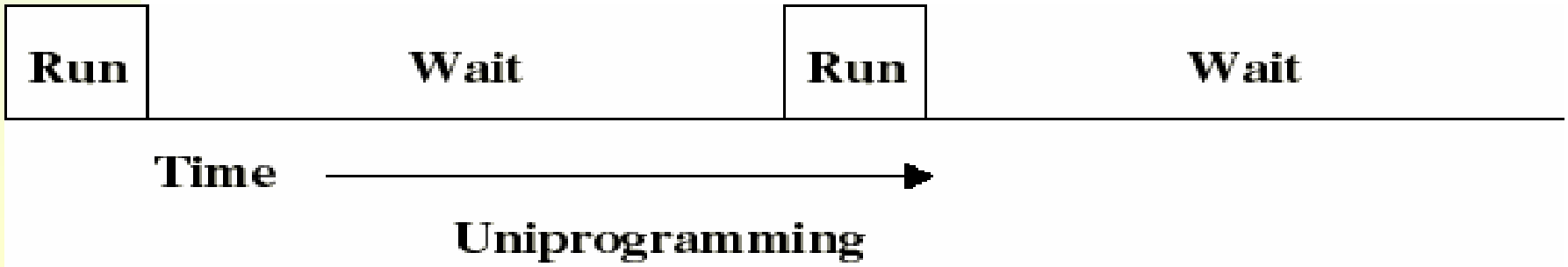
Les systèmes par lots

- **Ont été les premiers systèmes d`exploitation.**
- **Ils sont associés aux concepts suivants:**
  - langage de contrôle de travaux (JCL)
  - système d ’exploitation résident en mémoire
    - *kernel = noyau*
  - protection de mémoire
  - instructions privilégiées
    - modes usager-moniteur
  - interruptions
  - minuterie
- **Toutes ces caractéristiques se retrouvent dans les systèmes d’aujourd’hui**
- **Encore aujourd’hui on parle de jobs ‘par lots’ quand ils sont exécutés séquentiellement sans intervention humaine**
  - P.ex. salaires, comptabilité d’une compagnie

Phase 2.5: Traitement par lots multiprogrammé

- **Les opérations E/S sont extrêmement lentes (comparé aux autres instructions)**
  - P. ex. une boucle de programme pourrait durer 10 microsecondes, une opération disque 10 millisecondes
    - C'est la différence entre 1 heure et un mois et demi!

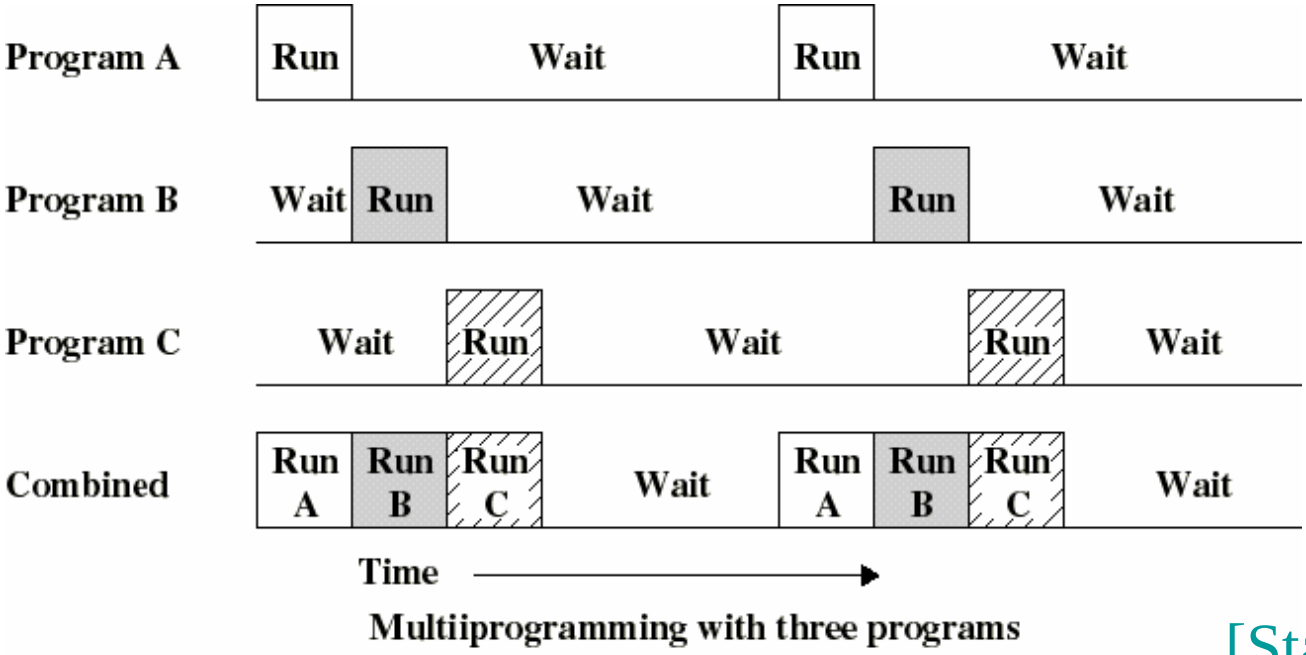
    Même avec peu d'E/S, un programme passe la majorité de son temps à attendre
- **Donc: pauvre utilisation de l'UCT lorsqu'un seul pgm usager se trouve en mémoire**



[Stallings]

Traitement par lots multiprogrammé

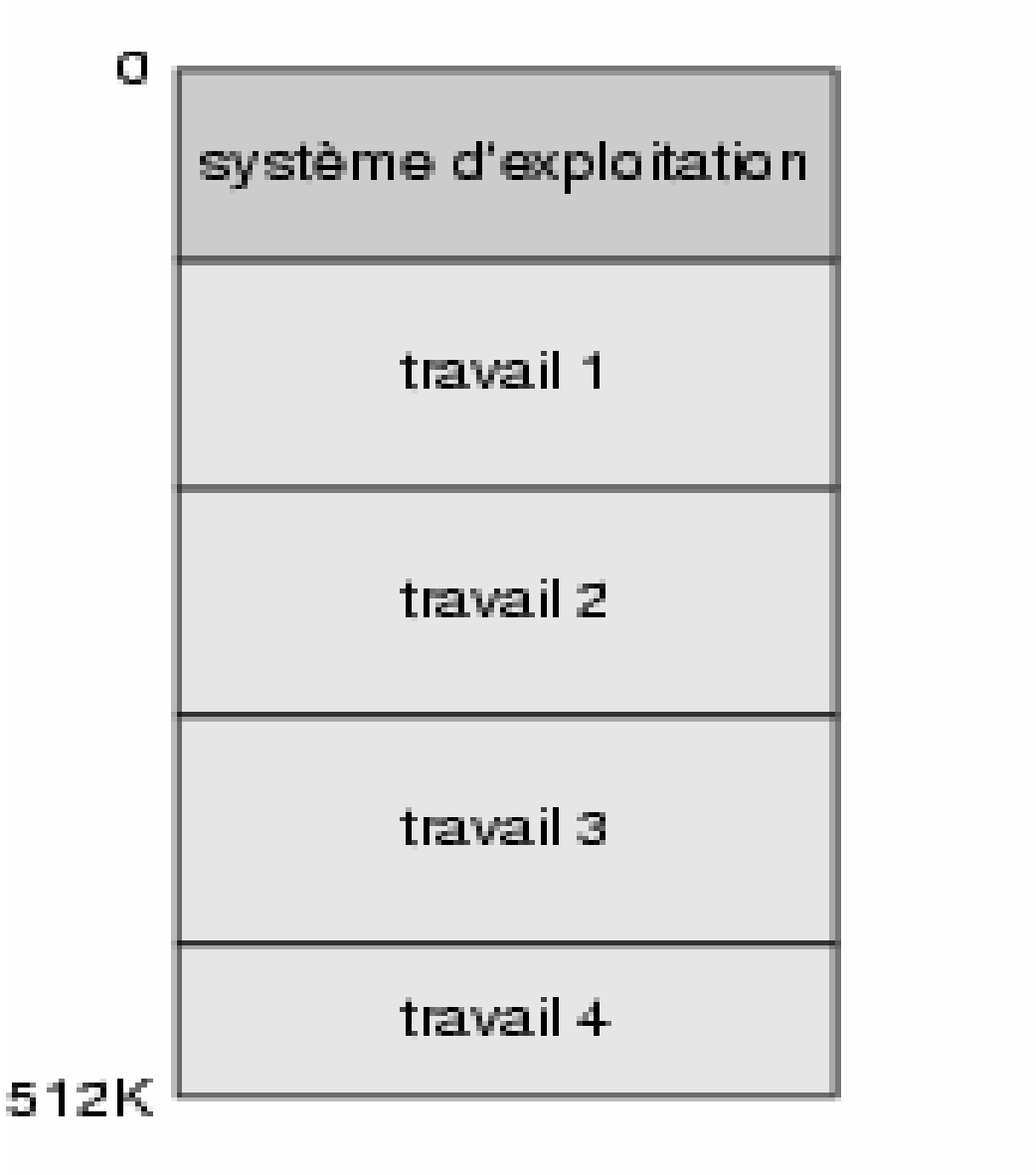
- **Si la mémoire peut contenir +sieurs pgms, l'UCT peut exécuter un autre pgm lorsqu'un pgm attend après E/S**
- **C'est la multiprogrammation**



Multiiprogramming with three programs

[Stallings]

Plusieurs programmes en mémoire pour la multiprogrammation



Exigences pour multiprogrammation

- **Interruptions**
  - afin de pouvoir exécuter d'autres jobs lorsqu'un job attend après E/S
- **Protection de la mémoire: isole les jobs**
- **Gestion du matériel**
  - plusieurs jobs prêts à être exécutées demandent des ressources:
    - UCT, mémoire, unités E/S
- **Langage pour gérer l'exécution des travaux: interface entre usager et OS**
  - jadis JCL, maintenant *shell, command prompt* ou semblables

Spoule ou spooling

- **Au lieu d ’exécuter les travaux au fur et à mesure qu’ils sont lus, les stocker sur une mémoire secondaire (disque)**

- **Puis choisir quels programmes exécuter et quand**
  - Ordonnanceur à long terme, à discuter

Équilibre de travaux

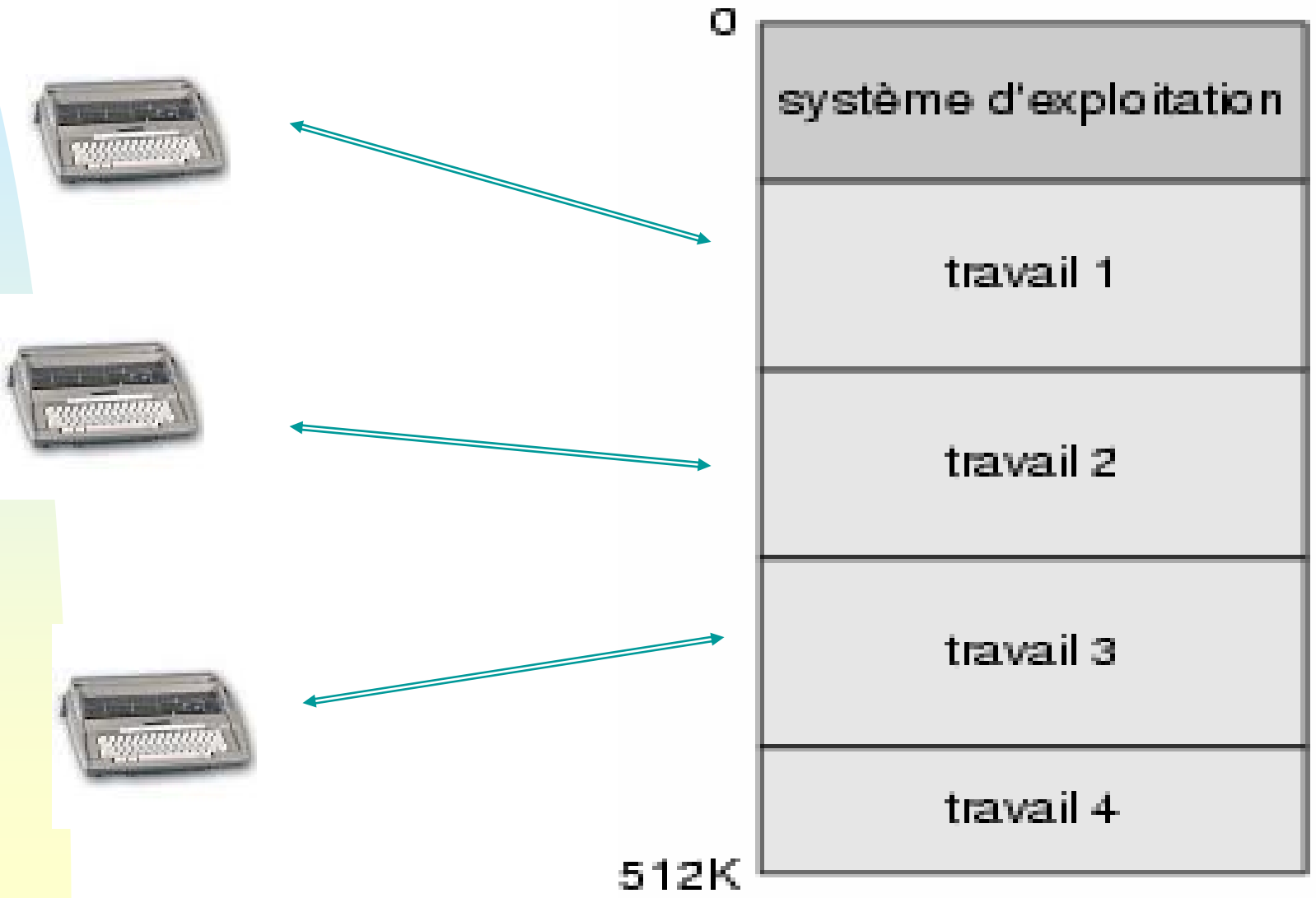
- **S`il y a un bon nombre de travaux à exécuter, on peut chercher à obtenir un équilibre**
- **Travaux qui utilisent peu l`UCT, beaucoup l ’E/S, sont appelés *tributaires de l`E/S***
- **Nous parlons aussi de travaux *tributaires de l ’UCT***
- **Le temps d`UCT non utilisé par des travaux trib. de l ’E/S peut être utilisé par des travaux trib. de l ’UCT et vice-versa.**
- **L ’obtention d`un tel équilibre est le but des ordonnanceurs à long terme et à moyen terme (à discuter).**
- **Dans les systèmes de multiprog. on a souvent coexistence de travaux longs et pas urgents avec travaux courts et urgents**
  - Le SE donne priorité aux deuxièmes et exécute les premiers quand il y a du temps de machine disponible.

Phase 3: Systèmes à temps partagé (TSS)

Terminaux
‘stupides’

ordinateur principal
(mainframe)

Chaque terminal a sa propre partition de mémoire



Systèmes à temps partagé (TSS)

- **Le traitement par lots multiprogrammé ne supporte pas l'interaction avec les usagers**
  - **excellente utilisation des ressources mais frustration des usagers!**
- **TSS permet à la multiprogrammation de desservir plusieurs usagers simultanément**
- **Le temps d 'UCT est partagé par plusieurs usagers**
- **Les usagers accèdent simultanément et interactivement au système à l'aide de terminaux**

Systèmes à temps partagé (TSS)

- **Le temps de réponse humain est lent: supposons qu`un usager nécessite, en moyenne, 2 sec du processeur par minute d'utilisation**
- **Environ 30 usagers peuvent donc utiliser le système sans délais notable du temps de réaction de l'ordinateur**
- **Les fonctionnalités du SE dont on a besoin sont les mêmes que pour les systèmes par lots, plus**
  - la communication avec usagers
  - le concept de *mémoire virtuelle* pour faciliter la gestion de mémoire
  - traitement central des données des usagers (partagées ou non)

MULTICS et UNIX

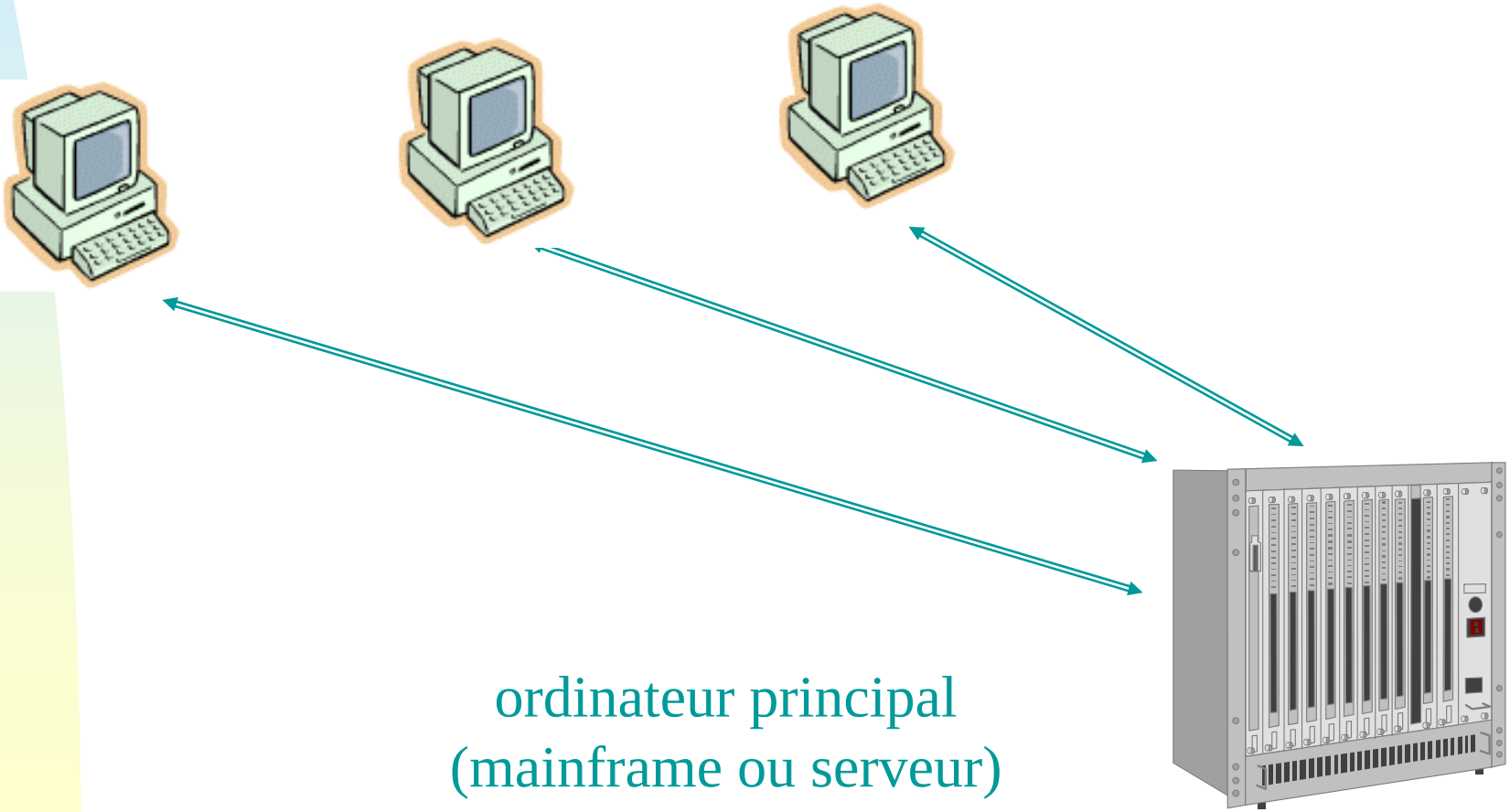
- **MULTICS a été un système TSS des années 60, très sophistiqué pour son époque**
- **Ne réussit pas à cause de la faiblesse du matériel de son temps**
- **Quelques unes de ses idées furent reprises dans le système UNIX**

Ordinateurs Personnels (PCs)

- **Au début, les PCs étaient aussi simples que les premiers ordinateurs**
- **Le besoin de gérer plusieurs applications en même temps conduit à redécouvrir la multiprogrammation**
- **Le concept de PC isolé évolue maintenant vers le concept d ’ordinateur de réseau *(network computer)*, donc extension des principes des TSS.**

Aujourd'hui

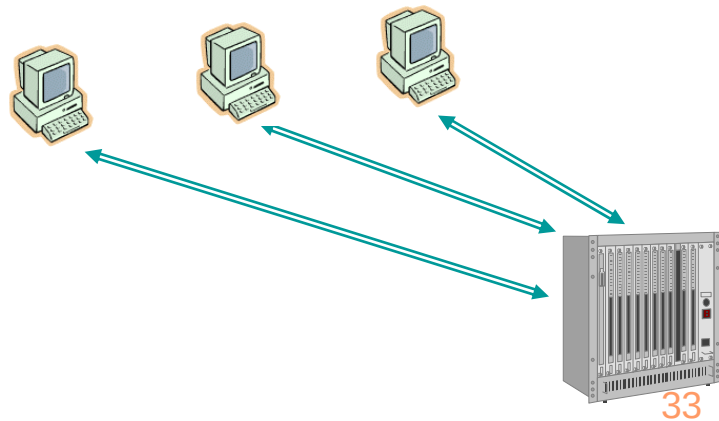
Terminaux
'intelligents' (PCs)'



ordinateur principal
(mainframe ou serveur)

Retour aux concepts de TSS

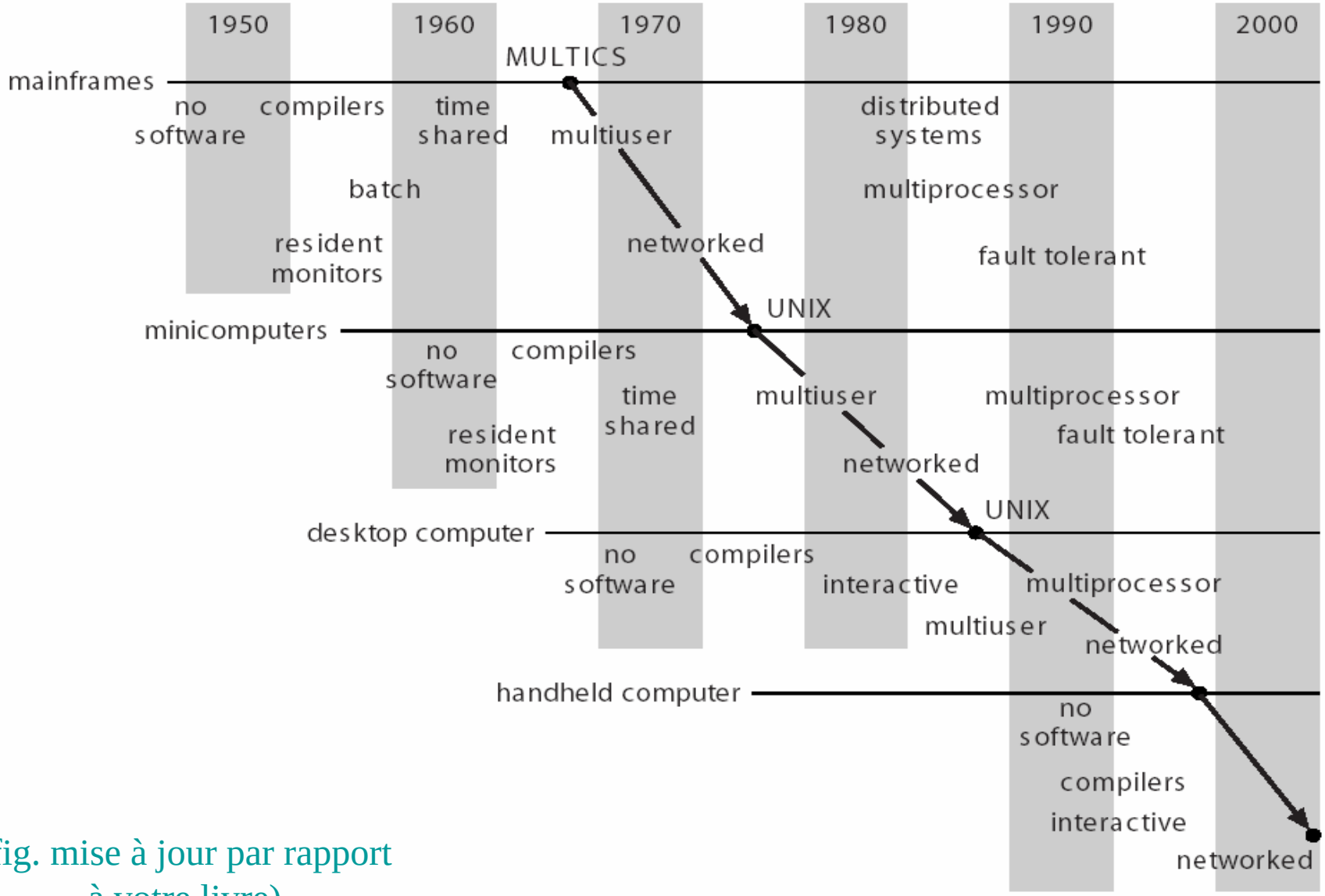
- **Plusieurs PC (clients) peuvent être desservis par un ordi plus puissant (serveur) pour des services qui sont trop complexes pour eux (clients/serveurs, bases de données, telecom)**
- **Les grands serveurs utilisent beaucoup des concepts développés pour les systèmes TSS**



Et puis…

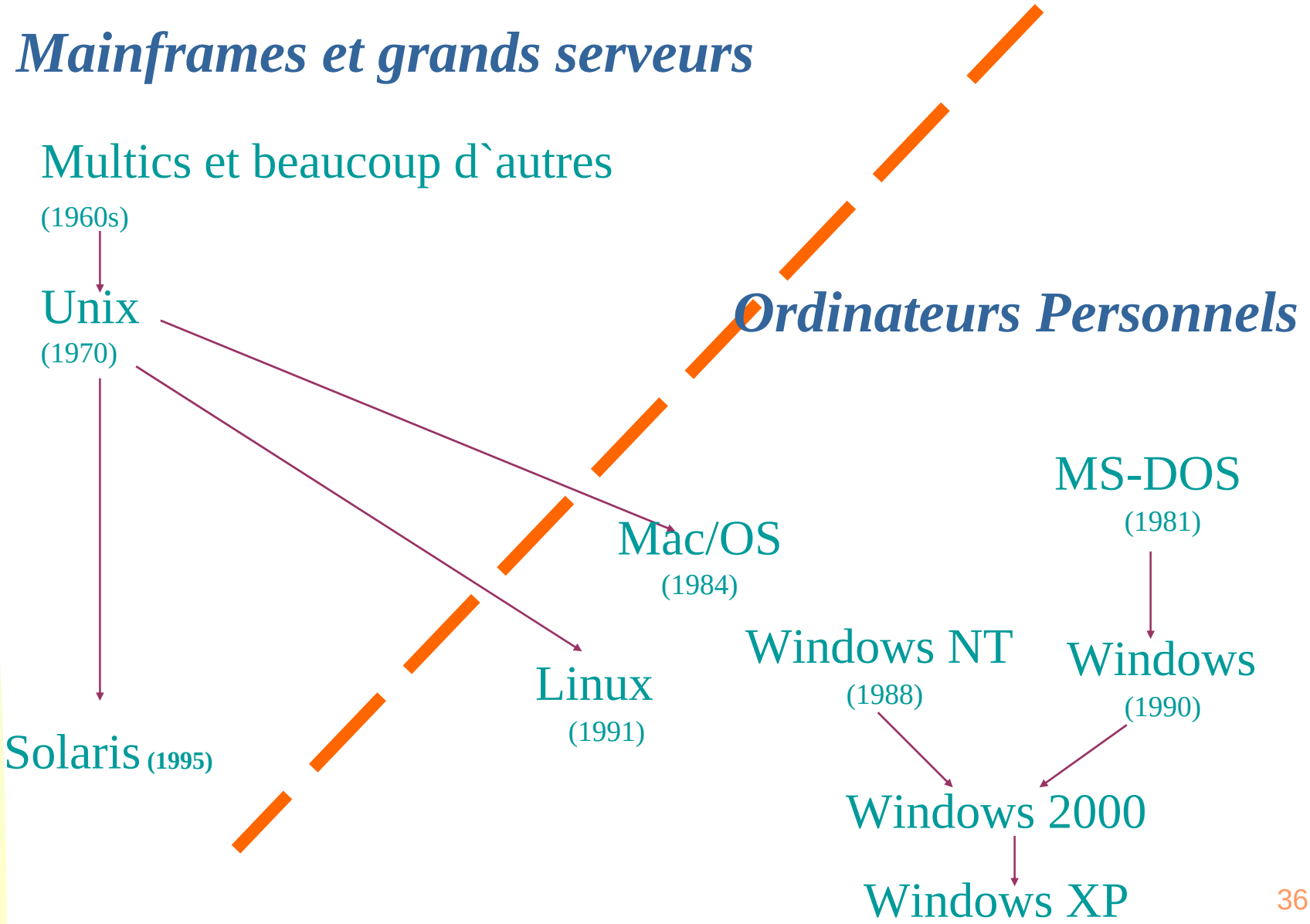
- **Systèmes d'exploitation répartis:**
  - Le SE exécute à travers un ensemble de machines qui sont reliées par un réseau
    - Pas discutés dans ce cours

Évolution des SE



(fig. mise à jour par rapport à votre livre)

Une synthèse historique



Maintenant, quelques définitions…

Systèmes parallèles *(tightly coupled)*

- **Le petit coût des puces rend possible leur composition dans systèmes multiprocesseurs**
- **Les ordinateurs partagent mémoire, horloge, etc.**
- **Avantages:**
  - plus de travail fait (throughput)
  - plus fiable:
    - dégradation harmonieuse (graceful degradation)

Systèmes parallèles

- **Symétriques**
  - Tous les UCTs exécutent le même SE
  - Elles sont fonctionnellement identiques
- **Asymétrique**
  - Les UCTs ont des fonctionnalités différentes, par exemple il y a un *maître* et des *esclaves*.
- **Aujourd'hui, tout ordinateur puissant est un système parallèle.**

Systèmes distribués (= répartis)

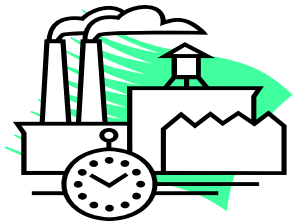
- **Les réseaux d ’ordinateurs sont en pleine émergence...**
- **Systèmes multiprocesseurs faiblement couplés *(loosely coupled)***
  - consistent d ’ordinateurs autonomes, qui communiquent à travers lignes de communication

Systèmes distribués (= répartis)

- **SE répartis**
  - il y a un SE qui fonctionne entre ordinateurs
  - l ’usager voit les ressources éloignées comme si elles étaient locales
- **SE en réseau *(network operating systems)* fournissent:**
  - partage de fichiers (systèmes client-serveur)
  - patrons de communication (protocoles)
  - autonomie des ordinateurs

Systèmes à temps réel

- **Doivent réagir à ou contrôler des événements externes (p.ex. contrôler une usine). Les délais de réaction doivent être bornés**
- **systèmes temps réel *souples:***
  - les échéances sont importantes, mais ne sont pas critiques (p.ex. systèmes téléphoniques)
- **systèmes temps réel *rigides (hard):***
  - le échéances sont critiques, p.ex.
    - contrôle d'une chaîne d`assemblage
    - graphiques avec animation



- (ma déf. de *souple* n'est pas la même que dans le livre)

Concepts importants du Chapitre 1

- **Que c'est que un SE**
- **Évolution historique**
  - Par lots
  - Multiprogrammés – balance de travaux
  - À partage de temps (time-sharing)
  - Parallèles:
    - Fortement couplés
      - Symétriques,
      - Asymétriques: maître-esclave
    - Faiblement couplés:
      - Répartis
      - Réseaux
- **Caractéristiques de matériel et logiciel requises pour cette évolution**
- **Systèmes à temps réel: durs, souples**

Dans le livre, pour ce chapitre, vous devez

- **Étudier le chapitre entier**

Terminologie: mémoire centrale et auxiliaire

- **La mémoire centrale est la mémoire RAM sur laquelle le CPU exécute les instructions**
- **Le cache est étroitement lié à la mémoire centrale, donc il est considéré partie de cette dernière**
- **Mémoires auxiliaires sont toutes les autres mémoires dans le système**
  - Disques
  - Flash-memory
  - Rubans…
- **Les m. aux. sont des périphériques**

Terminologie

- **Opérations d'E/S: Entrée ou Sortie, Input/Output**
  - Les opérations de lecture ou écriture en ou de mémoire centrale
  - Peuvent être directement ou indirectement demandées par le programme
    - Exemple d'indirectement: E/S occasionnées par la pagination
  - Entrée:
    - Read dans un programme
      - Lecture de disque
      - Caractères lu du clavier
      - Click du souris
      - Lecture de courriel
      - Lecture de page web (à être affichée plus tard, p.ex.)
  - Sortie:
    - Write dans un programme
      - Affichage sur l'écran
      - Impression
      - Envoi de courriel
      - Sortie de page web demandée par une autre machine

Terminologie

- **Travaux ‘en lots’ (batch)**
  - Travaux non-urgents qui sont soumis au système pour ramasser la réponse plus tard
    - Tri de fichier, calcul d’une fonction complexe, grosses impressions, sauvegarde régulière de fichiers usagers
    - Pour plus d’efficacité, peuvent être groupés et exécutés les uns après les autres
- **Interactifs**
  - Sont les travaux qui demandent une interaction continue avec l’ordinateur:
    - Édition de documents ou d’un programme
- **Les premiers ordinateurs n’avaient pas de mécanismes de communication aisée entre usager et machine, donc normalement les travaux étaient ‘par lots’**

Annexe historique sur les techniques de programmation

- **Les informaticiens d'aujourd'hui sont souvent surpris du fait qu'on pouvait faire quelque chose d'utile avec des ordinateurs aussi petits que ceux qui existaient dans les années '60**
- **Un exemple pourra aider à comprendre...**

Un programme Hello World du début des années '60

```
110016#T
OXXXXXX0
HELLO WORLD
```

Ce programme consiste en 27 octets (*en langage machine*)

Son adresse initiale est 0.

La première ligne dit d'imprimer à partir de l'adresse 16 pour longueur 11.

La deuxième ligne est l'instruction STOP.

La 3ème ligne est la constante à imprimer

Programmes Hello World d'aujourd'hui

```
class HelloWorld {
   public static void printHello( ) {
   System.out.println("Hello, World");
   }
}
class UseHello {
   public static void main(String[ ] args) {
   HelloWorld myHello = new HelloWorld( );
   myHello.printHello( );
   }
}
```

Dans l'article:
C. Hu. Dataless objects considered harmful.
Comm. ACM 48 (2), 99-101
http://portal.acm.org/citation.cfm?id=1042091.1042126#
l'auteur présente deux versions orientées objet de programmes 'Hello World'. Il critique la version ci-dessous disant qu'elle n'est pas vraiment orientée objet et qu'il faudrait vraiment utiliser la version à droite.

Quelle est la mémoire demandée par ces programmes?
Il serait intéressant de voir...

```
class Message {
   String messageBody;
   public void setMessage(String newBody) {
   messageBody = newBody;
   }
   public String getMessage( ) {
   return messageBody;
   }
   public void printMessage( ) {
   System.out.println(messageBody);
   }
}
public class MyFirstProgram {
   public static void main(String[ ] args) {
   Message mine = new Message ( );
   mine.setMessage("Hello, World");
   Message yours = new Message ( );
   yours.setMessage("This is my first program!");
   mine.printMessage( );
   System.out.println(yours.getMessage( ) + "—" +
mine.getMessage( ) );
   }
}
```

Structures d'ordinateurs (matériel)

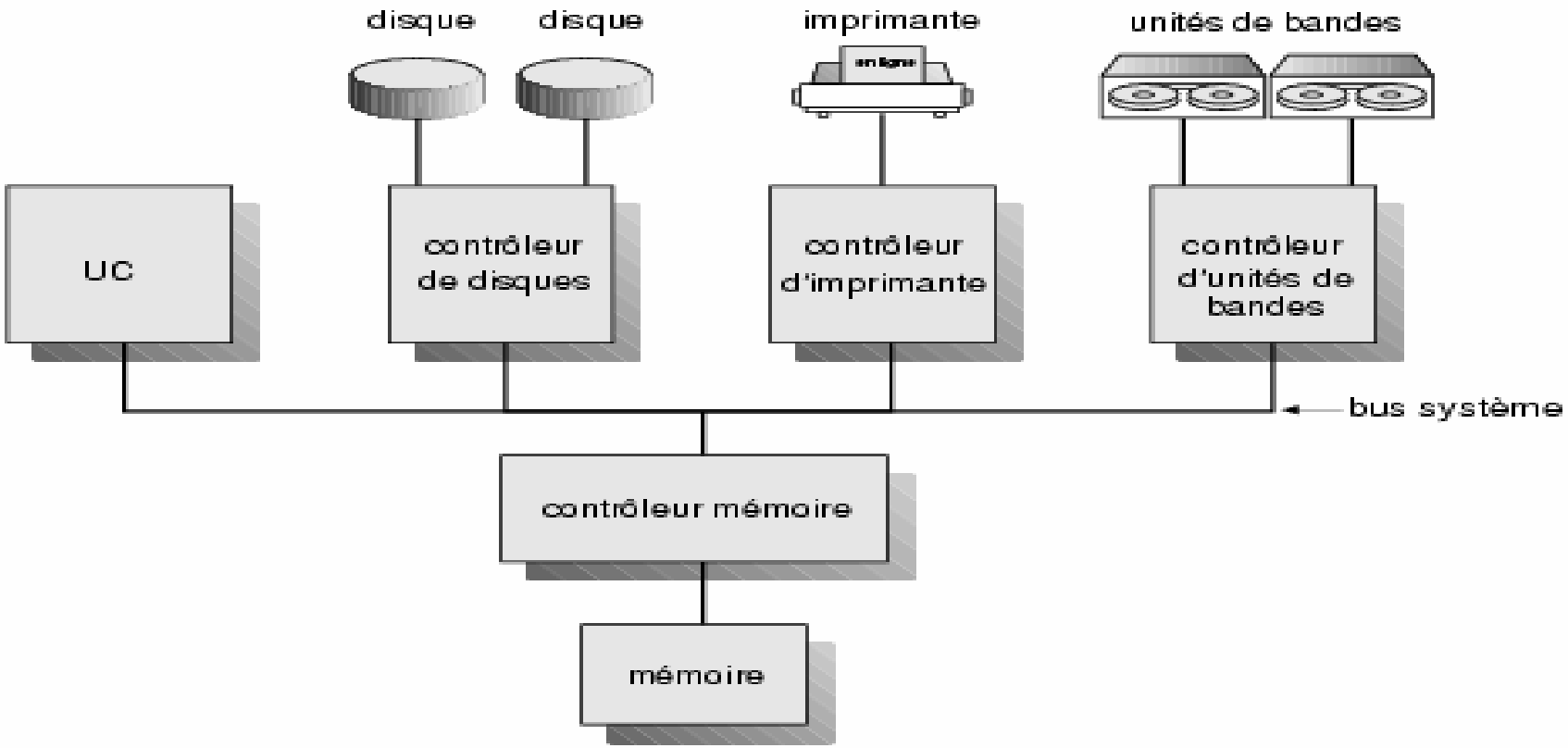
Chapitre 2

http://w3.uqo.ca/luigi/

Concepts importants du Chapitre 2

- **Registres d'UCT, tampons en mémoire, vecteurs d'interruption**
- **Interruption et polling**
- **Interruptions et leur traitement**
- **Méthodes d'E/S avec et sans attente, DMA**
- **Tableaux de statut de périphériques**
- **Hiérarchie de mémoire**
- **Protection et instructions privilégiées, modes d'exécution**
- **Registres bornes**
- **Appels de système**

Architecture d'ordinateurs



Dans beaucoup de systèmes, *ce n 'est que l`UCT qui peut adresser la mémoire*: les infos transférées entre les périphériques et la mémoire (ou même entre mémoire et mémoire) doivent passer à travers l 'UCT, dont le concept de `vol de cycles` (cycle stealing).

Registres de l'UCT (mémoire rapide dans UCT)

- **Registres de contrôle et statut**
  - Généralement non disponibles aux programmes de l'usager
  - l'UCT en utilise pour contrôler ses opérations
  - Le SE en utilise pour contrôler l'exécution des programmes
- **Registres Visibles (aux usagers)**
  - disponibles au SE et programmes de l'usager
  - visibles seulement en langage machine ou assembleur
  - contient données, adresses etc.

Exemples de registres de contrôle et statut

- **Le compteur d'instruction (PC)**
  - Contient l'adresse de la prochaine instruction à exécuter
- **Le registre d'instruction (IR)**
  - Contient l'instruction en cours d'exécution
- **Autres registres contenant, p.ex.**
  - bit d'interruption activé/désactivé
  - bit du mode d'exécution superviseur/usager
  - bornes de mémoire du programme en exec.
- **Registres de statut des périphériques**

Opération d`ordinateurs pour E/S

- **Unités d'E/S et UCT peuvent exécuter en même temps**
- **Chaque type d`unité a un contrôleur**
- **Chaque contrôleur a un tampon ou registre en mémoire principale (buffer)**
- **UCT transfère l 'information entre contrôleur et tampon *(vol de cycles)***
- **Le contrôleur informe l 'UCT que l'opération a terminé**
- **L'UCT a des registres qui contiennent le statut des différentes unités E/S.**

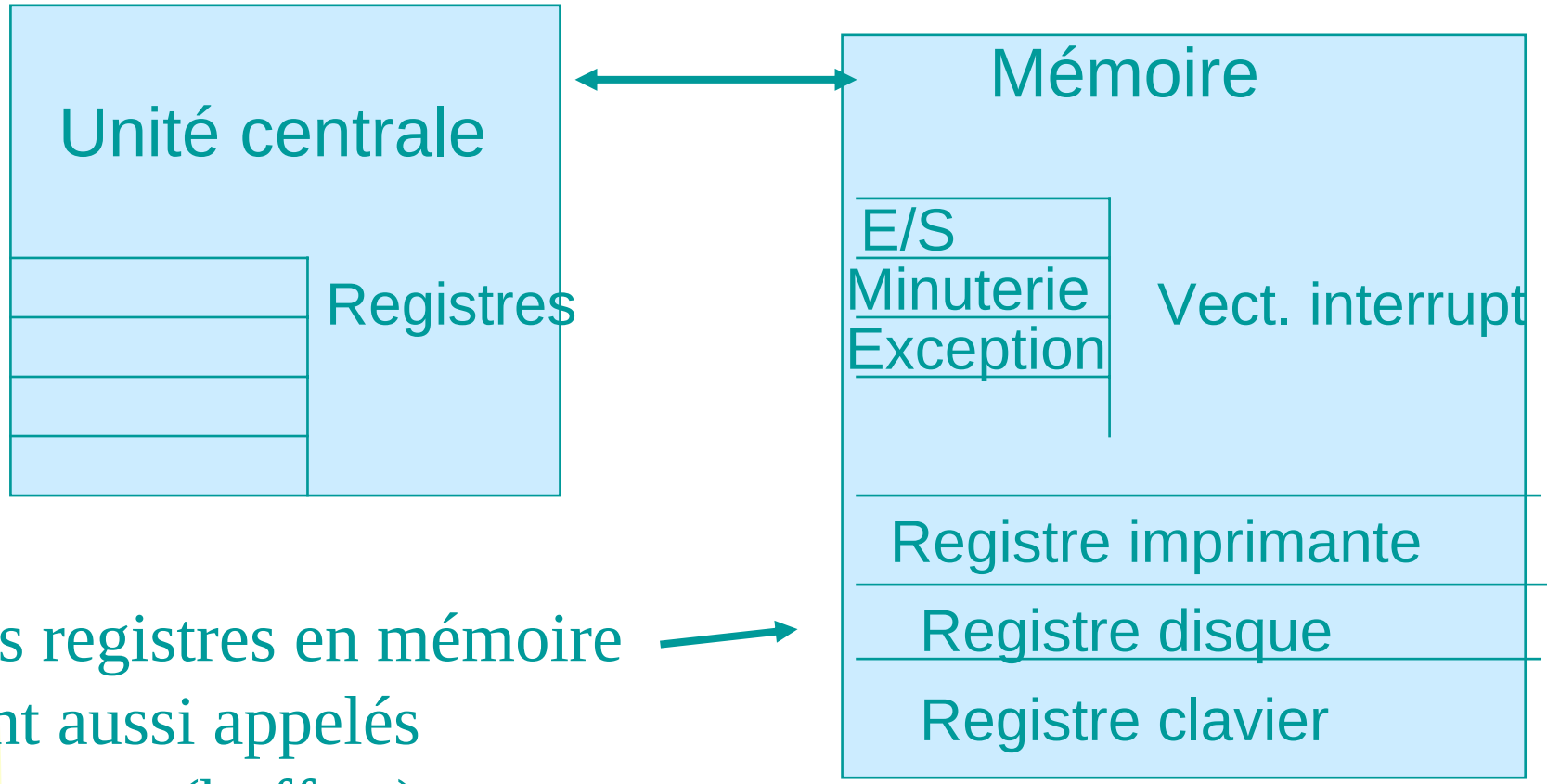
Deux façons différentes de traiter la communication entre UCT et unités E/S

- **Polling (E/S programmée, interrogation, scrutation): le programme interroge périodiquement les regs statut et détermine le statut de l 'unité E/S: *pour les unités E/S lentes***
- **Interruption: l 'UCT est interrompue entre instructions quand un événement particulier se produit (fin d 'E/S, erreur...)**
  - les interruptions peuvent être inhibées pendant l 'exécution de certaines parties critiques du programme (il y a une instruction pour faire ça).

Exemple

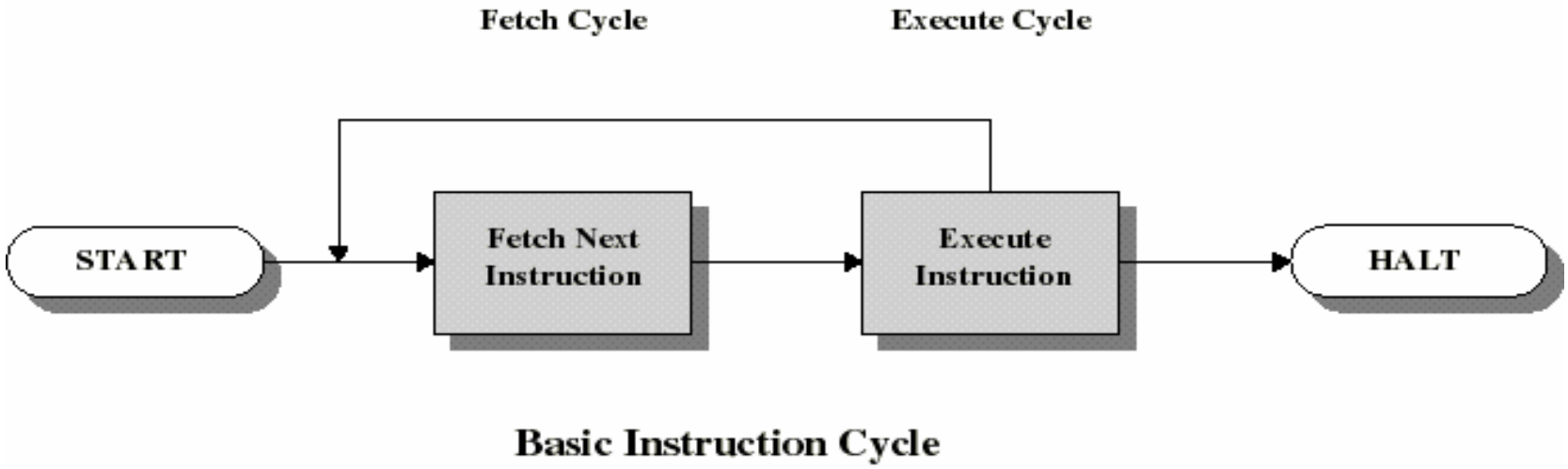
- **Courriel…**
  - J’utilise l’*interruption* si j’ai une sonnerie qui m’avertit quand un courriel arrive
  - J’utilise la *scrutation* (polling) si au lieu je regarde le courriel périodiquement de mon initiative

Registres, vecteurs d'interruptions, tampons



Les registres en mémoire sont aussi appelés tampons (buffers)

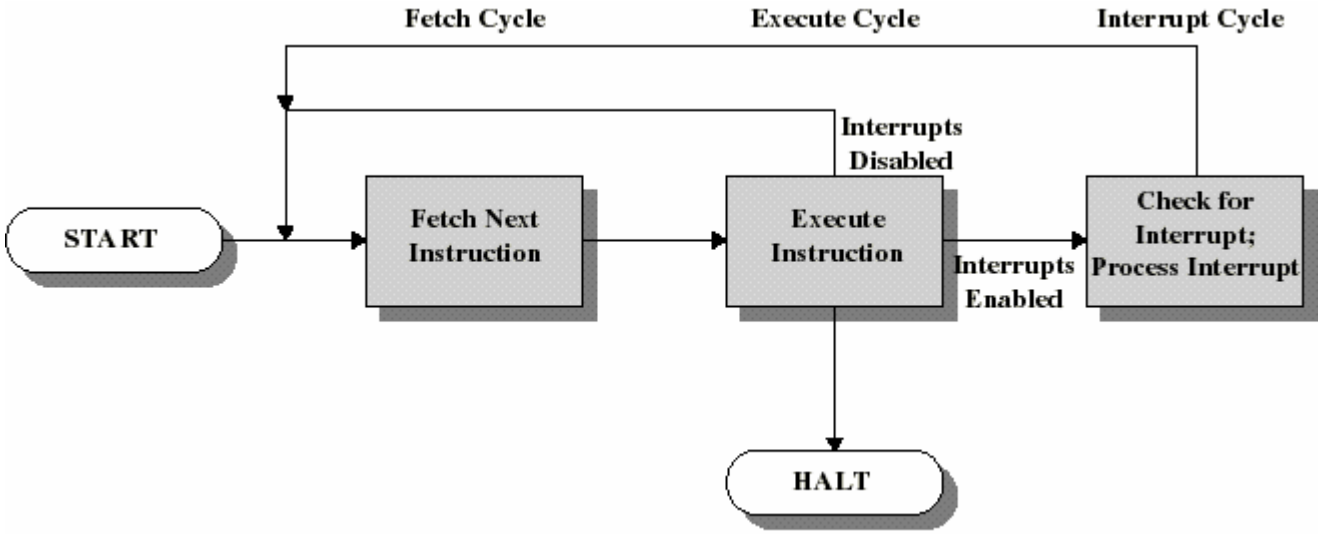
Le cycle d'instruction de base [Stallings]



Basic Instruction Cycle

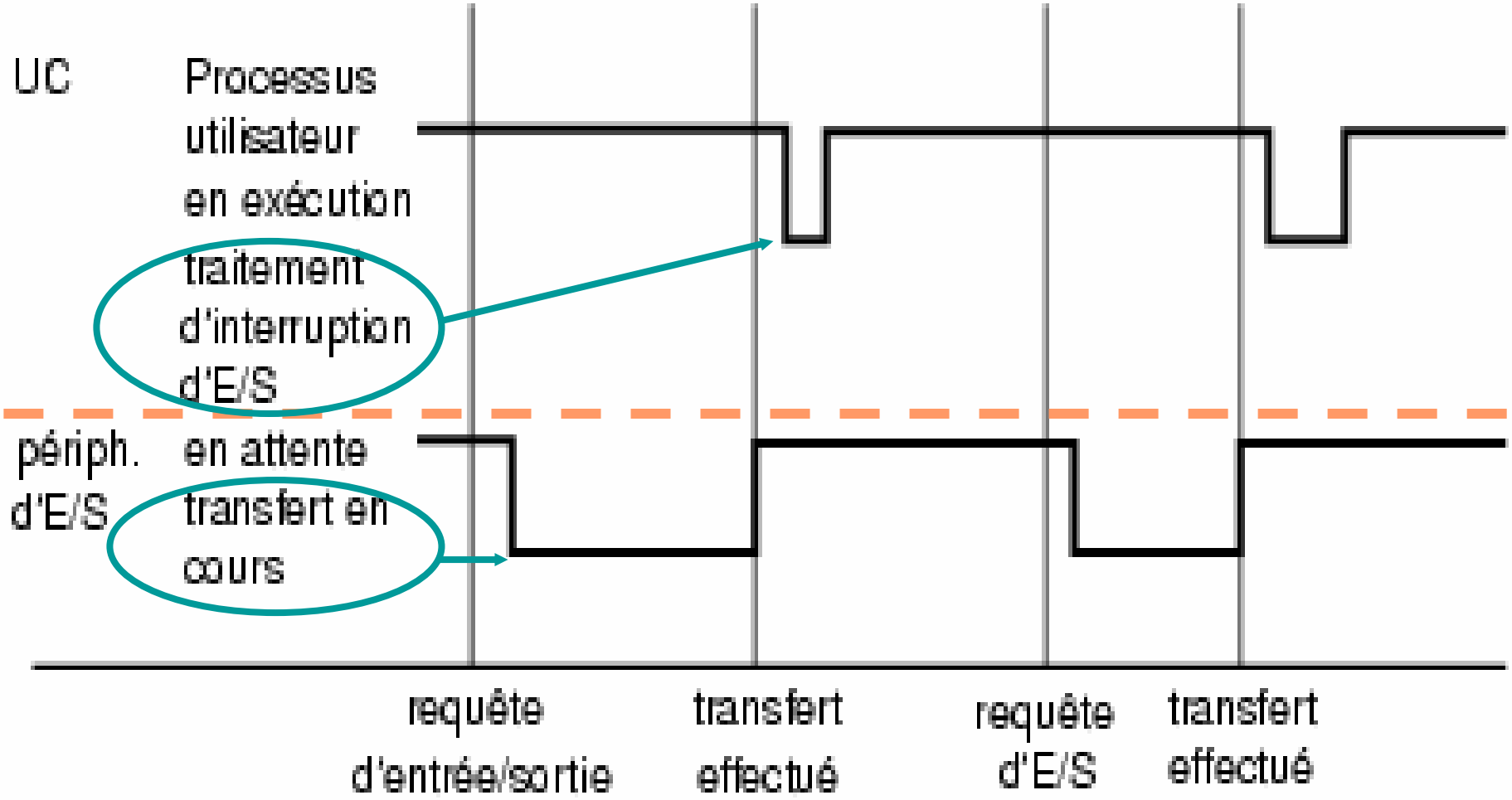
- **L'UCT extrait l'instruction de la mémoire.**
- **Ensuite l'UCT exécute cette instruction**
- **Le compteur d'instruction (PC) contient l'adresse de la prochaine instruction à extraire**
- **Le PC est incrémenté automatiquement après chaque extraction**

Le cycle d'instruction avec interruptions [Stallings]



- **Après chaque instruction, *si les interruptions sont habilitées*, l'UCT examine s'il y a eu une interruption**
- **S'il n'y en a pas, il extrait la prochaine instruction du programme**
- **S'il y en a, il suspend le pgm en cours et branche l'exécution à une position fixe de mémoire (déterminée par le type d 'interruption)**
  - une partie de la mémoire et réservée pour ça

Traitement des interruptions



Le pgm de gestion de l'interruption
(interrupt handler)

- **Est un pgm qui détermine la nature d'une interruption et exécute les actions requises**
- **L'exécution est transférée à ce pgm...**
- **…et doit revenir au programme initial au point d'interruption pour que celui-çi continue normalement ses opérations**
- **Le point d'interruption peut se situer n'importe où dans le pgm (excepté où les interruptions ne sont pas habilitées).**
- **L'on doit donc sauvegarder l'état du programme**
  - Registres UCT et autres infos nécessaires pour reprendre le programme après

Interruptions causées par les périphériques ou par le matériel

- **E/S**
  - lorsq'une opération E/S est terminée

- **Bris de matériel (ex: erreur de parité)**

Interruptions causées par le programme usager

- **Exception**
  - Division par 0, débordement
  - Tentative d'exécuter une instruction protégée
  - Référence au delà de l'espace mémoire du progr.
- **Appels du Système**
  - Demande d'entrée-sortie
  - Demande d'autre service du SE
- **Minuterie établie par programme lui-même**

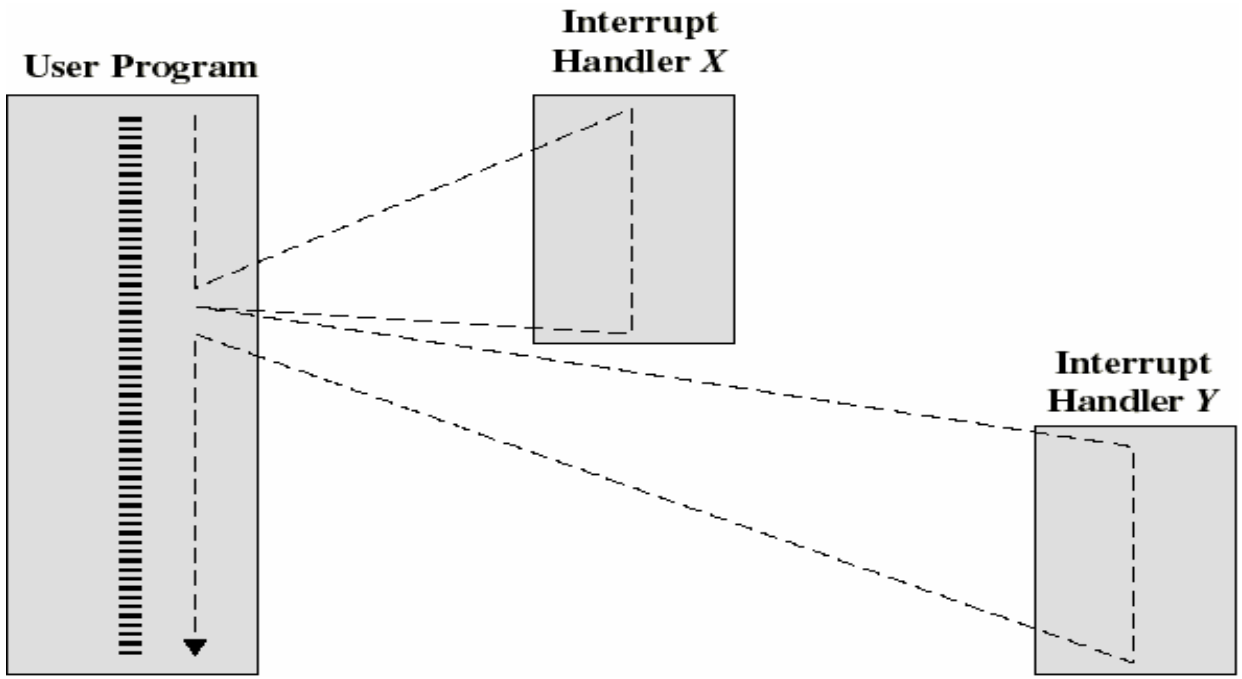
Interruptions causées par le SE

- **Minuterie établie par le SE**
- **Préemption: processus doit céder l'UCT à un autre processus**

Terminologie d`interruptions

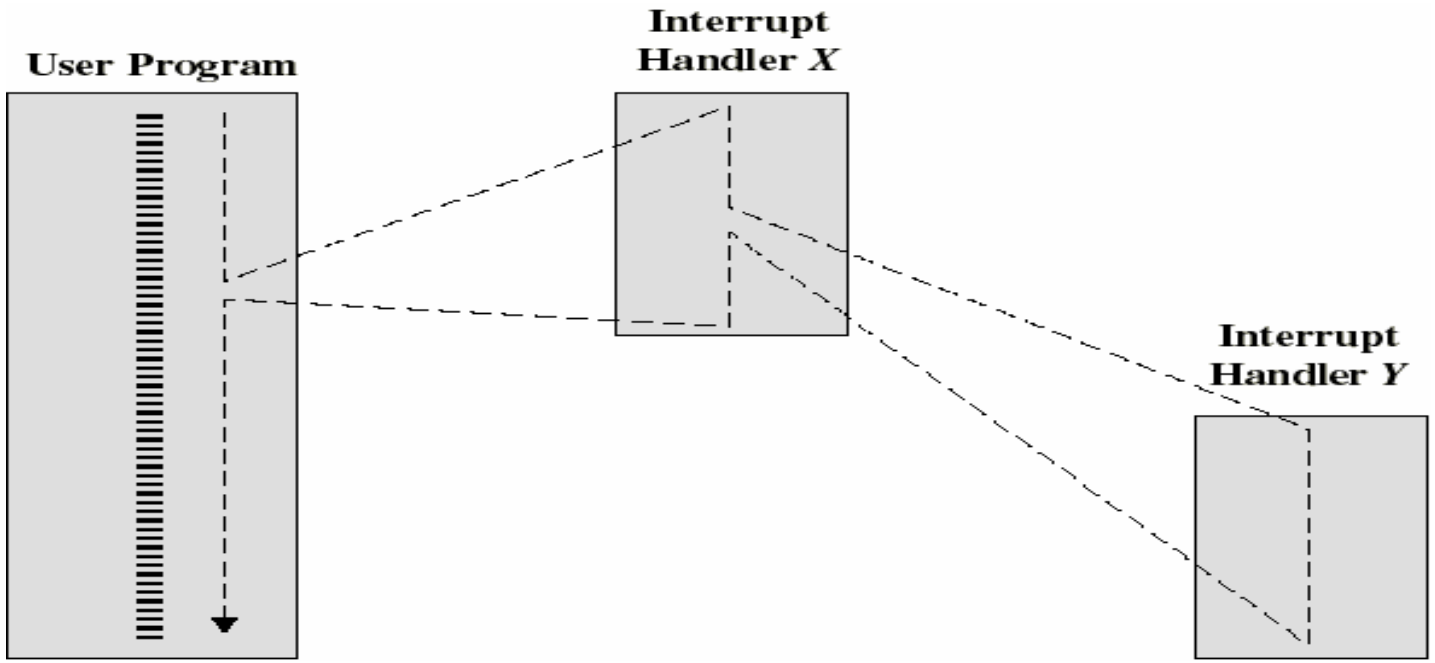
- **Pas normalisée, d'ailleurs les mécanismes de traitement sont pareils…**
- **C'est une bonne idée de distinguer entre:**
  - trappes: causées par le pgm en exécution: division par 0, accès illégal, appels du système...
  - interruptions: causées par événements indépendants: minuterie, fin d` E/S
  - fautes: ce mot est utilisé surtout par rapport à la pagination et la segmentation.

Ordre séquentiel des interruptions [Stallings]



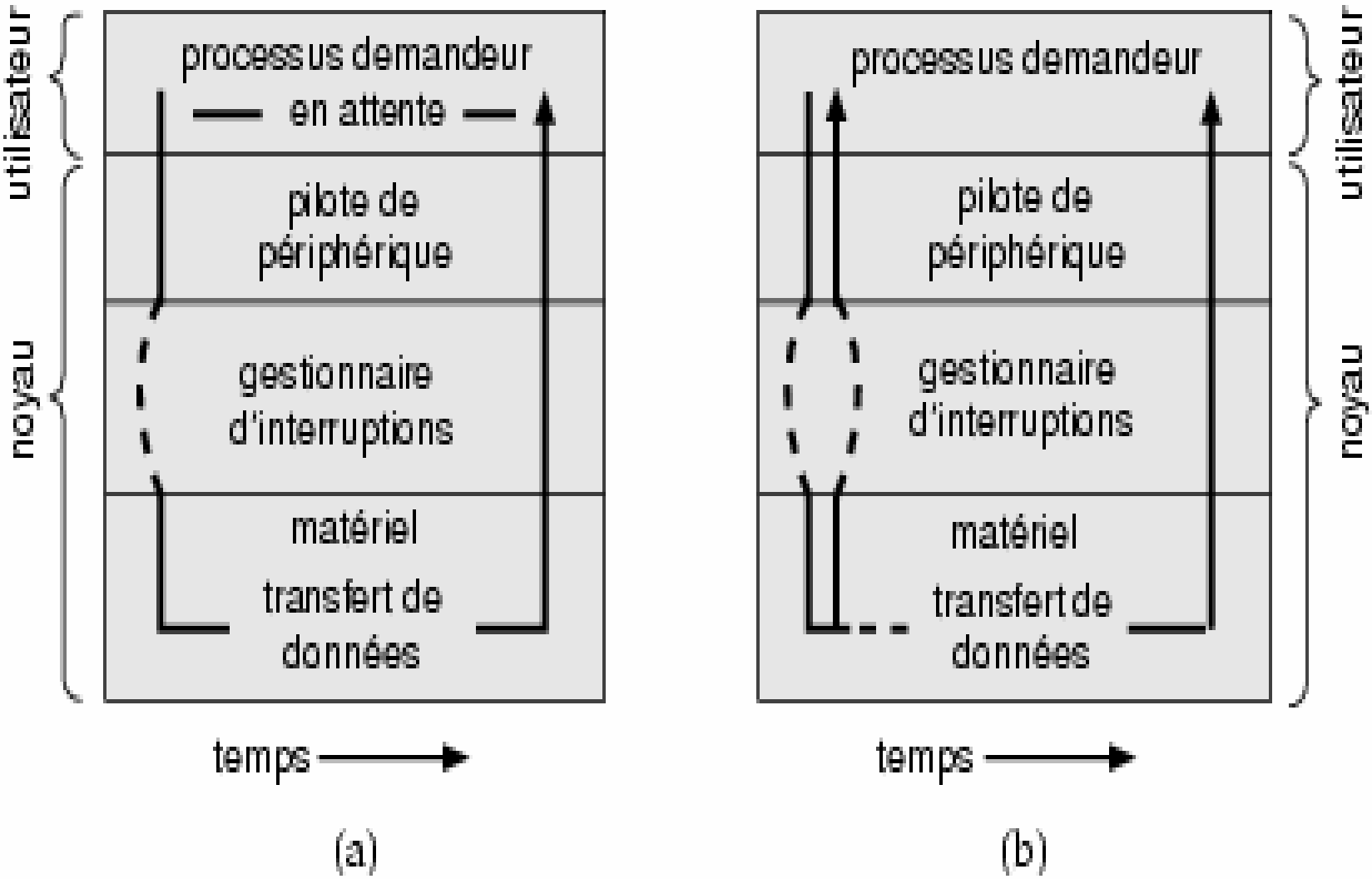
- **Interruption désactivée durant l'exécution d'un IH**
- **Les interruptions sont en attente jusqu'à ce que l'UCT active les interruptions (file d'attente en matériel).**
- **L'UCT examine s'il y a des interruptions en attente après avoir terminé d'exécuter l'IH**

Interruptions avec priorités



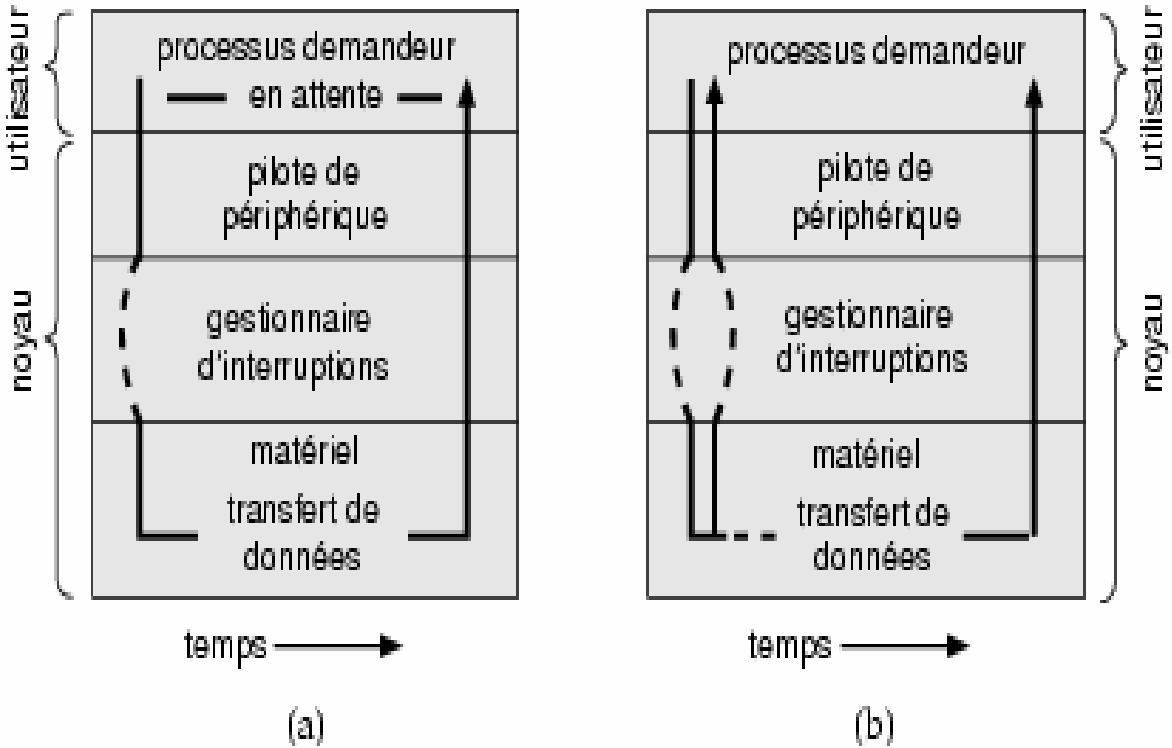
- **L'IH d'une interruption de priorité faible peut se faire interrompre par une interruption de priorité élevée**
- **Exemple: les données arrivant sur une ligne de communication doivent-être absorbées rapidement pour ne pas causer de retransmissions**

Deux méthodes d'E/S



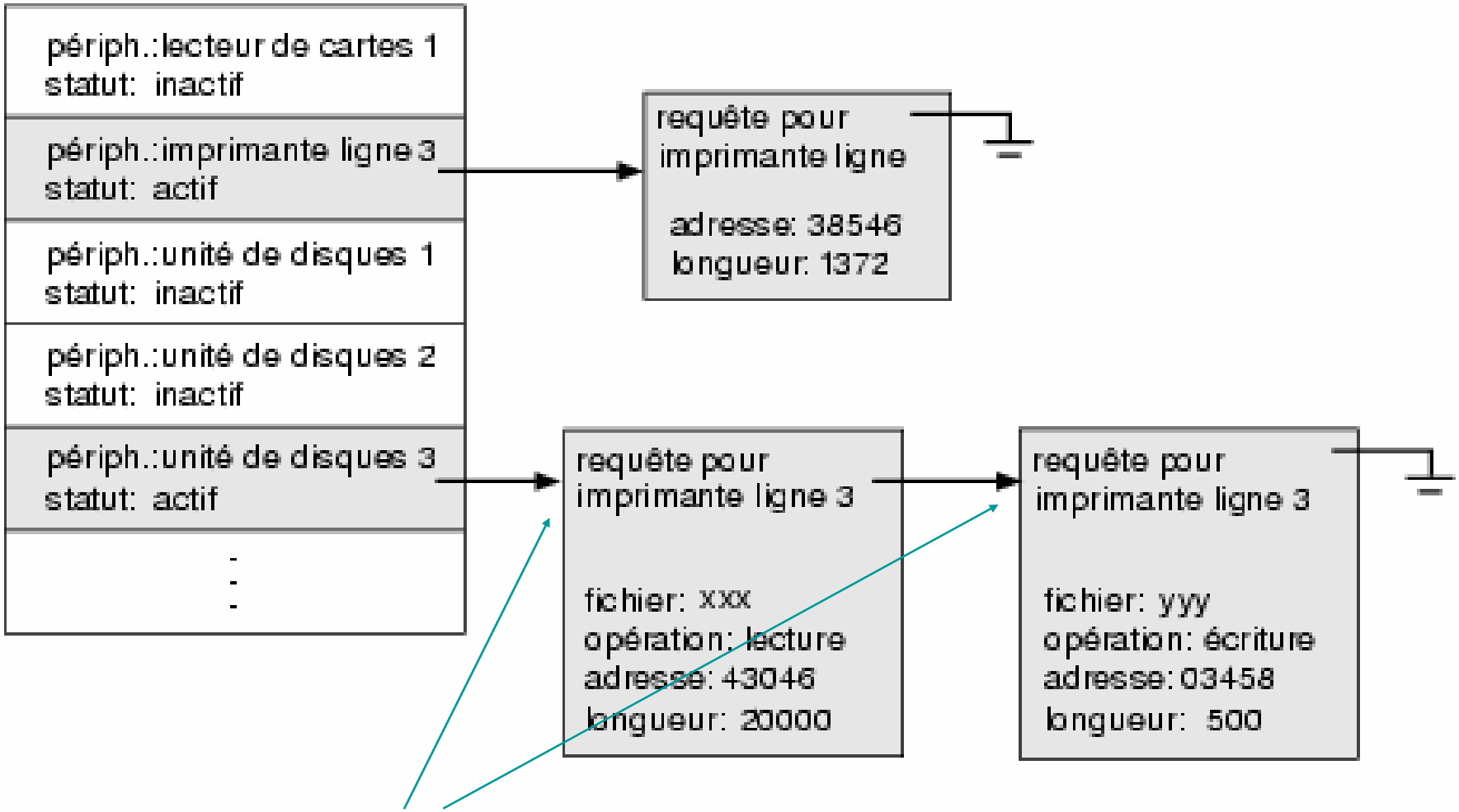
La Multiprogrammation est possible dans le cas de méthode (b)

- **Après l'initiation d'une op d'E/S, le contrôle retourne à l'UCT**
- **Qui peut utiliser le temps d'attente E/S pour exécuter un autre programme**



Pour gérerer les unités d'E/S (plus. E/S peuvent être en cours)

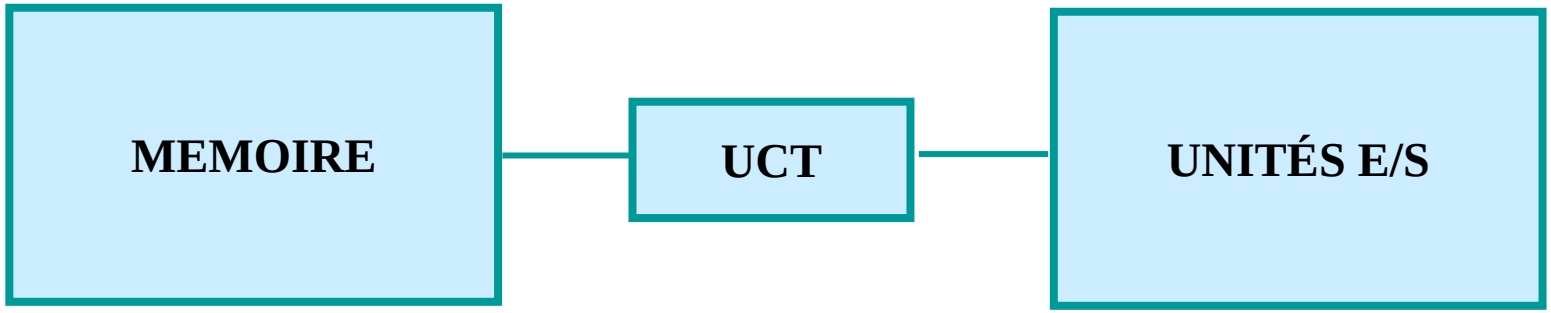
Tableau de statut des unités E/S



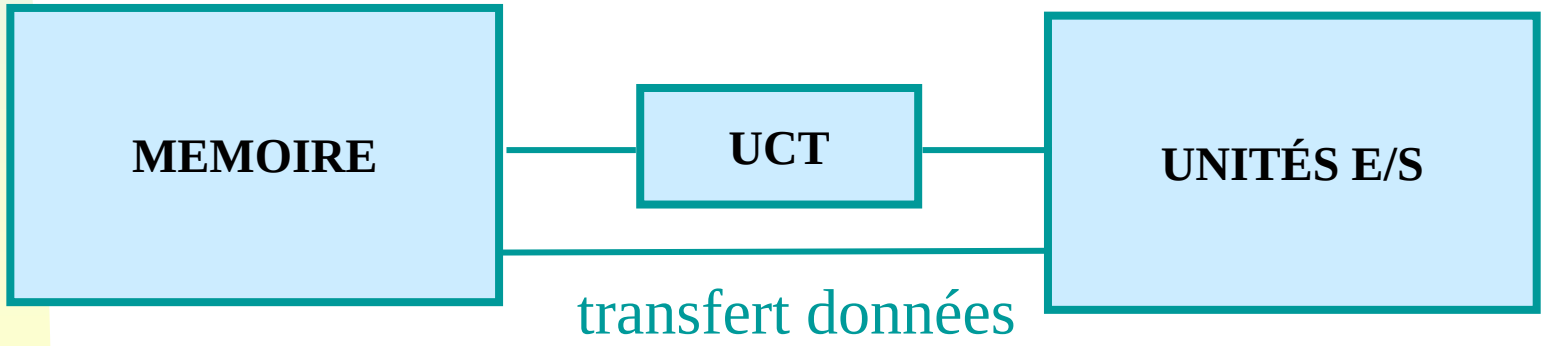
Erreur dans la figure: *imprimante* devrait être *disque 3* (2 fois)

Accès direct à la mémoire (DMA)

- **Sans DMA, tous les accès de mémoire passent à travers l'UCT**
  - Donc les E/S occupent une certaine portion du temps de l'UCT, même si l'UCT pourrait en même temps exécuter un processus (vol de cycles)
- **Avec DMA, les unités d'E/S transfèrent les données directement avec la mémoire**
  - L'UCT est impliquée seulement pour initier et terminer les E/S
  - L'UCT est complètement libre d'exécuter d'autres processus (pas de vol de cycles)



Sans DMA

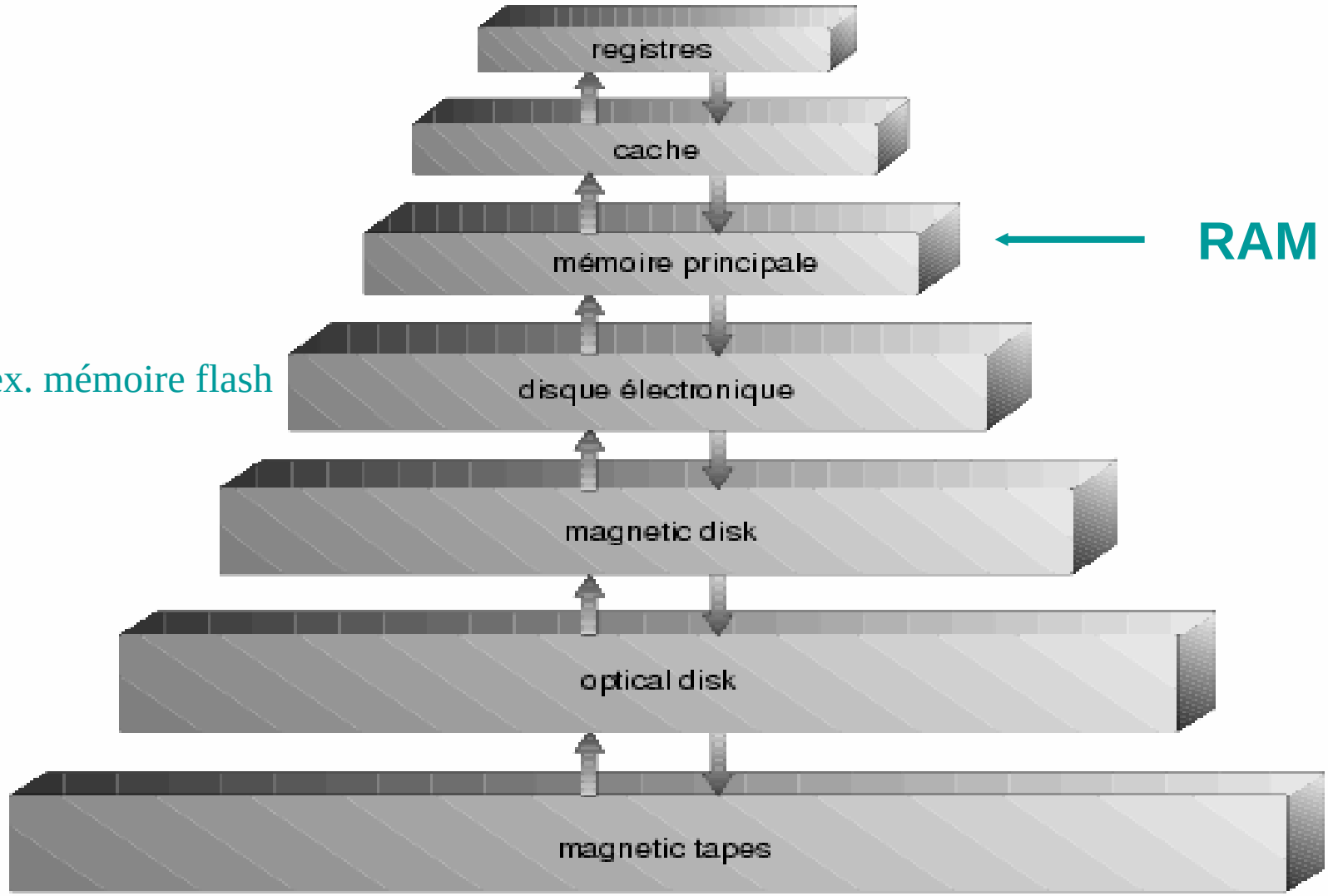


Accès directe (DMA)

Hiérarchie de mémoire

- **Différentes types de mémoire**
- **Constituent une hiérarchie**
  - vitesse (de plus vite à moins vite)
  - coût (de plus cher à moins cher)
  - permanence ou non

Hiérarchie de mémoire



Important pour la compréhension du concept de mémoire virtuelle

- **L'UCT ne peut pas accéder à une instruction ou à une donnée que s'ils se trouvent**
  - En cache dans les ordinateurs où il y a de cache
  - Ou sinon en mémoire centrale (RAM)
- **Donc ces données doivent être apportées en RAM ou cache au besoin**

Protection

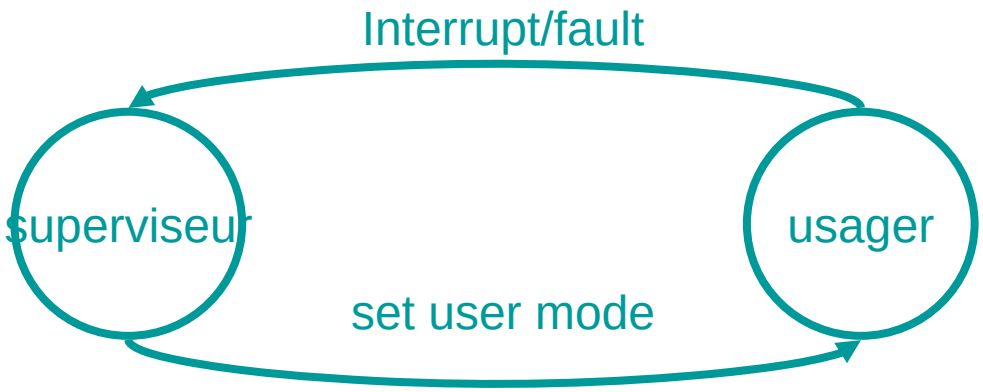
- **Plusieurs processus et le S/E partagent la mémoire, exécutant parfois les mêmes instructions**
- **Il faut empêcher que l ’un fasse des choses réservées à l ’autre**
- **Il faut les protéger les uns des autres**

- ***Protection d ’instructions***
- ***Protection de mémoire***

Instructions protégées = privilégiées

- **Ne peuvent être exécutées que par le S/E, en mode superviseur**
- **Exemples:**
  - Les instructions d'E/S
  - Instructions pour traiter les registres non-visibles d'UCT
  - Instructions pour la minuterie
  - Instructions pour changer les limites de mémoire
  - Instructions pour changer de mode d'exécution (superviseur,usager)
- **Le programme usager peut demander au SE que ces opérations soient exécutées, mais il ne peut pas les exécuter directement**

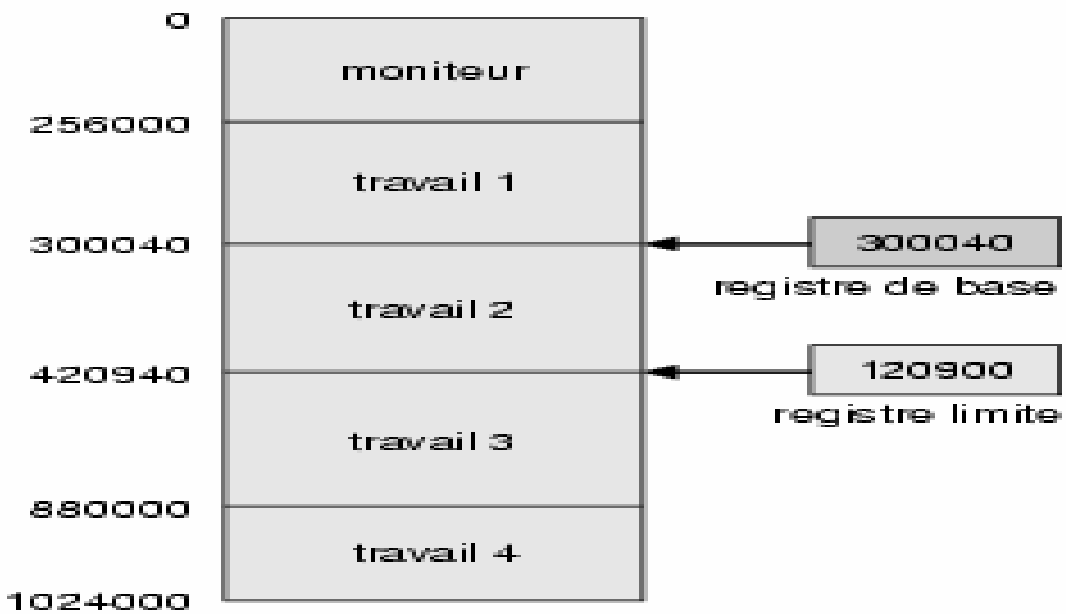
Fonctionnement *double mode*



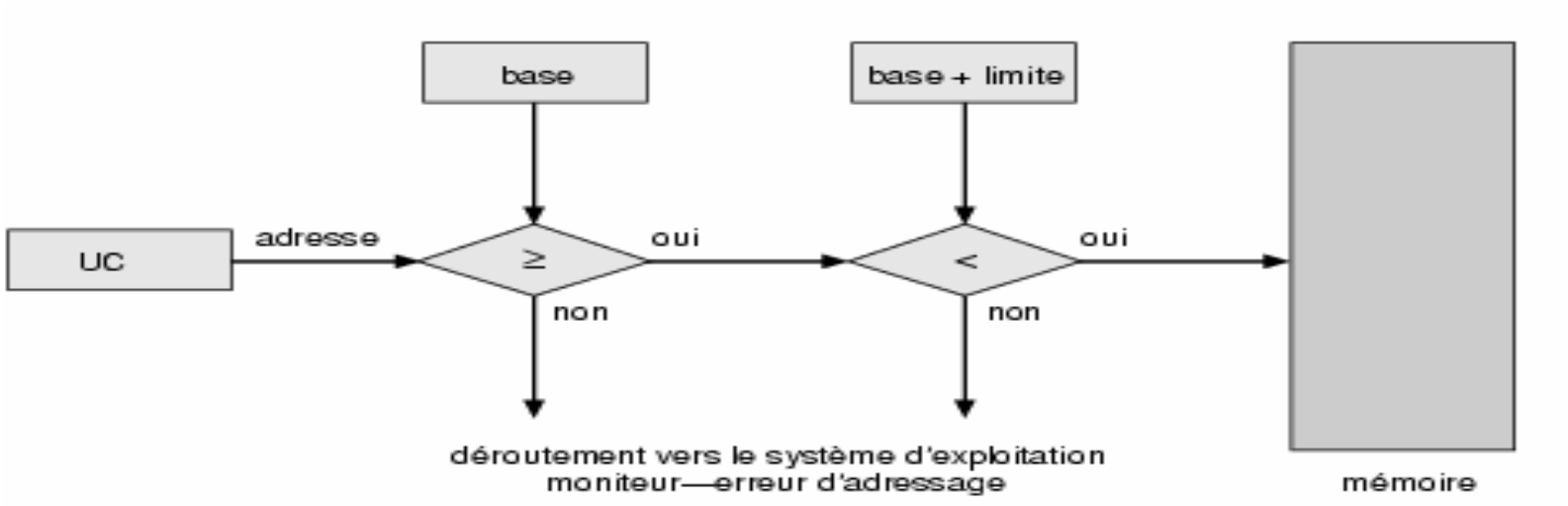
- **Un registre d'UCT contient un bit qui dit si l 'UCT exécute couramment en mode *superviseur* ou en mode *usager***
- **ce bit est changé automatiquement à mode superviseur lors d'une interruption**
- **certaines instructions ne peuvent être exécutées que en mode superviseur (instructions privilégiées):**
  - des tentatives de les exécuter en mode usager causeront une interruption, et retour à mode superviseur
- **le mode superviseur peut être changé à mode usager par une instruction privilégiée**
- **ces deux modes ont aussi des autres noms, v. livre**

Protection de mémoire: chaque processus doit rester dans ses propres bornes de mémoire

- **Solution typique: deux registres dans l 'UCT**
  - quand l 'UCT exécute un processus, elle sait quelle est la borne inférieure et supérieure de la zone de mémoire de ce processus
  - l 'adresse de chaque instruction est comparée à ces deux adresses avant l 'exécution
  - si un processus cherche à dépasser ses limites: **interruption**



Protection de mémoire



- l'adresse de chaque instruction est comparée à ces deux adresses avant l'exécution
  - seulement si le processus exécute en mode usager
- si un processus cherche à dépasser ses bornes:
  - Interruption → mode superviseur
- les instructions pour affecter les registres bornes sont privilégiées

Appels du système (system calls)

- **Quand un processus usager a besoin d ’un service du SE, par ex. E/S, il exécute un appel du système**
- **C’est une instruction qui cause une interruption (trap) et changement de mode (mode superviseur)**
- **Est associée à des paramètres qui indiquent le type de service désiré**
- **Le S/E prend la relève et exécute le service, il retourne puis au processus appelant avec des params qui indiquent le type de résultat**
  - changement de mode (mode usager)

Concepts importants du Chapitre 2

- Registres d'UCT, tampons en mémoire, vecteurs d'interruption
- Interruption et polling
- **Interruptions et leur traitement**
- Méthodes d'E/S avec et sans attente, DMA
- Tableaux de statut de périphériques
- Hiérarchie de mémoire
- Protection et instructions privilégiées, modes d'exécution
- Registres bornes ou limites
- Appels de système

Dans le livre, pour ce chapitre, vous devez

- **Étudier le chapitre entier**
- **La section 2.3 n'a pas été discutée en classe, cependant son contenu est important pour la compréhension du fonctionnement des systèmes informatiques et sûrement elle est enseignée dans d'autres cours…**
  - SVP réviser
  - à discuter plus tard (Chap. 13)

Informations additionnelles

- **Pour des explics claires sur comment différents parties d'un ordinateur fonctionnent, je vous recommande hautement**

http://computer.howstuffworks.com/

Un site qui contient des explications très claires sur un grand nombre de sujets

(malheureusement, aussi beaucoup de publicité)

Structure des Systèmes d'Exploitation

Chapitre 3

Beaucoup de choses dans ce chap. sont faciles à lire et je vais pas les discuter en classe.

Nous reviendrons sur plusieurs de ces concepts.

Section 3.7 sera discutée dans le lab.

http://w3.uqo.ca/luigi/

Concepts importants du Chapitre 3

- **Responsabilités et services d'un SE**
- **Le noyau**
- **Appels du système (system calls)**
- **Communication entre processus**
  - Messagerie et mémoire partagée
- **Structure à couches**
- **Machines virtuelles**

Gestion de processus et UCT

- **Un *processus=tâche* est un programme en exécution**
  - il a besoin de *ressources* pour exécuter (UCT, mémoire, unités E/S...)
- **Le SE est responsable pour:**
  - allocation de ressources aux processus
  - création, terminaison des processus
  - suspension, reprise des processus
  - synchronisation, communication entre processus

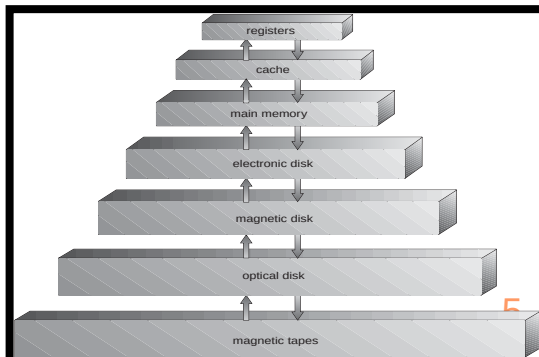
Gestion de mémoire principale (RAM)

- **Le SE est responsable pour:**
  - savoir quels processus utilisent quelles parties de la mémoire
  - savoir quels processus en demandent, et combien
  - allouer la mémoire quand elle devient disponible
  - libérer la mémoire

Gestion de mémoire virtuelle

- La mémoire principale est souvent trop petite pour contenir tous les processus en exécution
- La mémoire secondaire (disques, flash) est normalement utilisée pour contenir les parties d`un processus qui ne sont pas actives à l ’instant
- La mémoire principale et la mémoire secondaire forment donc une unité logique appelée *mémoire virtuelle*
- Pour implanter la mémoire virtuelle, le SE doit gérer de façon conjointe mémoire RAM et mémoire disque
- Mécanisme de va-et-vient (swap)

Hiérarchie de mémoire!



Services primaires des Systèmes d'exploitation

- **Exécution de programmes: chargement, exécution (load, run)**
- **Opérations E/S**
- **Manipulation fichiers**
- **Communication et synchronisation entre processus**
- **Détection et traitement d'erreurs**

Autres services importants

- **Allocation de ressources**
- **Protection de ressources**
- **Comptabilité**

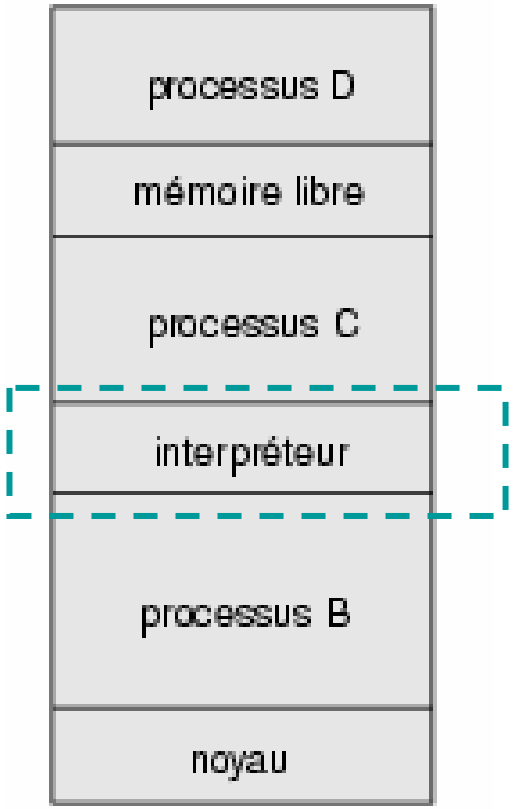
Le noyau (kernel) du SE

- **La partie résidente (toujours en RAM) du SE est appelée *Kernel = noyau***
- **Les autres parties sont amenées en RAM au besoin**
- **Contient les fonctionnalités critiques du SE: elles doivent toujours être prêtes à l ’utilisation**
  - traitement d ’interruptions
  - gestion de UCT
  - gestion mémoire
  - communication entre processus
  - etc.
- **À part ça, quoi exactement mettre dans le kernel est une question pour les concepteurs des SE**
- **La plupart des fonctionnalités discutées dans ce cours sont normalement dans le kernel**

Appels du système

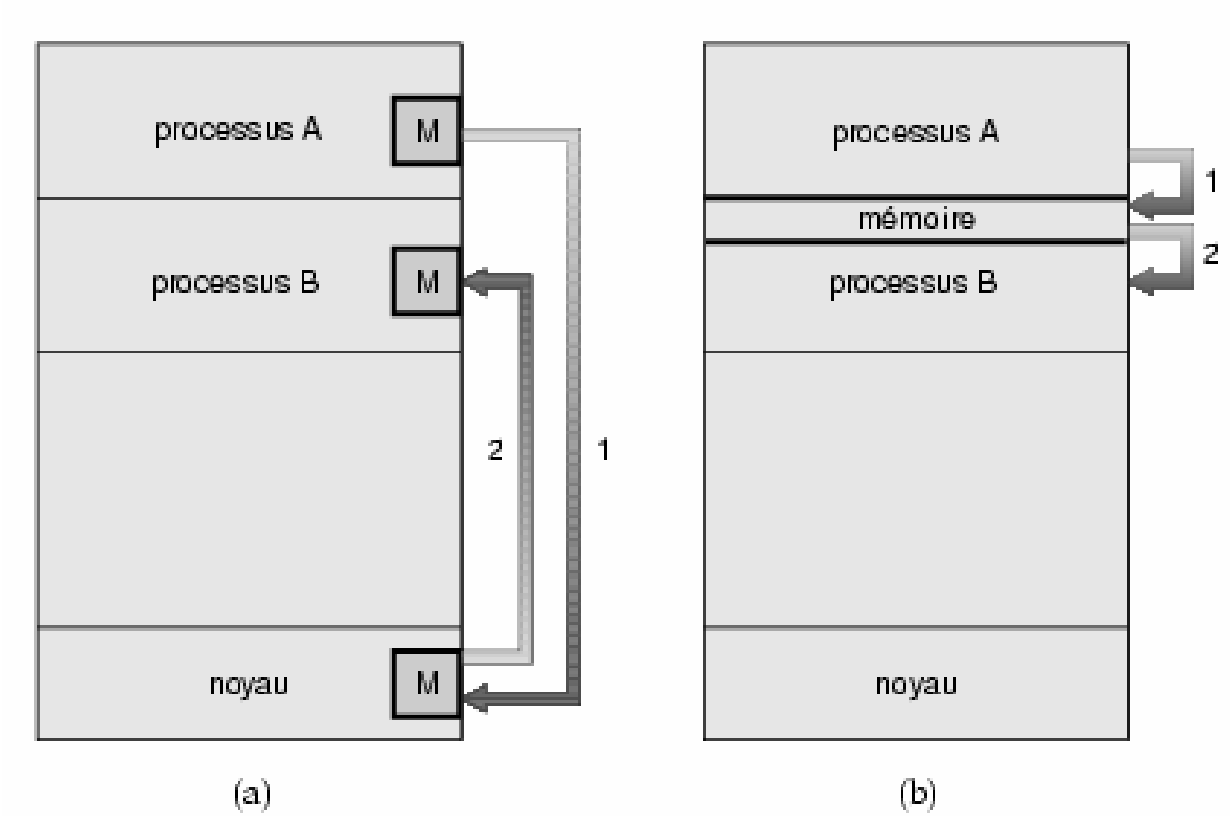
- **L'interface entre un processus et le SE**
  - directement disponibles dans les langages de programmation `bas niveau` (assembleur, C, C++)
  - ils sont cachés dans les langages a haut niveau (Java, Ada...)
- **Utilisent des paramètres pour transmettre**
  - la définition exacte des besoins de l 'usager
  - le résultat de l`appel (successful, unsuccessful...)

Interpréteur de commandes en UNIX



Le command interpreter (shell) peut démarrer et charger différents processus en mémoire, exécutant des appels de système appropriés (fork, exec). *Lire détails dans le livre, aussi v. sessions exercices.*

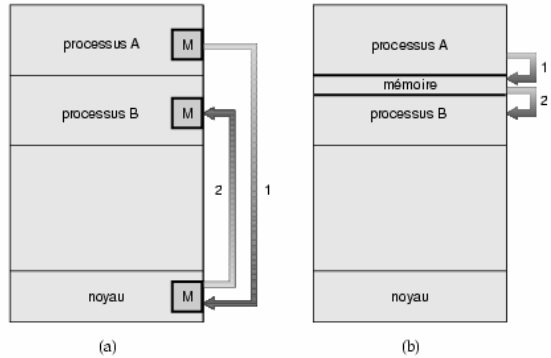
Deux modèles de communication entre processus par appels de système



a) transfert de messages entre processus (message passing)
   - utilisant le service de messagerie offert par le noyau
b) à travers mémoire partagée entre processus (shared memory)

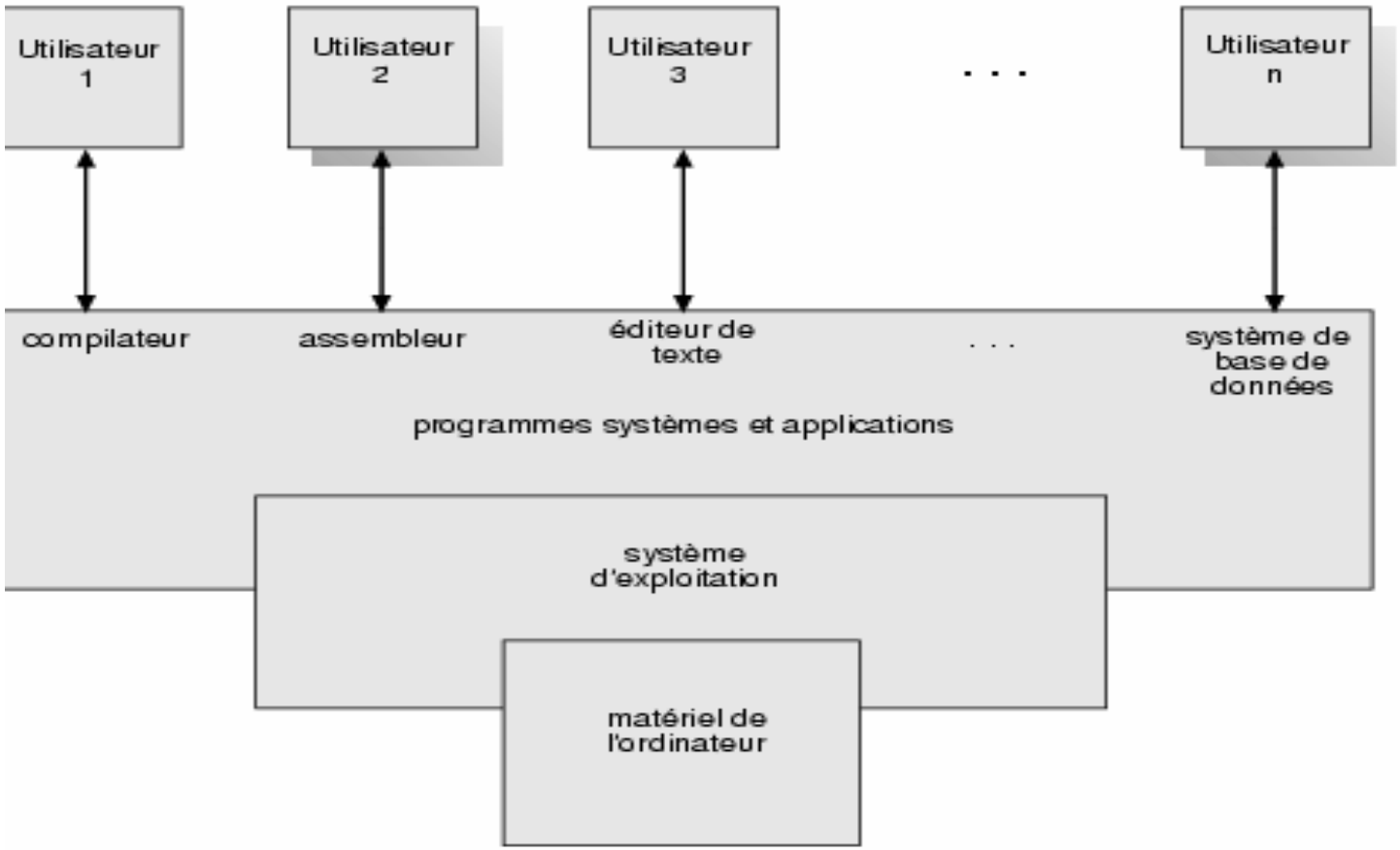
Messagerie et mémoire partagéé

- **Messagerie:**
  - il faut établir une connexion entre processus (appels de système open connection, accept connection, close connection, read/send message)
  - les processus s 'envoient des messages utilisant des identificateurs préalablement établis
- **Mémoire partagée**
  - il est nécessaire d 'établir une zone de communication entre processus
  - les processus doivent mutuellement synchroniser leur accès a cette zone
    - Pour ceci, il font appel au SE (Chap. 7)



v. chap. 7: synchro de proc.

Programmes système



Pas partie du kernel, en augmentent la fonctionnalité.

Voir discussion et exemples dans le livre

Partage de responsabilités entre programmes de systèmes et noyau

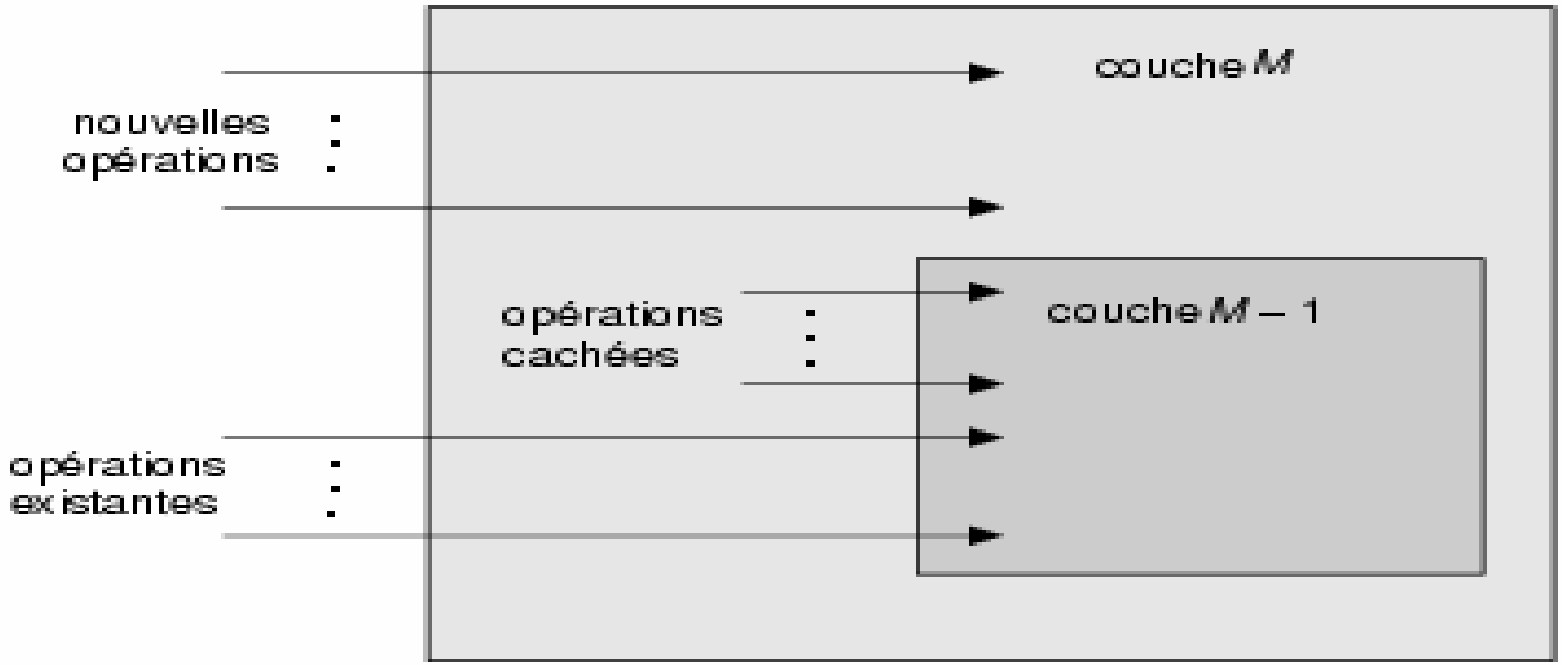
- **C’est en partie une décision de conception de SE de décider quelles fonctionnalités doivent être implémentées dans le kernel, et quelles dans les programmes de système.**
- **Dans l ’évolution des SE il y a eu un transfert de fonctionnalités vers l ’extérieur de la figure précédente**
- **Dans les SE modernes, les programmes de système sont l`interface entre usager et noyau**

Structure en couches dans les SE modernes

- **Un SE est divisé dans un certain nombre de couches, bâties les unes sur les autres**
  - la couche la plus basse est le matériel
  - la plus élevée est l ’interface usagers

- **Les couches supérieures utilisent les fonctionnalités fournies par les niveaux inférieurs**

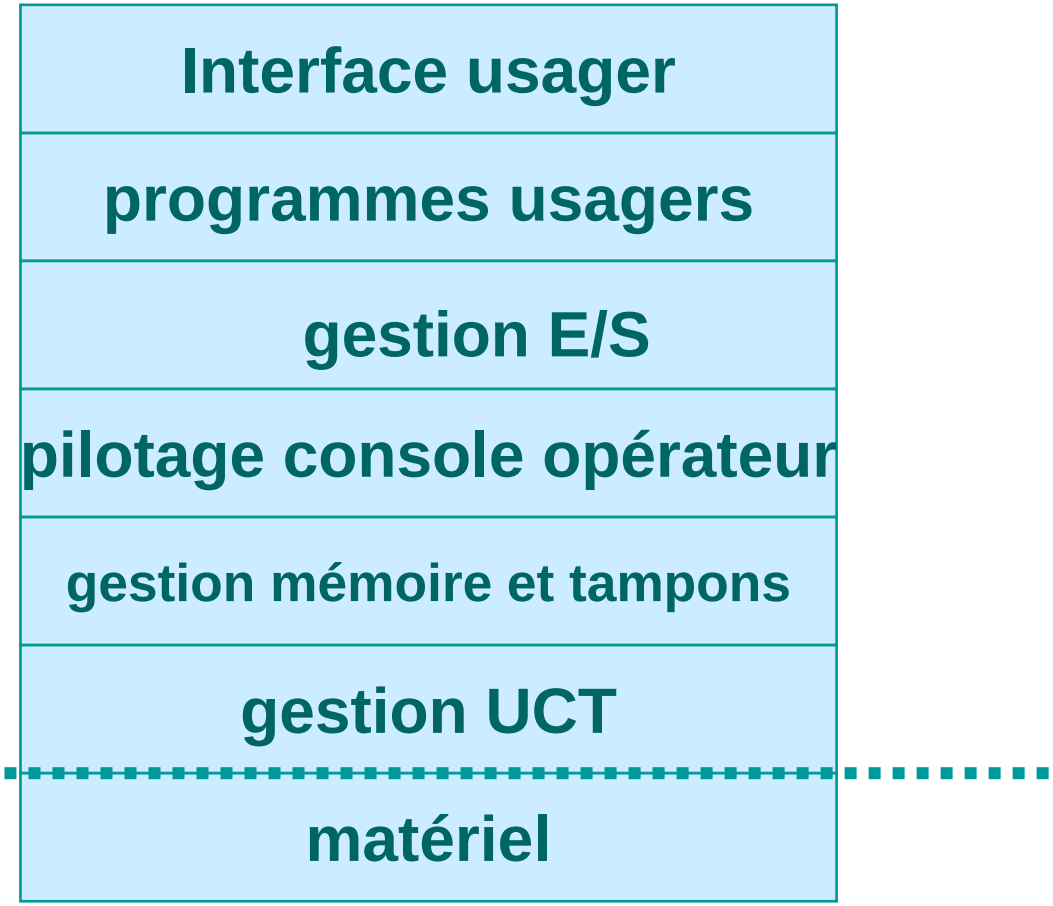
Structure à couches



- **opérations créées dans une couche pour les couches extérieures**
- **opérations fournies par une couche, utilisées par la prochaine couche, et cachées aux couches extérieures**
- **opérations d ’une couche intérieure rendues disponibles à une couche extérieure**
  - à éviter en principe, mais… v. après

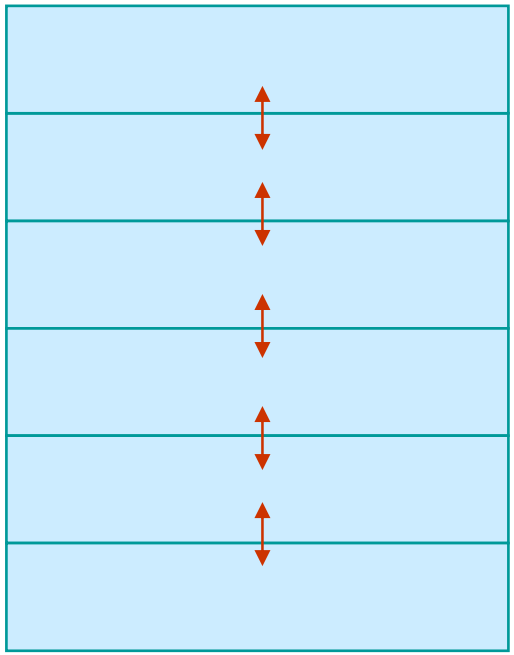
Structure à couches dans le système THE (1968)

- **La structure à couches fut inventée dans le système THE (E.W. Dijkstra) qui avait les couches suivantes:**

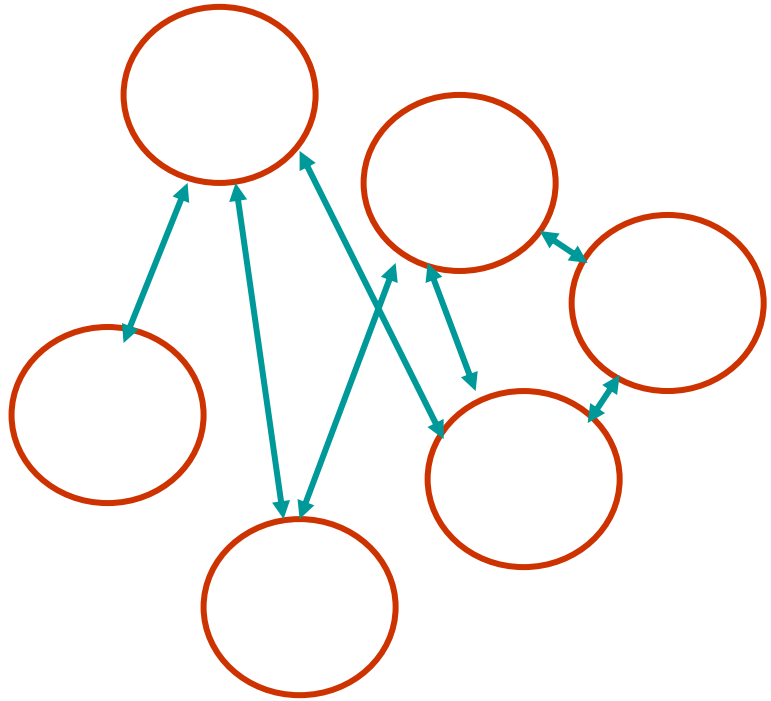


L'autre possibilité serait la structure *réseau*

- **Plus difficile à gérer, à cause des nombreuses interfaces possibles**



À couches



Réseau

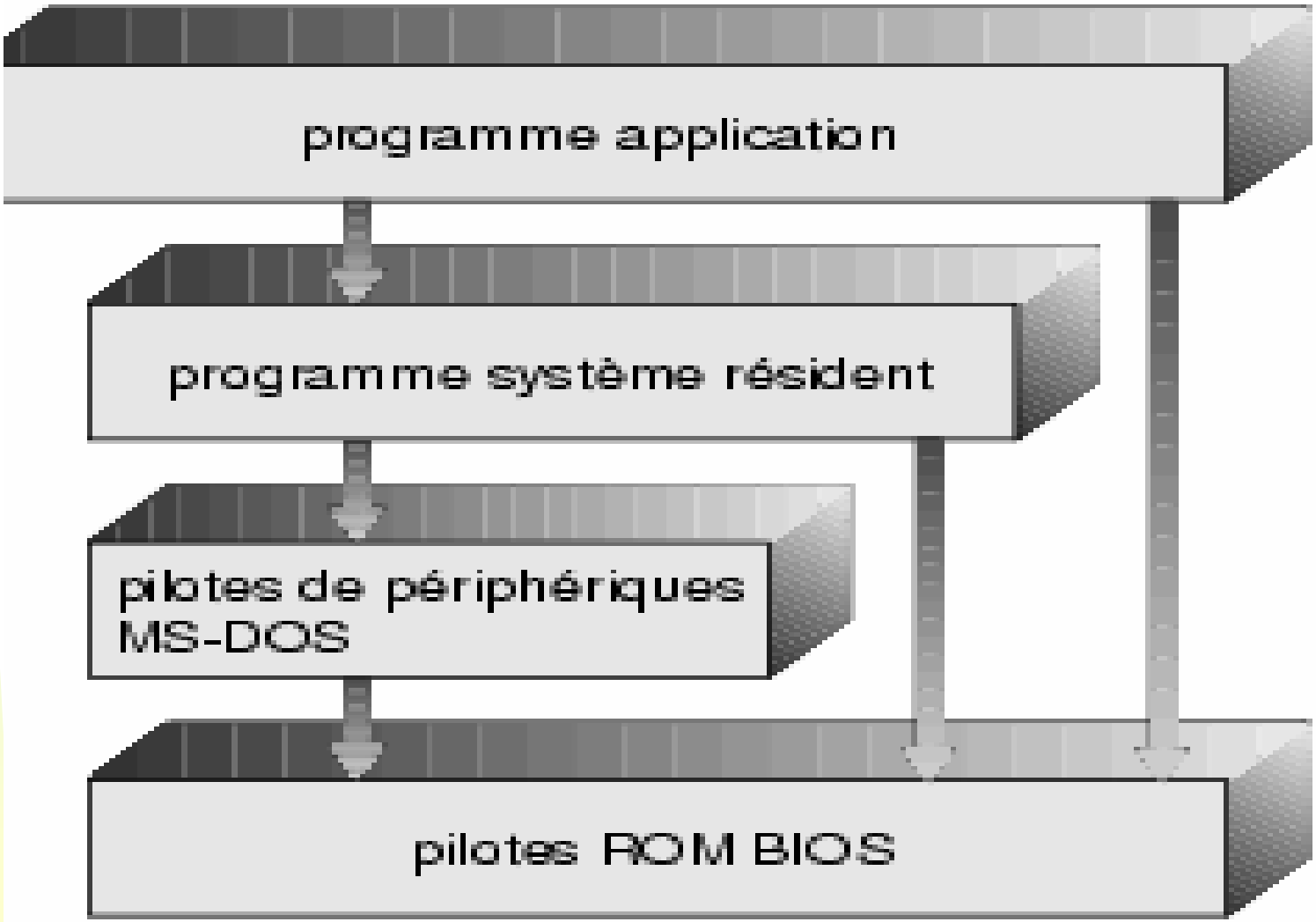
Avantages, désavantages de suivre fidèlement une structure en couches

- **Avantages:**
  - Chaque couche ne doit connaître que les fonctionnalités fournies par la couche immédiatement sous-jacente
  - Chaque couche ajoute ses propres fonctionnalités
  - Les erreurs peuvent plus facilement être isolés dans une couche spécifique
    - Maison construite un étage à la fois… poser l'étage n seulement quand l'n-1 est solide
- **Désavantages:**
  - Pas efficace car un appel des programmes usager à des ressources du matériel implique autant d'appels qu'il y a des couches intermédiaires
- **Excellent principe, pas toujours fidèlement suivi**
  - Pour des raisons d'efficacité

Structure de système - Approche simple

- **MS-DOS - cherchait à obtenir une fonctionnalité maximale avec des ressources limitées**
  - mono-tâche, mono-usager
  - pas très modularisé
    - manque de séparation claire entre couches
  - accès direct aux périphériques (écran, etc.) permis aux programmes d`application
    - manque de contrôles, vulnerabilité
  - malheureusement, il fut adapté à des fonctionnalités plus complexes...
    - Fut la prémière base de Windows et une grande partie de l'histoire de Windows a été un effort de dépasser les limitations de MS-DOS

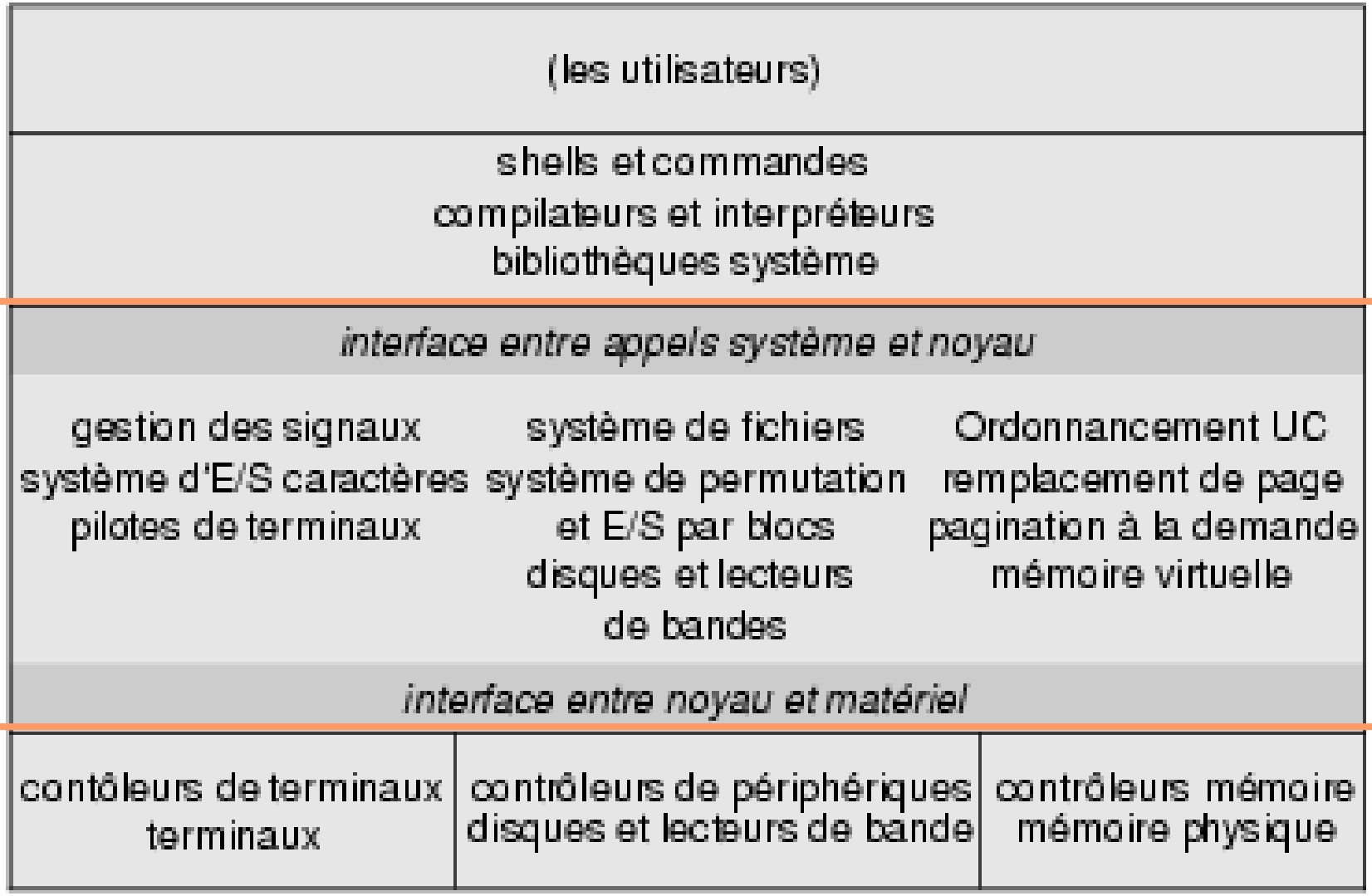
Couches du MS-DOS



Structure UNIX

- **Multi-tâches, multi-usagers depuis le début**
- **Le système UNIX initial était aussi préoccupé par les limitation du matériel**
- **Distinction entre:**
  - programmes du système
  - noyau
    - tout ce qu'il y a entre l 'interface des appels de système et le matériel
    - fournit dans une seule couche un grand nombre de fonctionnalités
      - système fichiers, ordonnancement UCT, gestion mémoire...
- **Plus modulaire et protégé que MS-DOS**

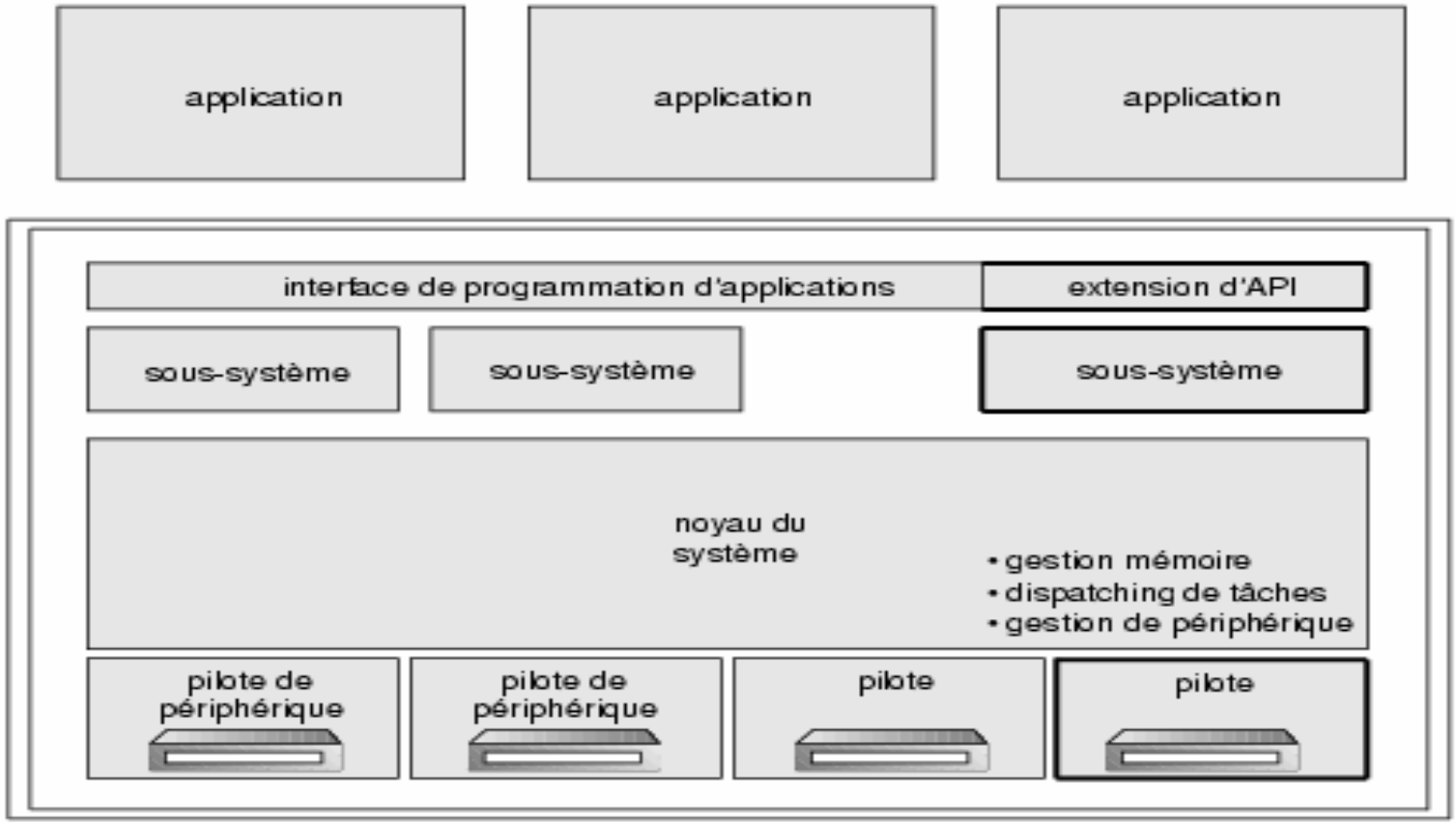
Structure UNIX: peu de couches



Micronoyaux (microkernels)

- **Dans les premiers SE, aussi UNIX, tout était dans le noyau**
- **Après, un effort fut fait pour laisser dans le noyau UNIX seulement les fonctionnalités absolument nécessaires**
- **Une des fonctionnalités du micronoyau UNIX est la communication par échange de messages**
  - utilisé pour la communication entre programme client et service

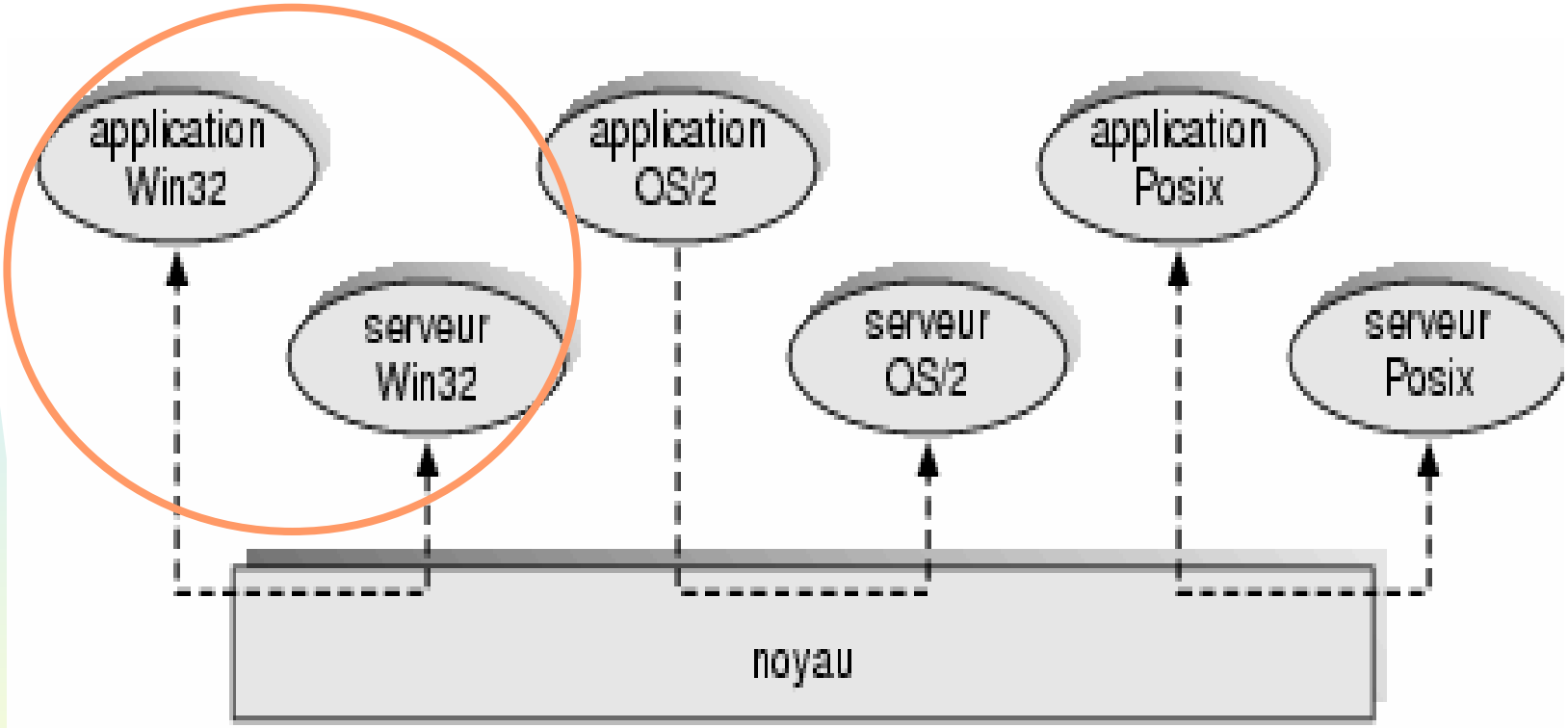
Structure à couches dans OS/2 (IBM)
suit les idées d'unix



L 'OS/2 était beaucoup mieux organisé que MS-DOS, et donc moins vulnérable. Cependant il était peu performant.

Win-NT a cherché à utiliser des principes semblables, mais avec une intégration meilleure des couches.

Structure client-serveur dans noyau Win-NT



Win-NT a un petit noyau qui fournit une structure client-serveur, en utilisant échanges de messages

Supporte différents SE: Win, OS/2, Posix

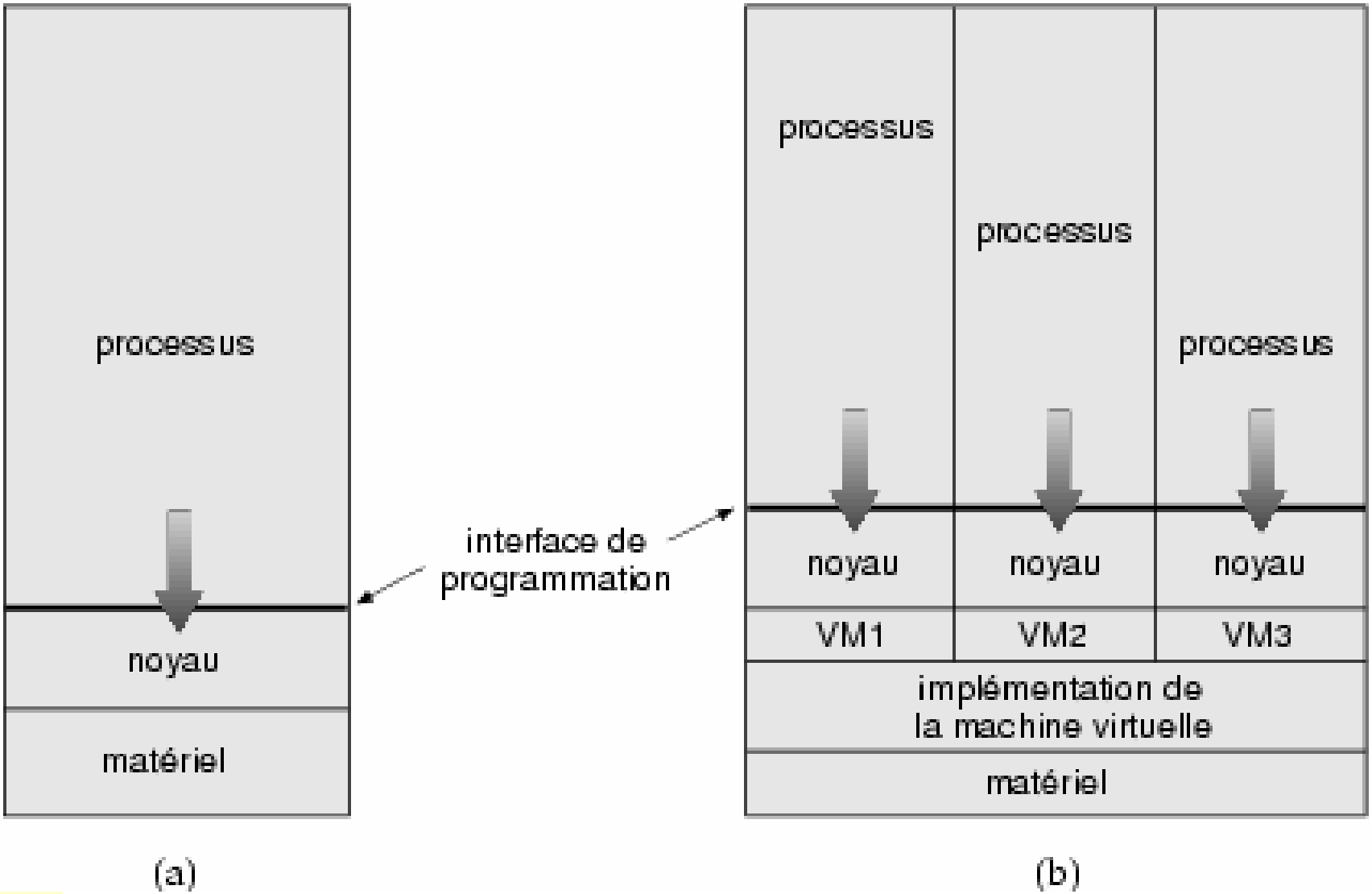
Machines virtuelles: le problème et la solution

- **Comment permettre de rouler différents SE sur une seule machine physique?**
- **Pas évident, car chaque SE demande accès direct au matériel**
- **SOLUTION: Un programme qui crée une couche qui met à disposition plusieurs machines physiques *virtuelles***
- ***Chaque machine se comporte comme une machine physique séparée***
- **Sur chacune, nous pouvons rouler un SE différent**

Machines Virtuelles

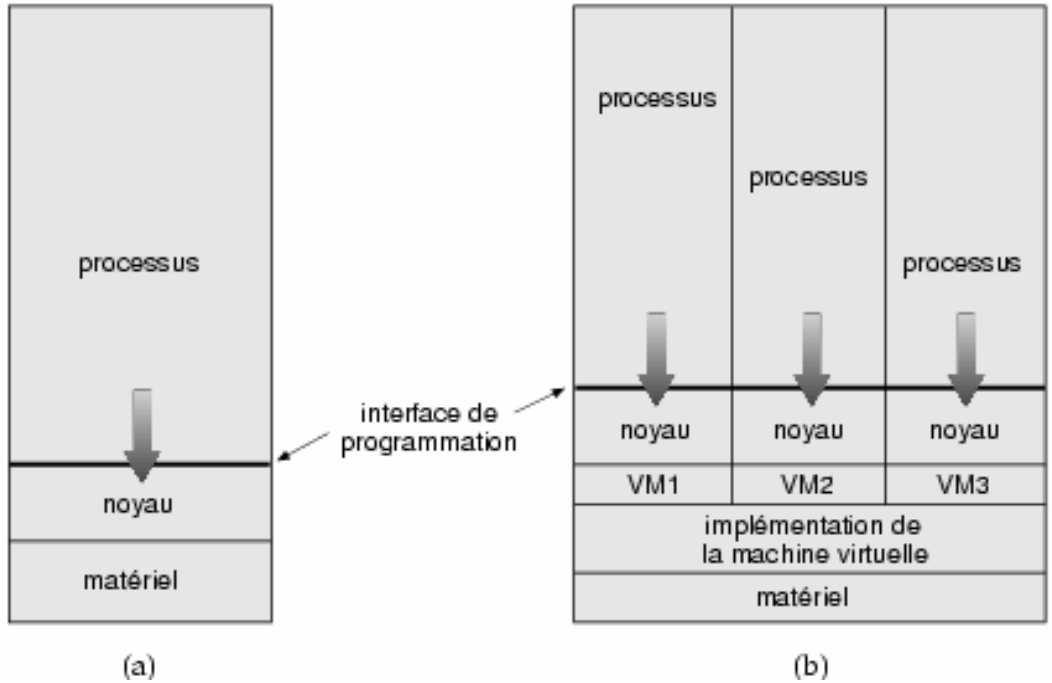
- ***Virtuel*** **en informatique dénote quelque chose qui n 'est pas *réel*, n 'est pas du matériel: il est construit par le logiciel sur la base des ressources fournies par le matériel**
- **Une machine virtuelle est une machine créée par des couche de logiciel**
- **Elle peut avoir des caractéristiques identiques à la machine physique du système:**
  - mêmes instructions, etc.
- **Ou elle peut 'simuler' une autre machine physique**
  - p.ex.pour exécuter Microsoft sur Apple
- **Plusieurs machines virtuelles peuvent être créées sur une machine physique donnée!**

(a) Une seule mach. réelle et un seul noyau
(b) plus. mach. virtuelles et plus. noyaux



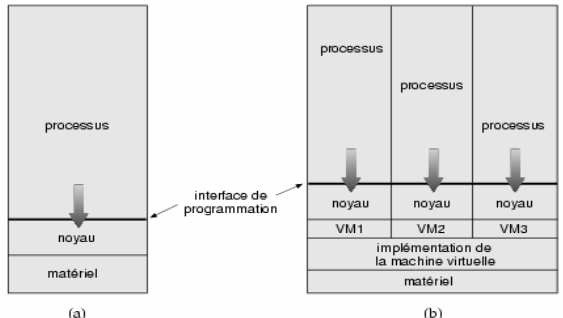
Fonctionnement typique

- **Le système VM laisse exécuter normalement les instructions non privilégiées**
- **Les appels au système sont exécutés par le système VM et les résultats sont passés à la machine virtuelle sur laquelle le processus exécute**



Avantages

- **Chaque machine virtuelle peut utiliser un SE différent!**
- **En théorie, on peut bâtir des machines virtuelles sur des machines virtuelles!**
- **Protection complète, car les machines virtuelles sont complètement isolées les unes des autres**
- **Un nouveau SE peut être développé sur une machine virtuelle sans déranger les autres**



Implémentations

- **Le concept de VM est très utilisé pour permettre de rouler un SE sur un autre**
- **P.ex. SUN, Apple, Linux permettent de rouler Windows sur leur plateforme,**
- **Ils doivent fournir à Windows un environnement que Windows reconnaît comme son environnement Intel usuel**

Concepts importants du Chapitre 3

- **Responsabilités et services d'un SE**
- **Le noyau**
- **Appels du système (system calls)**
- **Communication entre processus**
  - Messagerie et mémoire partagée
- **Structure à couches**
- **Machines virtuelles**

Par rapport au manuel…

- **Étudier sections 3.1 jusqu'à 3.6.**
- **La section 3.7 n'a pas été discutée en classe mais elle contient des concepts importants concernant Java donc c'est une excellente idée de la lire**
- **Les sections 3.8 et 3.9 ne sont pas sujet d'examen cependant il est utile de les lire.**

Gestion de Processus

Chapitre 4

http://w3.uqo.ca/luigi/

Concepts importants du Chapitre 4

- **Processus**
  - Création, terminaison, hiérarchie
- **États et transitions d'état des processus**
- **Process Control Block**
- **Commutation de processus**
  - Sauvegarde, rechargement de PCB
- **Files d'attente de processus et PCB**
- **Ordonnanceurs à court, moyen, long terme**
- **Processus communicants**
  - Producteurs et consommateurs

Processus et terminologie (aussi appelé job, task, user program)

- **Concept de processus: un programme en exécution**
  - Possède des ressources de mémoire, périphériques, etc
- **Ordonnancement de processus**
- **Opérations sur les processus**
- **Processus coopérants**
- **Processus communicants**

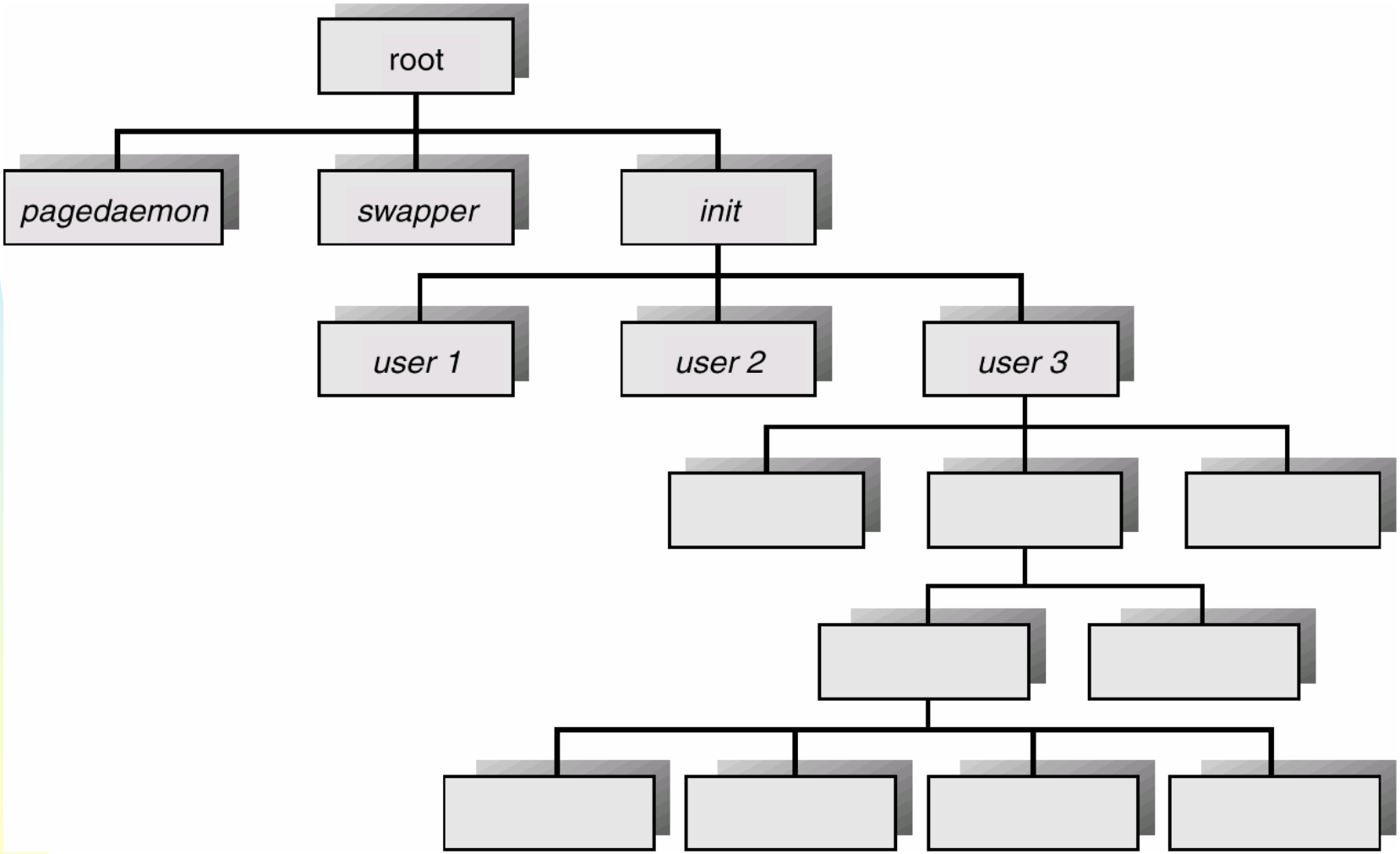
Création de processus

- **Les processus peuvent créer d'autres processus, formant une hiérarchie (instruction** fork **ou semblables)**

Terminaison de processus

- **Un processus exécute sa dernière instruction**
  - pourrait passer des données à son parent
  - ses ressources lui sont enlevées
- **Le parent termine l'exécution d'un fils (avortement) pour raisons différentes**
  - le fils a excédé ses ressources
  - le fils n`est plus requis
- **etc.**

Arbre de processus en UNIX

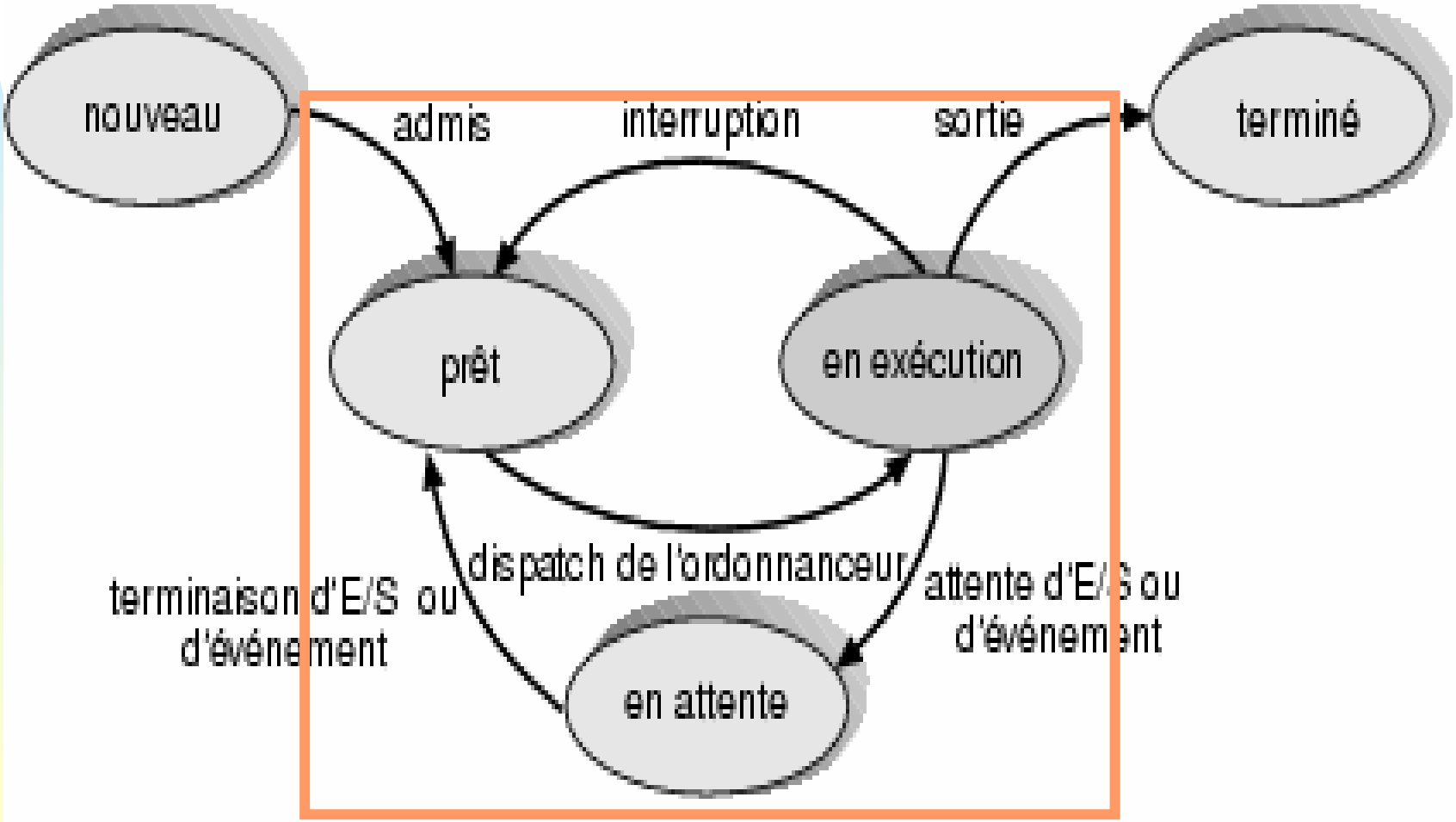


État de processus

IMPORTANT

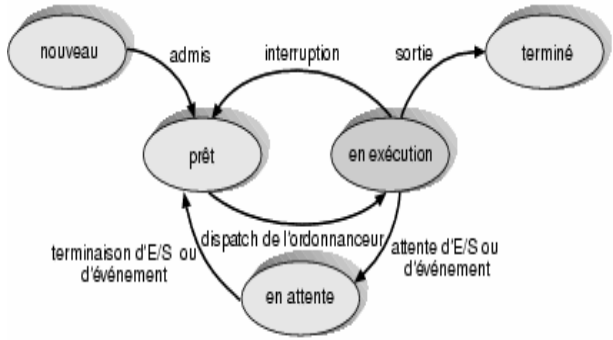
- **Au fur et a mesure qu'un processus exécute, il change d'état**
  - nouveau: le processus vient d 'être créé
  - exécutant-running: le processus est en train d 'être exécuté par l 'UCT
  - attente-waiting: le processus est en train d 'attendre un événement (p.ex. la fin d 'une opération d 'E/S)
  - prêt-ready: le processus est en attente d'être exécuté par l 'UCT
  - terminated: fin d 'exécution

Diagramme de transition d`états d`un processus



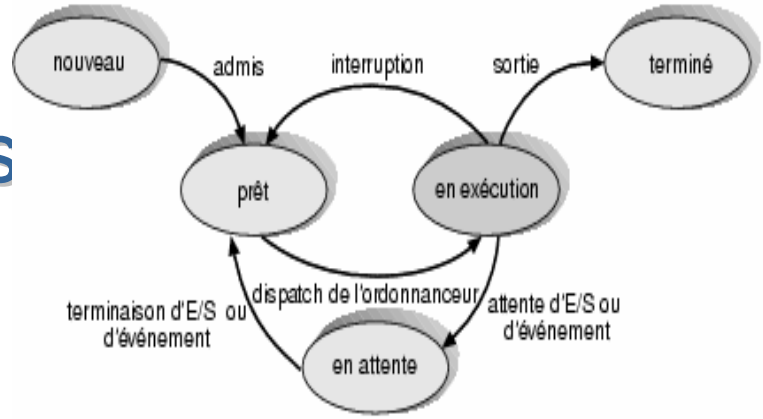
Ordonnanceur = angl. scheduler

États Nouveau, Terminé:



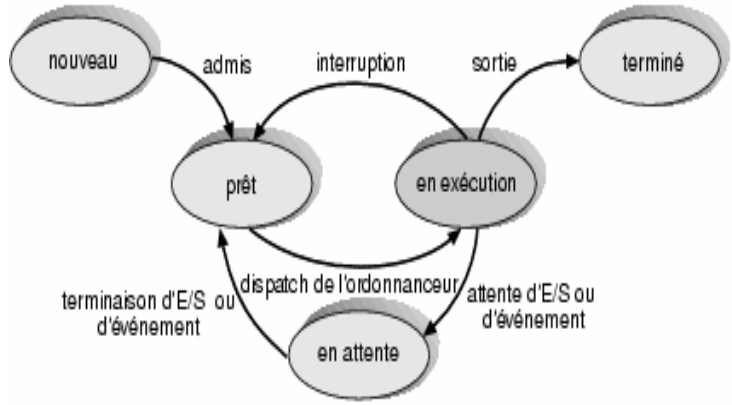
- **Nouveau**
  - Le SE a créé le processus
    - a construit un identificateur pour le processus
    - a construit les tableaux pour gérer le processus
  - mais ne s'est pas encore engagé à exécuter le processus (pas encore *admis*)
    - pas encore alloué des ressources
  - La file des nouveaux travaux est souvent appelée spoule travaux (job spooler)
- **Terminé:**
  - Le processus n 'est plus exécutable, mais ses données sont encore requises par le SE (comptabilité, etc.)

Transitions entre processus



- **Prêt → Exécution**
  - Lorsque l 'ordonnanceur UCT choisit un processus pour exécution
- **Exécution → Prêt**
  - Résultat d'une interruption causée par un événement indépendant du processus
    - Il faut traiter cette interruption, donc le processus courant perd l'UCT
      - Cas important: le processus à épuisé son intervalle de temps (minuterie)

Transitions entre processus



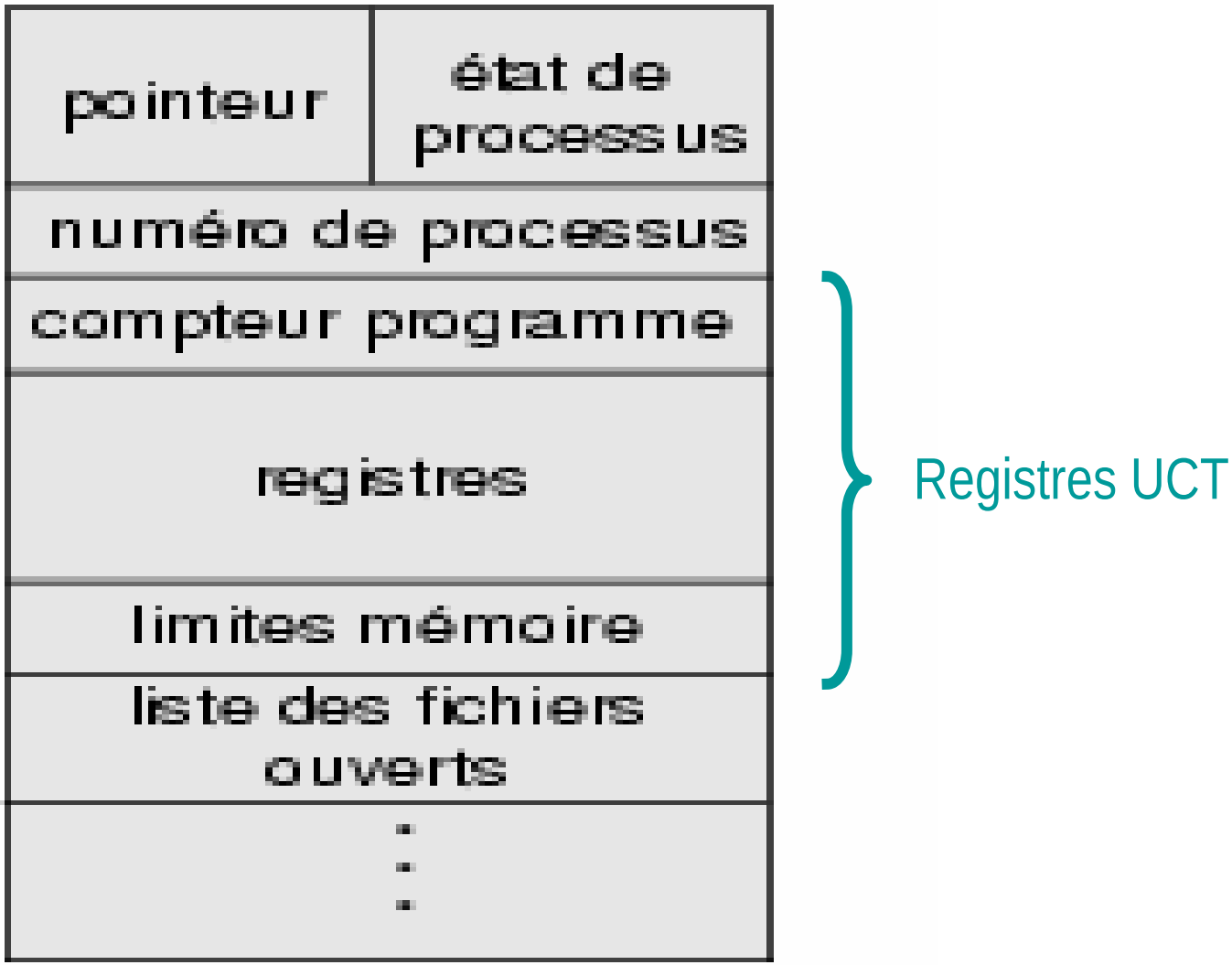
- **Exécution → Attente**
  - Lorsqu'un processus fait un appel de système (interruption causée par le processus lui-même)
    - initie une E/S: doit attendre le résultat
    - a besoin de la réponse d'un autre processus
- **Attente → Prêt**
  - lorsque l'événement attendu se produit

Sauvegarde d'informations processus

- **En multiprogrammation, un processus exécute sur l 'UCT de façon intermittente**
- **Chaque fois qu'un processus reprend l 'UCT (transition prêt → exécution) il doit la reprendre dans la même situation où il l'a laissée (même contenu de registres UCT, etc.)**
- **Donc au moment où un processus sort de l'état exécution il est nécessaire de sauvegarder ses informations essentielles, qu'il faudra récupérer quand il retourne à cet état**

PCB = Process Control Block:

Représente la situation actuelle d ’un processus, pour le reprendre plus tard



Process Control Block (PCB)

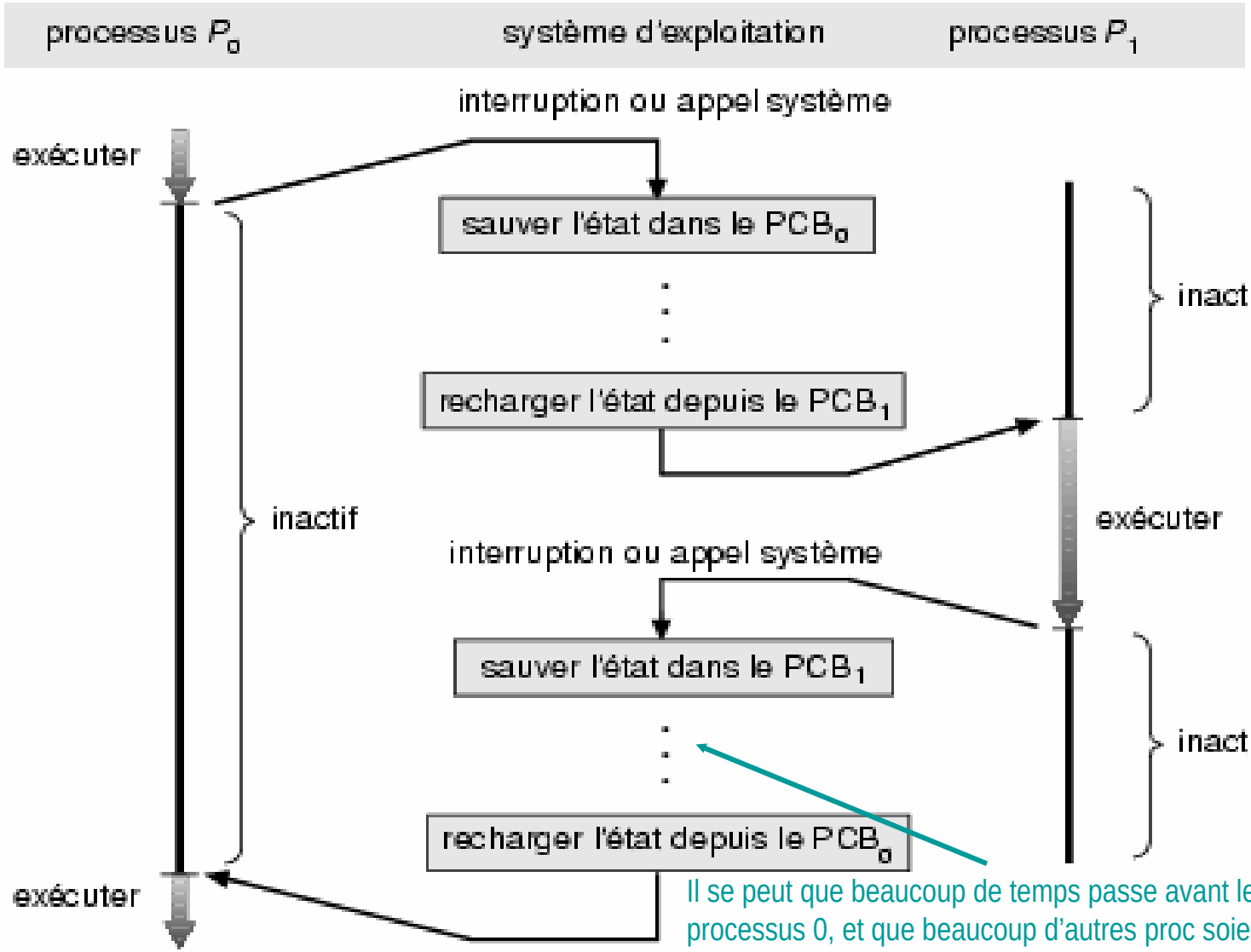
IMPORTANT

- pointeur: les PCBs sont rangés dans des listes enchaînées (à voir)
- état de processus: ready, running, waiting…
- compteur programme: le processus doit reprendre à l 'instruction suivante
- autres registres UCT
- bornes de mémoire
- fichiers qu'il a ouvert
- etc., v. manuel

Commutation de processeur
Aussi appélé commutation de contexte ou context switching

- **Quand l'UCT passe de l'exécution d 'un processus 0 à l 'exécution d`un proc 1, il faut**
  - mettre à jour et sauvegarder le PCB de **0**
  - reprendre le PCB de **1**, qui avait été sauvegardé avant
  - remettre les registres d 'UCT tels que le compteur d 'instructions etc. dans la même situation qui est décrite dans le PCB de **1**

Commutation de processeur (context switching)



Il se peut que beaucoup de temps passe avant le retour au processus 0, et que beaucoup d'autres proc soient exécutés entre temps

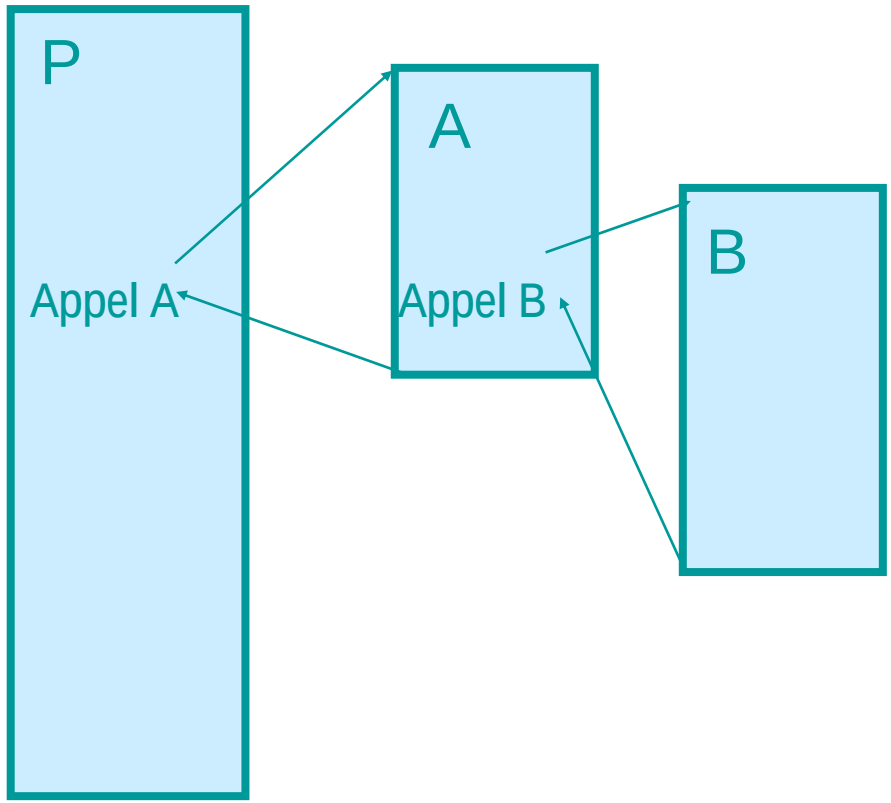
Le PCB n ’est pas la seule information à sauvegarder... (le manuel n’est pas clair ici)

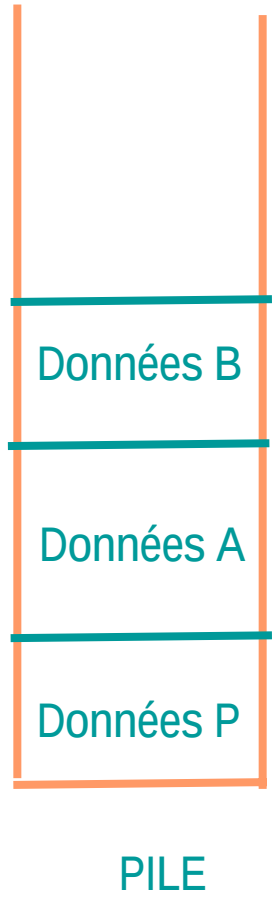
- **Il faut aussi sauvegarder l ’état des données du programme**
- **Ceci se fait normalement en gardant l *’image du programme* en mémoire primaire ou secondaire (RAM ou disque)**
- **Le PCB pointera à cette image**

La pile d'un processus (v. Stallings App. 1B)
aussi à sauvegarder

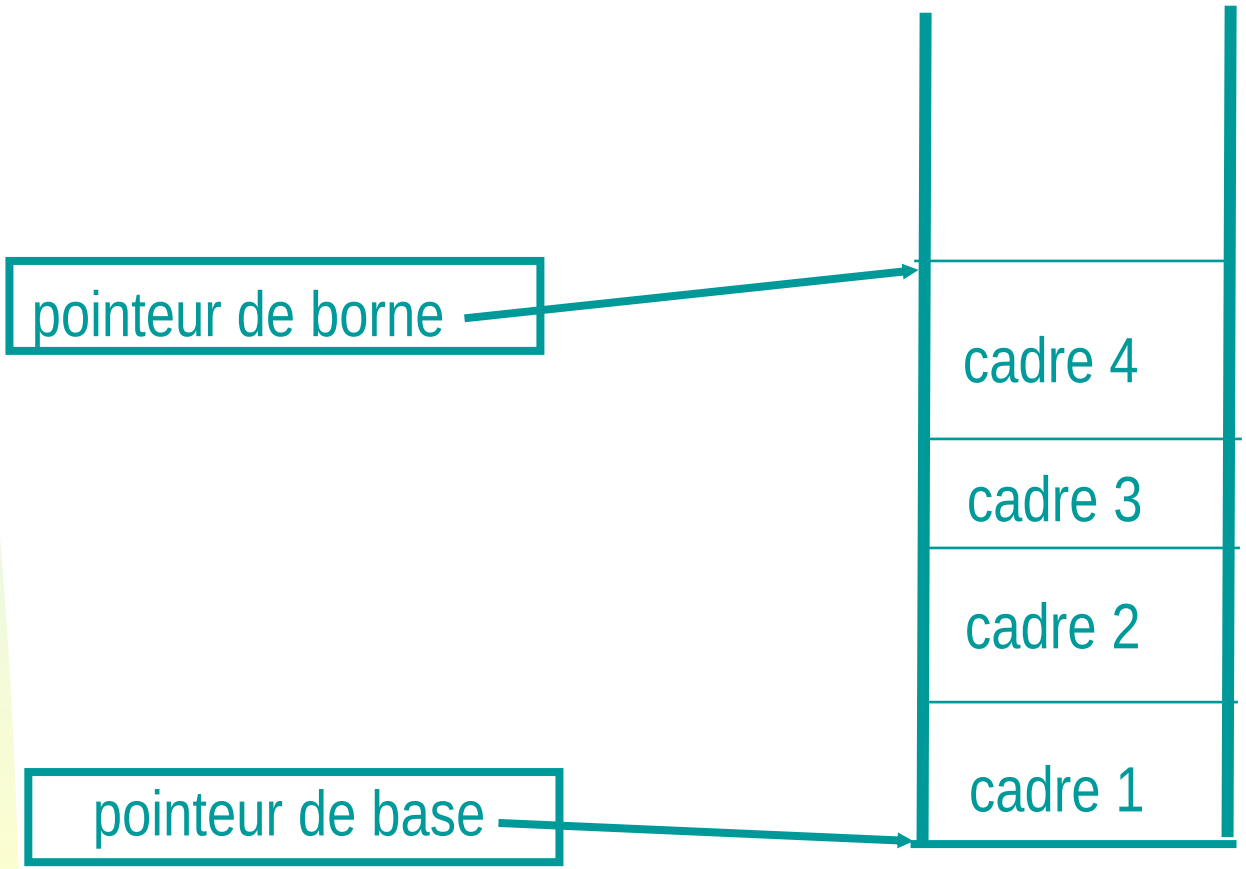
- **Quand un processus fait appel à une procédure, à une méthode, etc., il est nécessaire de mettre dans une pile l'adresse à laquelle le processus doit retourner après avoir terminé cette procédure, méthode, etc.**
- **Aussi on met dans cette pile les variables locales de la procédure qu'on quitte, les paramètres, etc., pour les retrouver au retour**
- **Chaque élément de cette pile est appelé stack frame ou cadre de pile**
- **Donc il y a normalement une pile d'adresses de retour après interruption et une pile d'adresses de retour après appel de procédure**
  - Ces deux piles fonctionnent de façon semblable, mais sont indépendantes
- **Les informations relatives à ces piles (base, pointeur…) doivent aussi être sauvegardées au moment de la commutation de contexte**

La Pile d'un processus



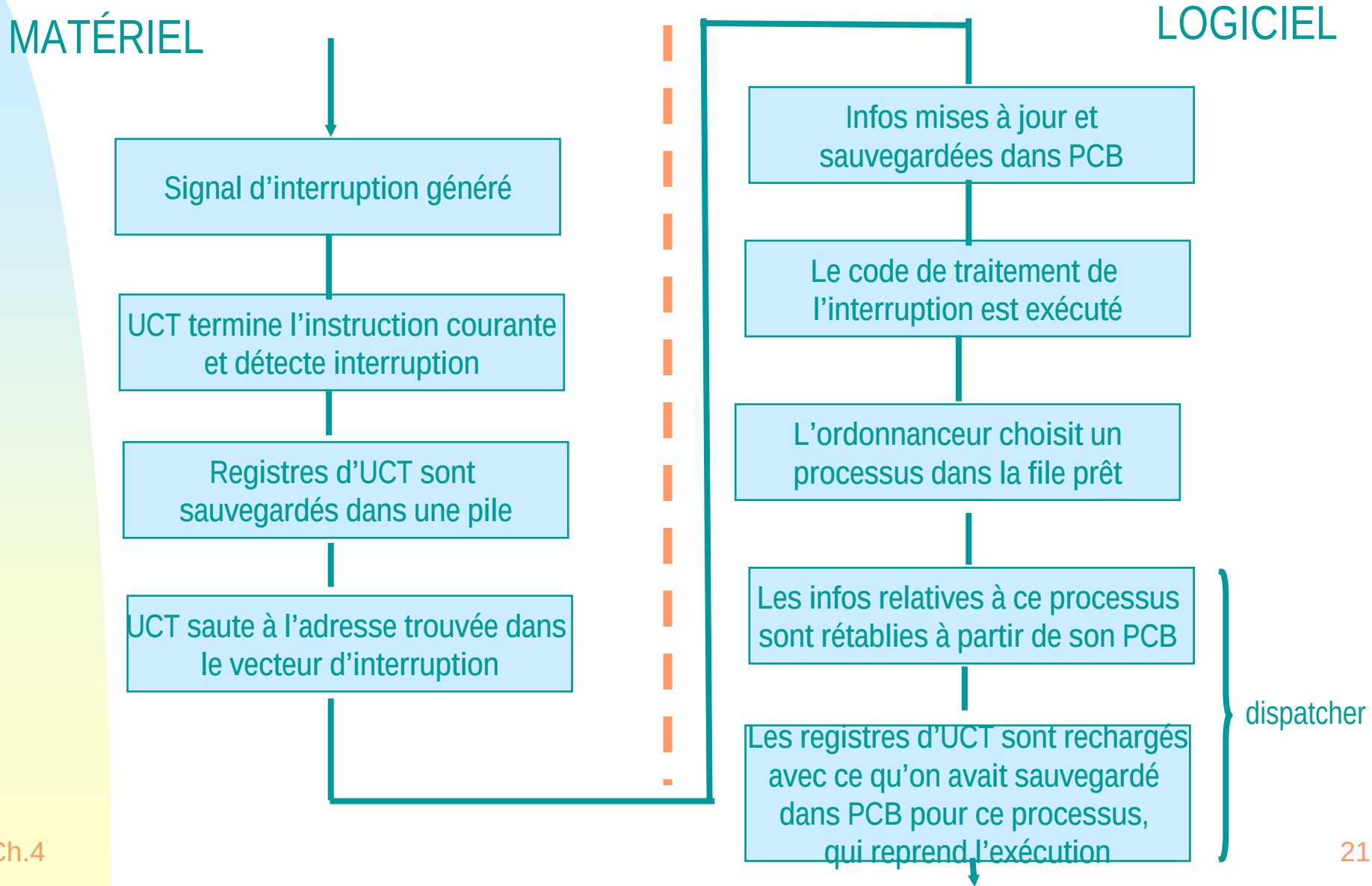


Pointeurs de pile processus à sauvegarder: base et borne



La pile fait normal. partie de l'image du programme, mais les pointeurs sont normal. des registres d'UCT donc il sont sauvegardés dans le PCB

Rôle du matériel et du logiciel dans le traitement d'interruptions



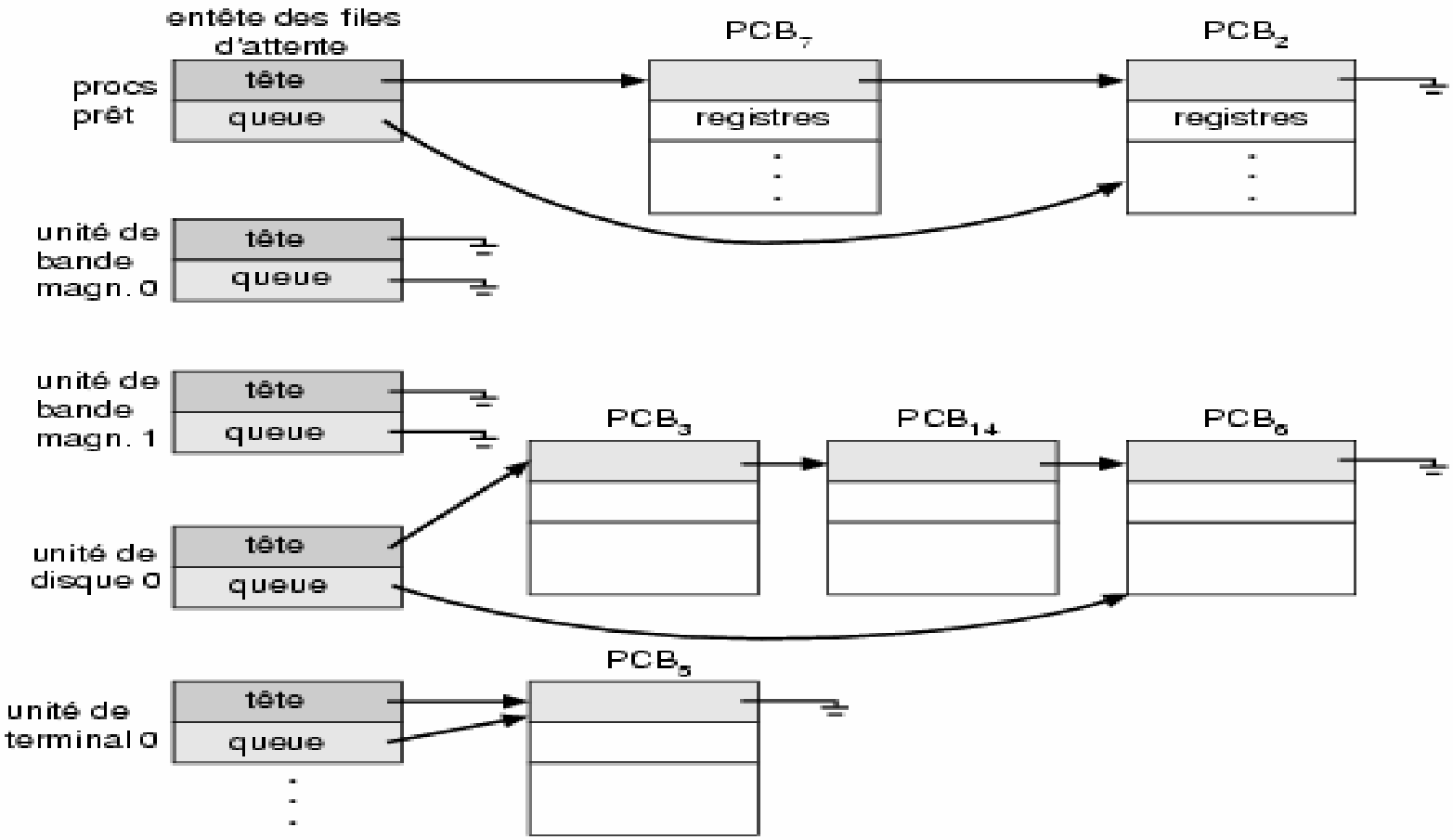
Files d'attente

IMPORTANT

- **Les ressources d 'ordinateur sont souvent limitées par rapport aux processus qui en demandent**
- **Chaque ressource a sa propre file de processus en attente**
- **À un moment donné, un proc ne peut se trouver que dans *une seule* des différentes files du SE**
- **En changeant d'état, les processus se déplacent d 'une file à l`autre**
  - File prêt: les processus en état prêt=ready
  - Files associés à chaque unité E/S
  - etc.

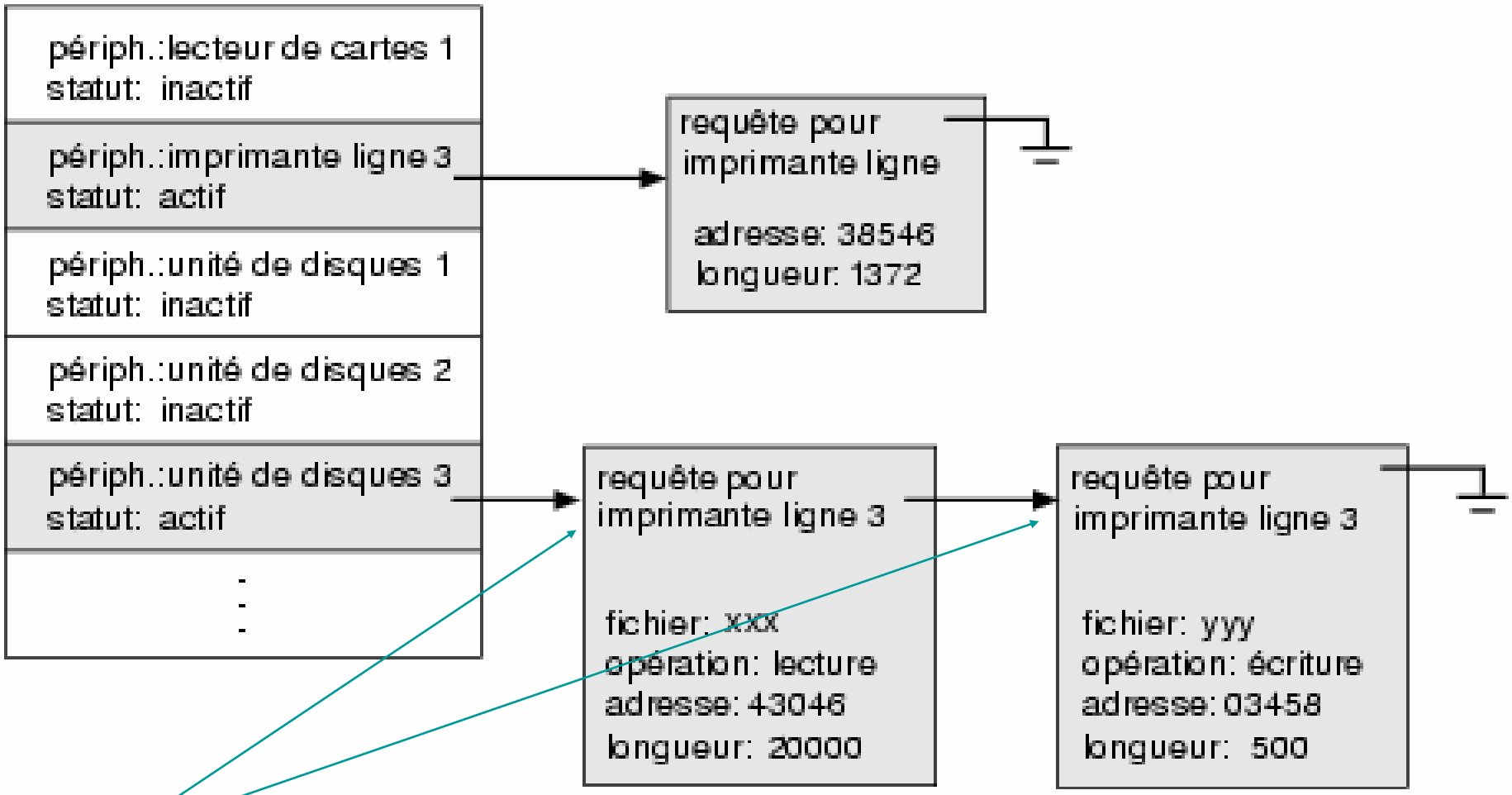
Ce sont les PCBs qui sont dans les files d'attente (dont le besoin d 'un *pointeur* dans le PCB)

file prêt



Nous ferons l'hypothèse que le premier processus dans une file est celui qui utilise la ressource: ici, proc7 exécute, proc3 utilise disque 0, etc.

Cet ensemble de files inclut donc la table de statut périphériques (fig. au Chap. 2)



2 fois la même erreur ici: imprimante devrait être disque 3

Une façon plus synthétique de décrire la même situation (pour les devoirs et les examens)

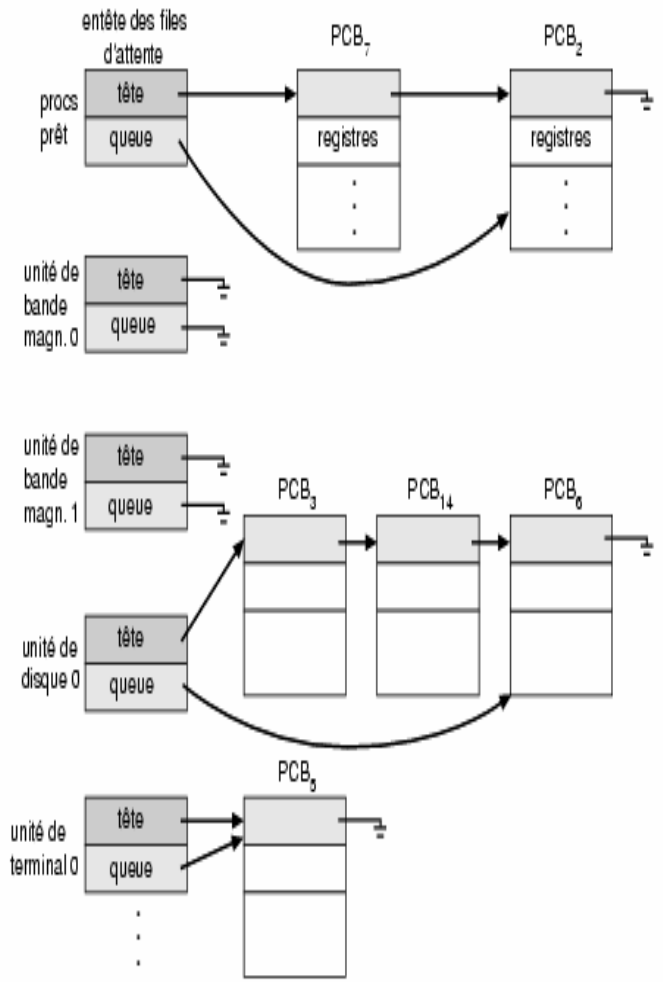
prêt → 7 → 2

bandmag0 →

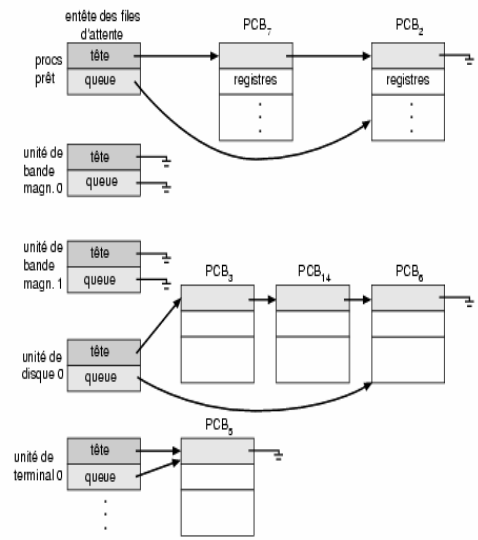
bandmag1 →

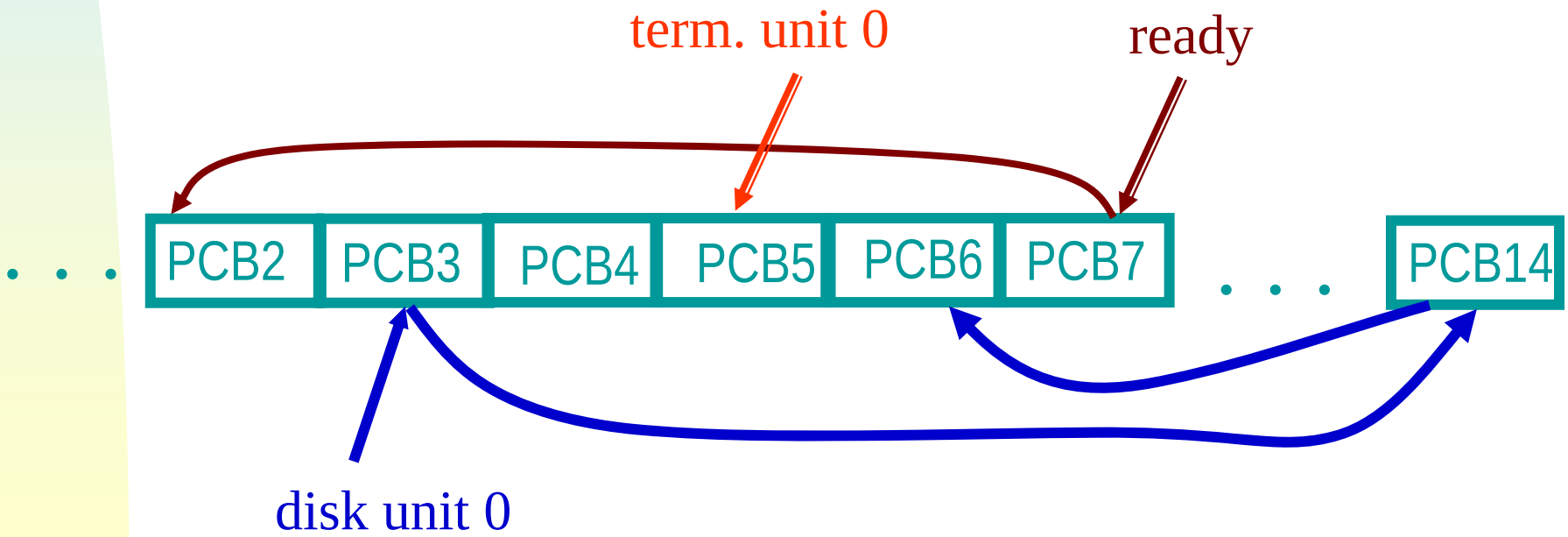
disq0 → 3 → 14 → 6

term0 → 5



Les PCBs ne sont pas déplacés en mémoire pour être mis dans les différentes files: ce sont les pointeurs qui changent.

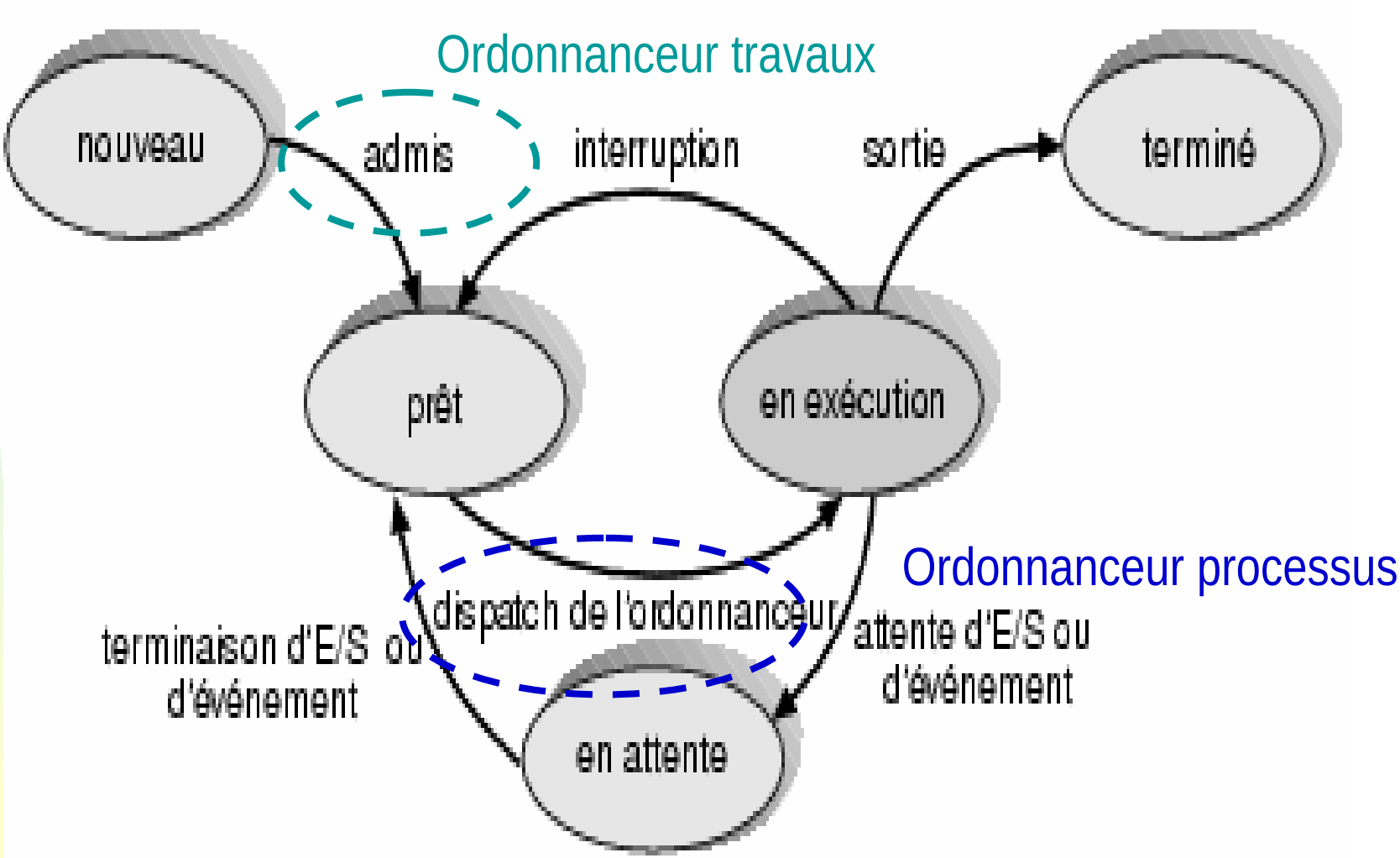




Ordonnanceurs (schedulers)

- **Programmes qui gèrent l 'utilisation de ressources de l`ordinateur**
- **Trois types d`ordonnanceurs :**
  - À court terme = ordonnanceur processus: sélectionne quel processus doit exécuter la transition prêt $\rightarrow$ exécution
  - À long terme = ordonnanceur travaux: sélectionne quels processus peuvent exécuter la transition nouveau $\rightarrow$ prêt (événement *admitted*) (de spoule travaux à file prêt)
  - À moyen terme: nous verrons

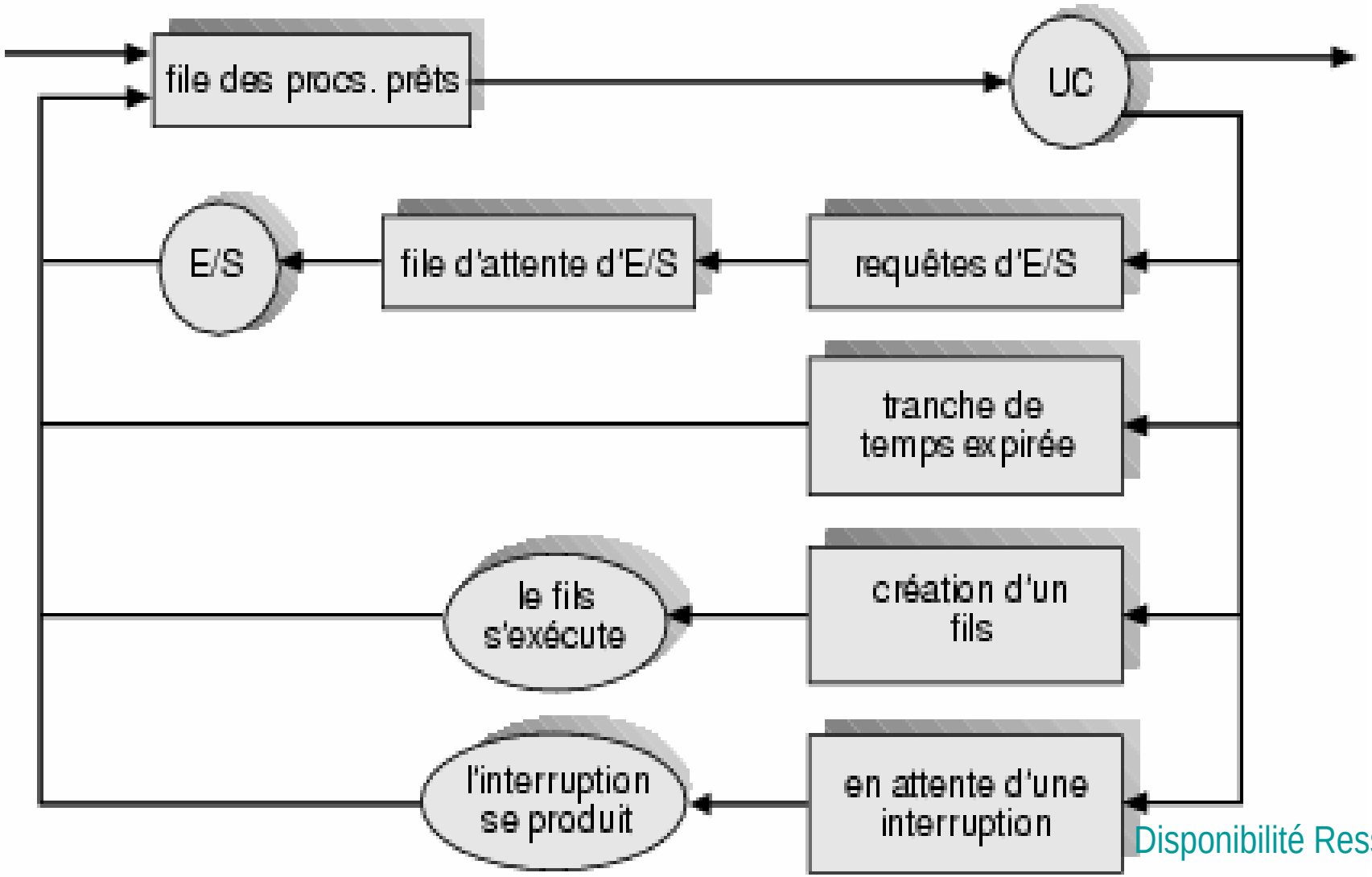
Ordonnanceur travaux = long terme et ordonnanceur processus = court terme



Ordonnanceurs

- **L`ordonnanceur à court terme est exécuté très souvent (millisecondes)**
  - doit être très efficace
- **L`ordonnanceur à long terme doit être exécuté beaucoup plus rarement: il contrôle le niveau de multiprogrammation**
  - Un des ses critères pourrait être la bonne utilisation des ressources de l'ordinateur
  - P.ex. établir une balance entre travaux liés à l'UCT et ceux liés à l 'E/S

Ordonnancement de processus (court terme)

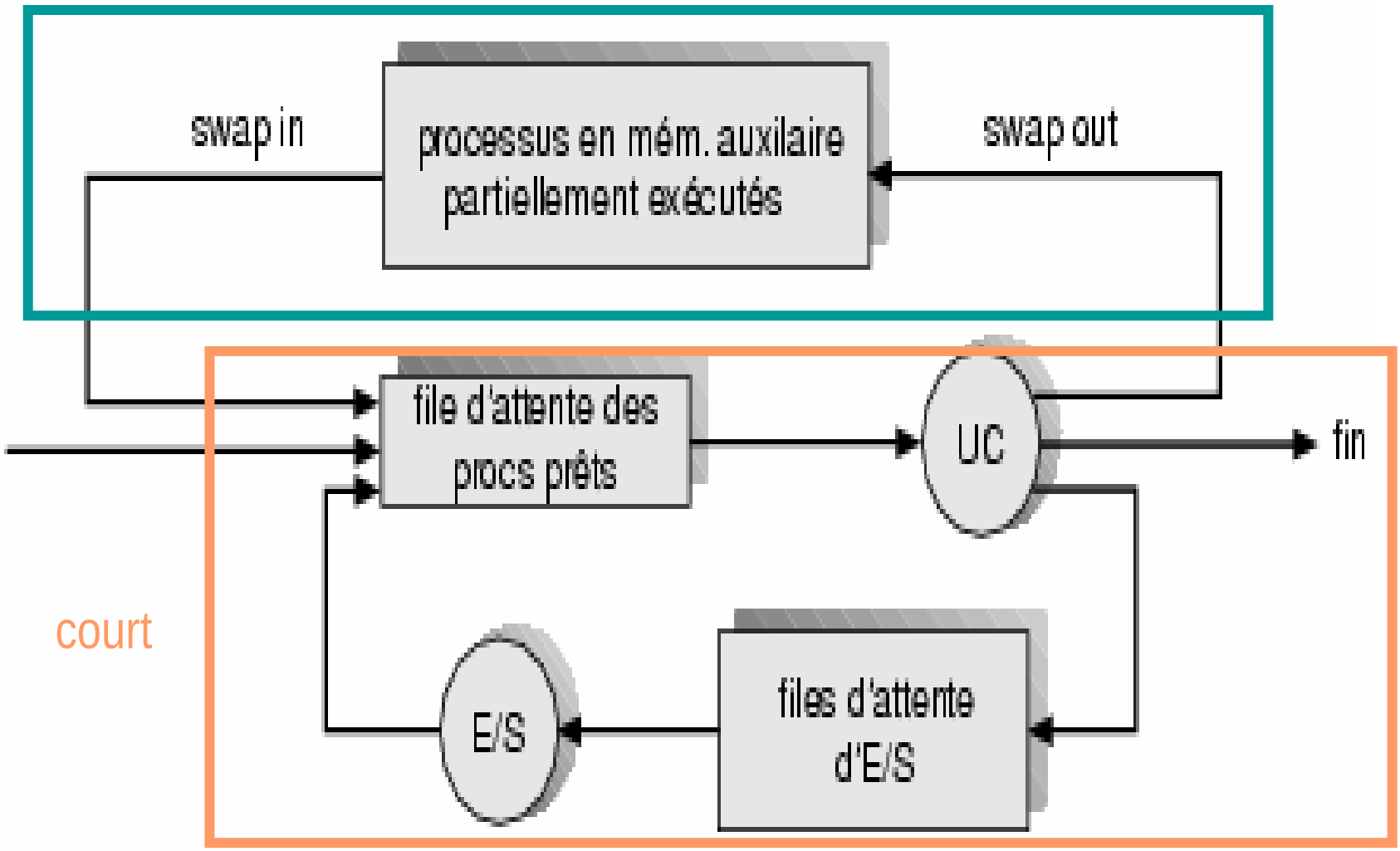


Disponibilité Ress.

Ordonnanceur à moyen terme

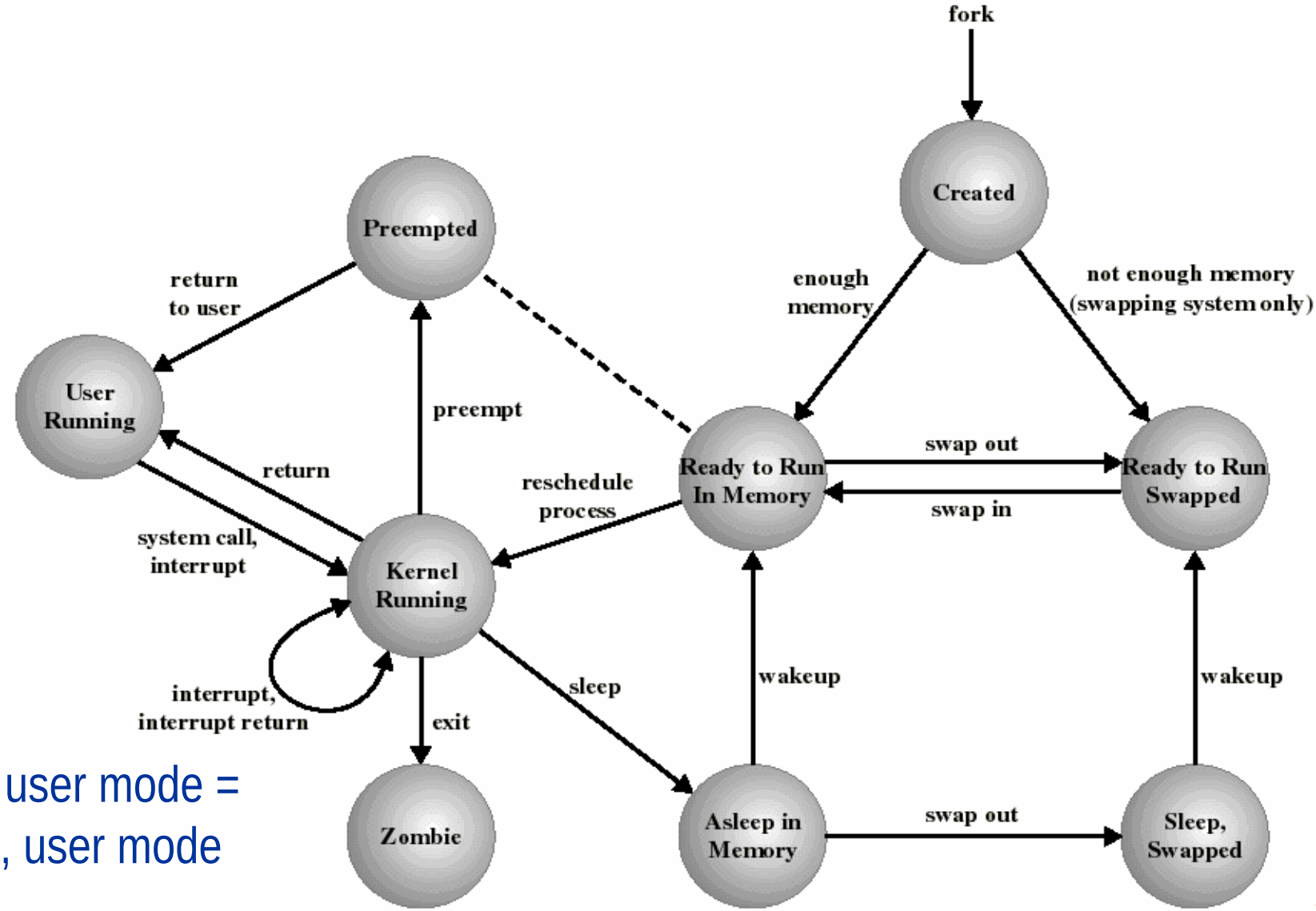
- **Le manque de ressources peut parfois forcer le SE à *suspendre* des processus**
  - ils seront plus en concurrence avec les autres pour des ressources
  - ils seront repris plus tard quand les ressources deviendront disponibles
- **Ces processus sont enlevés de mémoire centrale et mis en mémoire secondaire, pour être repris plus tard**
  - `swap out`, `swap in` , va-et-vien

Ordonnanceurs à court et moyen terme



États de processus dans UNIX SVR4 (Stallings)

Un exemple de diagramme de transitions d'états pour un SE réel



Kernel, user mode = monitor, user mode

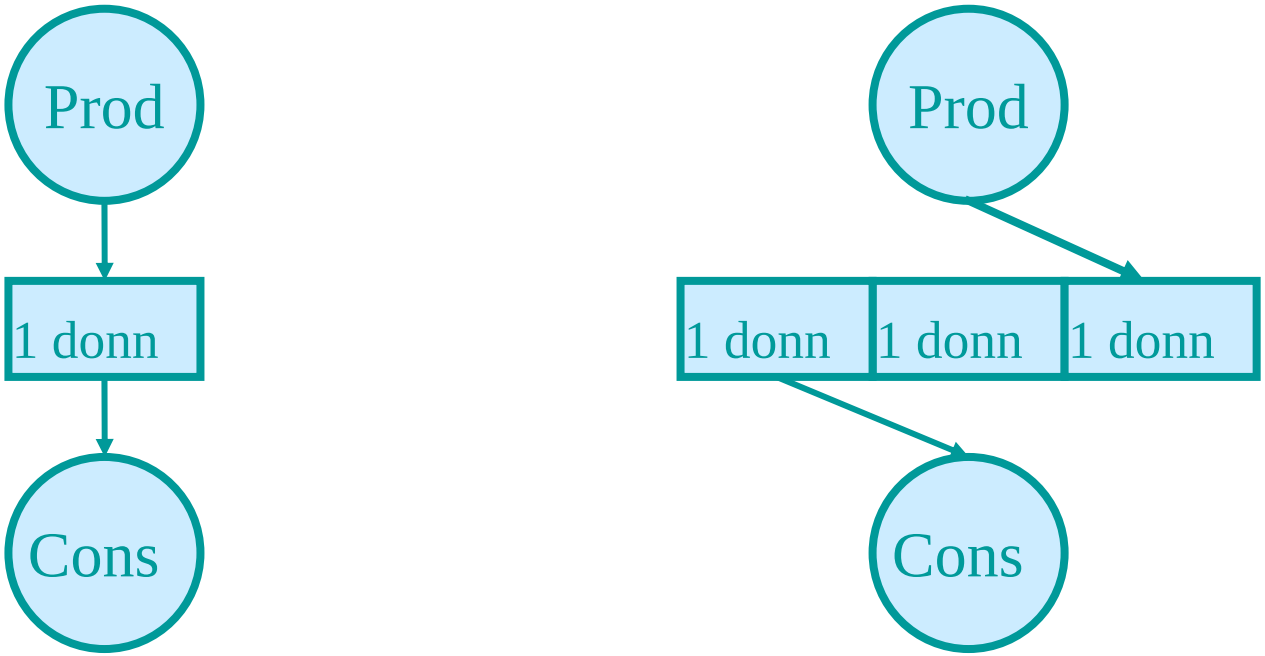
Processus coopérants

- **Les processus coopérants peuvent affecter mutuellement leur exécution**
- **Avantages de la coopération entre processus:**
  - partage de l ’information
  - efficacité en faisant des tâches en parallèle
  - modularité
  - la nature du problème pourrait le demander
    - P.ex. gestion d’événements indépendants
      - Un proc traite le clavier, un autre traite le modem

Le pb du producteur - consommateur

- **Un problème classique dans l ’étude des processus communicants**
  - un processus *producteur* produit des données (p.ex.des enregistrements d ’un fichier) pour un processus *consommateur*
  - un pgm d’impression produit des caractères -- consommés par une imprimante
  - un assembleur produit des modules objet qui seront consommés par le chargeur
- **Nécessité d’un tampon pour stocker les items produits (attendant d’être consommés**

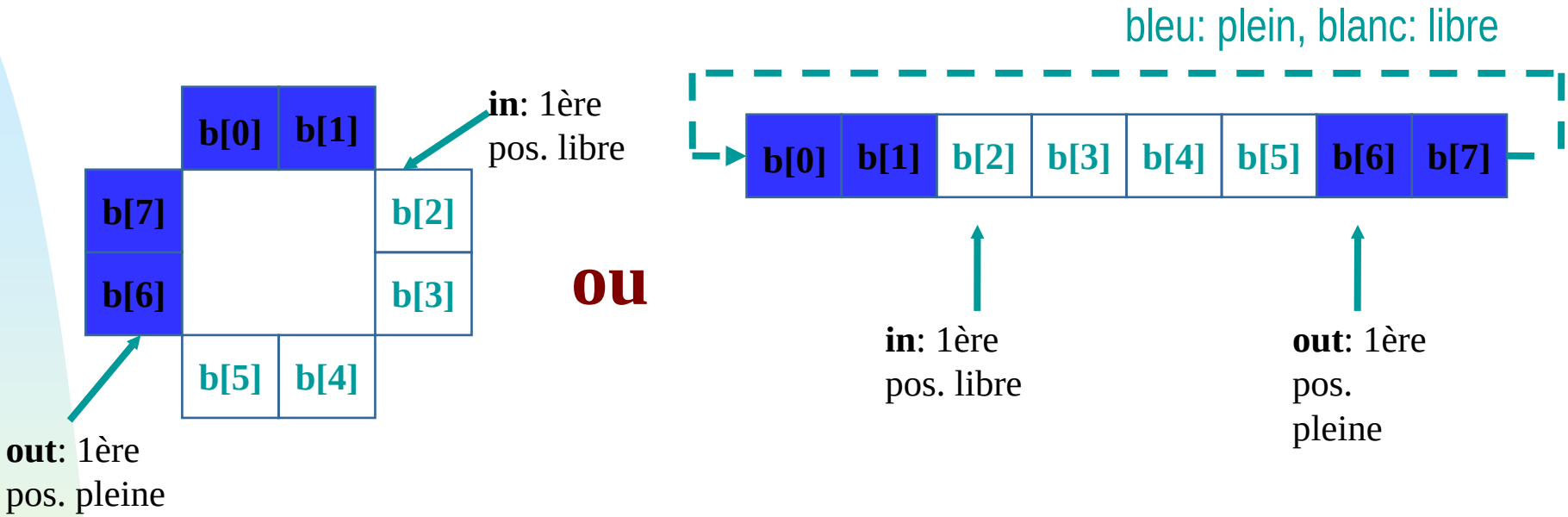
Tampons de communication



Si le tampon est de longueur 1, le producteur et consommateur doivent forcement aller à la même vitesse

Des tampons de longueur plus grandes permettent une certaine indépendance. P.ex. à droite le consommateur a été plus lent

Le tampon borné (bounded buffer)
une structure de données fondamentale dans les SE



Le tampon borné se trouve dans la mémoire partagée entre consommateur et usager

À l'écriture d'une nouvelle info dans le tampon, le producteur met à jour le pointeur in

Si le tampon est plein, le prod devra s'endormir, il sera plus tard réveillé par le consommateur

Le rôle du consommateur est symétrique (v. Chap. 7)

Utilisation du concept du tampon borné

- **Les tampons bornés sont partout en informatique, et partout dans les SE**
- **Les files utilisées dans un SE sont des tampons bornés:**
  - ‘pipes’ dans Unix
  - files d’attente pour ressources: file prêt, files pour imprimante, pour disque, etc.
- **Les protocoles de communications utilisent des tampons bornés: TCP, et autres**
- **Un client communique avec un serveur par des tampons bornés, etc.**

Concepts importants du Chapitre 4

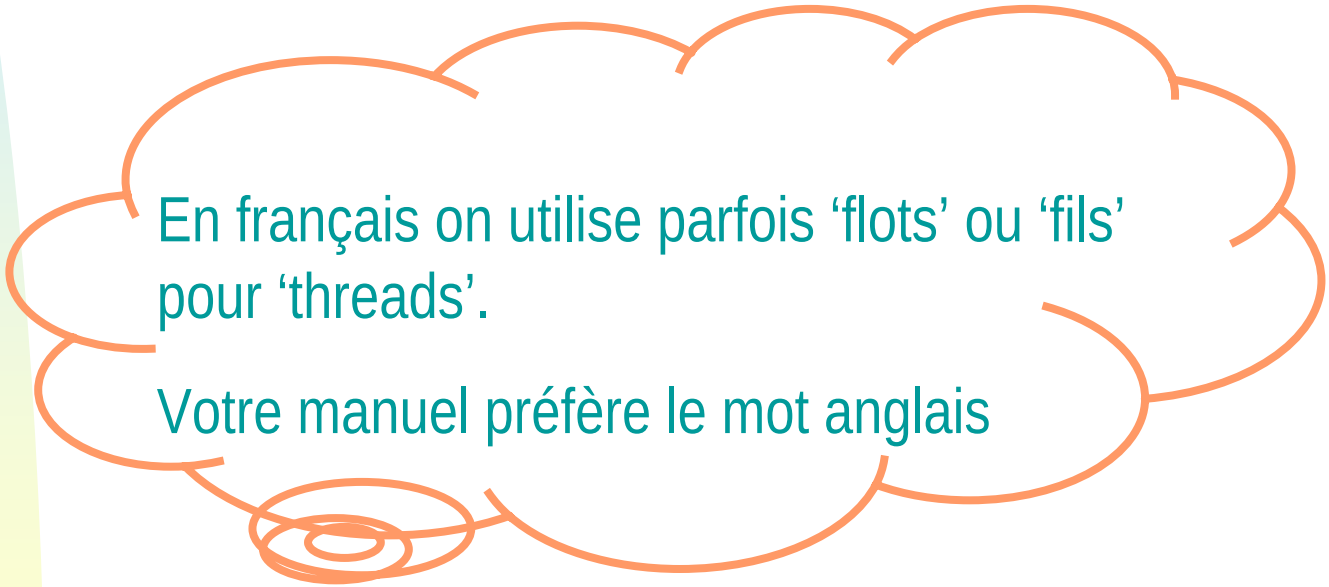
- **Processus**
  - Création, terminaison, hiérarchie
- **États et transitions d'état des processus**
- **Process Control Block PCB**
- **Commutation de processus**
  - Sauvegarde, rechargement de PCB
- **Files d'attente de processus et PCB**
- **Ordonnanceurs à court, moyen, long terme**
- **Processus communicants**
  - Producteurs et consommateurs

Par rapport au manuel…

- **Tout à étudier à l'exception des sections 4.5.1, 4.5.2, 4.5.3, 4.5.6 et 4.5.7**
- **Les exemples contenant du code Java et C seront expliqués aux sessions d'exercices**

Threads et Lightweight Processes

Chapitre 5



Concepts importants du Chap. 5

- **Threads et processus: différence**
- **Threads de noyau et d'usager: relations**
- **LWP: lightweight processes, threads légers**

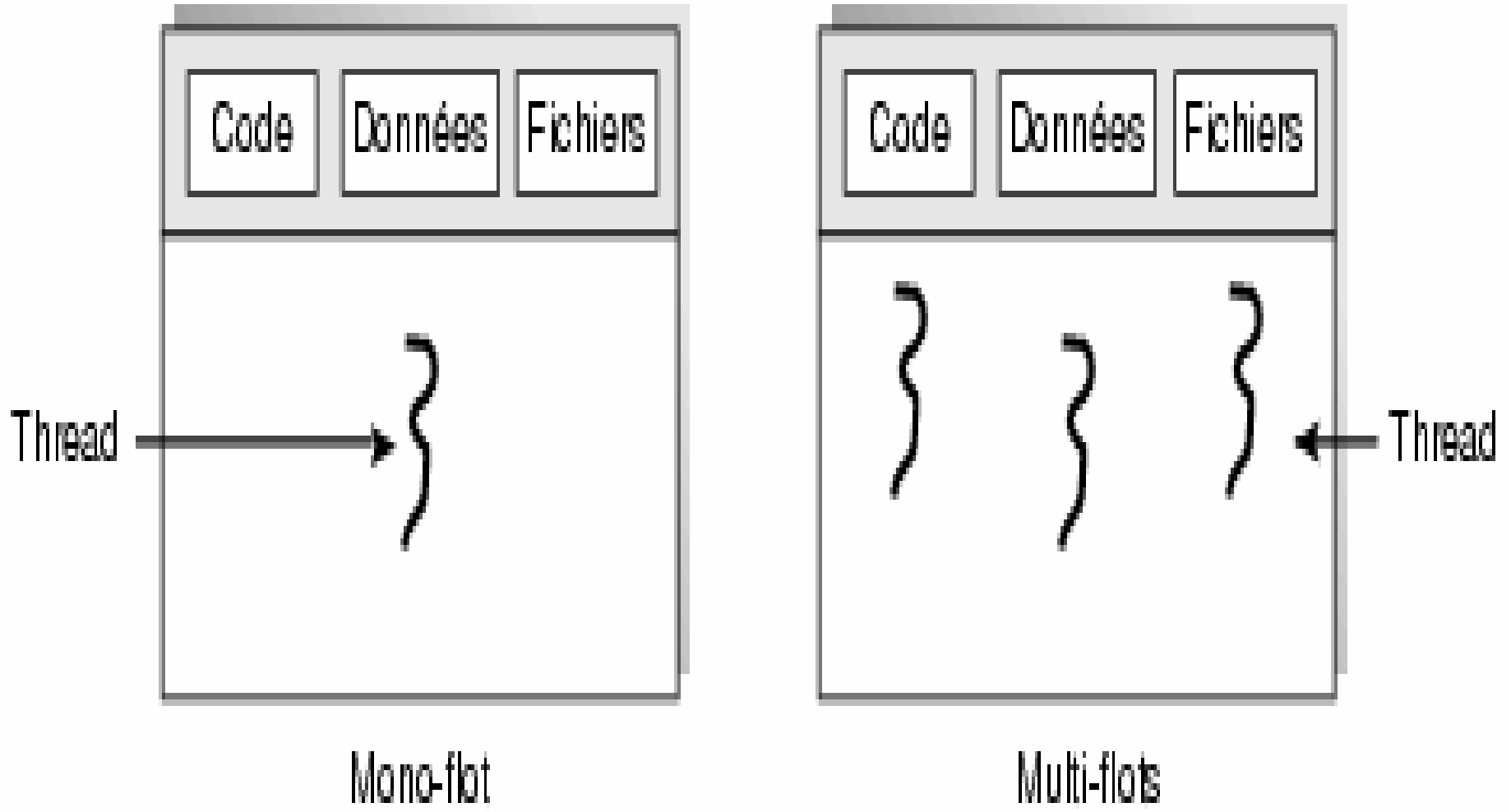
Flots = threads = lightweight processes

- **Un thread est une subdivision d`un processus**
  - Un fil de contrôle dans un processus
- **Les différents threads d ’un processus partagent l’espace adressable et les ressources d’un processus**
  - lorsqu’un thread modifie une variable (non locale), tous les autres threads voient la modification
  - un fichier ouvert par un thread est accessible aux autres threads (du même processus)

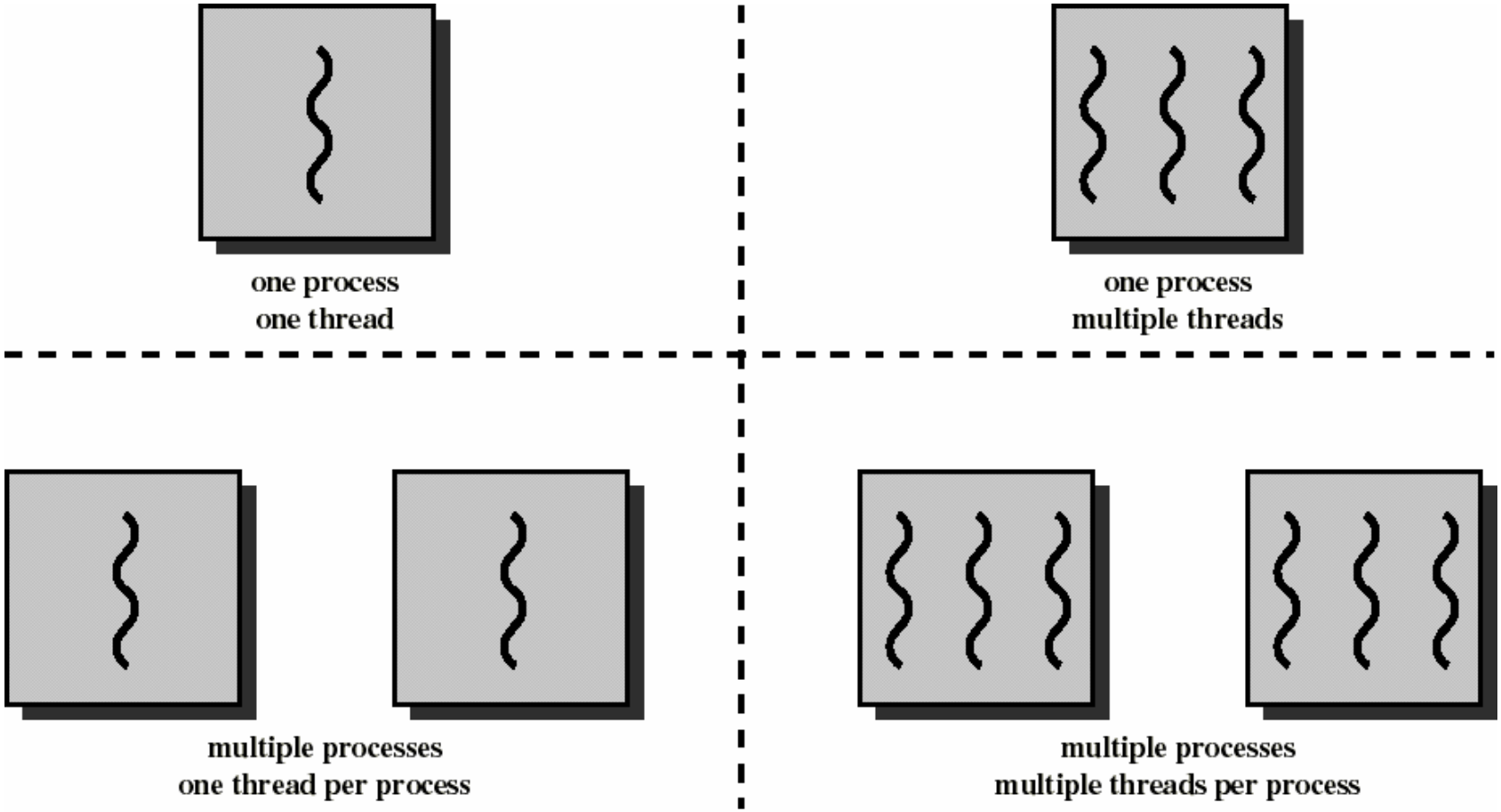
Exemple

- **Le processus MS-Word implique plusieurs threads:**
  - Interaction avec le clavier
  - Rangement de caractères sur la page
  - Sauvegarde régulière du travail fait
  - Contrôle orthographe
  - Etc.
- **Ces threads partagent tous le même document**

Processus à un thread et à plusieurs threads



Threads et processus [Stallings]



Processus

- **Possède sa mémoire, ses fichiers, ses ressources, etc.**
- **Accès protégé à la mémoire, fichiers, ressources d'autres processus**

Thread

- **Possède un état d'exécution (prêt, bloqué…)**
- **Possède sa pile et un espace privé pour variables locales**
- **A accès à l'espace adressable, fichiers et ressources du processus auquel il appartient**
  - En commun avec les autres threads du même proc

Pourquoi les threads

- **Reactivité: un processus peut être subdivisé en plusieurs threads, p.ex. l'un dédié à l'interaction avec les usagers, l'autre dédié à traiter des données**
  - L'un peut exécuter tant que l'autre est bloqué
- **Utilisation de multiprocesseurs: les threads peuvent exécuter en parallèle sur des UCT différentes**

La commutation entre threads est moins dispendieuse que la commutation entre processus

- **Un processus possède mémoire, fichiers, autres ressources**
- **Changer d`un processus à un autre implique sauvegarder et rétablir l'état de tout ça**
- **Changer d'un thread à un autre *dans le même proc* est bien plus simple, implique sauvegarder les registres de l 'UCT, la pile, et peu d 'autres choses**

La communication aussi est moins dispendieuse entre threads que entre processus

- **Étant donné que les threads partagent leur mémoire,**
  - la communication entre threads dans un même processus est plus efficace
  - que la communication entre processus

La création est moins dispendieuse

- **La création et terminaison de nouveaux threads dans un proc existant est aussi moins dispendieuse que la création d'un proc**

Threads de noyau (kernel) et d'utilisateur

- **Où implémenter les threads:**
  - Dans les bibliothèques d'usager
    - contrôlés par l'usager
  - Dans le noyau du SE:
    - contrôlés par le noyau
  - Solutions mixtes

Threads d'utilisateur et de noyau (kernel)

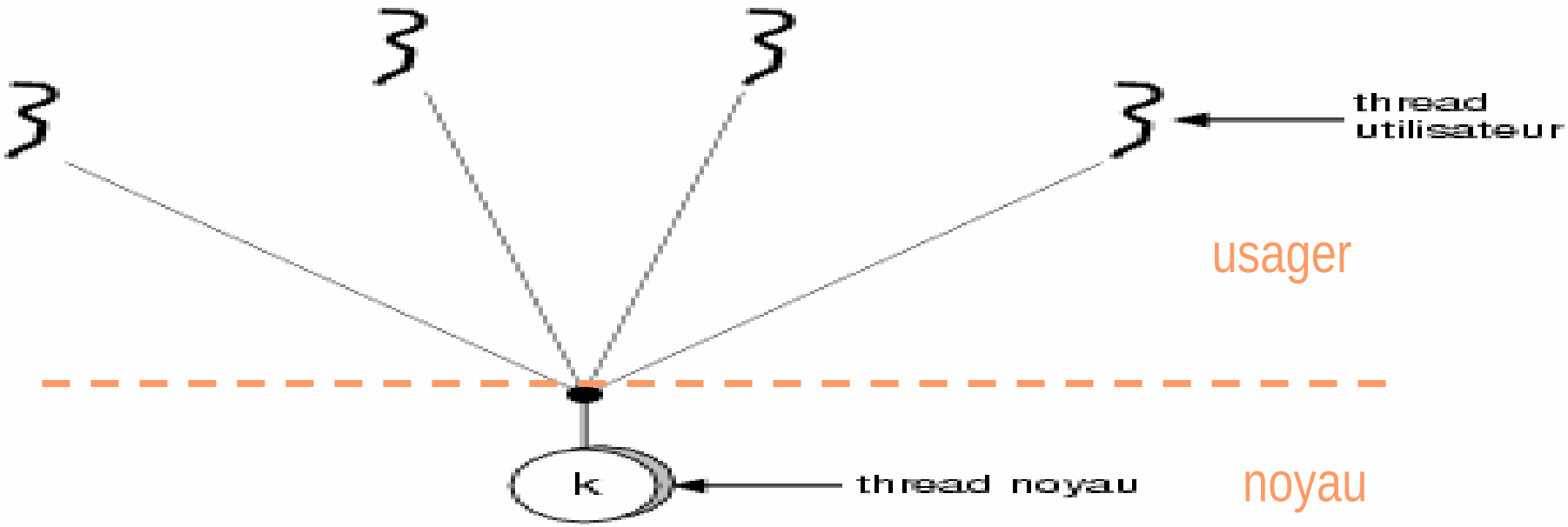
- **threads d'utilisateur: supportés par des bibliothèques d'usager ou langage de prog**
  - efficace car les ops sur les threads ne demandent pas des appels du système
  - désavantage: le noyau n 'est pas capable de distinguer entre état de processus et état des threads dans le processus
    - blocage d 'un thread implique blocage du processus
- **threads de noyau: supportés directement par le noyau du SE (WIN NT, Solaris)**
  - le noyau est capable de gérer directement les états des threads
  - Il peut affecter différents threads à différentes UCTs

Solutions mixtes: threads utilisateur et noyau

- **Relation entre threads utilisateur et threads noyau**
  - plusieurs à un
  - un à un
  - plusieurs à plusieurs

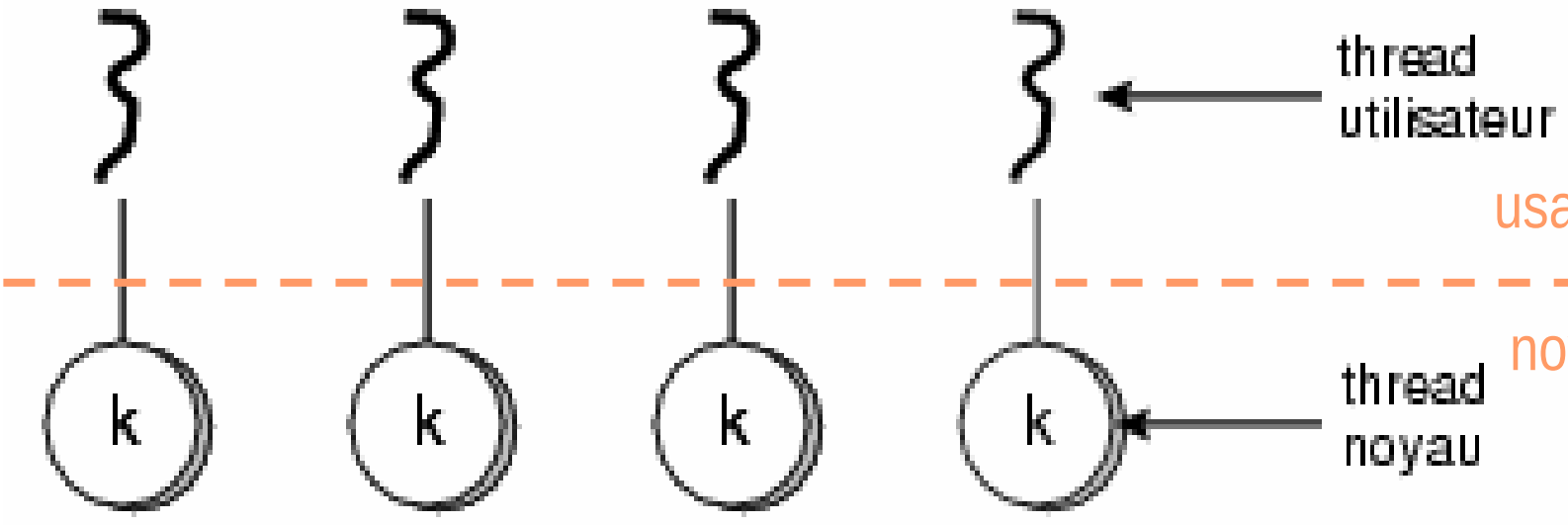
- **Nous devons prendre en considération plusieurs niveaux:**
  - Processus
  - Thread usager
  - Thread noyau
  - Processeur (UCT)

Plusieurs threads utilisateur pour un thread noyau: l'usager contrôle les threads



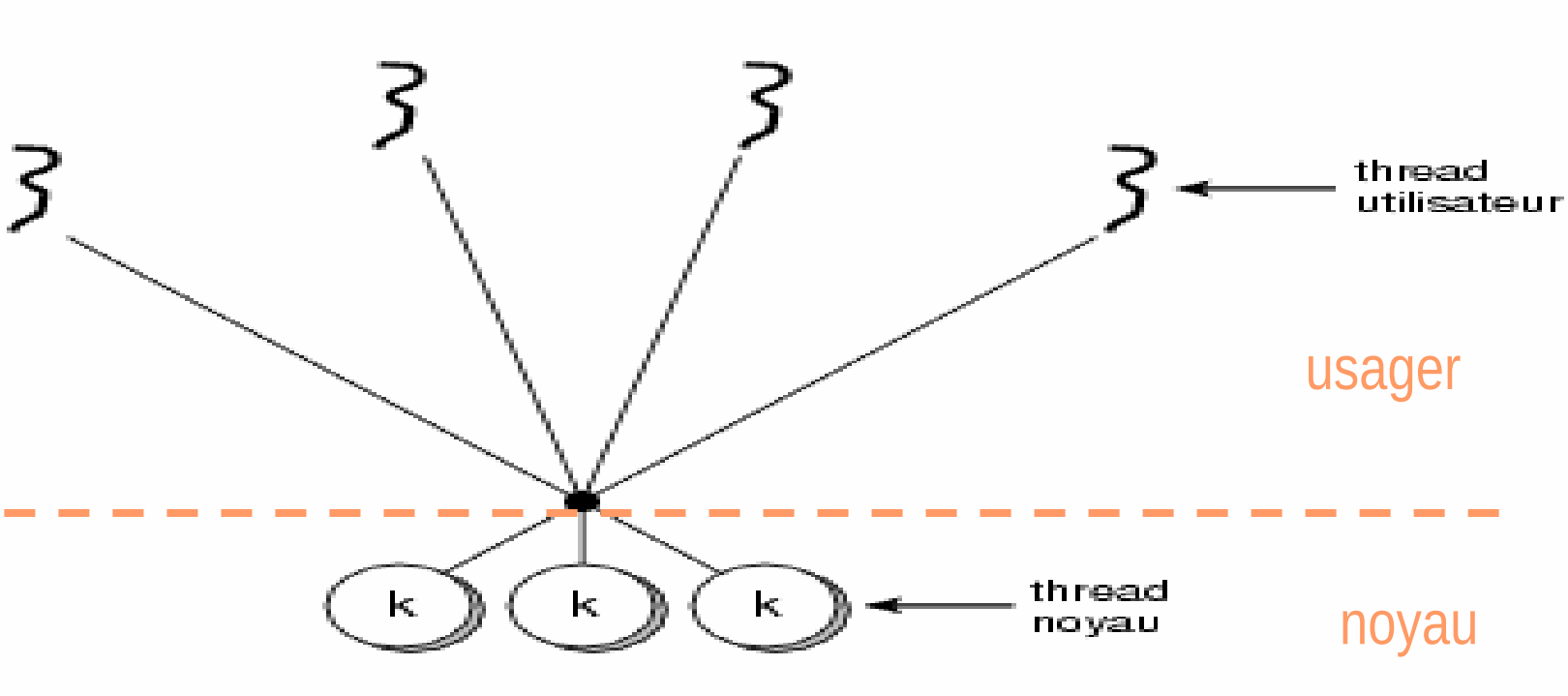
- **Le SE ne connaît pas les threads utilisateur**
  - v. avantages et désavantages mentionnés avant

Un vers un: le SE contrôle les threads



- **Les ops sur les threads sont des appels du système**
- **Permet à un autre thread d'exécuter lorsqu'un thread exécute un appel de système bloquant**
- **Win NT, XP, OS/2**

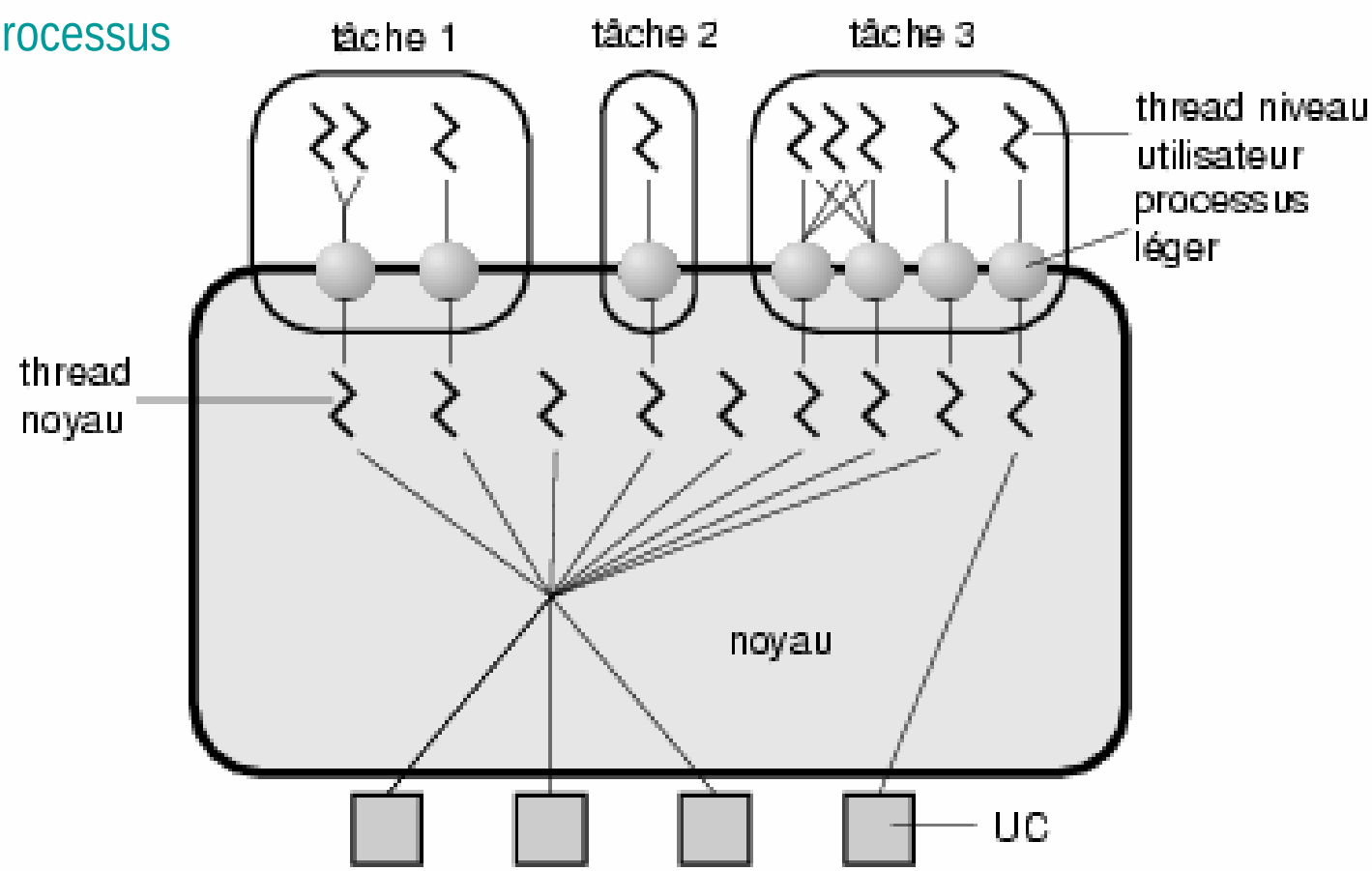
Plusieurs à plusieurs: solution mixte



- **Flexibilité pour l 'usager d 'utiliser la technique qu'il préfère**
- **Si un thread utilisateur bloque, son kernel thread peut être affecté à un autre**
- **Si plus. UCT sont disponibles, plus. kernel threads peuvent exécuter en même temps**
- **Quelques versions d'Unix, dont Solaris**

Threads dans Solaris 2 (une version de Unix)

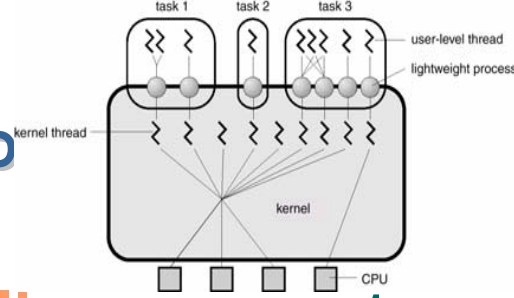
Tâche = processus



Plusieurs à plusieurs,

usager et noyau

Processus légers (lightweight, LWP



- **Fonctionnent comme des UCT virtuelles, pouvant exécuter des threads niveau usager**
- **Sont entre les threads usager et les threads noyau**
- **Il y a un thread noyau pour chaque LWP, chaque LWP est lié à son propre thread noyau**
- **Chaque processus doit contenir au moins un LWP**
- **La bibliothèque des threads exécute les threads utilisateur sur les LWP disponibles**
- **Seulement les threads qui sont associés à un LWP peuvent exécuter, les autres sont en attente d 'exécuter (bloqués)**
  - similarité avec ordonnancement UCT

Threads à niveau usager: liés et libres

- **Un thread à niveau usager est *lié* s'il est attaché à un LWP de façon permanente**
- **Est *non lié* sinon**

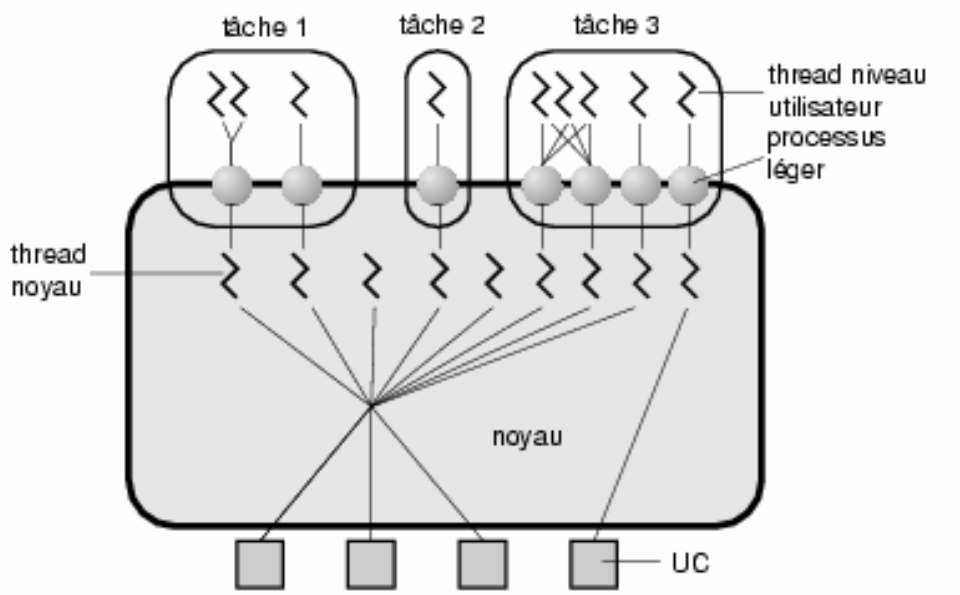
Utilisation des LWP

- **Les LWP sont demandés quand un thread usager a besoin de communiquer avec le noyau**
  - Appel système (E/S…)
  - car dans ce cas un nouveau thread peut exécuter tant que l'autre est bloqué en attente

- **Il y a besoin d 'un LWP pour chaque thread qui devient bloqué pour un appel de système**
  - p.ex. s 'il y a 5 threads qui demandent de l 'E/S, nous avons besoin de 5 LWP
  - s`il y a seul. 4 LWP, une des demandes d 'E/S doit attendre qu 'un LWP devienne libre ou soit créé

Exécution des LWP

- **Les threads de noyau qui implémentent les LWP exécutent sur les UCT qui deviennent disponibles**
- **Si un thread noyau se bloque, son LWP se bloque aussi,**
  - mais un processus (tâche) peut en obtenir un autre, ou un nouveau LWP peut être créé
- **Si un LWP devient bloqué, l'UCT qui l'exécute peut être affectée à un autre thread**

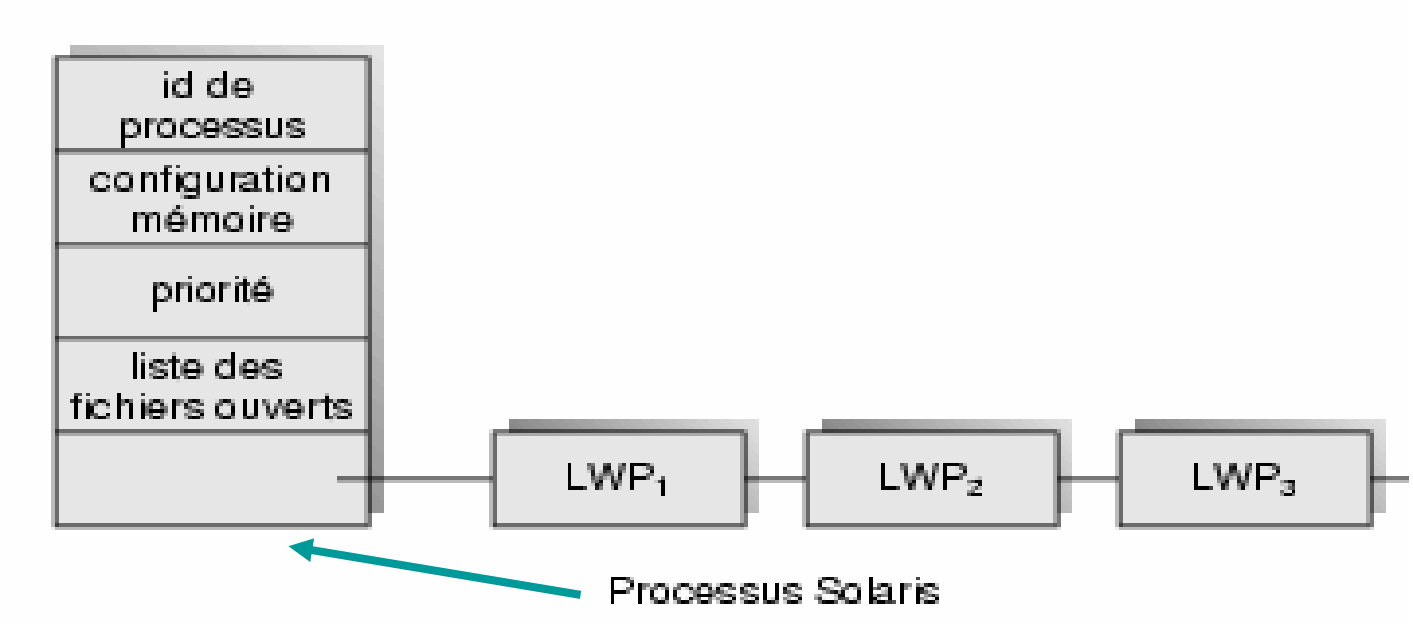


Création et terminaison dynamique de LWP

- **La biblio. de threads à niveau usager crée automatiquement des nouveaux LWP au besoin**
- **Termine les LWP qui ne sont plus demandés**

Structures de données: Solaris

- **Une tâche=processus Solaris peut être associée à plusieurs LWP**
- **Un LWP contient un ensemble de registres, des infos de mémoire et de comptabilisation**
- **Les structures des données sont essentiellement des PCBs enchaînés**



Les Java threads implémentent ces idées

Il y aura discussion sur les Java threads dans le lab

Multithreads et monothreads

- **MS-DOS supporte un processus usager à monothread**
- **UNIX SVR4 supporte plusieurs processus à monothread**
- **Solaris, Widows NT, XP et OS2 supportent plusieurs processus multithreads**

Concepts importants du Chap. 5

- **threads et processus: différence**
- **threads de noyau et d'usager: relations**
- **LWP: lightweight processes, threads légers**
- **Implémentation en utilisant UCT physiques**

Quoi lire dans le livre

- **En classe, nous avons vu seulement 5.1- 5.5**
- **Mais pendant les sessions exercices vous verrez la partie Java**
- **Donc tout le chapitre…**

Ordonnancement Processus

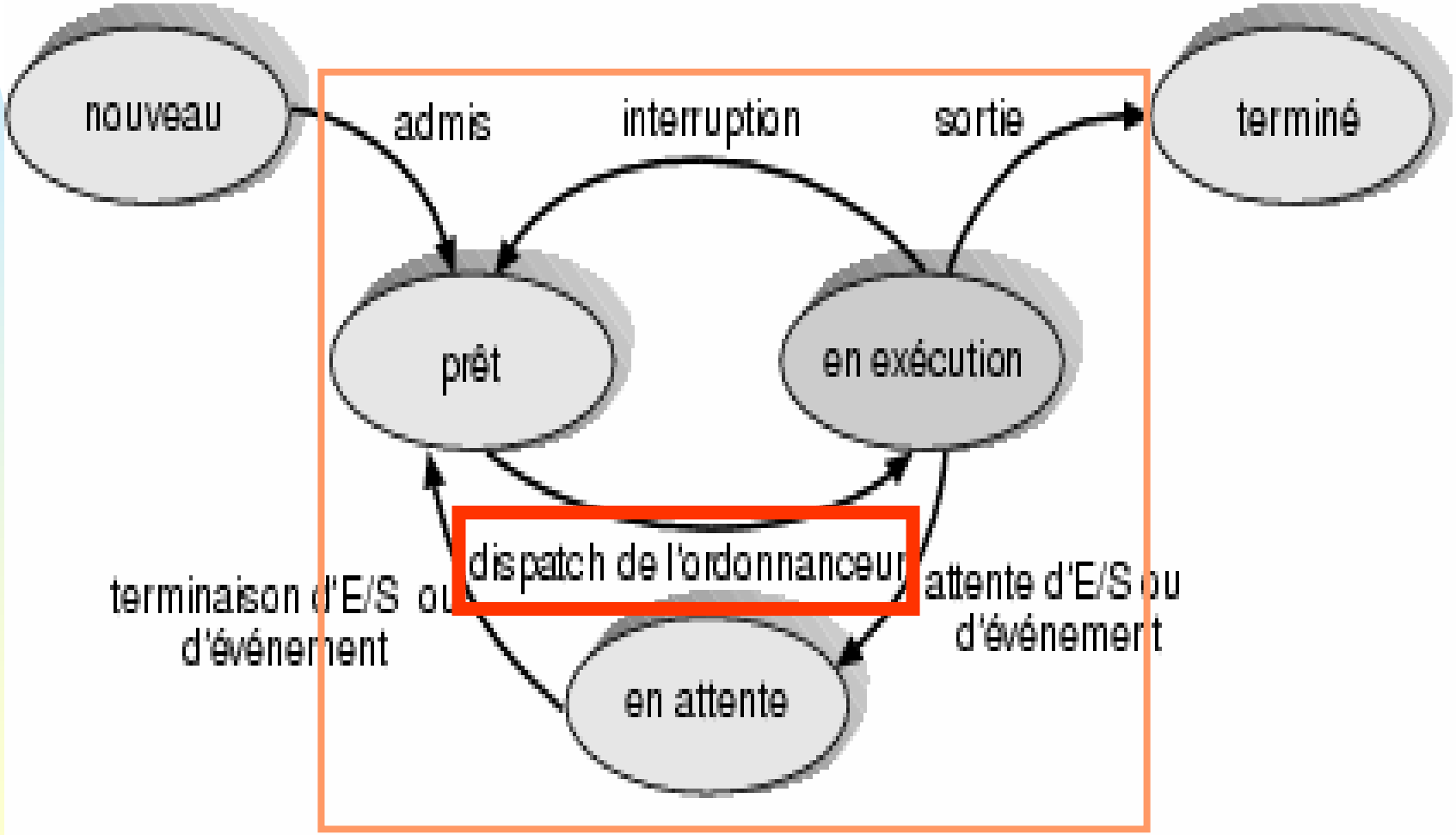
Chapitre 6

http://w3.uqo.ca/luigi/

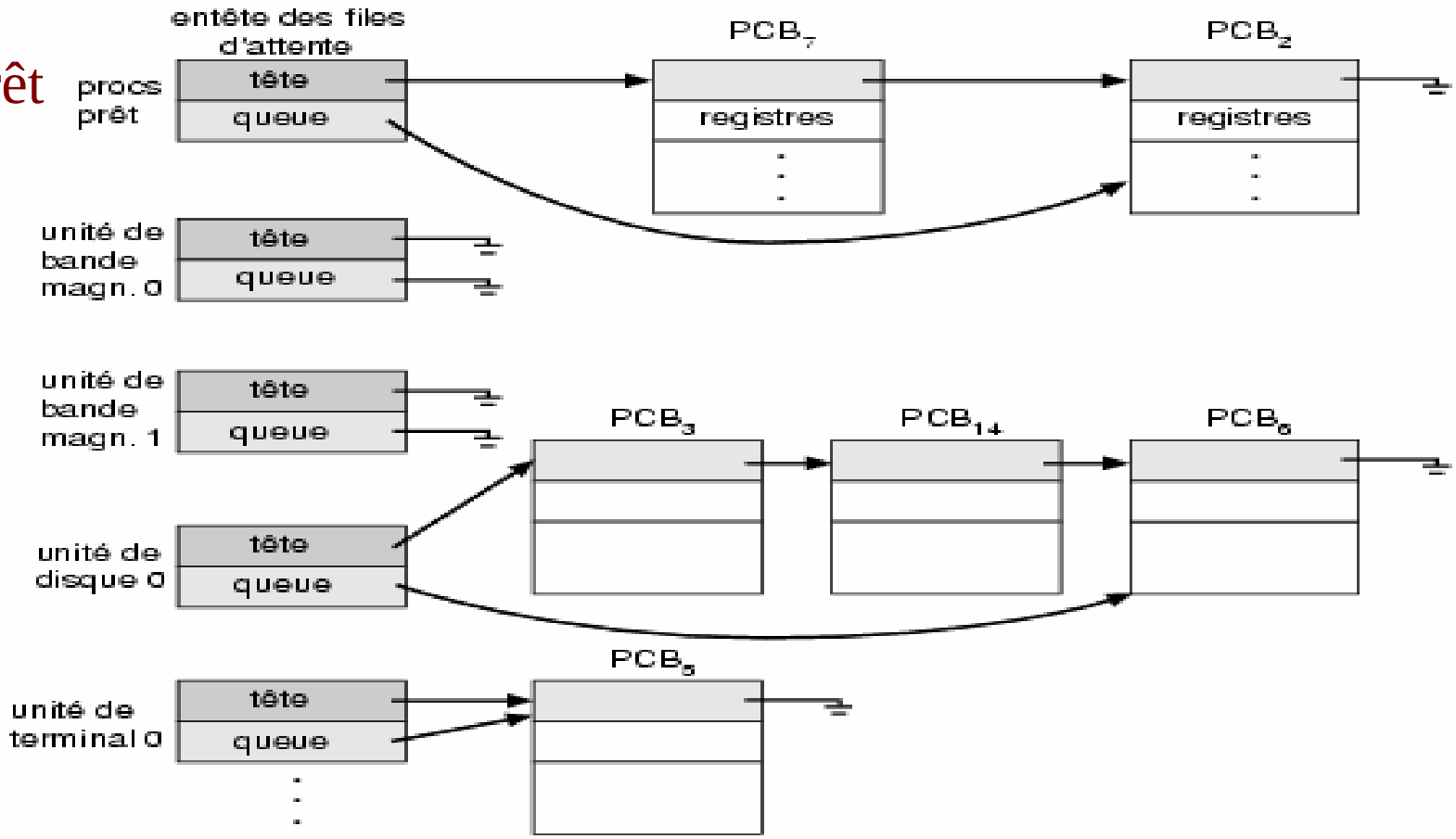
Aperçu du chapitre

- **Concepts de base**
- **Critères d'ordonnancement**
- **Algorithmes d'ordonnancement**
- **Ordonnancement de multiprocesseurs**
- **Ordonnancement temps réel**
- **Évaluation d'algorithmes**

Diagramme de transition d`états d`un processus



Files d'attente de processus pour ordonnancement

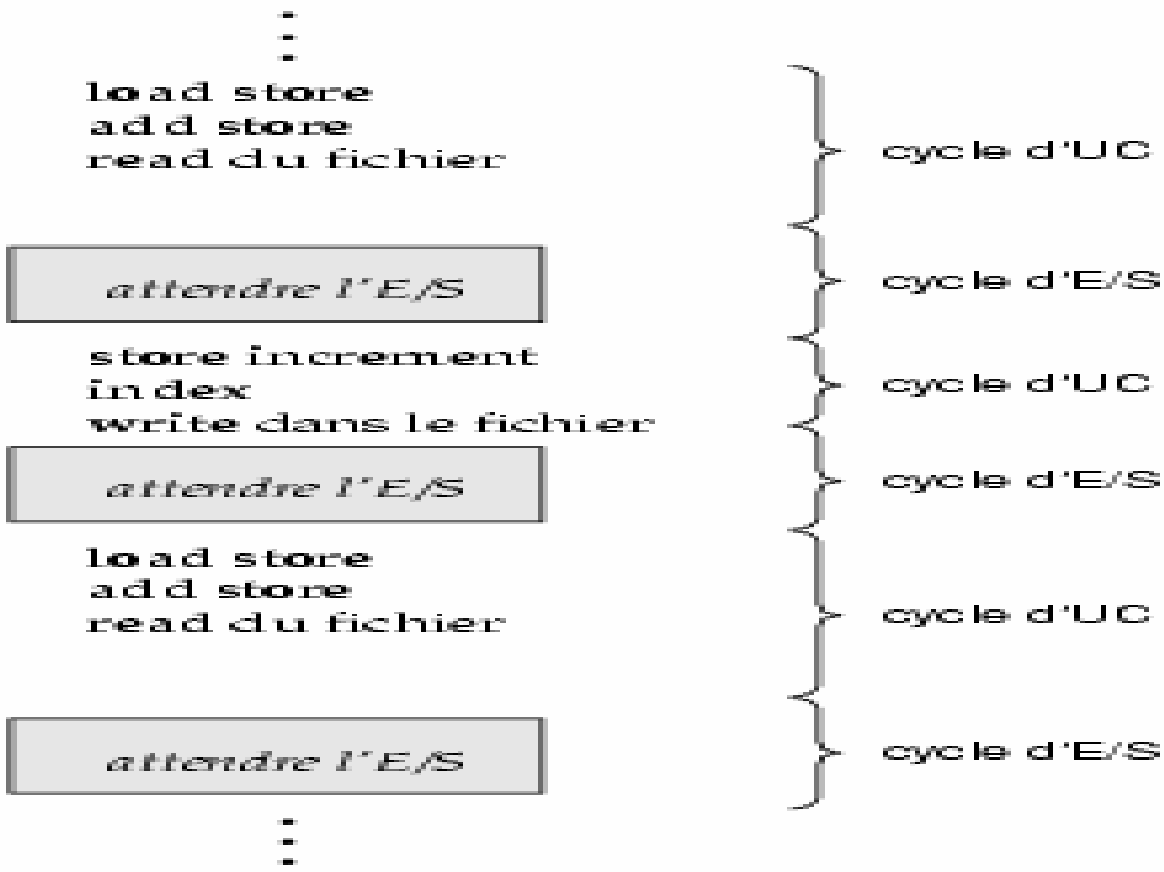


Nous ferons l'hypothèse que le premier processus dans une file est celui qui utilise la ressource: ici, proc7 exécute

Concepts de base

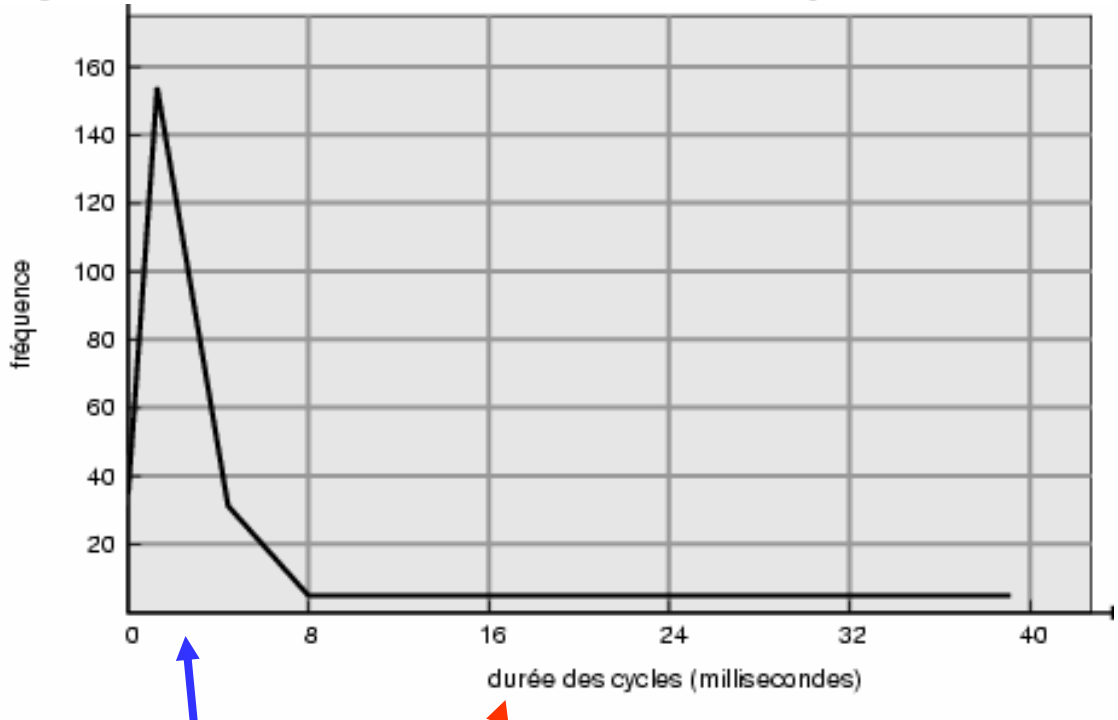
- **La multiprogrammation vise à obtenir une**
  - utilisation optimale des ressources, surtout l'UCT
  - et aussi à un bon temps de réponse pour l'usager
- **L`ordonnanceur UCT est la partie du SE qui décide quel processus dans la file ready/prêt obtient l 'UCT quand elle devient libre**
- **L 'UCT est la ressource la plus précieuse dans un ordinateur, donc nous parlons d'elle**
  - Cependant, les principes que nous verrons s 'appliquent aussi à l 'ordonnancement des autres ressources (unités E/S, etc).

Les cycles d'un processus



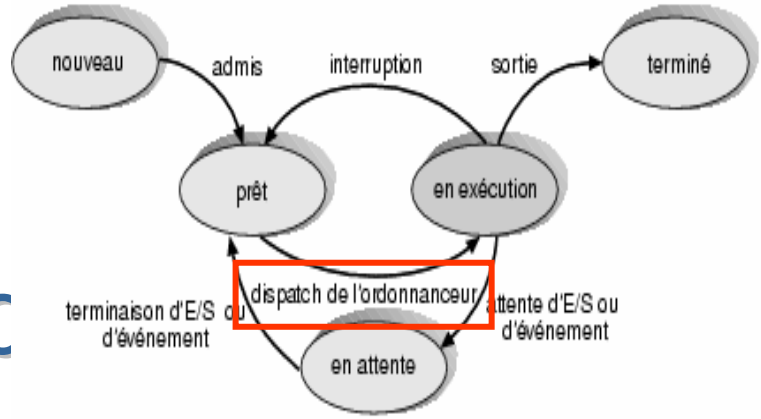
- **Cycles (bursts) d'UCT et E/S: l'exécution d'un processus consiste de séquences d'exécution sur UCT et d'attentes E/S**

Histogramme de durée des cycles UCT



- **Observation expérimentale:**
  - dans un système typique, nous observerons un grand nombre de court cycles, et un petit nombre de long cycles
- **Les programmes tributaires de l 'UCT auront normalm. un petit nombre de long cycles UCT**
- **Les programmes tributaires de l'E/S auront normalm. un grand nombre de court cycles UCT**

Quand invoquer l'ordonnanceur UC



- **L 'ordonnanceur UCT doit prendre sa décision chaque fois que le processus exécutant est interrompu, c'e-à.-d.**
  1. un processus se se présente en tant que nouveau ou se termine
  2. un processus exécutant devient bloqué en attente
  3. un processus change d'exécutant/running à prêt/ready
  4. un processus change de attente à prêt/ready
     - en conclusion, tout événement dans un système cause une interruption de l'UCT et l'intervention de l'ordonnanceur,
     - qui devra prendre une décision concernant quel proc ou thread aura l'UCT après
- **Préemption: on a préemption si on enlève l'UCT à un processus qui l'avait et ne l'a pas laissée de propre initiative**
  - P.ex. préemption dans le cas 3, pas de préemption dans le cas 2
- **Plusieurs pbs à résoudre dans le cas de préemption, v. manuel**

Dispatcheur

- **Le processus qui donne le contrôle au processus choisi par l'ordonnanceur. Il doit se préoccuper de:**
  - changer de contexte
  - changer à mode usager
  - réamorcer le processus choisi
- **Attente de dispatcheur (dispatcher latency)**
  - le temps nécessaire pour exécuter les fonctions du dispatcheur
  - il est souvent négligé, il faut supposer qu'il soit petit par rapport à la longueur d'un cycle

Critères d'ordonnancement

- **Il y aura normalement plusieurs processus dans la file prêt**
- **Quand l'UCT devient disponible, lequel choisir?**
- **Critères généraux:**
  - Bonne utilisation de l'UCT
  - Réponse rapide à l'usager
- **Mais ces critères peuvent être jugés différemment...**

Critères spécifiques d'ordonnancement

- **Utilisation UCT: pourcentage d 'utilisation**
- **Débit = Throughput: nombre de processus qui complètent dans l 'unité de temps**
- **Temps de rotation = turnaround: le temps pris par le proc de son arrivée à sa termin.**
- **Temps d'attente: attente dans la file prêt (somme de tout le temps passé en file prêt)**
- **Temps de réponse (pour les systèmes interactifs): le temps entre une demande et la réponse**

Critères d'ordonnancement: maximiser/minimiser

- **Utilisation UCT: pourcentage d'utilisation**
  - ceci est à maximiser
- **Débit = Throughput: nombre de processus qui complètent dans l 'unité de temps**
  - ceci est à maximiser
- **Temps de rotation (turnaround): temps terminaison moins temps arrivée**
  - à minimiser
- **Temps d'attente: attente dans la file prêt**
  - à minimiser
- **Temps de réponse (pour les systèmes interactifs): le temps entre une demande et la réponse**
  - à minimiser

Examinons maintenant plusieurs méthodes d'ordonnancement et voyons comment elles se comportent par rapport à ces critères

nous étudierons des cas spécifiques

l'étude du cas général demanderait recours à techniques probabilistes ou de simulation

Premier arrive, premier servi (First come, first serve, FCFS)

Exemple:	Processus	Temps de cycle
	P1	24
	P2	3
	P3	3

Si les processus arrivent au temps 0 dans l'ordre: P1 , P2 , P3

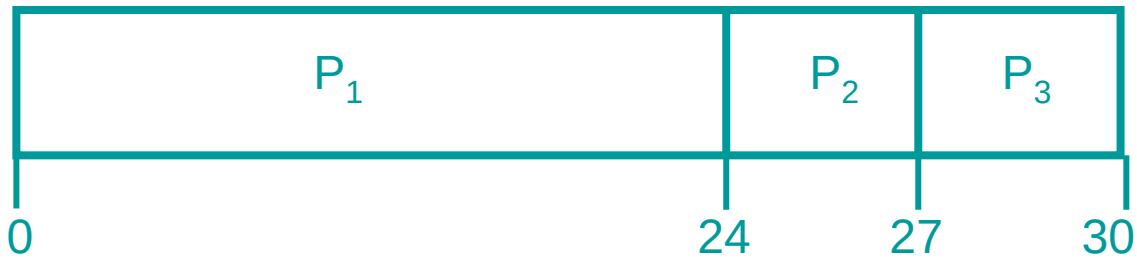
Le diagramme Gantt est:

Temps d'attente pour P1= 0; P2= 24; P3= 27

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

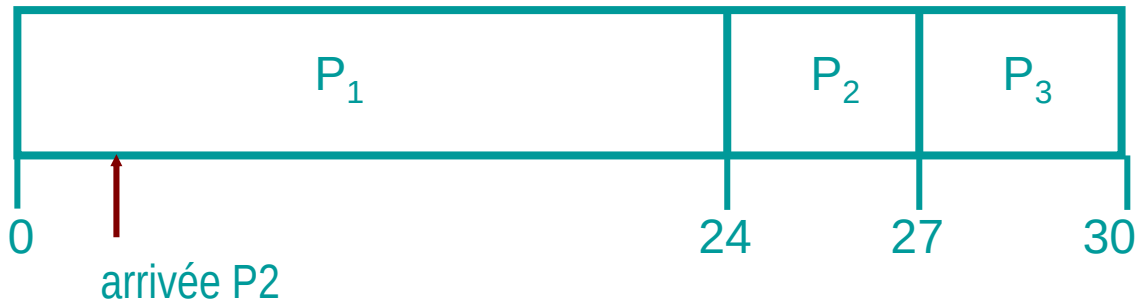
Premier arrive, premier servi

- **Utilisation UCT = 100%**
- **Débit = 3/30 = 0,1**
  - 3 processus complétés en 30 unités de temps
- **Temps de rotation moyen: (24+27+30)/3 = 27**



Tenir compte du temps d'arrivée!

- **Dans le cas où les processus arrivent à moment différents, il faut soustraire les temps d'arrivée**
- **Exercice: répéter les calculs si:**
  - P1 arrive à temps 0 et dure 24
  - P2 arrive à temps 2 et dure 3
  - P3 arrive à temps 5 et dure 3
- **Donc P1 attend 0 comme avant**
- **Mais P2 attend 24-2, etc.**



FCFS Scheduling (Cont.)

Si les mêmes processus arrivent à 0 mais dans l'ordre

$$P_2 , P_3 , P_1 .$$

Le diagramme de Gantt est:

- **Temps d'attente pour $P_1 = 6$ $P_2 = 0$ $P_3 = 3$**
- **Temps moyen d'attente: (6 + 0 + 3)/3 = 3**
- **Temps de rotation moyen: (3+6+30)/3 = 13**
- **Beaucoup mieux!**
- **Donc pour cette technique, les temps peuvent varier grandement par rapport à l'ordre d'arrivée de différent processus**
- ***Exercice: calculer aussi le débit, etc.***

Effet d'accumulation (convoy effect) dans FCFS

- **Supposons un processus tributaire de l'UCT et plusieurs tributaires de l`E/S (situation assez normale)**
- **Les processus tributaires de l'E/S attendent pour l 'UCT: E/S sous-utilisée (*)**
- **Le processus tributaire de l'UCT fait une E/S: les autres proc exécutent rapidement leur cycle UCT et retournent sur l'attente E/S: UCT sous-utilisée**
- **Processus tributaire de l'UCT fini son E/S, puis les autres procs aussi : retour à la situation (*)**
- **Donc dans ce sens FCFS favorise les procs tributaires de l'UCT**
- **Et peut conduire à une très mauvaise utilisation des ressources**
  - Tant d'UCT que de périphériques
- **Une possibilité: interrompre de temps en temps les proc tributaires de l'UCT pour permettre aux autres procs d'exécuter (préemption)**

Plus Court d'abord = Shortest Job First (SJF)

- **Le processus qui demande moins d'UCT part le premier**
- **Optimal en principe du point de vue du temps d'attente moyen**
  - (v. le dernier exemple)
- **Mais comment savons-nous quel processus demande moins d'UCT!**

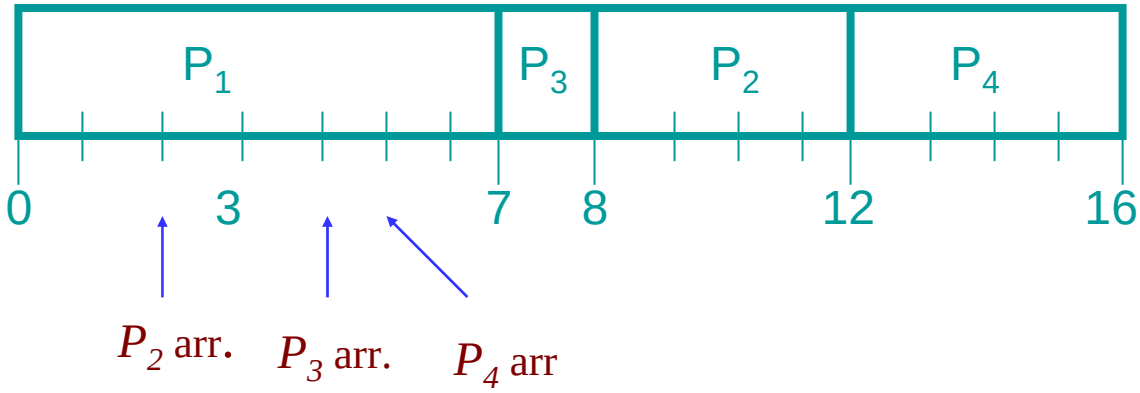
SJF avec préemption ou non

- **Avec préemption: si un processus qui dure moins que le *restant* du processus courant se présente plus tard, l'UCT est enlevée au proc courant et donnée à ce nouveau processus**
  - SRTF: shortest remaining-time first
- **Sans préemption: on permet au processus courant de terminer son cycle**
  - Observation: SRTF est logique pour l'UCT car le processus exécutant sera interrompu par l'arrivée du nouveau processus
    - Il est retourné à l'état prêt
  - Il est impossible pour les unités qui ne permettent pas de préemption
    - p.ex. une unité disque, imprimante…

Example de SJF sans préemption

Processus	Arrivée	Cycle
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **SJF (sans préemption)**

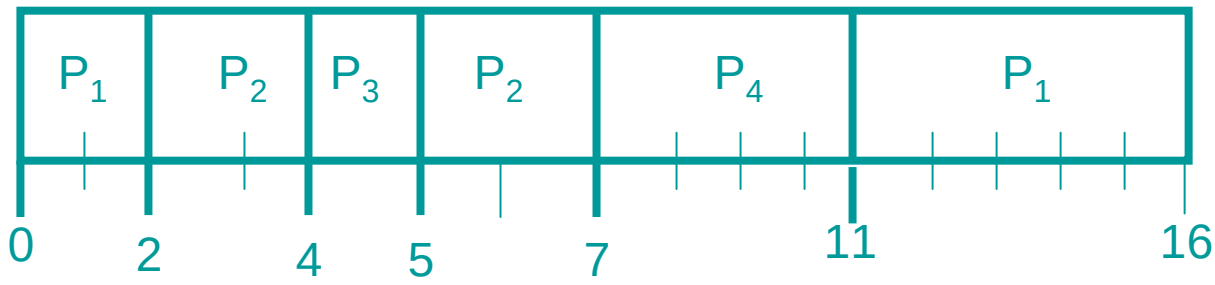


- **Temps d'attente moyen =** (0+(8-2)+(7-4)+(12-5))/4
  - (0 + 6 + 3 + 7)/4 = 4
- **Temps de rotation moyen =** 7+(12-2)+(8-4)+(16-5) = 8

Exemple de SJF avec préemption

Processus	Arrivée	Cycle
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **SJF (préemptive)**



P_2 arr. P_3 arr. P_4 arr

- **Temps moyen d`attente = (9 + 1 + 0 +2)/4 = 3**
  - **P1 attend de 2 à 11, P2 de 4 à 5, P4 de 5 à 7**
- **Temps de rotation moyen =** 16+ (7-2) + (5-4) + (11-5) = 7

Comment déterminer la longueur des cycles à l'avance?

- **Quelques méthodes proposent de déterminer le comportement futur d 'un processus sur la base de son passé**
  - p.ex. moyenne exponentielle

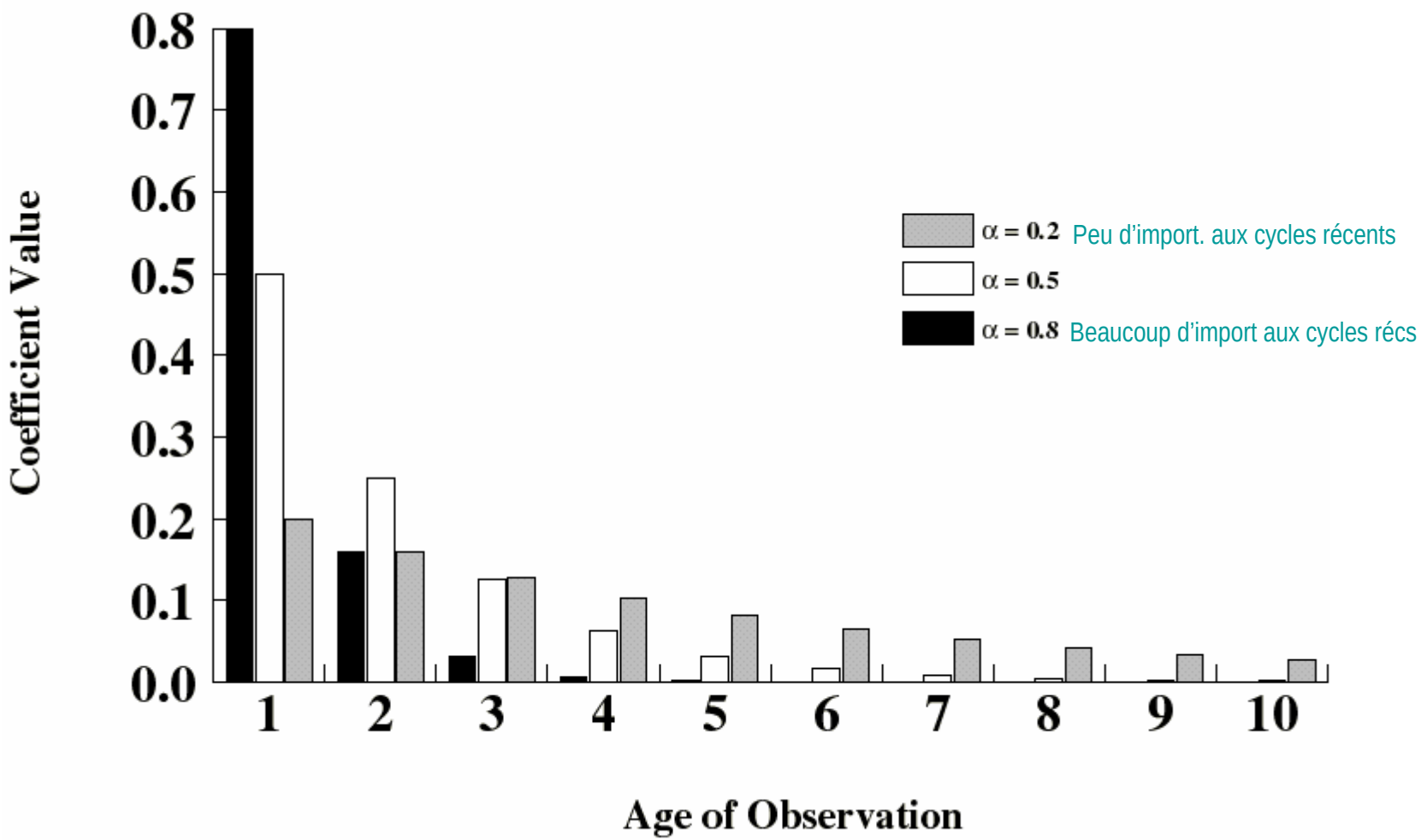
Estimation de la durée du prochain cycle

- **T_i : la durée du *i*ème cycle de l'UCT pour ce processus**
- **S_i : la valeur *estimée* du le *i*ème cycle de l'UCT pour ce processus. Un choix simple est:**
  - $S_{n+1} = (1/n) \sum_{i=1 \text{ to } n} T_i$ (une simple moyenne)
- **Nous pouvons éviter de recalculer la somme en récrivant:**
  - $S_{n+1} = (1/n) T_n + ((n-1)/n) S_n$
- **Ceci donne un poids identique à chaque cycle**

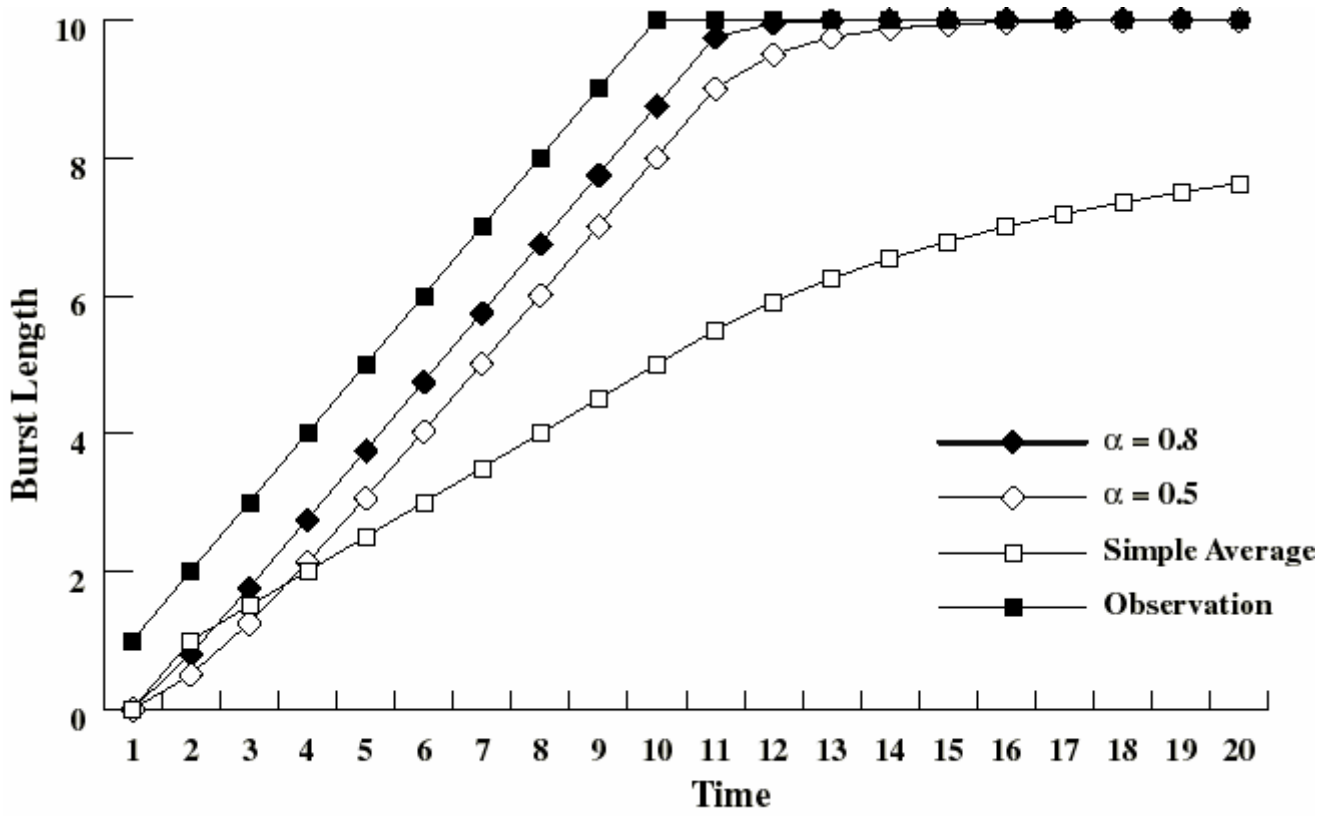
Estimation de la durée du prochain cycle

- **Mais les cycles récents peuvent être plus représentatifs des comportements à venir**
- **La moyenne exponentielle permet de donner différents poids aux cycles plus ou moins récents:**
  - $S_{n+1} = \alpha\, T_n + (1-\alpha)\, S_n \;\; ; \;\; 0 <= \alpha <= 1$
- **Par expansion, nous voyons que le poids de chaque cycle décroît exponentiellement**
  - $S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1$
- **la valeur estimée S_1 du 1er cycle peut être fixée à 0 pour donner priorité max. aux nouveaux processus**

Importance de différents valeurs de coefficients [Stallings]

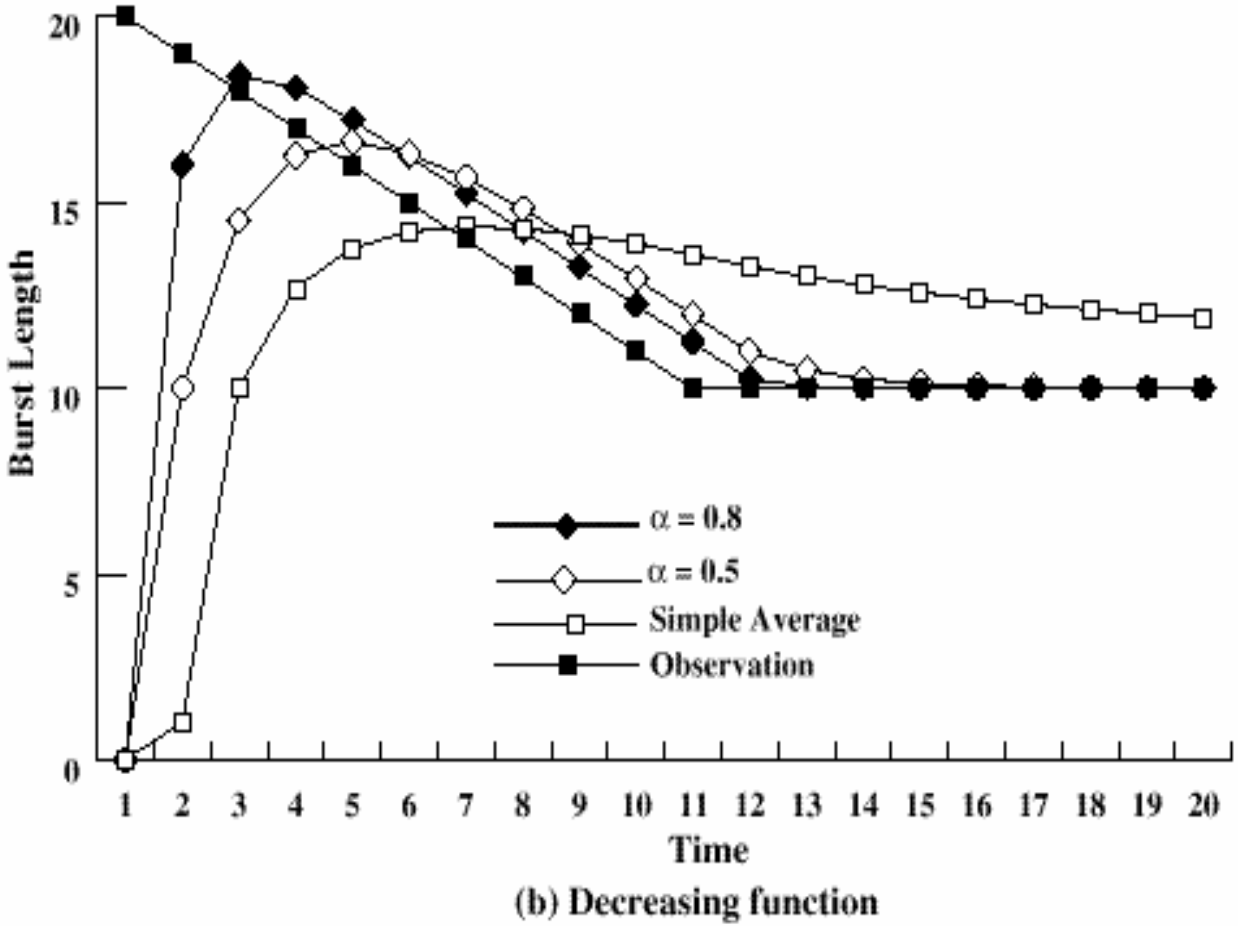


Importance de différents valeurs de coefficients [Stallings]



- $S_1 = 0$ (priorité aux nouveaux processus)
- Un coefficient plus élevé réagit plus rapidement aux changements de comportement

Un deuxième exemple [Stallings]



(b) Decreasing function

Stallings

Comment choisir le coefficient α

- **Un petit coefficient est avantageux quand un processus peut avoir des anomalies de comportement, après lesquelles il reprend son comportement précédent (il faut ignorer son comportement récent)**
  - cas limite: α **= 0** on reste sur l 'estimée initiale
- **Un coefficient élevé est avantageux quand un processus est susceptible de changer rapidement de type d'activité et il reste sur ça**
  - cas limite: $\alpha = 1$: $S_{n+1} = T_n$
    - Le dernier cycle est le seul qui compte

Le plus court d'abord SJF: critique

- **Difficulté d'estimer la longueur à l'avance**
- **Plus pratique pour l'ordonnancement travaux que pour l'ordonnancement processus**
  - on peut plus facilement prévoir la durée d'un travail entier que la durée d'un cycle
- **Il y a assignation implicite de priorités: préférences aux travaux plus courts**

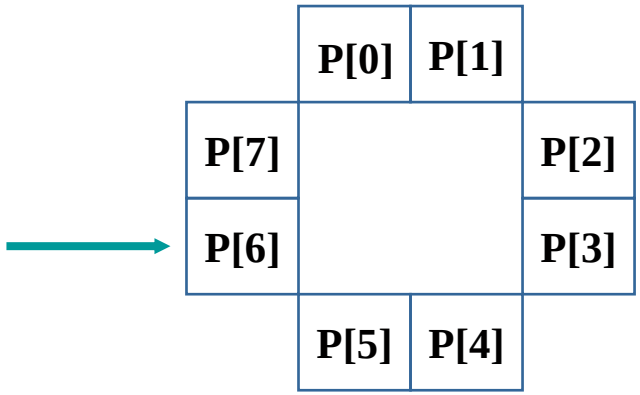
Difficultés majeures avec les méthodes discutées

- **Un processus long peut monopoliser l'UCT s'il est le 1er à entrer dans le système et il ne fait pas d'E/S**
- **Dans le cas de SJF, les processus longs souffriront de famine lorsqu'il y a un apport constant de processus courts**

Tourniquet = Round-Robin (RR)

Le plus utilisé en pratique

- **Chaque processus est alloué une tranche de temps (p.ex. 10-100 millisecs.) pour exécuter**
  - Tranche aussi appelée *quantum*
- **S'il exécute pour tranche entière sans autres interruptions, il est interrompu par la minuterie et l 'UCT est donnée à un autre processus**
- **Le processus interrompu redevient prêt (à la fin de la file)**
- **Méthode préemptive**



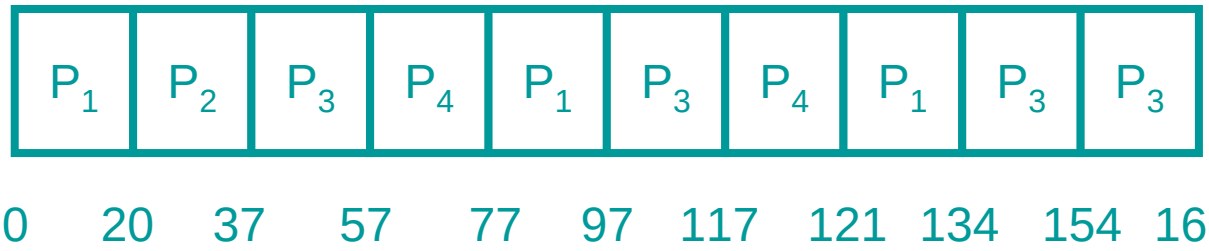
La file prêt est un cercle (dont RR)

Performance de tourniquet

- **S ’il y a n processus dans la file prêt et la tranche est t, alors chaque processus reçoit 1/n du temps UCT dans unités de durée max. t**
- **Si t grand $\Rightarrow$ FIFO**
- **Si t petit... nous verrons**

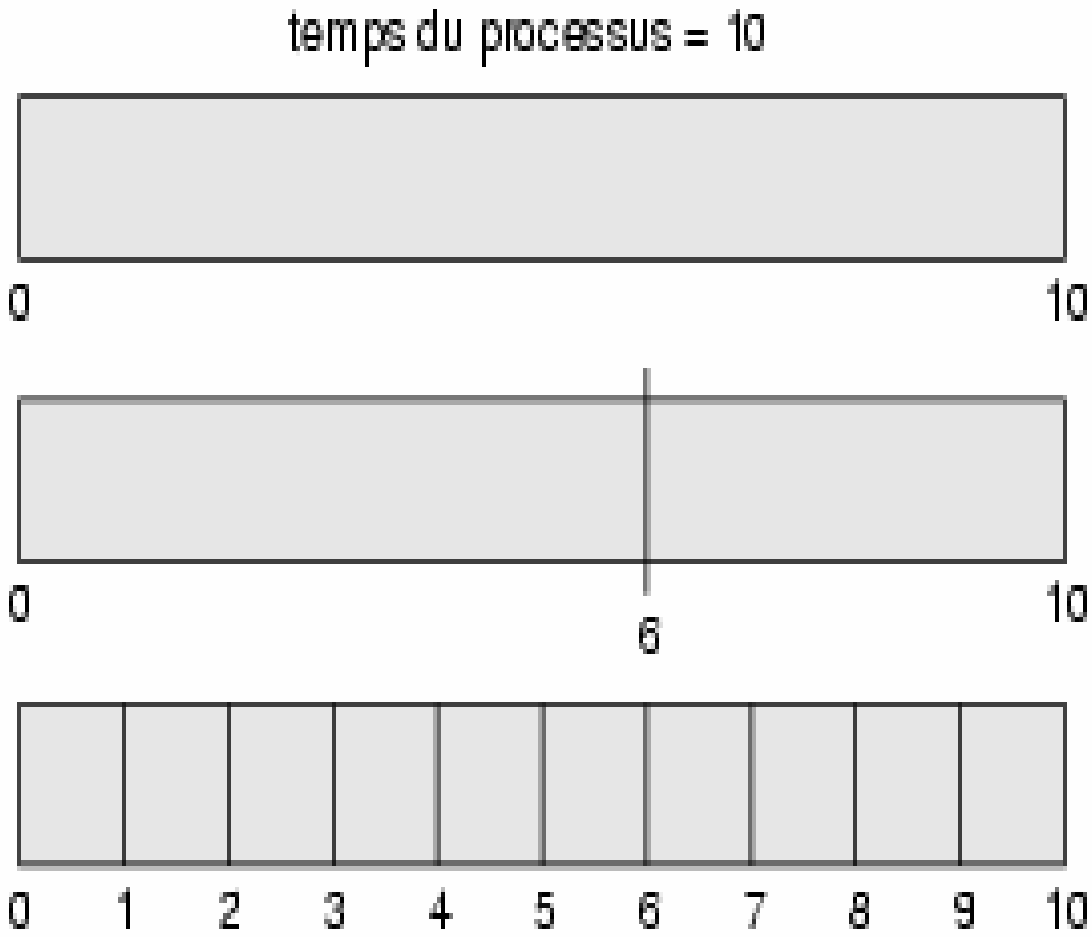
Exemple: Tourniquet tranche = 20

Processus	Cycle
P_1	53
P_2	17
P_3	68
P_4	24



- **Observez**
  - temps de rotation et temps d'attente moyens beaucoup plus élevés que SJF
  - mais aucun processus n'est favorisé

Une petite tranche augmente les commutations de contexte (temps de SE)

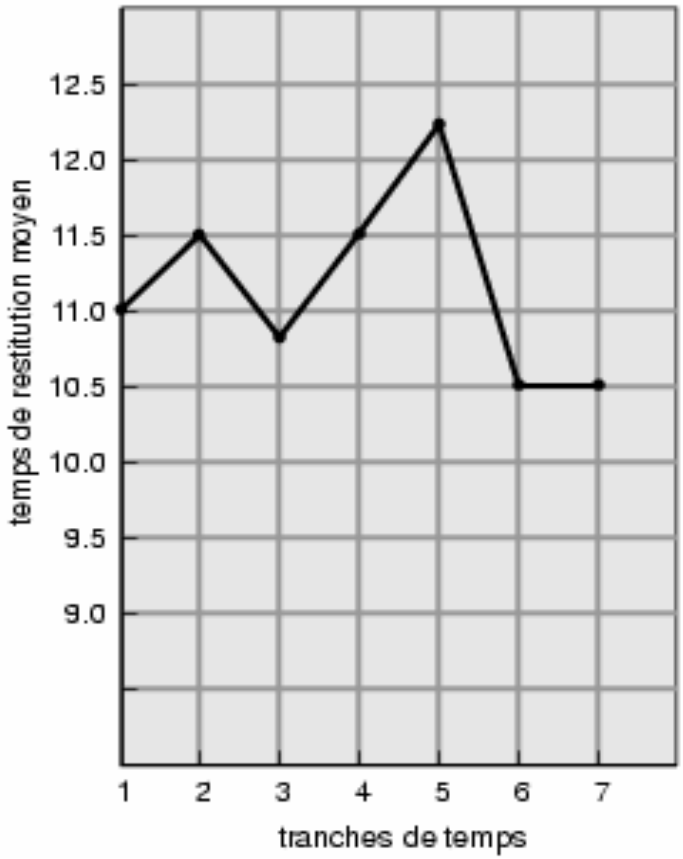


tranches de temps	changement de contexte
12	0
6	1
1	9

Exemple pour voir l'importance d'un bon choix de tranche (à développer comme exercice)

- **Trois cycles:**
  - A, B, C, toutes de 10
- **Essayer avec:**
  - t=1
  - t=10
- **Dans ce deuxième cas, tourniquet fonctionne comme FIFO et le temps de rotation moyen est meilleur**

Le temps de rotation (turnaround) varie avec la tranche



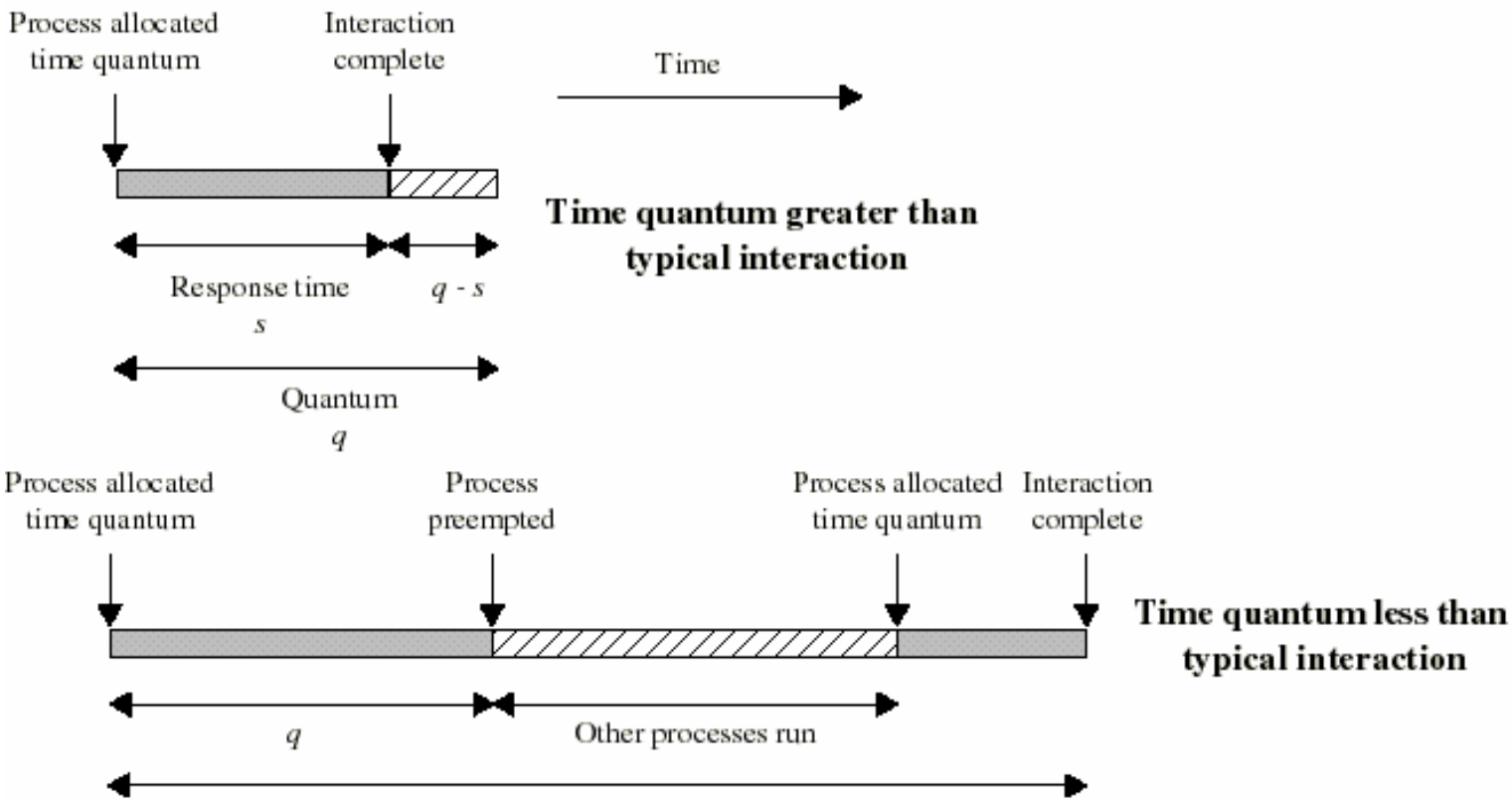
processus	temps
P_1	6
P_2	3
P_3	1
P_4	7

= FIFO

Exemple qui montre que le temps de rotation moyen n 'améliore pas nécessairement en augmentant la tranche (sans considérer les temps de commutation contexte)

Choix de la tranche pour le tourniquet [Stallings]

- **doit être beaucoup plus grande que le temps requis pour exécuter le changement de contexte**
- **doit être un peu plus grand que le cycle typique (pour donner le temps à la plupart des proc de terminer leur cycle, mais pas trop pour éviter de pénaliser les processus courts)**



 v. ex. prec. où les point optimaux sont 6 et 7, et il y a deux cycles de 6 et 7

Priorités

- **Affectation d'une priorité à chaque processus (p.ex. un nombre entier)**
  - souvent les petits chiffres dénotent des hautes priorités
    - 0 la plus haute
- **L'UCT est donnée au processus prêt avec la plus haute priorité**
  - avec ou sans préemption
  - il y a une file *prêt* pour chaque priorité

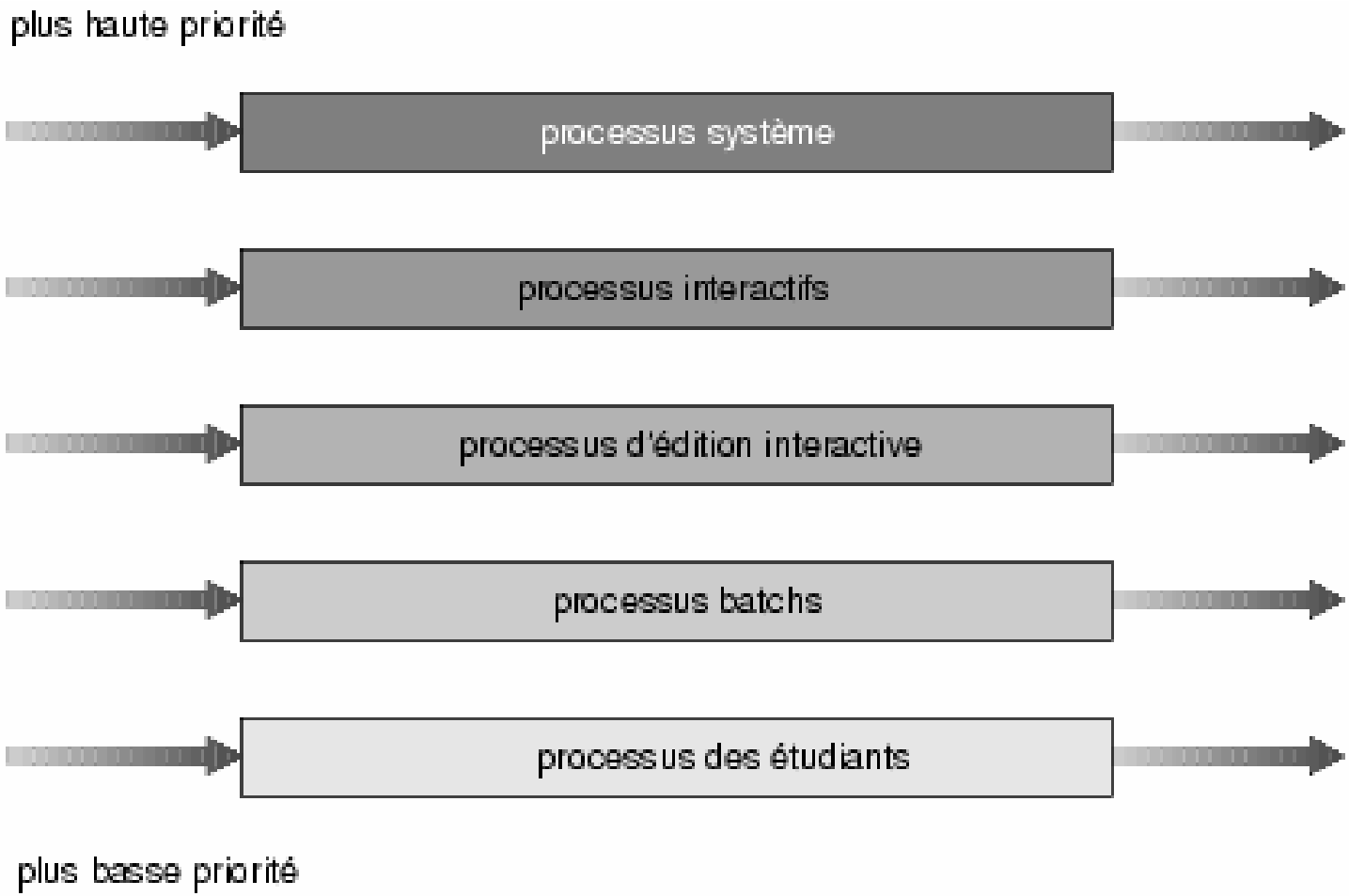
Problème possible avec les priorités

- **Famine: les processus moins prioritaires n'arrivent jamais à exécuter**
- **Solution: vieillissement:**
  - modifier la priorité d 'un processus en fonction de son âge et de son historique d 'exécution
  - le processus change de file d`attente
- **Plus en général, la modification dynamique des priorités est une politique souvent utilisée (v. files à rétroaction ou retour)**

Files à plusieurs niveaux (multiples)

- **La file *prêt* est séparée en plusieurs files, p.ex.**
  - travaux `d'arrière-plan` (background - batch)
  - travaux `de premier plan` (foreground - interactive)
- **Chaque file a son propre algorithme d 'ordonnancement, p.ex.**
  - FCFS pour arrière-plan
  - tourniquet pour premier plan
- **Comment ordonnancer entre files?**
  - Priorité fixe à chaque file $\rightarrow$ famine possible, ou
  - Chaque file reçoit un certain pourcentage de temps UCT, p.ex.
    - 80% pour arrière-plan
    - 20% pour premier plan

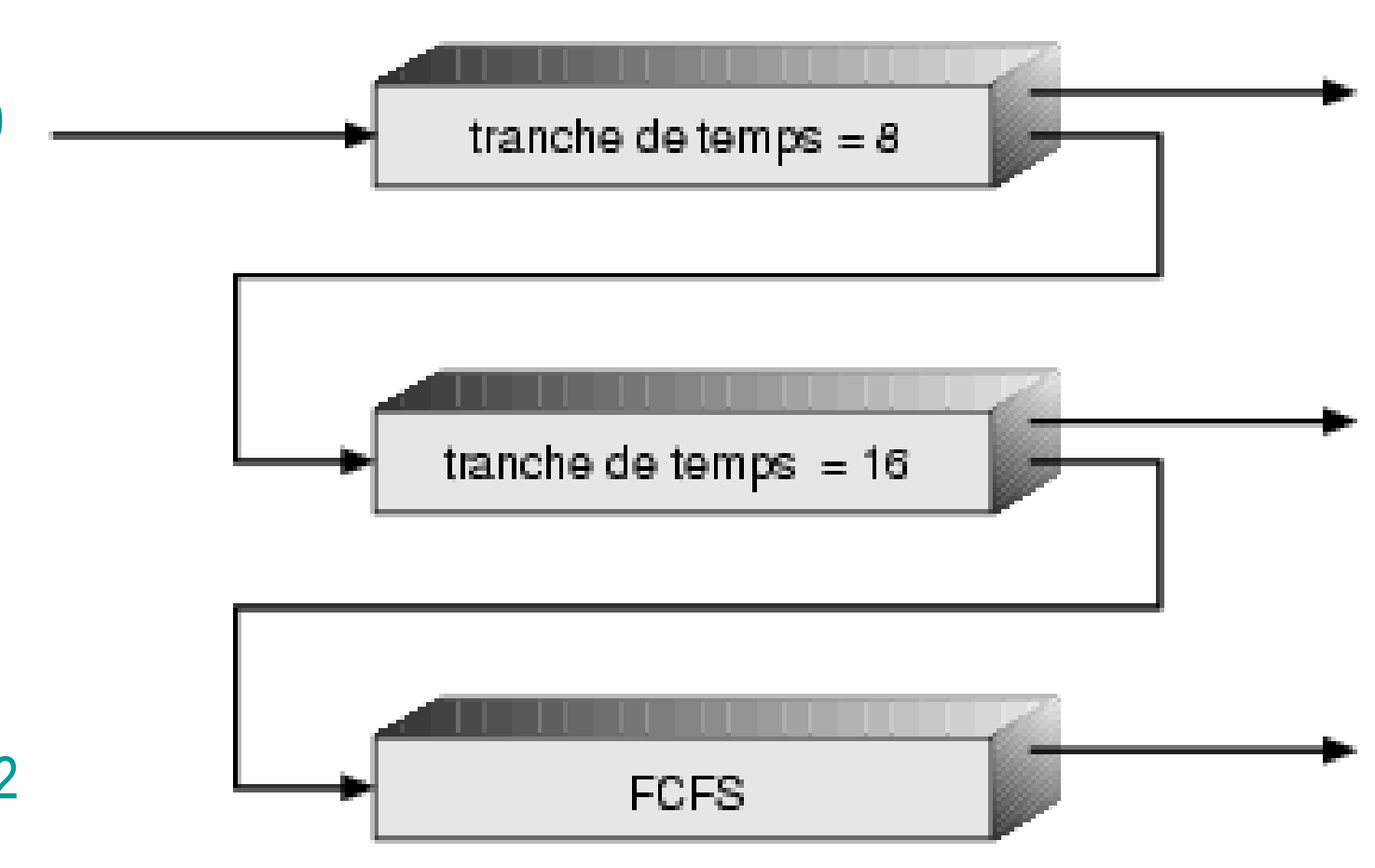
Ordonnancement avec files multiples



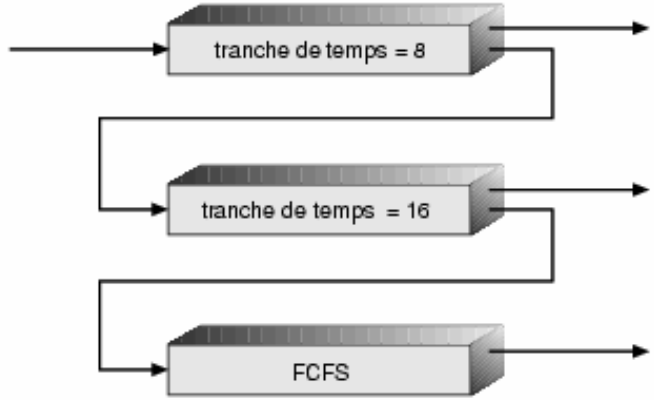
Files multiples et à retour

- **Un processus peut passer d ’une file à l ’autre, p.ex. quand il a passé trop de temps dans une file**
- **À déterminer:**
  - nombre de files
  - algorithmes d ’ordonnancement pour chaque file
  - algorithmes pour décider quand un proc doit passer d ’une file à l`autre
  - algorithme pour déterminer, pour un proc qui devient prêt, sur quelle file il doit être mis

Files multiples et à retour



Exemple de files multiples à retour



- **Trois files:**
  - Q0: tourniquet, tranche = 8 msecs
  - Q1: tourniquet, tranche = 16 msec
  - Q2: FCFS
- **Ordonnancement:**
  - Un nouveau processus entre dans Q0, il reçoit 8 msecs d ’UCT
  - S ’il ne finit pas dans les 8 msecs, il est mis dans Q1, il reçoit 16 msecs additionnels
  - S ’il ne finit pas encore, il est interrompu et mis dans Q2
  - Si plus tard il commence à avoir des cycles plus courts, il pourrait retourner à Q0 ou Q1

En pratique...

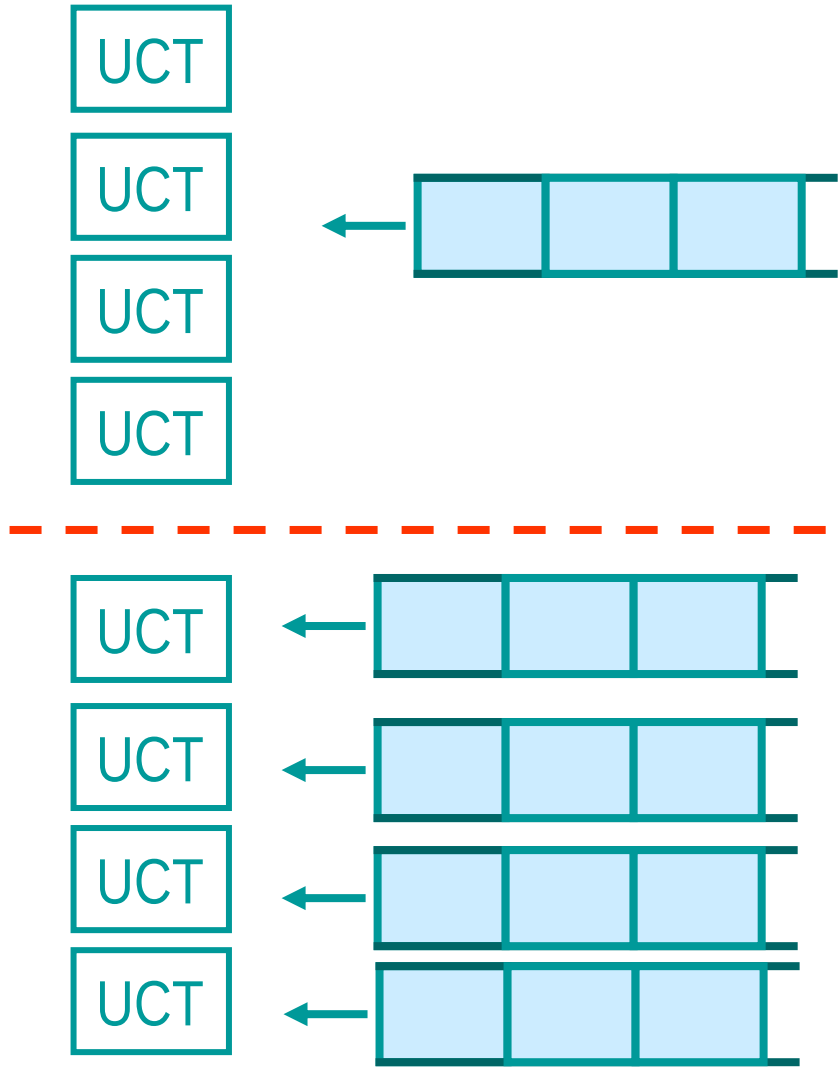
- **Les méthodes que nous avons vu sont toutes utilisées en pratique (sauf plus court servi *pur* qui est impossible)**
- **Les SE sophistiqués fournissent au gérant du système une librairie de méthodes, qu'il peut choisir et combiner au besoin après avoir observé le comportement du système**
- **Pour chaque méthode, plusieurs params sont disponibles: p.ex. durée des tranches, coefficients, etc.**
- **Ces méthodes évidemment sont importantes seulement pour les ordis qui ont des fortes charges de travail**

Aussi…

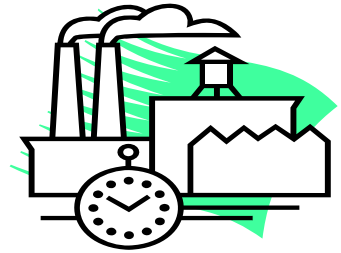
- **Notre étude des méthodes d'ordonnancement est théorique, ne considère pas en détail tous les problèmes qui se présentent dans l'ordonnancement UCT**
- **P.ex. les ordonnanceurs UCT ne peuvent pas donner l'UCT à un processus pour tout le temps dont il a besoin**
  - Car en pratique, l'UCT sera souvent interrompue par quelque événement externe avant la fin de son cycle
- **Cependant les mêmes principes d'ordonnancement s'appliquent à unités qui ne peuvent pas être interrompues, comme une imprimante, une unité disque, etc.**
- **Dans le cas de ces unités, on pourrait avoir des infos complètes concernant le temps de cycle prévu, etc.**
- **Aussi, cette étude ne considère pas du tout les temps d'exécution de l'ordonnanceur, du dispatcheur, etc.**

Ordonnancement avec plusieurs UCTs identiques: *homogénéité*

- **Méthodes symétriques: chaque UCT peut exécuter l'ordonnancement et la répartition**
  - Une seule liste *prêt* pour toutes les UCTs (division travail = load sharing)
- **Méthodes asymétriques: certaines fonctions sont réservées à une seule UCT**
  - Files d'attentes séparées pour chaque UCT



Systèmes temps réel



- **systèmes temps réel *rigides (hard):***
  - les échéances sont critiques (p.ex. contrôle d'une chaîne d`assemblage, animation graphique)
  - il est essentiel de connaître la durée des fonctions critiques
  - il doit être possible de garantir que ces fonctions sont effectivement exécutées dans ce temps (réservation de ressources)
  - ceci demande une structure de système très particulière
- **systèmes temps réel *souples (soft):***
  - les échéances sont importantes, mais ne sont pas critiques (p.ex. systèmes téléphoniques)
  - les processus critiques reçoivent la *priorité*

Systèmes temps réel:
Problèmes d'attente dans plus. systèmes (ex. UNIX)

- **Dans UNIX 'classique' il n 'est pas permis d 'effectuer changement de contexte pendant un appel du système - et ces appels peuvent être longs**
- **Pour le temps réel il est nécessaire de permettre la préemption des appels de systèmes ou du noyau en général**
- **Donc ce système n'est pas considéré approprié pour le temps réel**
- **Mais des variétés appropriées de UNIX ont été conçues (p.ex. Solaris)**

Inversion de priorité et héritage de priorités

- **Quand un processus de haute priorité doit attendre pour des processus de moindre priorité (p.ex. a besoin de données produites par ces derniers)**

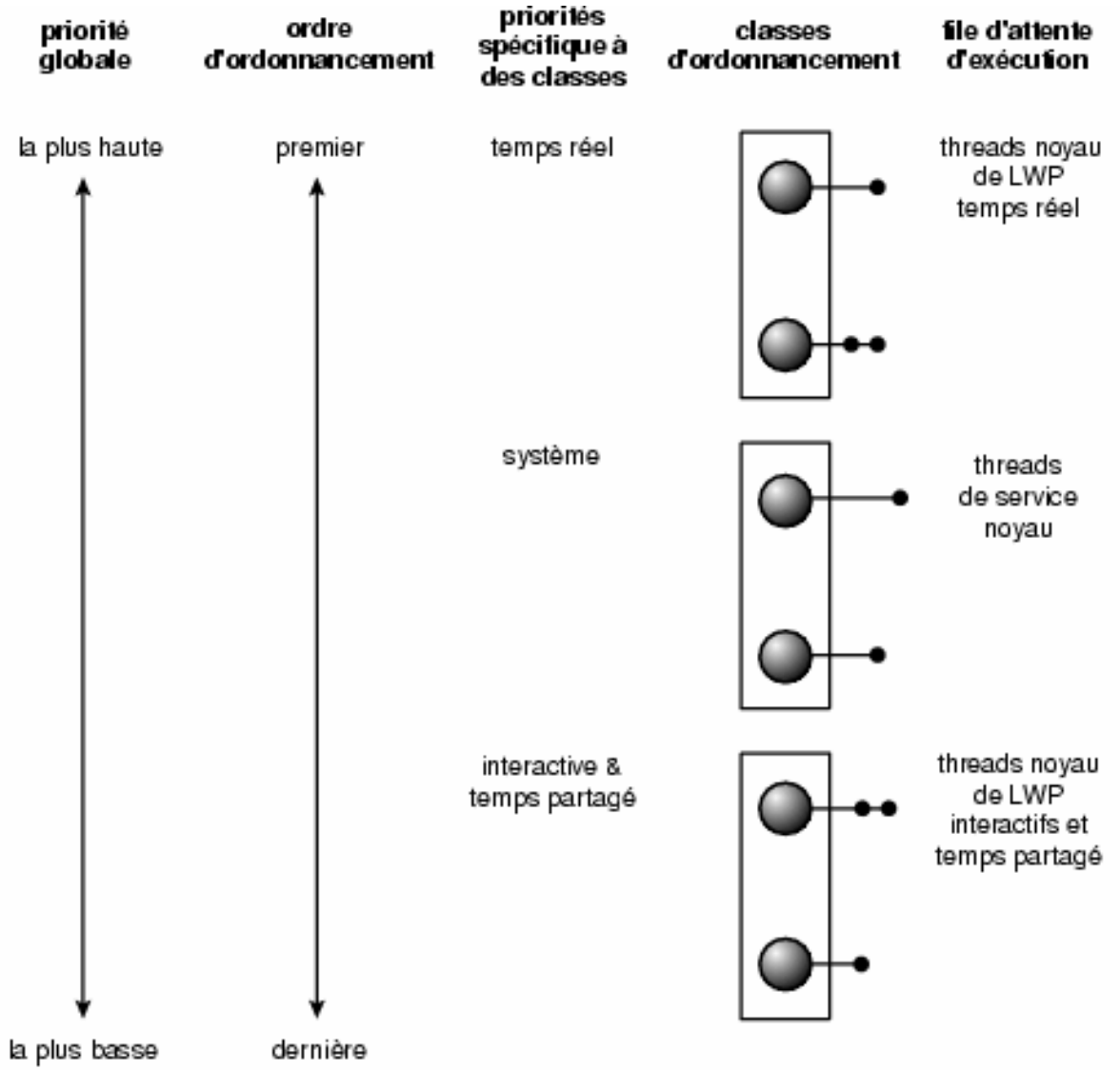
- **Pour permettre à ces derniers de finir rapidement, on peut lui faire *hériter* la priorité du processus plus prioritaire**

Ordonnancement de threads

- **Local: la librairie des threads pour une application donnée décide quel thread usager obtient un LWP disponible**

- **Global: le noyau décide quel thread de noyau exécute sur l'UCT**

Ordonnancement et priorités en Solaris 2



Solaris 2: lire dans le manuel pour voir l'application pratique de plusieurs concepts discutés

- **Priorités et préemption**
- **Files multiniveau à retour avec changement de priorité**
- **Tranches plus grandes pour les processus moins prioritaires**
- **Les procs interactifs sont plus prioritaires que les les procs tributaires de l'UCT**
- **La plus haute priorité aux procs temps réel**
- **Tourniquet pour les fils de priorités égales**

Méthode d'évaluation et comparaison d'algorithmes (section plutôt à lire)

- **Modélisation déterministe**
- **Modèles de files d'attente (queuing theory)**
- **Simulation**
- **Implantation**

Modélisation déterministe

- **Essentiellement, ce que nous avons déjà fait en étudiant le comportement de plusieurs algorithmes sur plusieurs exemples**

Utilisation de la théorie des files (queuing th.)

- **Méthode analytique basée sur la théorie des probabilités**
- **Modèle simplifié: notamment, les temps du SE sont ignorés**
- **Cependant, elle rend possibles des estimées**

Théorie des files: la formule de Little

- **Un résultat important:**

  $$\mathbf{n = \lambda \times W}$$

  - n: longueur moyenne de la file d 'attente
  - λ : débit d 'arrivée de travaux dans file
  - W: temps d 'attente moyen dans la file (temps de service)
- **P. ex.**
  - λ si les travaux arrivent 3 par sec.
  - **W** et il restent dans la file 2 secs
  - **n** la longueur moyenne de la file sera???
- **Exercice: Résoudre aussi pour λ et W**
- **Observer que afin que n soit stable, $\lambda \times$ W doit être stable**
  - Un débit d'arrivée plus rapide doit impliquer un temps de service mineur, et vice-versa
    - Si n doit rester 6 et λ monte à 4, quel doit être W?

Simulation

- **Construire un modèle *(simplifié...)* de la séquence d'événements dans le SE**
- **Attribuer une durée de temps à chaque événement**
- **Supposer une certaine séquence d'événements extérieurs (p.ex. arrivée de travaux, etc.)**
- **Exécuter le modèle pour cette séquence afin d'obtenir des stats**

Implémentation

- **Implémenter l'algorithme**
- **Exécuter dans le système réel ou des mélanges de travaux typiques (benchmark)**
- **Obtenir des statistiques,**
  - en tirer des conclusions...

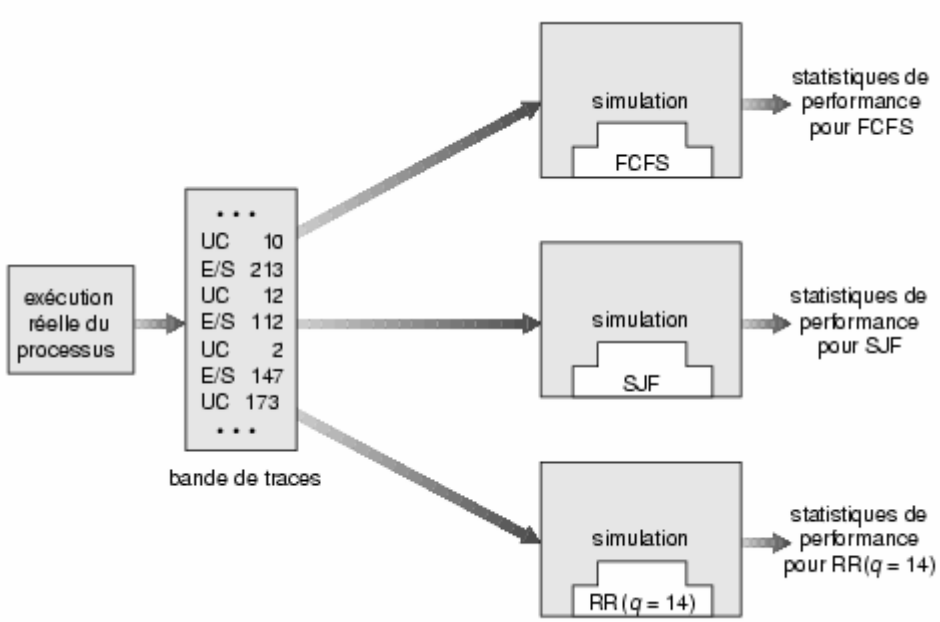


Tableau de comparaison

Le tableau suivant fait une comparaison globale des différentes techniques étudiées

	Critère sélection	Préempt	Motivation	Temps de rotat. et att.	Temps de système	Effect sur processus	Famine
FCFS	Max [w]	non	**Simplicité**	**Variable**	Minim.	**Favor. proc. trib. UCT**	Non
Tourniq.	Tour fixe	oui	**Equité**	**Variable selon tranche, Normalement élevé**	Élevé pour tranche courte	Équitable	Non
SJF	Min[s]	non	Optimisation des temps	**Optimal, mais souffre si des procs longs arrivent au début**	**Peut être élevé (pour estimer les longs. des trames)**	**Bon pour proc. courts, pénalise proc. longs**	Possible
SJF préemp.	Min[s-e]	oui	**Évite le problème de procs longs au début**	**Meilleur, même si des procs longs arrivent au début**	**Peut être élevé**	**Pénalise plus encore proc. longs**	Possible
Files multiniv.	v. détails	oui	**Varie la longueur des tranches en fonction des besoins**	**Variable**	**Peut être élévé**	**Variable**	Possible

w: temps total dans système jusqu'à présent; e: temps en exec jusqu'à présent

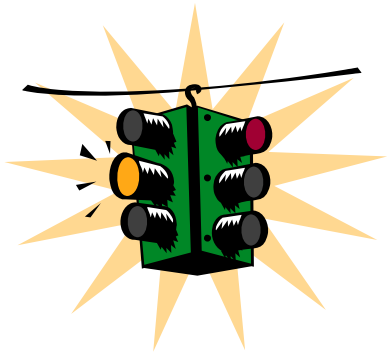
s: temps demandé; Famine est 'renvoi infini'

Points importants dans ce chapitre

- **Files d ’attente pour UCT**
- **Critères d ’ordonnancement**
- **Algorithmes d ’ordonnancement**
  - FCFS: simple, non optimal
  - SJF: optimal, implantation difficile
    - moyennage exponentiel
  - Priorités
  - Tourniquet: sélection du quantum
- **Évaluation des méthodes, théorie des files,**
  - formule de Little

Synchronisation de Processus
(ou threads, ou fils ou tâches)

Chapitre 7



http://w3.uqo.ca/luigi/

Problèmes avec concurrence = parallélisme

- **Les threads concurrents doivent parfois partager données (fichiers ou mémoire commune) et ressources**
  - On parle donc de tâches *coopératives*
- **Si l'accès n'est pas contrôlé, le résultat de l'exécution du programme pourra dépendre de l'ordre d'entrelacement de l'exécution des instructions (*non-déterminisme*).**
- **Un programme pourra donner des résultats différents et parfois indésirables de fois en fois**

Un exemple

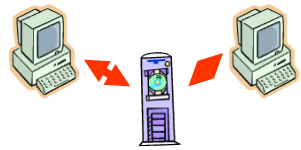
- **Deux threads exécutent cette même procédure et partagent la même base de données**
- **Ils peuvent être interrompus n'importe où**
- **Le résultat de l 'exécution concurrente de P1 et P2 dépend de l`ordre de leur *entrelacement***

M. X demande une réservation d'avion

Base de données dit que fauteuil A est disponible

Fauteuil A est assigné à X et marqué occupé

Vue globale d'une exécution possible



P1

P2

Interruption
ou retard

M. Leblanc demande une réservation d'avion

M. Guy demande une réservation d'avion

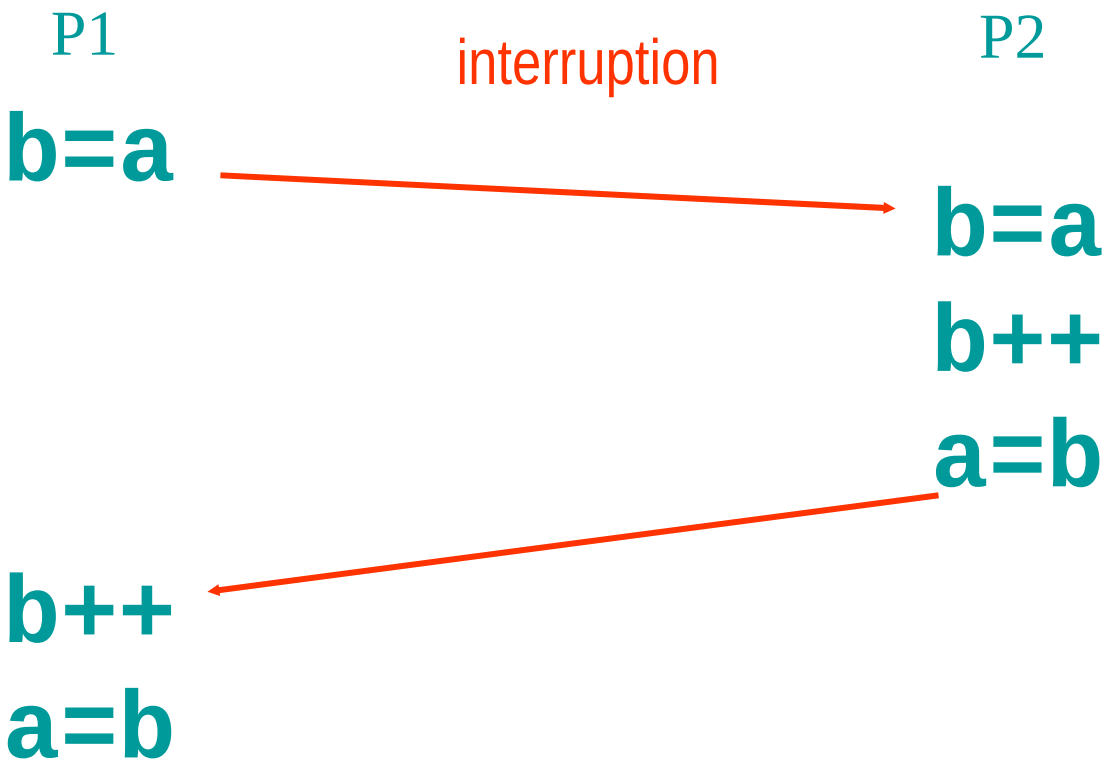
Base de données dit que fauteuil 30A est disponible

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Leblanc et marqué occupé

Fauteuil 30A est assigné à Guy et marqué occupé

Deux opérations en parallèle sur une var a partagée (b est privé à chaque processus)



Supposons que a soit 0 au début *(il pourrait être un compteur d'accès à une page web)*

P1 travaille sur le vieux a donc le résultat final sera a=1.

Sera a=2 si les deux tâches sont exécutées l'une après l'autre

Si a était sauvegardé quand P1 est interrompu, il ne pourrait pas être partagé avec P2 (il y aurait deux a tandis que nous en voulons une seule)

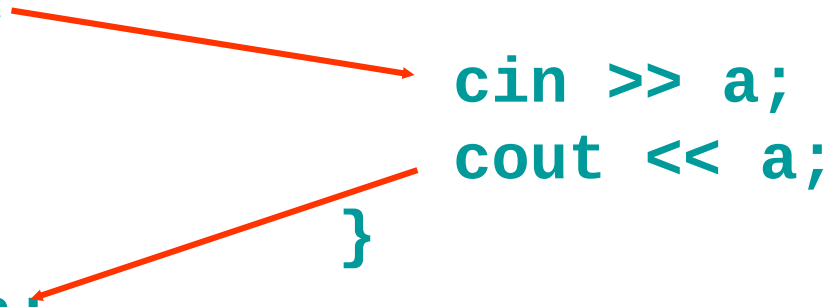
3ème exemple

Thread P1

```
static char a;

void echo()
{
   cin >> a;

   cout << a;
}
```

Thread P2

```
static char a;

void echo()
{
   cin >> a;
   cout << a;
}
```



Si la var a est partagée, le premier a est effacé
Si elle est privée, l'ordre d'affichage est renversé

Autres exemples

- **Des threads qui travaillent en simultanéité sur une matrice, par ex. un pour la mettre à jour, l`autre pour en extraire des statistiques**
- **Problème qui affecte le programme du *tampon borné*, v. manuel**
- **Quand plusieurs threads exécutent en parallèle, nous ne pouvons pas faire d'hypothèses sur la vitesse d'exécution des threads, ni leur entrelacement**
  - Peuvent être différents à chaque exécution du programme

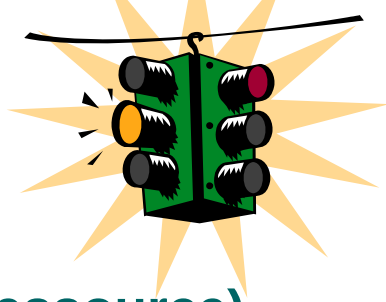
Section Critique

- **Partie d'un programme dont l'exécution de doit pas *entrelacer* avec autres programmes**

- **Une fois qu'un tâche y entre, il faut lui permettre de terminer cette section sans permettre à autres tâches de jouer sur les mêmes données**

  - La section critique doit être verrouillée

Le problème de la section critique



- **Lorsqu'un thread manipule une donnée (ou ressource) partagée, nous disons qu'il se trouve dans une section critique (SC) (associée à cette donnée)**
- **Le problème de la section critique est de trouver un algorithme d`*exclusion mutuelle* de threads dans l`exécution de leur SCs afin que le résultat de leurs actions *ne dépendent pas* de *l'ordre d'entrelacement* de leur exécution (avec un ou plusieurs processeurs)**
- **L'exécution des sections critiques doit être mutuellement exclusive: à tout instant, un seul thread peut exécuter une SC pour une var donnée (même lorsqu'il y a plusieurs processeurs)**
- **Ceci peut être obtenu en plaçant des instructions spéciales dans les sections d`entrée et sortie**
- **Pour simplifier, dorénavant nous faisons l'hypothèse qu'il n'y a q'une seule SC dans un programme.**

Structure du programme

- **Chaque thread doit donc demander une permission avant d'entrer dans une section critique (SC)**
- **La section de code qui effectue cette requête est la section d'entrée**
- **La section critique est normalement suivie d'une section de sortie**
- **Le code qui reste est la section restante (SR): non-critique**

```
repeat
   section d'entrée
    section critique
   section de sortie
    section restante
forever
```

Application

M. X demande une réservation d'avion

Section d'entrée

Section critique

Base de données dit que fauteuil A est disponible

Fauteuil A est assigné à X et marqué occupé

Section de sortie

Critères nécessaires pour solutions valides

- **Exclusion Mutuelle**
  - À tout instant, au plus un thread peut être dans une section critique (SC) pour une variable donnée
- **Non interférence:**
  - Si un thread s'arrête dans sa section restante, ceci ne devrait pas affecter les autres threads

- **Mais on fait l 'hypothèse qu'un thread qui entre dans une section critique, en sortira.**

Critères nécessaires pour solutions valides

- **Progrès:**
  - absence d`interblocage (Chap 8)
  - si un thread demande d`entrer dans une section critique à un moment où aucun autre thread en fait requête, il devrait être en mesure d'y entrer
    - Donc si un thread veut entrer dans la SC à répétition, et les autres ne sont pas intéressés, il doit pouvoir le faire

Aussi nécessaire

- **Absence de famine: aucun thread éternellement empêché d'atteindre sa SC**
- **Difficile à obtenir, nous verrons…**

Types de solutions

- **Solutions par logiciel**
  - des algorithmes dont la validité ne s'appuie pas sur l'existence d`instruction spéciales
- **Solutions fournies par le matériel**
  - s'appuient sur l'existence de certaines instructions (du processeur) spéciales
- **Solutions fournies pas le SE**
  - procure certains appels du système au programmeur
- **Toutes les solutions se basent sur l'atomicité de l'accès à la mémoire centrale: une adresse de mémoire ne peut être affectée que par une instruction à la fois, donc par un thread à la fois.**
- ***Plus en général, toutes les solutions se basent sur l 'existence d'instructions atomiques, qui fonctionnent comme SCs de base***

Atomicité = indivisibilité

Solutions par logiciel

(pas pratiques, mais intéressantes pour comprendre le pb)

- **Nous considérons d'abord 2 threads**
  - Algorithmes 1 et 2 ne sont pas valides
    - Montrent la difficulté du problème
  - Algorithme 3 est valide (algorithme de Peterson)

- **Notation**
  - Débutons avec 2 threads: T0 et T1
  - Lorsque nous discutons de la tâche Ti, Tj dénotera toujours l'autre tâche (i != j)

Algorithme 1: threads se donnent mutuellement le tour

- La variable partagée turn est initialisée à 0 ou 1
- La SC de Ti est exécutée ssi turn = i
- Ti est occupé à attendre si Tj est dans SC.
- Fonctionne pour l'exclusion mutuelle!
- Mais critère du progrès n'est pas satisfait car l'exécution des SCs doit strictement alterner

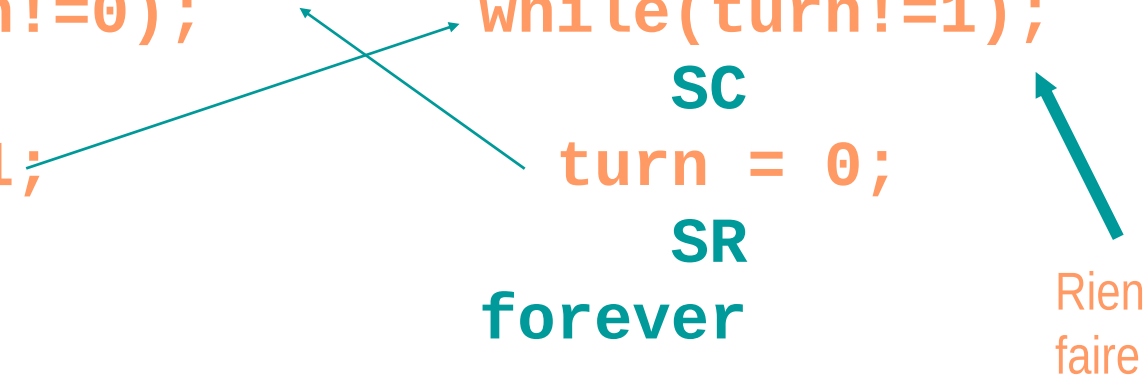
```
Thread Ti:
repeat
  while(turn!=i);
     SC
  turn = j;
     SR
forever
```

Rien faire

T0→T1→T0→T1… même si l'un des deux n'est pas intéressé du tout

initialisation de turn à 0 ou 1

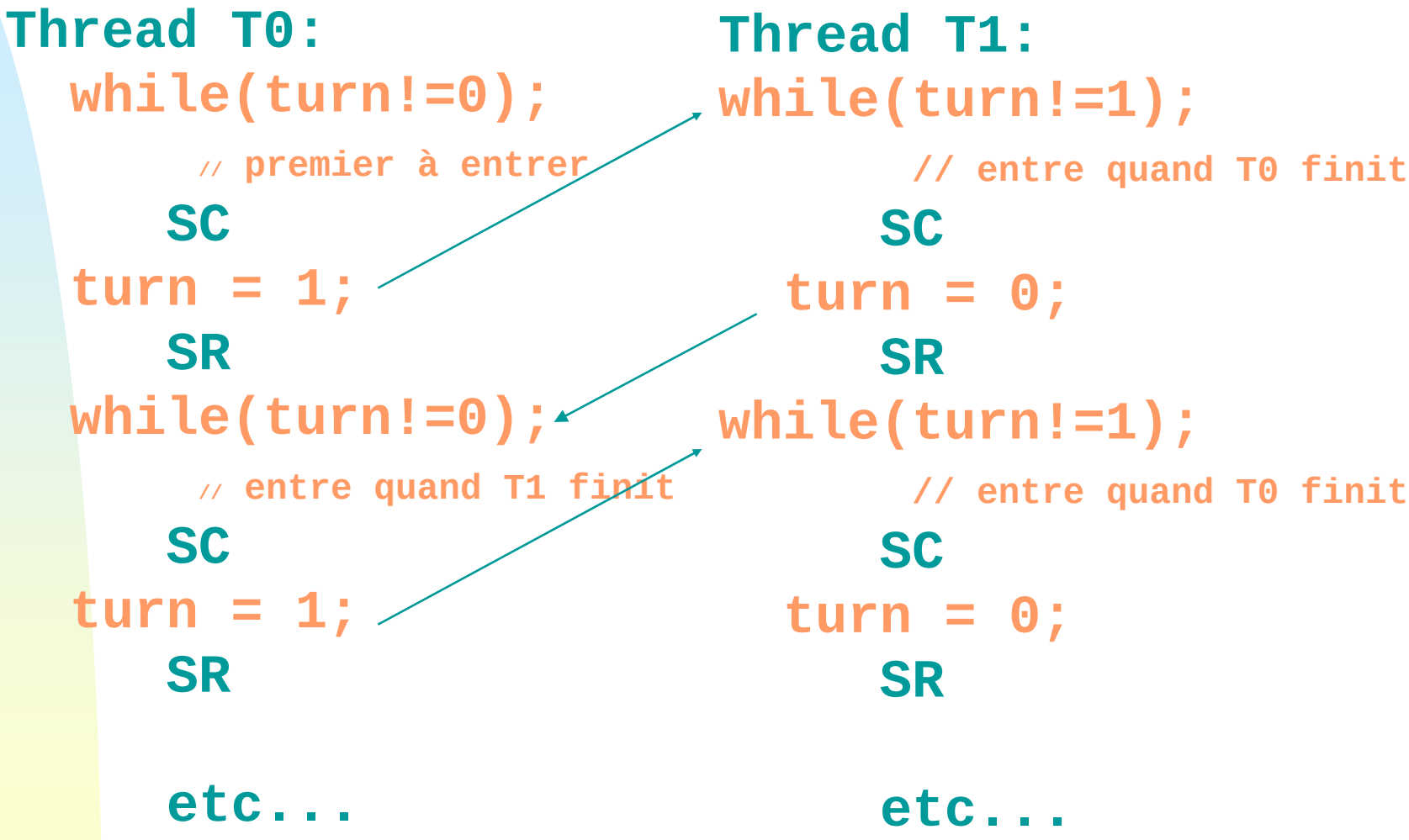
```
Thread T0:                    Thread T1:
repeat                        repeat

while(turn!=0);               while(turn!=1);
     SC                            SC
  turn = 1;                     turn = 0;
     SR                            SR
forever                       forever
```



Algorithme 1 vue globale

Ex 2: Généralisation à n threads: chaque fois, avant qu'un thread puisse rentrer dans sa section critique, il lui faut attendre que tous les autres aient eu cette chance!

Exemple: supposez que turn=0 au début

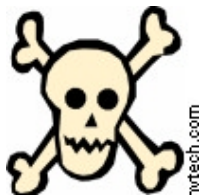


Algorithme 2 ou l'excès de courtoisie...

- **Une variable Booléenne par Thread: flag[0] et flag[1]**
- **Ti signale qu'il désire exécuter sa SC par: flag[i] =vrai**
- **Mais il n'entre pas si l'autre est aussi intéressé!**
- **Exclusion mutuelle ok**
- **Progrès pas satisfait:**
- **Considérez la séquence:**
  - T0: flag[0] = vrai
  - T1: flag[1] = vrai
    - Chaque thread attendra indéfiniment pour exécuter sa SC: on a un *interblocage*

```
Thread Ti:
repeat
   flag[i] = vrai;
   while(flag[j]);
      SC
   flag[i] = faux;
      SR
forever
```

rien faire



Après vous, monsieur

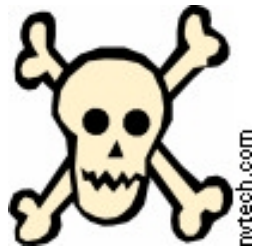
Après vous, monsieur

```
Thread T0:
repeat
   flag[0] = vrai;
   while(flag[1]);
      SC
   flag[0] = faux;
      SR
forever
```

```
Thread T1:
repeat
   flag[1] = vrai
   while(flag[0]);
      SC
   flag[1] = faux;
      SR
forever
```

Algorithme 2 vue globale

T0: flag[0] = vrai
T1: flag[1] = vrai
interblocage!



Algorithme 3 (dit de Peterson): bon!

combine les deux idées: flag[i]=intention d'entrer; turn=à qui le tour

- **Initialisation:**
  - flag[0] = flag[1] = faux
  - turn = i ou j
- **Désire d'exécuter SC est indiqué par flag[i] = vrai**
- **flag[i] = faux à la section de sortie**

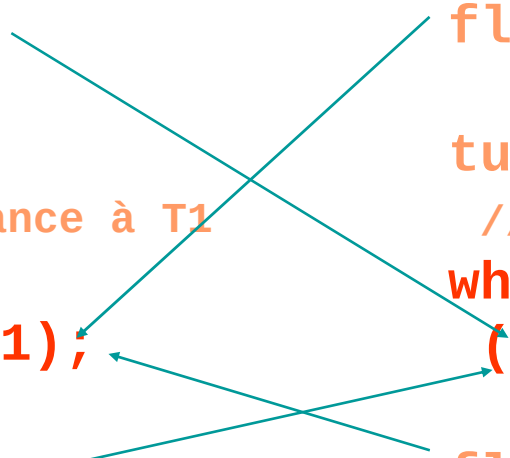
```
Thread Ti:
repeat
   flag[i] = vrai;
      // je veux entrer
   turn = j;
      // je donne une chance à l'autre
   do while
    (flag[j] && turn==j);
      SC
   flag[i] = faux;
      SR
forever
```

Entrer ou attendre?

- **Thread Ti attend si:**
  - Tj veut entrer est c'est la chance à Tj
    - `flag[j]==vrai et turn==j`
- **Un thread Ti entre si:**
  - Tj ne veut pas entrer ou c'est la chance à Ti
    - `flag[j]==faux ou turn==i`
- **Pour entrer, un thread dépend de l'autre qu'il lui donne la chance!**

```
Thread T0:
repeat
   flag[0] = vrai;
     // T0 veut entrer
   turn = 1;
    // T0 donne une chance à T1
   while
    (flag[1]&&turn=1);
      SC
   flag[0] = faux;
    // T0 ne veut plus entrer
      SR
forever
```

```
Thread T1:
repeat
   flag[1] = vrai;
     // T1 veut entrer
   turn = 0;
    // T1 donne une chance à 0
   while
    (flag[0]&&turn=0);
      SC
   flag[1] = faux;
    // T1 ne veut plus entrer
      SR
forever
```



Algorithme de Peterson vue globale

Scénario pour le changement de contrôle

```
Thread T0:                          Thread T1:

    ...                                 ...
    SC                              flag[1] = vrai;
flag[0] = faux;                       // T1 veut entrer
 // T0 ne veut plus entrer          turn = 0;
    SR                               // T1 donne une chance à T0
                                    while
    ...                            (flag[0]&&turn=0) ;
                                      //test faux, entre (F&&V)

                                        ...
```

T1 prend la relève, donne une chance à T0 mais T0 a dit qu'il ne veut pas entrer.
T1 entre donc dans la SC

Autre scénario de changem. de contrôle

Thread T0:

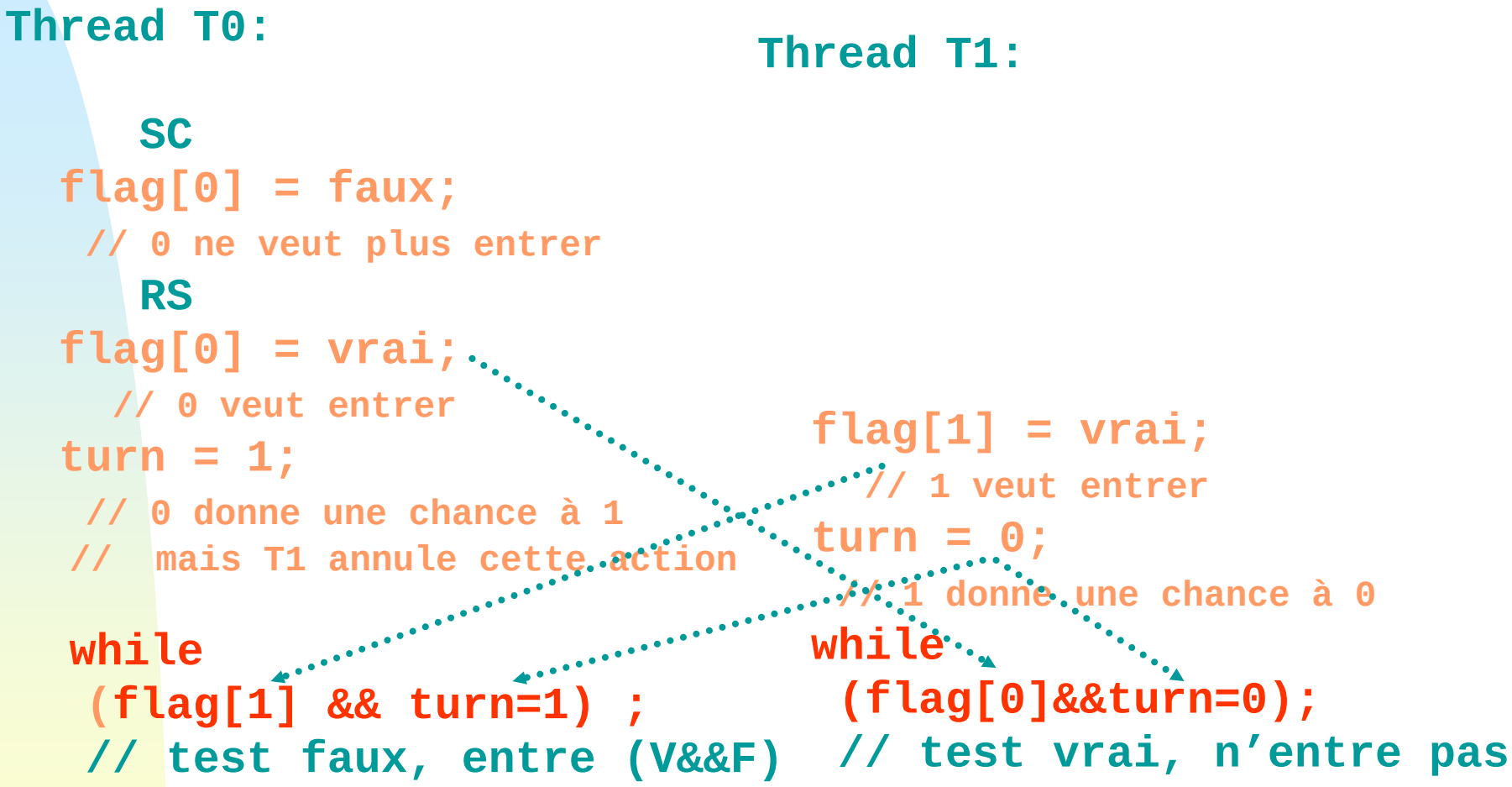
```
    SC
flag[0] = faux;
 // T0 ne veut plus entrer
    SR
flag[0] = vrai;
  // T0 veut entrer
turn = 1;
  // T0 donne une chance à T1
while
  (flag[1]==vrai&&turn=1) ;
   // test vrai, n'entre pas
       (V&&V)
```

Thread T1:

```
flag[1] = vrai;
  // T1 veut entrer
turn = 0;
// T1 donne une chance à T0
 // mais T0 annule cette action

while
 (flag[0]&&turn=0) ;
   //test faux, entre
       (V&&F)
```

T0 veut rentrer mais est obligé de donner une chance à T1, qui entre

Mais avec un petit décalage, c'est encore T0!

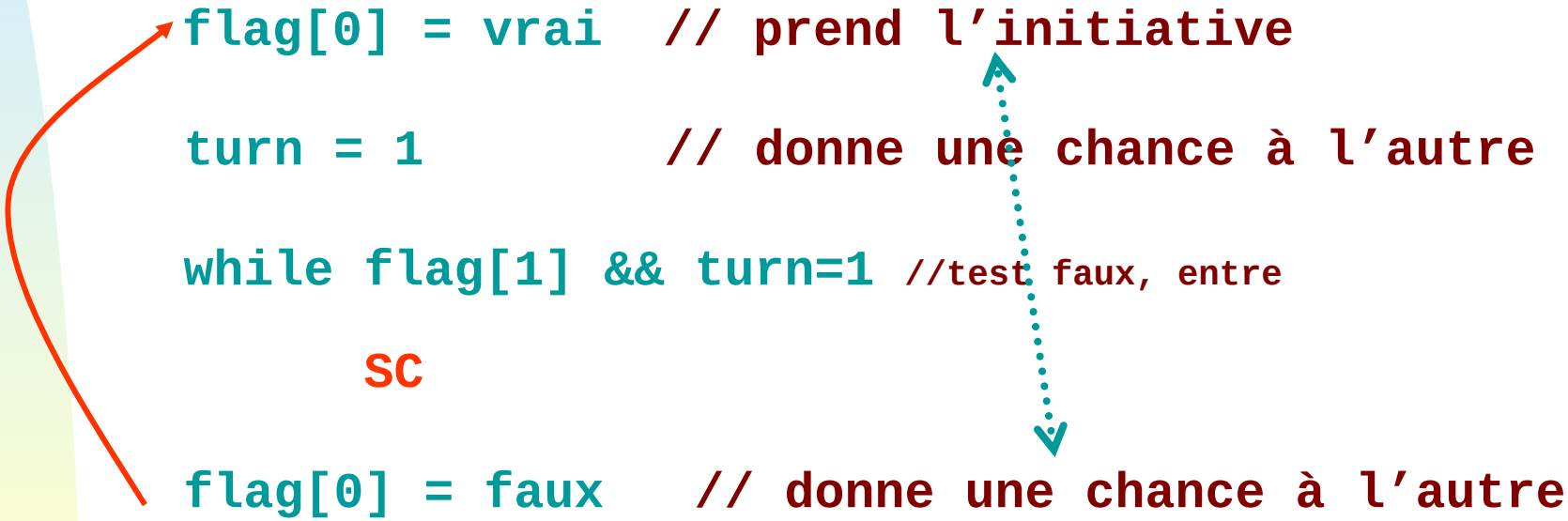


Si T0 et T1 tentent simultanément d'entrer dans SC, seule une valeur pour turn survivra:

non-déterminisme (on ne sait pas qui gagnera), mais l'exclusion fonctionne

Donc cet algo. n'oblige pas une tâche d'attendre pour d'autres qui pourraient ne pas avoir besoin de la SC

Supposons que T0 soit le seul à avoir besoin de la SC, ou que T1 soit lent à agir: T0 peut rentrer de suite (**`flag[1]==faux`** la dernière fois que T1 est sorti)



Cette propriété est désirable, mais peut causer *famine* pour T1

Algorithme 3: preuve de validité
(pas matière d'examen, seulement pour les intéressés...)

- **Exclusion mutuelle est assurée car:**
  - T0 et T1 sont tous deux dans SC seulement si turn est simultanément égal à 0 et 1 (impossible)

- **Démontrons que progrès et attente limitée sont satisfaits:**
  - Ti ne peut pas entrer dans SC seulement si en attente dans la boucle while() avec condition: flag[j] == vrai and turn = j.
  - Si Tj ne veut pas entrer dans SC alors flag[j] = faux et Ti peut alors entrer dans SC

Algorithme 3: preuve de validité (cont.)

- Si Tj a effectué flag[j]=vrai et se trouve dans le while(), alors turn==i ou turn==j
- Si
  - turn==i, alors Ti entre dans SC.
  - turn==j alors Tj entre dans SC mais il fera flag[j] =false à la sortie: permettant à Ti d'entrer CS
- mais si Tj a le temps de faire flag[j]=true, il devra aussi faire turn=i
- Puisque Ti ne peut modifier turn lorsque dans le while(), Ti entrera SC après au plus une entrée dans SC par Tj (attente limitée)

A propos de l'échec des threads

- **Si une solution satisfait les 3 critères (EM, progrès et attente limitée), elle procure une robustesse face à l'échec d'un thread dans sa section restante (SR)**
  - un thread qui échoue dans sa SR est comme un thread qui ne demande jamais d'entrer...
- **Par contre, aucune solution valide ne procure une robustesse face à l'échec d'un thread dans sa section critique (SC)**
  - un thread Ti qui échoue dans sa SC n'envoie pas de signal aux autres threads: pour eux Ti est encore dans sa SC...

Extension à >2 threads

- **L ’algorithme de Peterson peut être généralisé au cas de >2 threads**
- **Cependant, dans ce cas il y a des algorithmes plus élégants, comme l’algorithme du boulanger, basée sur l’idée de ‘prendre un numéro au comptoir’...**
  - Pas le temps d’en parler…

Une leçon à retenir…

- **À fin que des threads avec des variables partagées puissent réussir, il est nécessaire que tous les threads impliqués utilisent le même algorithme de coordination**
  - Un protocole commun

Critique des solutions par logiciel

- **Difficiles à programmer! Et à comprendre!**
  - Les solutions que nous verrons dorénavant sont toutes basées sur l'existence d'instructions spécialisées, qui facilitent le travail.

- **Les threads qui requièrent l'entrée dans leur SC sont occupés à attendre (busy waiting); consommant ainsi du temps de processeur**
  - Pour de longues sections critiques, il serait préférable de bloquer les threads qui doivent attendre...

Solutions matérielles: désactivation des interruptions

- **Sur un uniprocesseur: exclusion mutuelle est préservée mais l'efficacité se détériore: lorsque dans SC il est impossible d'entrelacer l'exécution avec d'autres threads dans une SR**
- **Perte d'interruptions**
- **Sur un multiprocesseur: exclusion mutuelle n'est pas préservée**
- **Pas bon en général**

```
Process Pi:
repeat
   désactiver interrupt
    section critique
   rétablir interrupt
    section restante
forever
```

Solutions matérielles: instructions machine spécialisées

- **Normal: pendant qu'un thread ou processus fait accès à une adresse de mémoire, aucun autre ne peut faire accès à la même adresse en même temps**
- **Extension: instructions machine exécutant plusieurs actions (ex: lecture et écriture) sur la même case de mémoire de manière atomique (indivisible)**
- **Une instruction atomique ne peut être exécutée que par un thread à la fois (même en présence de plusieurs processeurs)**

L'instruction test-and-set

- Une version C de test-and-set:

```
bool testset(int& i)
{
   if (i==0) {
      i=1;
      return true;
   } else {
      return false;
   }
}
```

Instruction **atomique**!

- Un algorithme utilisant testset pour Exclusion Mutuelle:
- Variable partagée b est initialisée à 0
- Le 1er Pi qui met b à 1 entre dans SC

```
Tâche Pi:
   while testset(b)==false ;
      SC //entre quand vrai
   b=0;
      SR
```

L’instruction test-and-set (cont.)

- **Exclusion mutuelle est assurée: si Ti entre dans SC, l’autre Tj est occupé à attendre**
- **Problème: utilise encore occupé à attendre**
- **Peut procurer facilement l’exclusion mutuelle mais nécessite algorithmes plus complexes pour satisfaire les autres exigences du problème de la section critique**
- **Lorsque Ti sort de SC, la sélection du Tj qui entrera dans SC est arbitraire: pas de limite sur l’attente: possibilité de famine**

Instruction ‘Échange’

- **Certains UCTs (ex: Pentium) offrent une instruction xchg(a,b) qui interchange le contenue de a et b de manière *atomique*.**
- **Mais xchg(a,b) souffre des même lacunes que test-and-set**

Utilisation de xchg pour exclusion mutuelle (Stallings)

- **Variable *partagée* b est initialisée à 0**
- **Chaque Ti possède une variable *locale* k**
- **Le Ti pouvant entrer dans SC est celui qui trouve b=0**
- **Ce Ti exclue tous les autres en assignant b à 1**
  - Quand SC est occupée, k et b seront 1 pour un autre thread qui cherche à entrer
  - Mais k est 0 pour le thread qui est dans la SC

usage:

```
Thread Ti:
repeat
   k = 1
 while k!=0 xchg(k,b);
 SC
 xchg(k,b);
 SR
forever
```

Solutions basées sur des instructions fournies par le SE (appels du système)

- **Les solutions vues jusqu'à présent sont difficiles à programmer et conduisent à du mauvais code.**
- **On voudrait aussi qu`il soit plus facile d'éviter des erreurs communes, comme interblocages, famine, etc.**
  - Besoin d'instruction à plus haut niveau
- **Les méthodes que nous verrons dorénavant utilisent des instructions puissantes, qui sont implantées par des appels au SE (system calls)**

Sémaphores

- **Un sémaphore S est un entier qui, sauf pour l'Initialisation, est accessible seulement par ces 2 opérations atomiques et mutuellement exclusives:**
  - wait(S) (appelé P dans le livre)
  - signal(S) (appelé V dans le livre)
- **Il est partagé entre tous les procs qui s`intéressent à la même section critique**
- **Les sémaphores seront présentés en deux étapes:**
  - sémaphores qui sont occupés à attendre (busy waiting)
  - sémaphores qui utilisent des files d 'attente
- **On fait distinction aussi entre sémaphores compteurs et sémaphores binaires, mais ce derniers sont moins puissants (v. livre).**

Spinlocks d'Unix: Sémaphores occupés à attendre
(busy waiting)

- La façon la plus simple d'implanter les sémaphores.
- Utiles pour des situations où l'attente est brève, ou il y a beaucoup d'UCTs
- S est un entier initialisé à une valeur positive, de façon que un premier thread puisse entrer dans la SC
- Quand S>0, jusqu'à n threads peuvent entrer
- S ne peut pas être négatif

```
wait(S):
while S=0 ;
 S--;
```

Attend si no. de threads qui peuvent entrer = 0

```
signal(S):
 S++;
```

Augmente de 1 le no des threads qui peuvent entrer

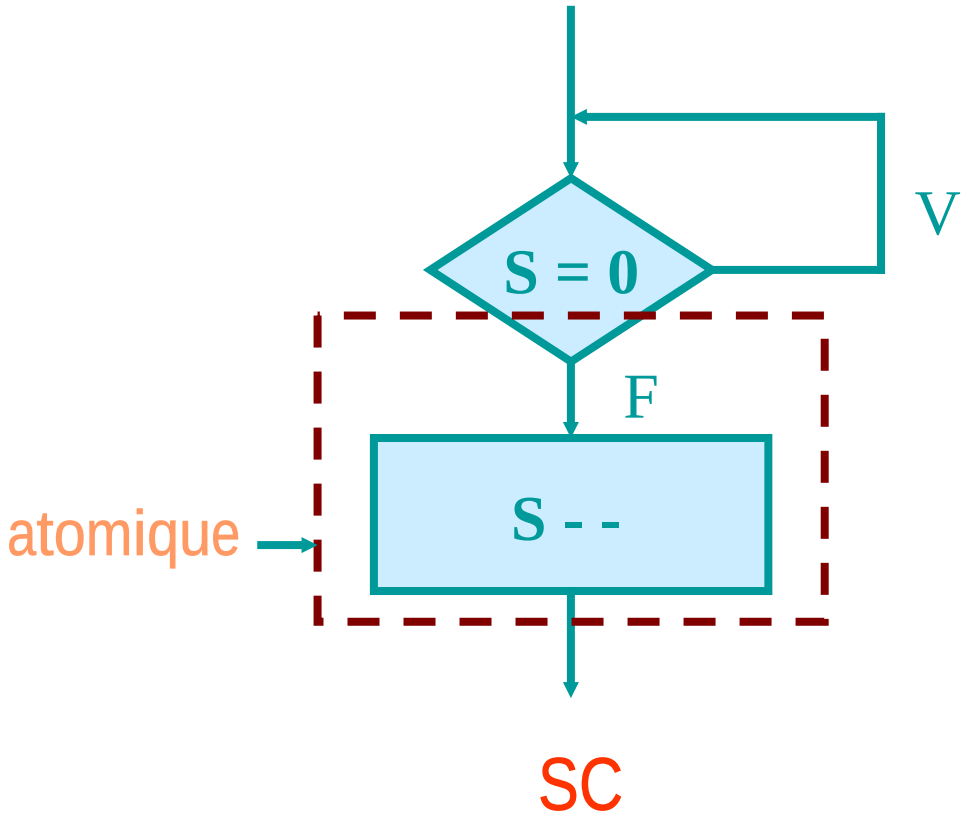
Atomicité

Wait: La séquence test-décrément est atomique, mais pas la boucle!

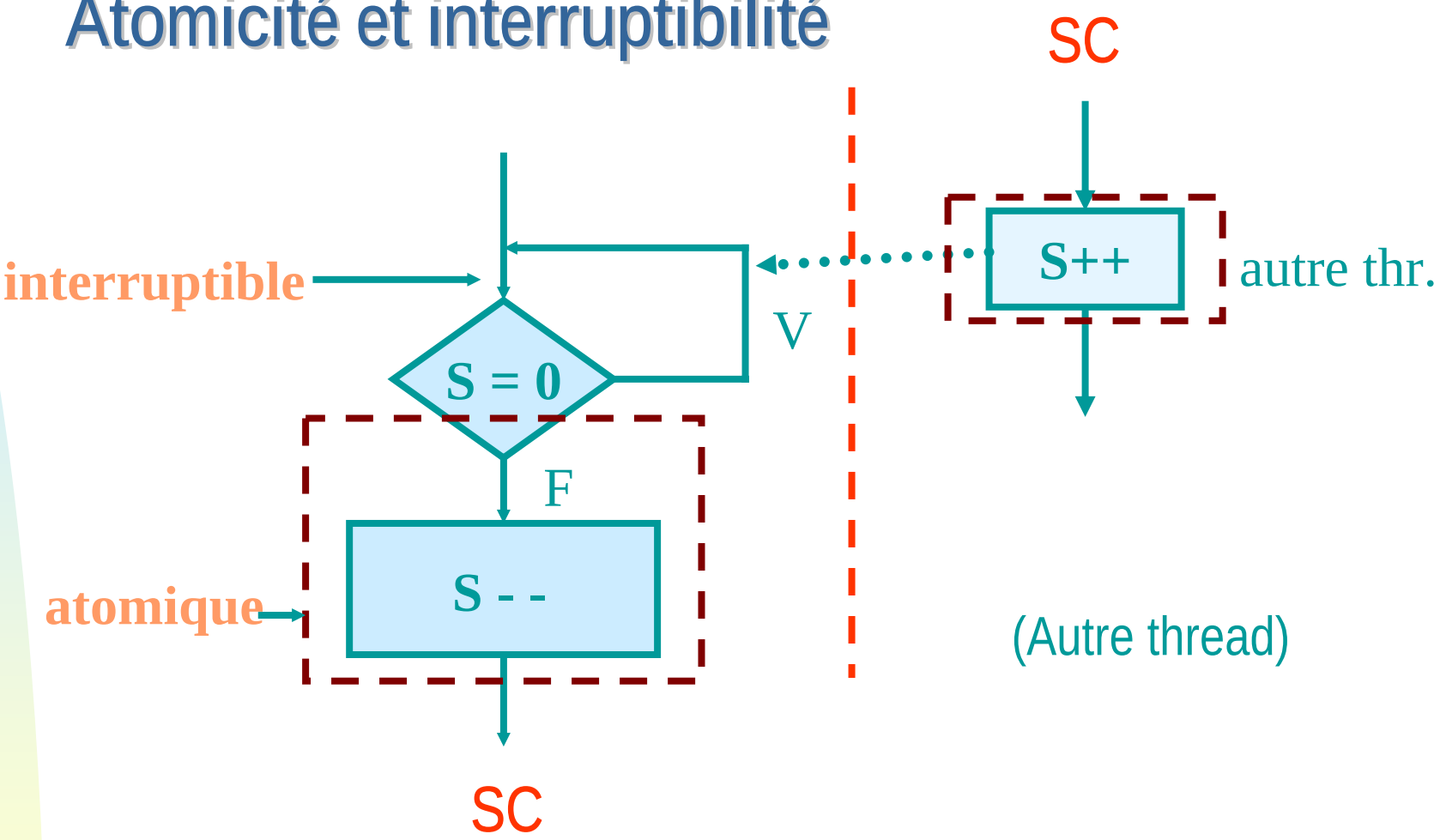
Signal est atomique.

Rappel: les sections atomiques ne peuvent pas être exécutées simultanément par différent threads

(ceci peut être obtenu un utilisant un des mécanismes précédents)

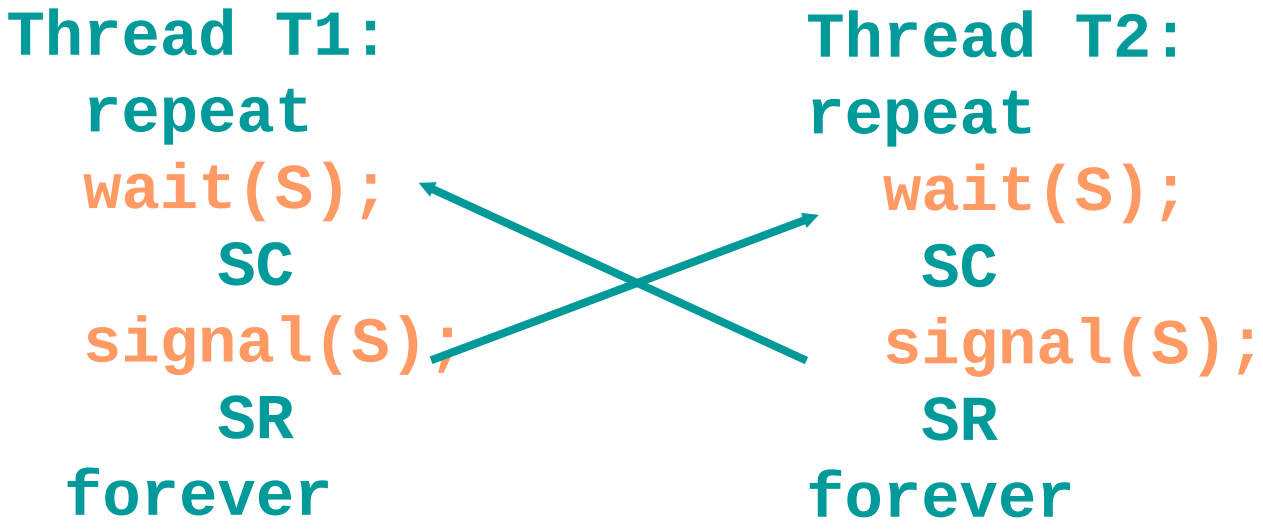


Atomicité et interruptibilité



La boucle n'est pas atomique pour permettre à un autre thread d'interrompre l'attente sortant de la SC

Initialise S à >=1



Semaphores: vue globale

Peut être facilement généralisé à plus. threads

Utilisation des sémaphores pour sections critiques

- Pour n threads
- Initialiser S à 1
- Alors 1 seul thread peut être dans sa SC
- Pour permettre à k threads d'exécuter SC, initialiser S à k

```
Thread Ti:
repeat
   wait(S);
    SC
   signal(S);
    SR
forever
```

Utilisation des sémaphores pour synchronisation de threads

- **On a 2 threads: T1 et T2**
- **Énoncé S1 dans T1 doit être exécuté avant énoncé S2 dans T2**
- **Définissons un sémaphore S**
- **Initialiser S à 0**

- **Synchronisation correcte lorsque T1 contient:**

```
S1;
signal(S);
```

- **et que T2 contient:**

```
wait(S);
S2;
```

Interblocage et famine avec les sémaphores

- **Famine: un thread peut n'arriver jamais à exécuter car il ne teste jamais le sémaphore au bon moment**
- **Interblocage: Supposons S et Q initialisés à 1**

T0 T1

wait(S)

wait(Q)

wait(Q) ⟷ wait(S)

Sémaphores: observations

```
wait(S):
while S=0 ;
 S--;
```

- **Quand S >= 0:**
  - Le nombre de threads qui peuvent exécuter wait(S) sans devenir bloqués = S
    - S threads peuvent entrer dans la SC
    - noter puissance par rapport à mécanismes déjà vus
    - dans les solutions où S peut être >1 il faudra avoir un 2ème sém. pour les faire entrer un à la fois (excl. mutuelle)
- **Quand S devient > 1, le thread qui entre le premier dans la SC est le premier à tester S (choix aléatoire)**
  - ceci ne sera plus vrai dans la solution suivante

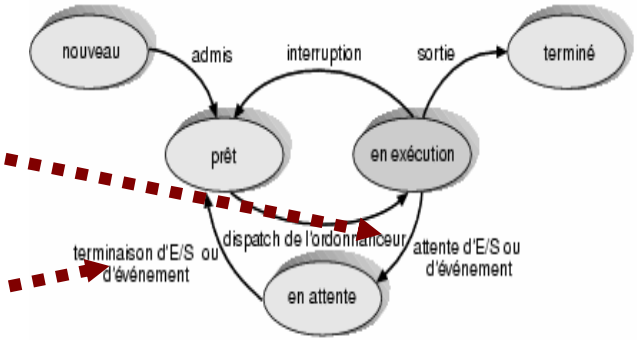
Comment éviter l'attente occupée et le choix aléatoire dans les sémaphores

- **Quand un thread doit attendre qu'un sémaphore devienne plus grand que 0, il est mis dans une file d'attente de threads qui attendent sur le même sémaphore.**
- **Les files peuvent être PAPS (FIFO), avec priorités, etc. Le SE contrôle l`ordre dans lequel les threads entrent dans leur SC.**
- ***wait* et *signal* sont des appels au système comme les appels à des opérations d'E/S.**
- **Il y a une file d 'attente pour chaque sémaphore comme il y a une file d'attente pour chaque unité d'E/S.**

Sémaphores sans attente occupée

- **Un sémaphore S devient une structure de données:**
  - Une valeur
  - Une liste d'attente L

- **Un thread devant attendre un sémaphore S, est bloqué et ajouté la file d'attente S.L du sémaphore (v. état bloqué = attente chap 4).**



- **signal(S) enlève (selon une politique juste, ex: PAPS/FIFO) un thread de S.L et le place sur la liste des threads prêts/ready.**

Implementation

(les boîtes réprésentent des séquences non-interruptibles)

```
wait(S):  S.value --;
          if S.value < 0   {        // SC occupée
                    add this thread to S.L;
                    block           // thread mis en état attente (wait)
                    }
```

```
signal(S): S.value ++;
          if S.value ≤ 0 {          // des threads attendent
                    remove a process P from S.L;
                    wakeup(P)       // thread choisi devient prêt
                    }
```

S.value doit être initialisé à une valeur non-négative (dépendant de l'application, v. exemples)

Figure montrant la relation entre le contenu de la file et la valeur de S

(Stallings)

Quand S < 0: le nombre de threads qui attendent sur S est = |S|

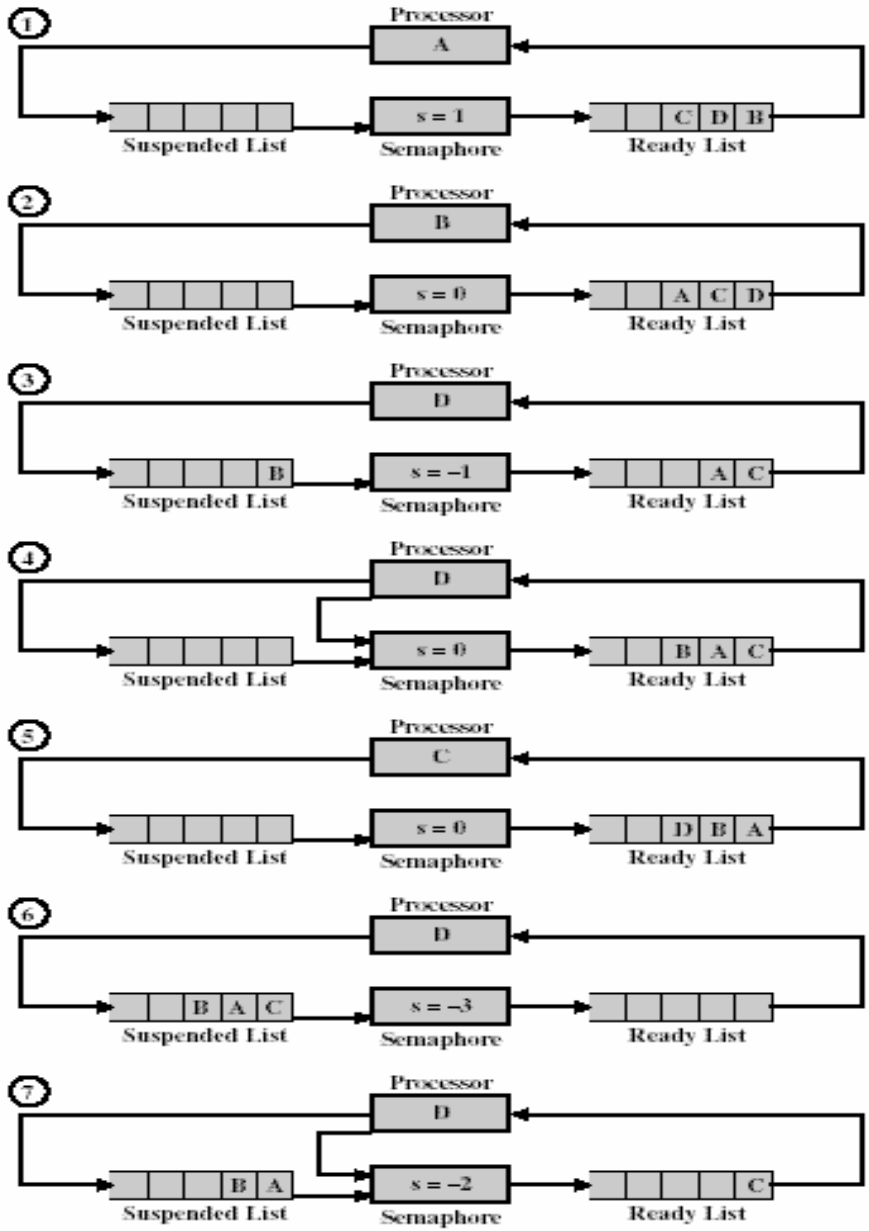


Figure 5.8 Example of Semaphore Mechanism

Wait et signal contiennent elles mêmes des SC!

- **Les opérations *wait* et *signal* doivent être exécutées atomiquement (un seul thr. à la fois)**
- **Dans un système avec 1 seule UCT, ceci peut être obtenu en inhibant les interruptions quand un thread exécute ces opérations**
- **Normalement, nous devons utiliser un des mécanismes vus avant (instructions spéciales, algorithme de Peterson, etc.)**
- **L'attente occupée dans ce cas ne sera pas trop onéreuse car *wait* et *signal* sont brefs**

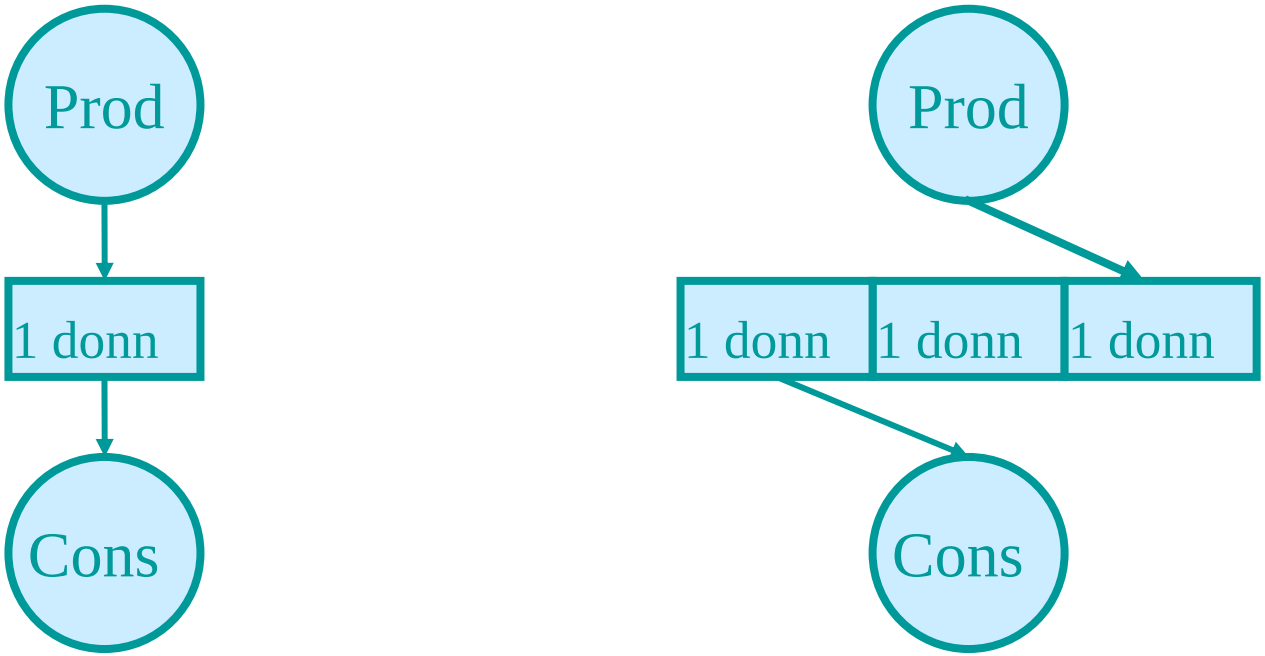
Problèmes classiques de synchronisation

- **Tampon borné (producteur-consommateur)**
- **Écrivains - Lecteurs**
- **Les philosophes mangeant**

Le pb du producteur - consommateur

- **Un problème classique dans l ’étude des threads communicants**
  - un thread *producteur* produit des données (p.ex.des enregistrements d ’un fichier) pour un thread *consommateur*

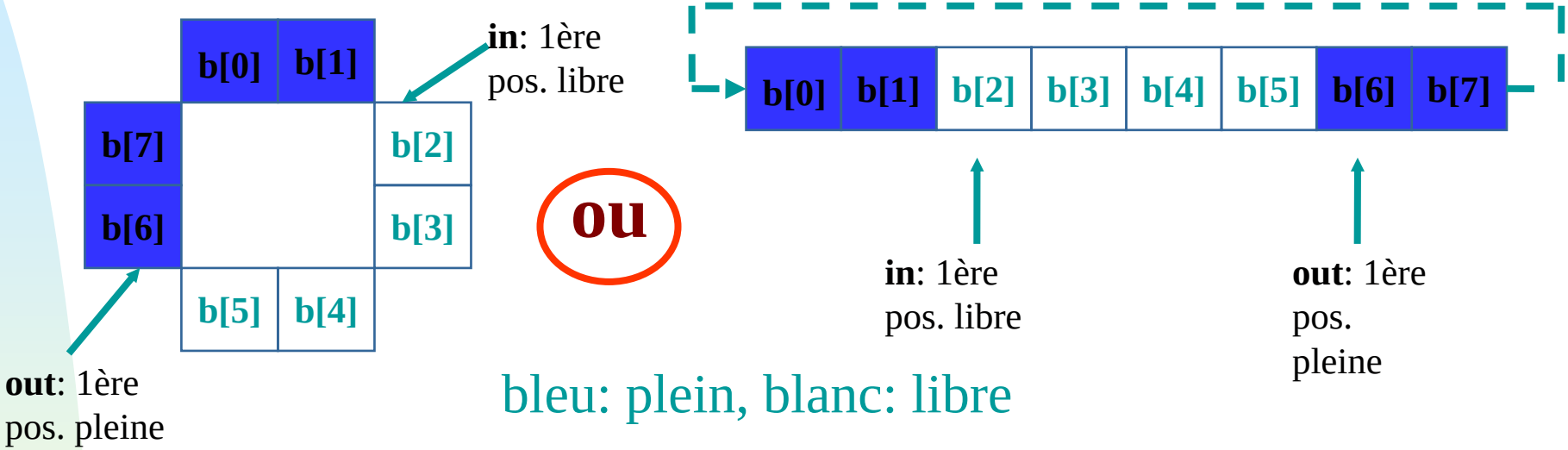
Tampons de communication



Si le tampon est de longueur 1, le producteur et consommateur doivent forcement aller à la même vitesse

Des tampons de longueur plus grandes permettent une certaine indépendance. P.ex. à droite le consommateur a été plus lent

Le tampon borné (bounded buffer)
une structure de données fondamentale dans les SE



Le tampon borné se trouve dans la mémoire partagée entre consommateur et usager

Pb de sync entre threads pour le tampon borné

- **Étant donné que le prod et le consommateur sont des threads indépendants, des problèmes pourraient se produire en permettant accès simultané au tampon**
- **Les sémaphores peuvent résoudre ce problème**

Sémaphores: rappel.

- **Soit S un sémaphore sur une SC**
  - il est associé à une file d ’attente
  - S positif: S threads peuvent entrer dans SC
  - S zéro: aucun thread ne peut entrer, aucun thread en attente
  - S négatif: |S| thread dans file d ’attente
- **Wait(S): S - -**
  - si après S >= 0, thread peut entrer dans SC
  - si S < 0, thread est mis dans file d ’attente
- **Signal(S): S++**
  - si après S<= 0, il y avait des threads en attente, et un thread est réveillé
- **Indivisibilité = atomicité de ces ops**

Solution avec sémaphores

- **Un sémaphore S pour exclusion mutuelle sur l'accès au tampon**
  - Les sémaphores suivants ne font pas l'EM
- **Un sémaphore N pour synchroniser producteur et consommateur sur le nombre d'éléments consommables dans le tampon**
- **Un sémaphore E pour synchroniser producteur et consommateur sur le nombre d'espaces libres**

Solution de P/C: tampon circulaire fini de dimension k

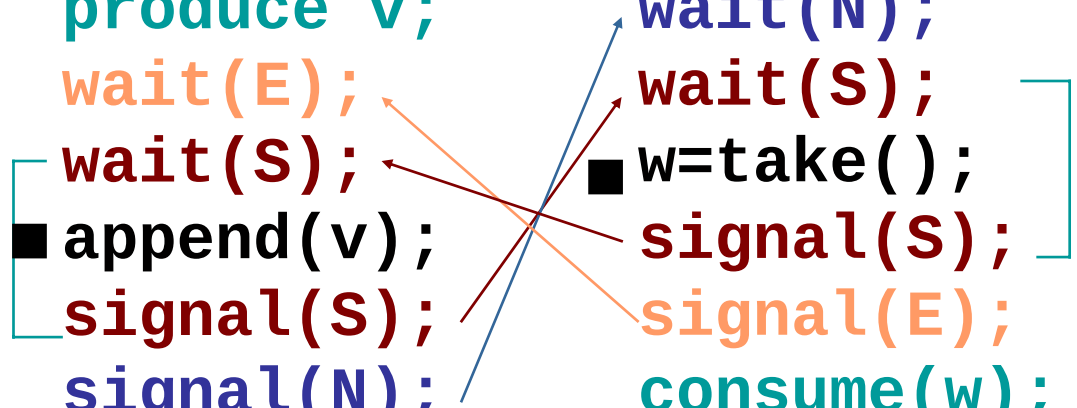
```
Initialization: S.count=1; //excl. mut.
                N.count=0; //esp. pleins
                E.count=k; //esp. vides
```

```
append(v):
  b[in]=v;
  In ++ mod k;
```

```
take():
  w=b[out];
  Out ++ mod k;
  return w;
```

```
Producer:
repeat
  produce v;
  wait(E);
  wait(S);
■ append(v);
  signal(S);
  signal(N);
forever
```

```
Consumer:
repeat
  wait(N);
  wait(S);
■ w=take();
  signal(S);
  signal(E);
  consume(w);
forever
```



■ Sections critiques

Points importants à étudier

- **dégâts possibles en inter changeant les instructions sur les sémaphores**
  - ou en changeant leur initialisation
- **Généralisation au cas de plus. prods et cons**

Concepts importants de cette partie du Chap 7

- **Le problème de la section critique**
- **L'entrelacement et l'atomicité**
- **Problèmes de famine et interblocage**
- **Solutions logiciel**
- **Instructions matériel**
- **Sémaphores occupés ou avec files**
- **Fonctionnement des différentes solutions**
- **L'exemple du tampon borné**
- **Par rapport au manuel: ce que nous n'avons pas vu en classe vous le verrez au lab**

Glossaire

- **Atomicité, non-interruptibilité:**
  - La définition précise d'atomicité, non-déterminisme etc. est un peu compliquée, et il y en a aussi des différentes… (les curieux pourraient faire une recherche Web sur ces mot clé)
  - Ce que nous discutons dans ce cours est une 1ère approximation: une séquence d'ops est ***atomique*** si elle est exécutée toujours sans être interrompue par aucune autre séquence sur les mêmes données
    - Ou son résultat ne dépend jamais de l'existence d'autres séquences en parallèle…

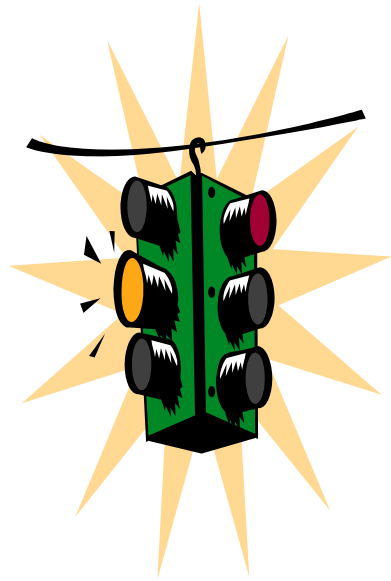
Non-déterminisme et conditions de course

- ***Non-déterminisme*: une situation dans laquelle il y a plusieurs séquences d'opérations possibles à un certain moment, même avec les mêmes données. Ces différentes séquences peuvent conduire à des résultats différents**

- ***Conditions de course*: Les situations dans lesquelles des activités exécutées en parallèle sont 'en course' les unes contre les autres pour l`accès à des ressources (variables partagées, etc.), sont appelées 'conditions de course '.**

Chapitre 7 continuation

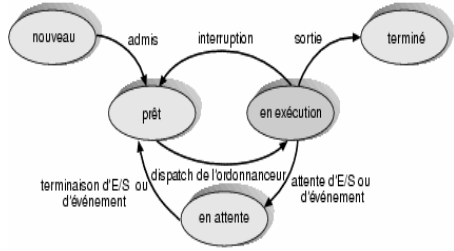
- **Problèmes classiques de synchronisation**
- **Lecteurs - Rédacteurs**
- **Les philosophes mangeant**
- **Moniteurs**
- **Threads en Java**



http://w3.uqo.ca/luigi/

Sémaphores: rappel

(les boîtes représentent des séquences indivisibles)




```
wait(S):  S.value --;
          if S.value < 0   {        // SC occupée
                ajouter ce thread à S.L;
                block               // thread mis en état attente (wait)
                }
```

```
signal(S): S.value ++;
          if S.value ≤ 0 {          // des threads attendent
                enlever un thread P de S.L;
                wakeup(P)           // thread choisi devient prêt
                }
```

S.value doit être initialisé à une valeur non-négative

dépendant de l'application, v. exemples

Sémaphores: rappel.

- **Soit S un sémaphore sur une SC**
  - il est associé à une file d ’attente
  - S positif: S thread peuvent entrer dans SC
  - S zéro: aucun thread ne peut entrer, aucun thread en attente
  - S négatif: |S| thread dans file d ’attente
- **Wait(S): S - -**
  - si après S >= 0, thread peut entrer dans SC
  - si S < 0, thread est mis dans file d ’attente
- **Signal(S): S++**
  - si après S<= 0, il y avait des threads en attente, et un thread est transféré à la file prêt
- **Indivisibilité = atomicité de wait et signal**

Problème des lecteurs - rédacteurs

- **Plusieurs threads peuvent accéder à une base de données**
  - Pour y lire ou pour y écrire
- **Les rédacteurs doivent être synchronisés entre eux et par rapport aux lecteurs**
  - il faut empêcher à un thread de lire pendant l'écriture
  - il faut empêcher à deux rédacteurs d 'écrire simultanément
- **Les lecteurs peuvent y accéder simultanément**

Une solution (n'exclut pas la famine)

- **Variable readcount: nombre de threads lisant la base de données**
- **Sémaphore mutex: protège la SC où readcount est mis à jour**
- **Sémaphore wrt: exclusion mutuelle entre rédacteurs et lecteurs**
- **Les rédacteurs doivent attendre sur wrt**
  - les uns pour les autres
  - et aussi la fin de toutes les lectures
- **Les lecteurs doivent**
  - attendre sur wrt quand il y a des rédacteurs qui écrivent
  - bloquer les rédacteurs sur wrt quand il y a des lecteurs qui lisent
  - redémarrer les rédacteurs quand personne ne lit

Les données et les rédacteurs

Données: deux sémaphores et une variable

```
mutex, wrt: semaphore (init. 1);
readcount : integer (init. 0);
```

Rédacteur

```
wait(wrt);
            . . .
     // écriture
            . . .
signal(wrt);
```

Les lecteurs

```
wait(mutex);
      readcount ++ ;
      if readcount == 1 then wait(wrt);
signal(mutex);

//SC: lecture

wait(mutex);
      readcount -- ;
      if readcount == 0 then signal(wrt);
signal(mutex):
```

Le premier lecteur d 'un groupe pourrait devoir attendre sur wrt, il doit aussi bloquer les rédacteurs. Quand il sera entré, les suivants pourront entrer librement

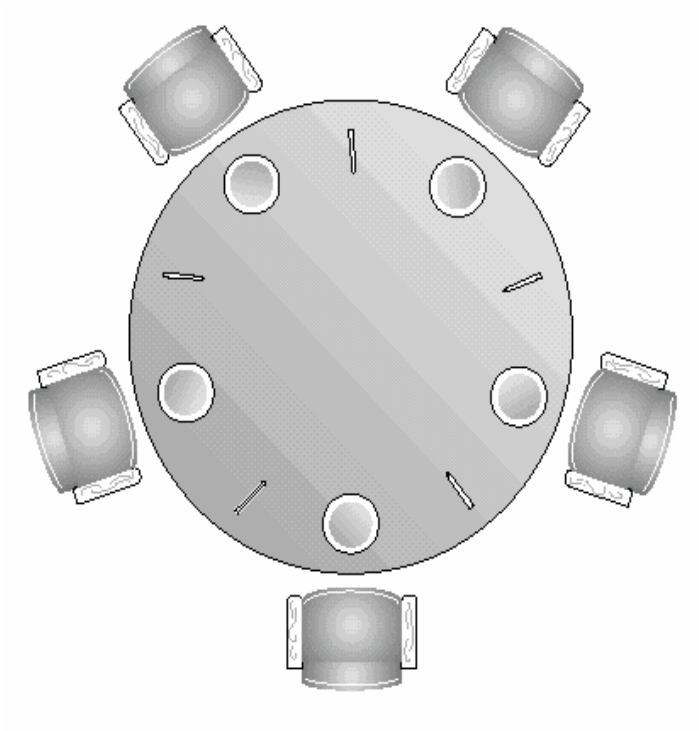
Le dernier lecteur sortant doit permettre l`accès aux rédacteurs

Observations

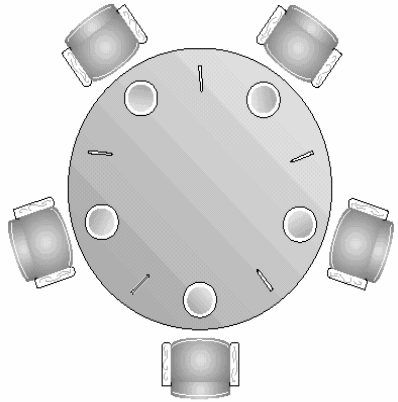
- **Le 1er lecteur qui entre dans la SC bloque les rédacteurs**
  - wait (wrt)
- **le dernier les remet en marche**
  - signal (wrt)
- **Si 1 rédacteur est dans la SC, 1 lecteur attend sur wrt, les autres sur mutex**
- **un signal(wrt) peut faire exécuter un lecteur ou un rédacteur**

Le problème des philosophes mangeant

- 5 philosophes qui mangent et pensent
- Pour manger il faut 2 fourchettes, droite et gauche
- On en a seulement 5!
- Un problème classique de synchronisation
- Illustre la difficulté d'allouer ressources aux threads tout en évitant interblocage et famine



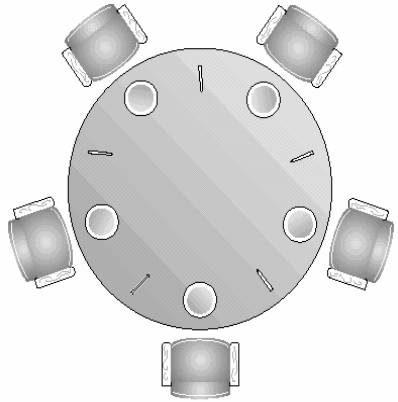
Le problème des philosophes mangeant



- **Un thread par philosophe**
- **Un sémaphore par fourchette:**
  - fork: array[0..4] of semaphores
  - Initialisation: fork[i] =1 for i:=0..4
- **Première tentative:**
  - interblocage si chacun débute en prenant sa fourchette gauche!
    - `wait(fork[i])`

```
Thread Pi:
repeat
 think;
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
forever
```

Le problème des philosophes mangeant



- **Une solution: admettre seulement 4 philosophes à la fois qui peuvent tenter de manger**
- **Il y aura touj. au moins 1 philosophe qui pourra manger**
  - même si tous prennent 1 fourchette
- **Ajout d'un sémaphore T qui limite à 4 le nombre de philosophes "assis à la table"**
  - initial. de T à 4
- **N'empêche pas famine!**

```
Thread Pi:
repeat
 think;
 wait(T);
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
 signal(T);
forever
```

Avantage des sémaphores
(par rapport aux solutions précédentes)

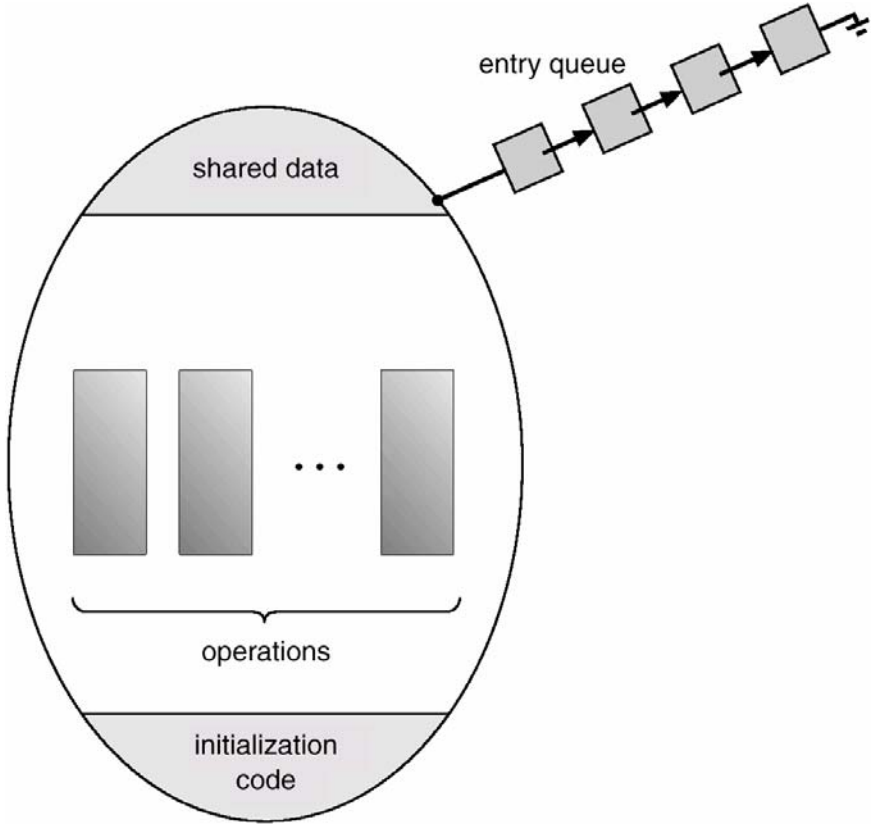
- **Une seule variable partagée par section critique**
- **deux seules opérations: wait, signal**
- **contrôle plus localisé (que avec les précéds)**
- **extension facile au cas de plus. threads**
- **possibilité de faire entrer plus. threads à la fois dans une section critique**
- **gestion de files d`attente par le SE: famine évitée si le SE est équitable (p.ex. files FIFO)**

Problème avec sémaphores: difficulté de programmation

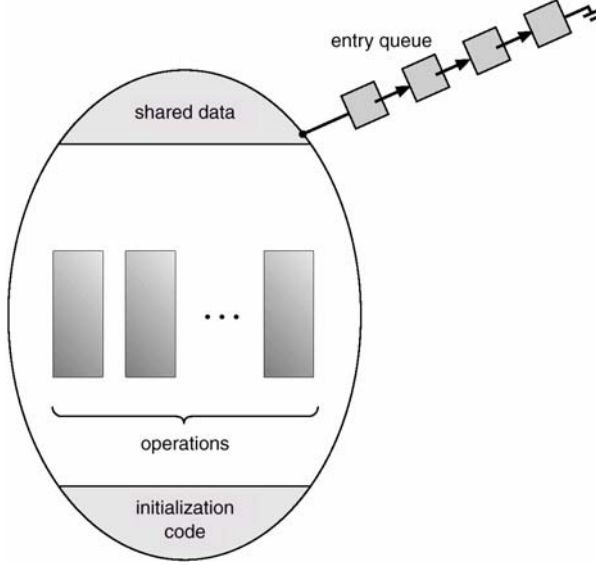
- **wait et signal sont dispersés parmi plusieurs threads, mais ils doivent se correspondre**
  - Utilisation doit être correcte dans tous les threads
- **Un seul “mauvais” thread peut faire échouer toute une collection de threads (p.ex. oublie de faire signal)**
- **Considérez le cas d`un thread qui a des waits et signals dans des boucles et des tests...**

Moniteurs: une autre solution

- **Constructions (en langage de haut-niveau) qui procurent une fonctionnalité équivalente aux sémaphores mais plus facile à contrôler**
- **Disponibles en:**
  - Concurrent Pascal, Modula-3...
    - *synchronized method* en Java (moniteurs simplifiés)

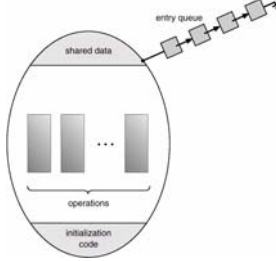


Moniteur



- **Est un module contenant:**
  - une ou plusieurs procédures
  - une séquence d'initialisation
  - variables locales
- **Caractéristiques:**
  - variables locales accessibles seulement à l'aide d'une procédure du moniteur
  - un thread entre dans le moniteur en invoquant une de ses procédures
  - *un seul thread peut exécuter dans le moniteur* à tout instant (mais plus. threads peuvent être en attente dans le monit.)

Moniteur



- **Il assure à lui seul l'exclusion mutuelle: pas besoins de le programmer explicitement**

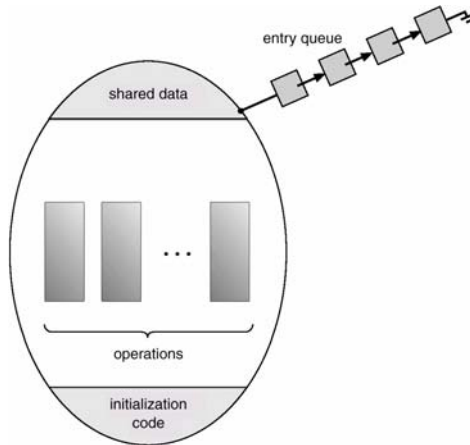
- **On assure la protection des données partagées en les plaçant dans le moniteur**
  - Le moniteur verrouille les données partagées lorsqu'un thread y entre

- **Synchronisation de threads est effectuée en utilisant des variables conditionnelles qui représentent des conditions après lesquelles un thread pourrait attendre avant d'exécuter dans le moniteur**

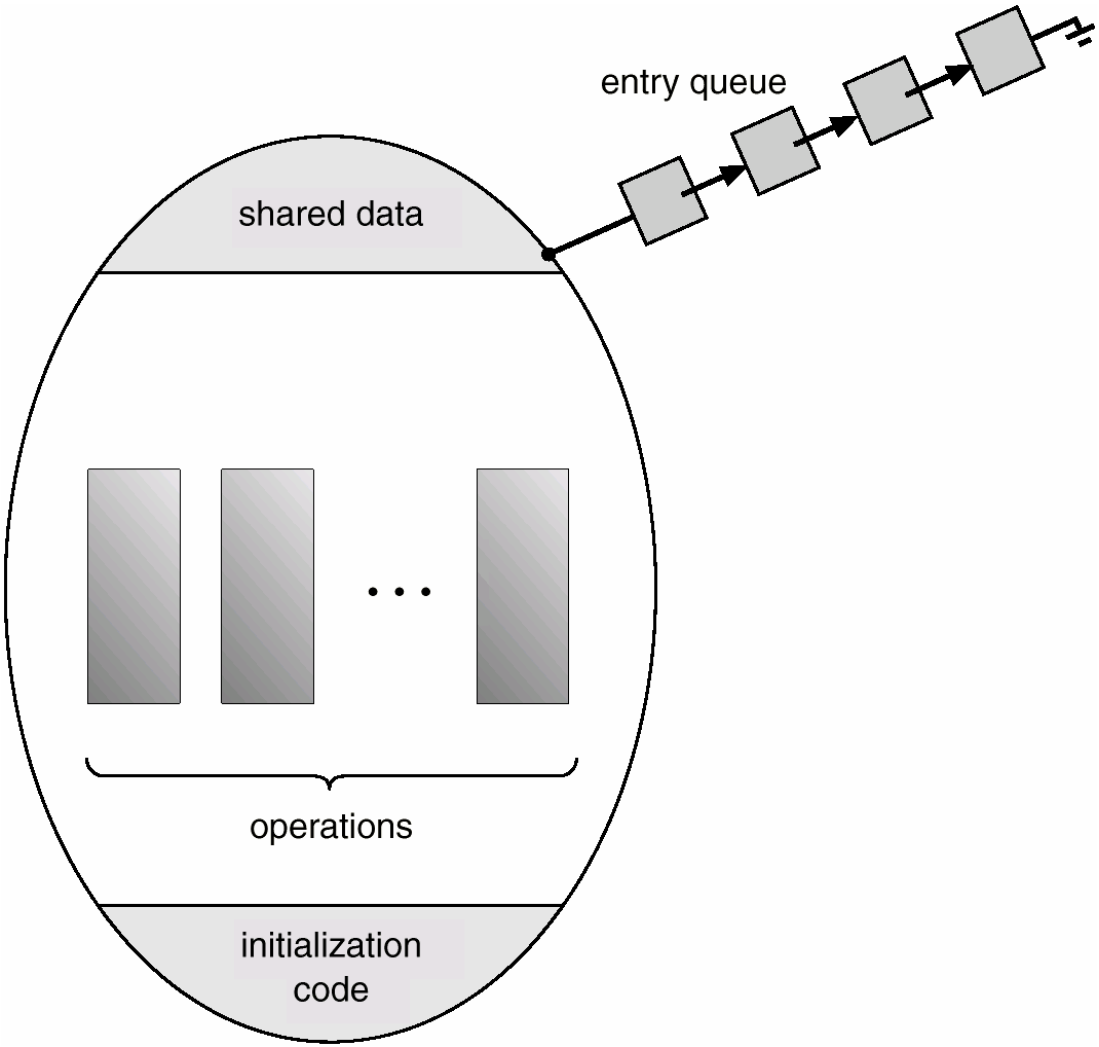
Structure générale du moniteur (style Java)

```
monitor nom-de-moniteur
{ // déclarations de vars
        public entry p1(. . .) {code de méthode p1}
        public entry p2(. . .) {code de méthode p2}
        . . .
}
```

La seule façon de manipuler les vars internes au moniteur est d'appeler une des méthodes d'entrée



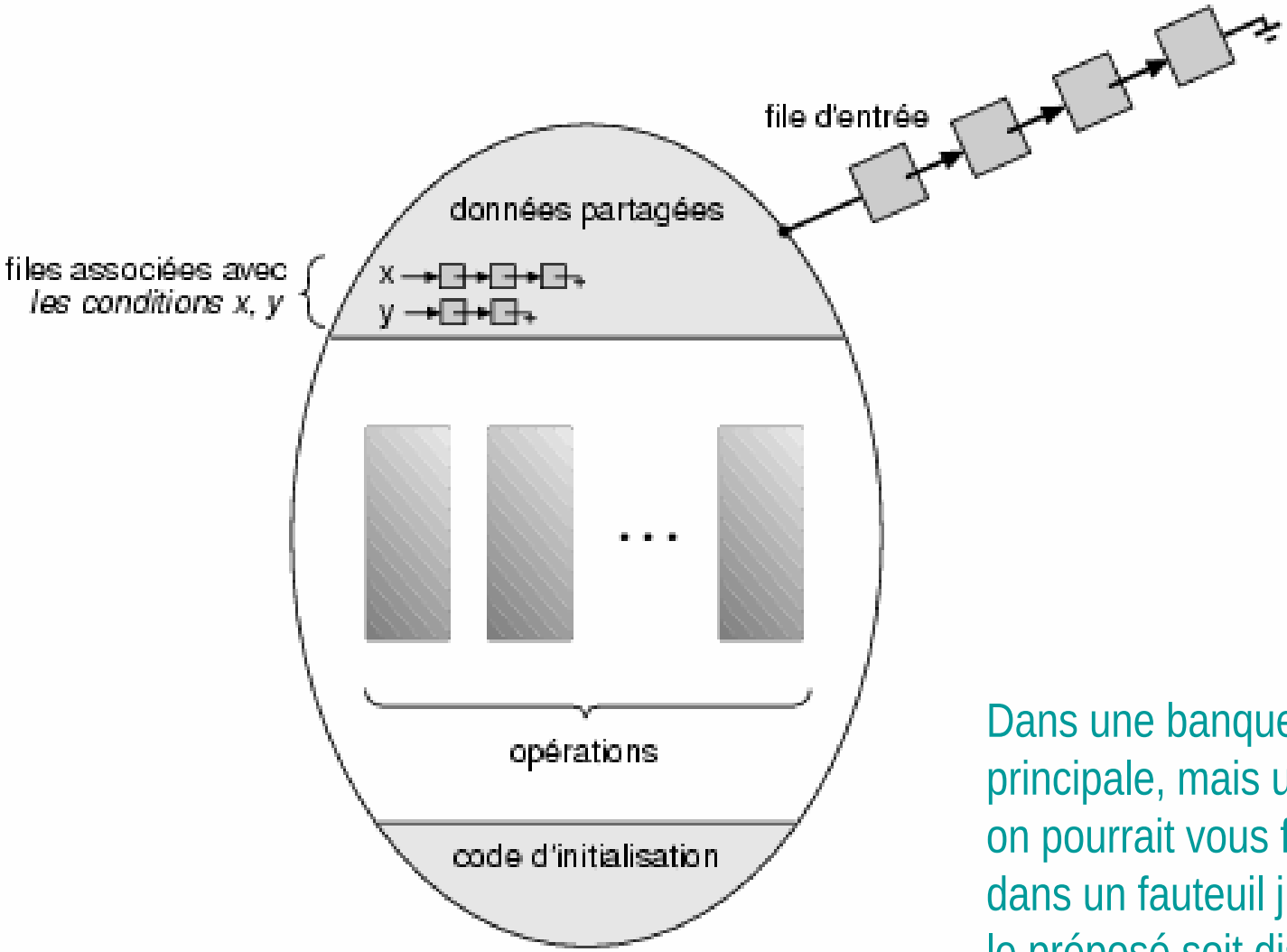
Moniteur: Vue schématique simplifiée style Java



Variables conditionnelles (n'existent pas en Java)

- **sont accessibles seulement dans le moniteur**
- **accessibles et modifiables seulement à l'aide de 2 fonctions:**
  - x: wait bloque l'exécution du thread exécutant sur la condition x
    - le thread pourra reprendre l'exécution seulement si un autre thread exécute x: signal)
  - x: signal reprend l'exécution d'un thread bloqué sur la condition x
    - S'il en existe plusieurs: en choisir un (file?)
    - S'il n'en existe pas: ne rien faire

Moniteur avec variables conditionnelles



Dans une banque, il y a une file principale, mais une fois entré on pourrait vous faire attendre dans un fauteuil jusqu'à ce que le préposé soit disponible

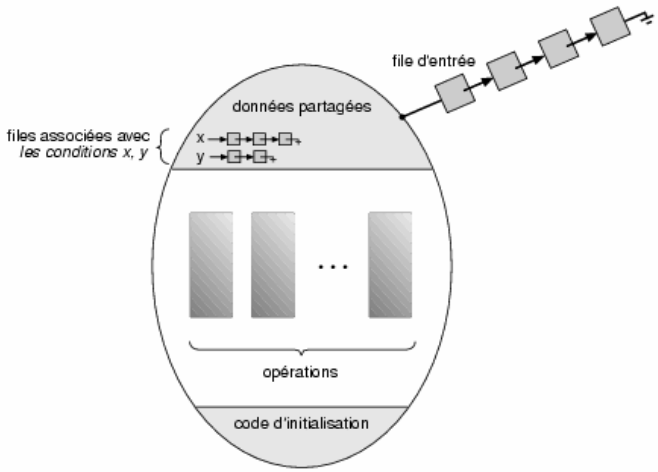
Un concept plus général: Variables condition

- **On appelle *variable condition* une var qui peut être testée et**
  - endorme le thread qui la teste si la condition est fausse
  - le réveille quand la condition devient vraie

- **Sont employées dans les moniteurs, mais peuvent aussi être utilisées indépendamment**

Blocage dans les moniteurs

- **threads attendent dans la file d'entrée ou dans une file de condition (ils n 'exécutent pas)**
- **sur x.wait: le thread est placé dans la file de la condition (il n 'exécute pas)**
- **x.signal amène dans le moniteur 1 thread de la file x (si x vide, aucun effet)**



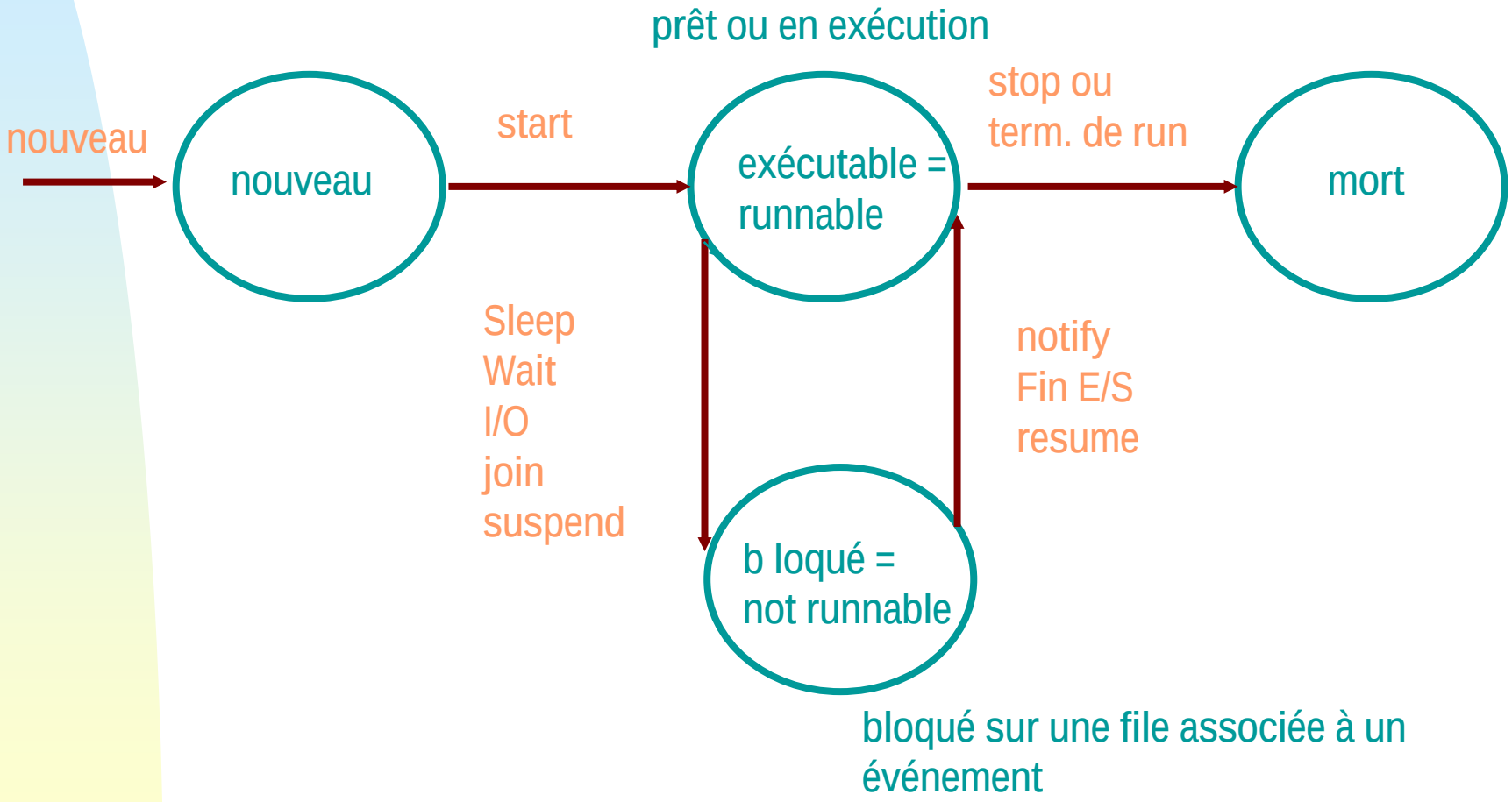
Un pb concernant le signal

- **Quand un thread P exécute x.signal et libère un thr. Q, il pourrait y avoir 2 thr. qui peuvent exécuter, P et Q, ce qui est défendu. Deux solutions possibles:**
  - P pourrait attendre jusqu` à ce que Q sorte du moniteur, p.ex. dans une file spéciale (dite *urgente*) (v. Stallings)
  - Q pourrait attendre jusqu'à ce que P sorte du moniteur

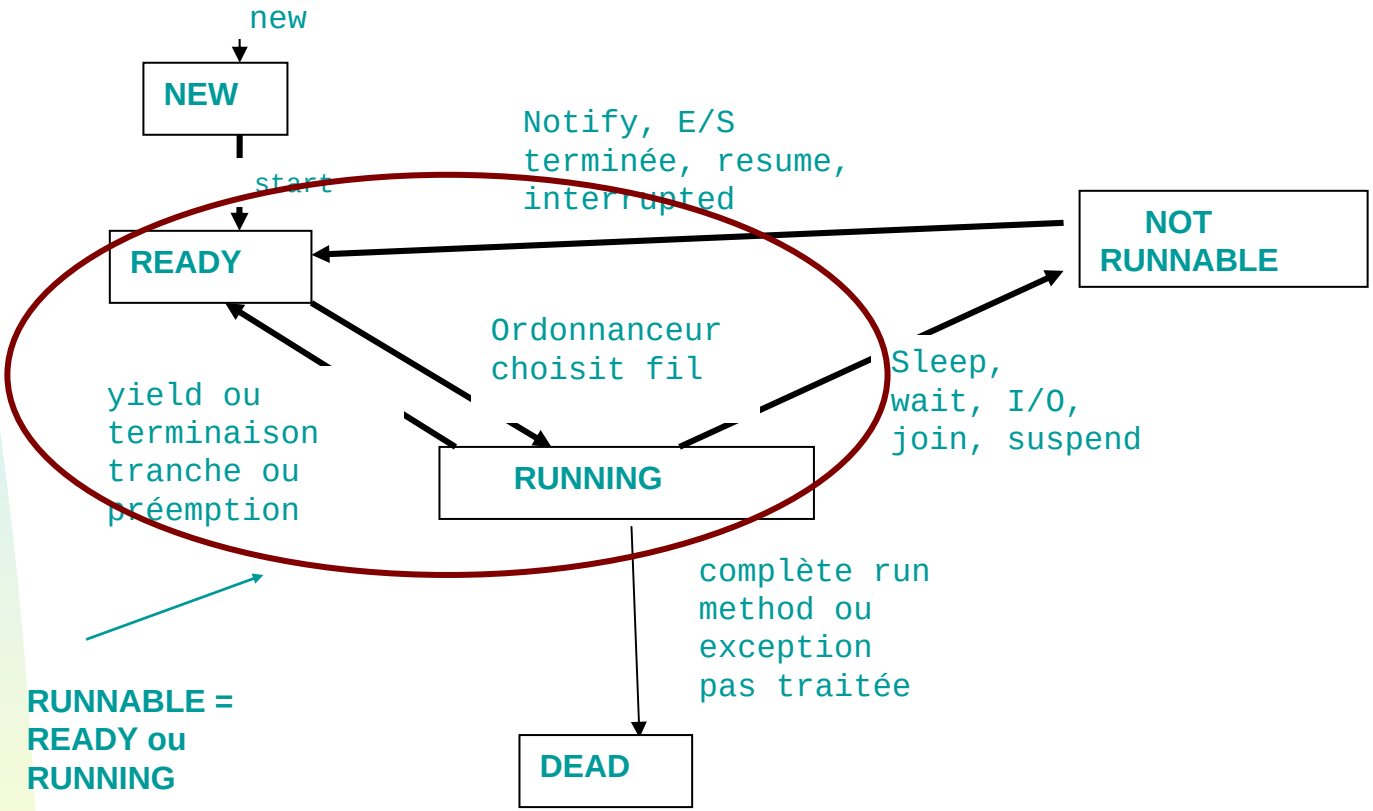
Terminologie Java (davantage au lab)

- **Les méthodes synchronisées de Java sont essentiellement des moniteurs**
  - Un seul thread à la fois peut les exécuter
- **Il y a 2 files pour un objet:**
  - File d'entrée
  - File d'attente (méthode wait)
- **Un thread ne peut avoir que 1 file wait**
  - Limitation importante qui complique les choses en Java…
- **Wait existe en Java + ou – comme décrit pour les moniteurs**
- **Signal s'appelle notify**
  - Notify() libère 1 seul thread
  - NotifyAll les libères tous
  - Mais ils n'exécutent pas: ils sont mis dans la file d'entrée

Java: diagramme simplifié de transition d'état threads
(sur la base de la fig. 5.10 du manuel)



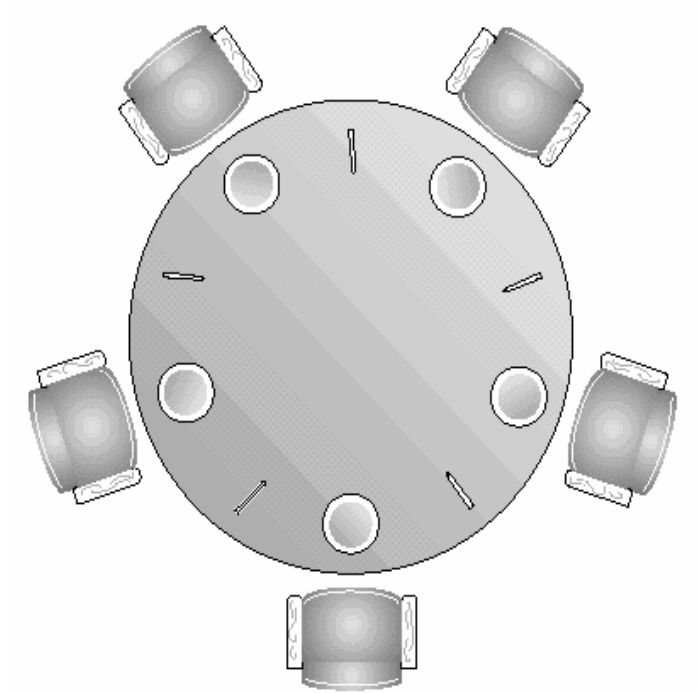
Un diagramme plus complet



Les méthodes *suspend*, *resume*, *stop* ne sont pas recommandées aujourd'hui *(deprecated)*.
Malheureusement le manuel s'en sert…

Retour au problème des philosophes mangeant

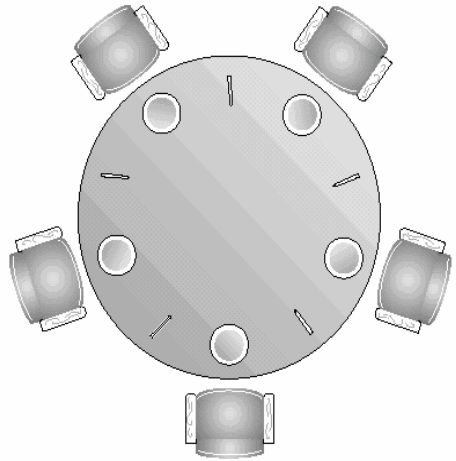
- 5 philosophes qui mangent et pensent
- Pour manger il faut 2 baguettes, droite et gauche
- On en a seulement 5!
- Un problème classique de synchronisation
- Illustre la difficulté d'allouer ressources aux threads tout en évitant interblocage et famine



Philosophes mangeant structures de données

- **Chaque philos. a son propre state qui peut être (thinking, hungry, eating)**
  - philosophe i peut faire state[i] = eating ssi les voisins ne mangent pas

- **Chaque condition a sa propre condition self**
  - le philosophe i peut attendre sur self [i] si veut manger, mais ne peut pas obtenir les 2 baguettes



Chaque philosophe exécute à jamais:

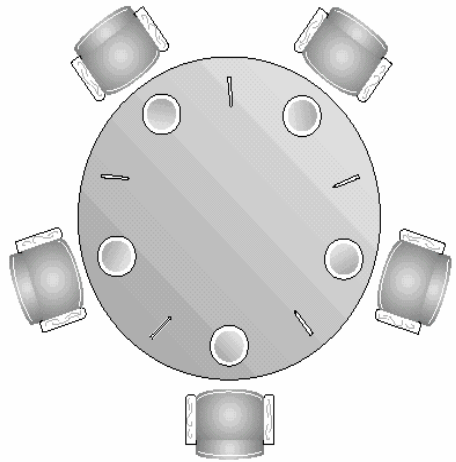
repeat

pickup

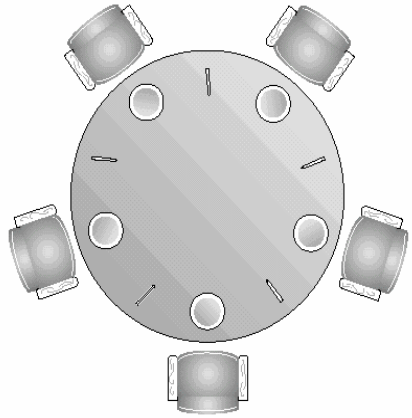
eat

putdown

forever



Un philosophe mange



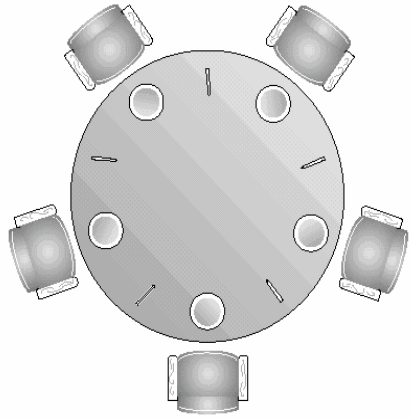
```
private test(int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING;
                self[i].signal;
        }
}
```

Un philosophe mange si ses voisins ne mangent pas et s'il a faim.

Une fois mangé, il signale de façon qu'un autre pickup soit possible, si pickup s'était arrêté sur wait

Il peut aussi sortir sans avoir mangé si le test est faux

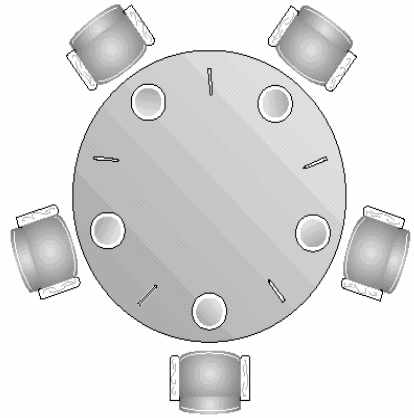
Chercher de prendre les baguettes



```
public entry pickUp(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait;
}
```

Phil. cherche à manger en testant, s'il sort de test qu'il n'est pas mangeant il attend – un autre pickup n'est pas possible avant un self[i] signal

Déposer les baguettes



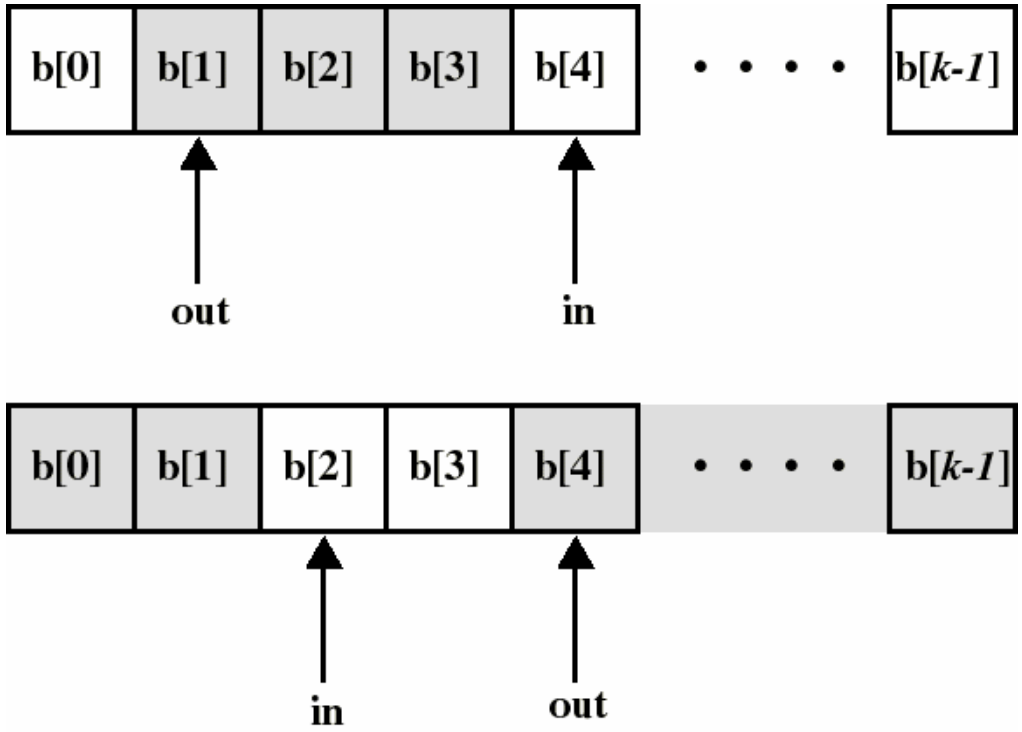
```
public entry putDown(int i) {
        state[i] = THINKING;
            // tester les deux voisins
        test((i + 4) % 5);
        test((i + 1) % 5);
}
```

Une fois fini de manger, un philosophe se préoccupe de faire manger ses voisins en les testant

Une solution ingénieuse, cependant les baguettes ne sont que implicites

Il vaut la peine de l'étudier

P/C: tampon circulaire de dimension k



- **Peut consommer seulement si le nombre N d'éléments consommables est au moins 1 (N = in-out)**
- **Peut produire seulement si le nombre E d'espaces libres est au moins 1 (E = out-in)**

Variables conditionnelles utilisées

- **Si le tampon est plein, le producteur doit attendre qu'il devienne non-plein**
  - Var conditionnelle notfull
- **Si le tampon est vide, le consommateur doit attendre qu'il devienne non-vide**
  - Var conditionnelle notempty

Moniteur pour P/C avec tampon fini
(syntaxe un peu différente, pas orienté objet)

```
Monitor boundedbuffer:
  buffer: vecteur[0..k-1] de items;
  nextin = 0, nextout = 0, count = 0 ;
  notfull, notempty: condition;

  Produce(v):
    if (count==k) notfull.wait;
    buffer[nextin] = v;
    nextin = (nextin+1 mod k);
    count ++;
    notempty.signal;

  Consume(v):
    if (count==0) notempty.wait;
    v = buffer[nextout];
    nextout = (nextout+1 mod k);
    count --;
    notfull.signal;
```

Variable conditionnelle sur laquelle le producteur attend s'il n'y a pas d'espaces libres

Variable conditionnelle sur laquelle le consommateur attend s'il n'y a pas d'espaces occupés

La solution Java est plus compliquée surtout à cause du fait que Java n’a pas de variables conditionnelles nommées

V. manuel

Relation entre moniteurs et autre mécanismes

- **Les moniteurs sont implantés utilisant les sémaphores ou les autres mécanismes déjà vus**
- **Il est aussi possible d`implanter les sémaphores en utilisant les moniteurs!**
  - les laboratoires vont discuter ça

Le problème de la SC en pratique...

- **Les systèmes réels rendent disponibles plusieurs mécanismes qui peuvent être utilisés pour obtenir la solution la plus efficace dans différentes situations**

Synchronisation en Solaris 2 (avec UCT multiples)

- **Plusieurs mécanismes utilisés:**
  - *adaptive mutex* protège l 'accès aux données partagées pour des SC courtes
  - sémaphores et *condition variables* protègent des SC plus importantes
  - serrures lecteurs-rédacteurs (reader-writers locks) protègent des données qui normalement ne sont que lues
  - les mêmes mécanismes sont disponibles aux usagers et dans le noyau

Adaptive mutex en Solaris 2

- **Utilisés pour des SC courtes: quand un thread veut accéder à des données partagées:**
  - Si les données sont couramm. utilisées par un thread exécutant sur un autre UCT, l 'autre thread fait une attente occupée
  - Sinon, le thread est mis dans une file d 'attente et sera réveillé quand les données deviennent disponibles

Windows NT: aussi plus. mécanismes

- **exclusion mutuelle sur données partagées: un fil doit demander accès et puis libérer**
- **section critiques: semblables mais pour synchroniser entre fils de threads différents**
- **sémaphores**
- **event objects: semblables à *condition variables***

Concepts importants du Chapitre 7

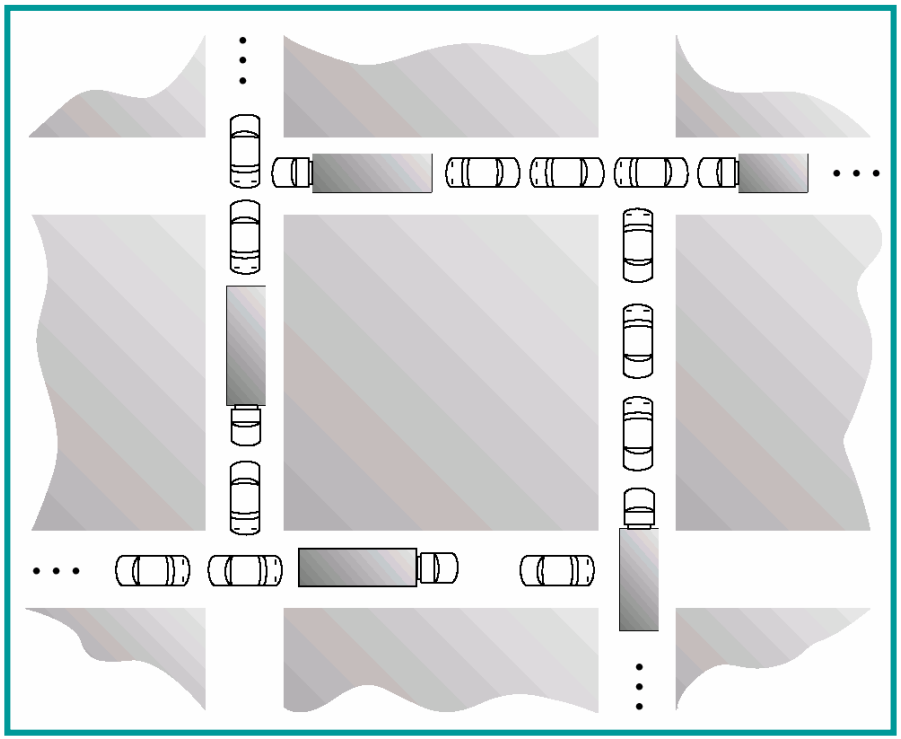
- **Sections critiques: pourquoi**
- **Difficulté du problème de la synch sur SC**
  - Bonnes et mauvaises solutions
- **Accès atomique à la mémoire**
- **Solutions logiciel `pures`**
- **Solution matériel: test-and-set**
- **Solutions par appels du système:**
  - Sémaphores, moniteurs, fonctionnement
- **Problèmes typiques: tampon borné, lecteurs-écrivains, philosophes**

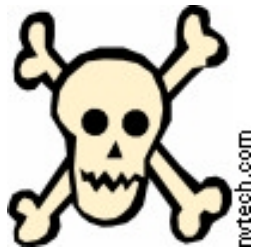
Par rapport au manuel

- **Le manuel couvre + ou – la même matière, mais en utilisant une approche Java**
- **Pour le test et examen, suivre ma présentation**
- **Pour les travaux de programmation, utiliser les exemples du manuel**

Interblocage = impasse (Deadlock)

Chapitre 8





http://w3.uqo.ca/luigi/

Interblocages: concepts importants

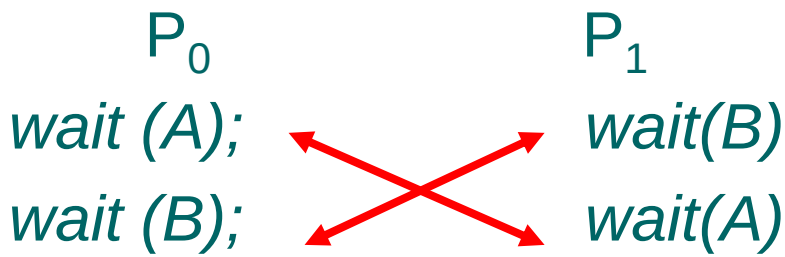
- **Caractérisation: les 4 conditions**
- **Graphes allocation ressources**
- **Séquences de terminaison**
- **États sûrs et no-sûrs**
- **Prévenir les interblocages**
- **Éviter les interblocages**
- **Détecter les interblocages**
- **Récuperer d'un interblocage**

Exemple 1

- **Deux processus coexistent dans un système, qui a 2 lignes téleph. seulement**
- **Le proc 1 a besoin de**
  - une ligne téléphonique pour démarrer
  - la ligne précédente, et une additionnelle, pour terminer
- **Le proc 2 est pareil**
- **Scénario d 'interblocage:**
  - proc 1 demande 1 ligne
  - proc 2 demande 1 ligne: les deux lignes sont engagées
  - interblocage! aucun proc ne peut compléter
    - à moins qu 'un des proc ne puisse être suspendu
    - ou puisse retourner en arrière
- **Observez que l 'interblocage n 'est pas inévitable, p.ex. si 1 complète avant le début de 2**
  - Éviter / détecter le risque d'interblocage
  - Éviter / détecter qu'un interblocage se vérifie

Exemple 2

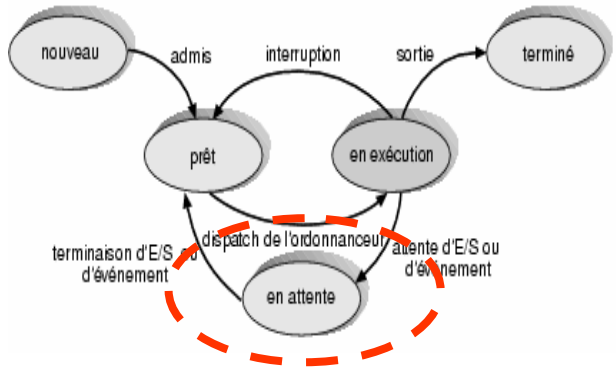
- **Sémaphores**



- **Scénario d'interblocage:**
  - initialisation de A et B à 1
  - P0 exécute wait(A), A=0
  - P1 exécute wait(B), B=0
  - P0 et P1 ne peuvent pas aller plus loin
    - Qu'arrive au lieu si P0 exécute entièrement avant P1?

Définition (Tanenbaum)

- **Un ensemble de processus est en *interblocage* si chaque processus attend un événement que seul un autre processus de l'ensemble peut provoquer**
- **L'événement est une *libération de ressource***
  - Prenant ce mot dans le sens le plus vaste: ressource peut être un signal, un message,un sémaphore, etc.
  - Exemple intéressant: interblocage entre lecteurs ou écrivains sur une base de données???
- **Un processus en interblocage est en état attente**

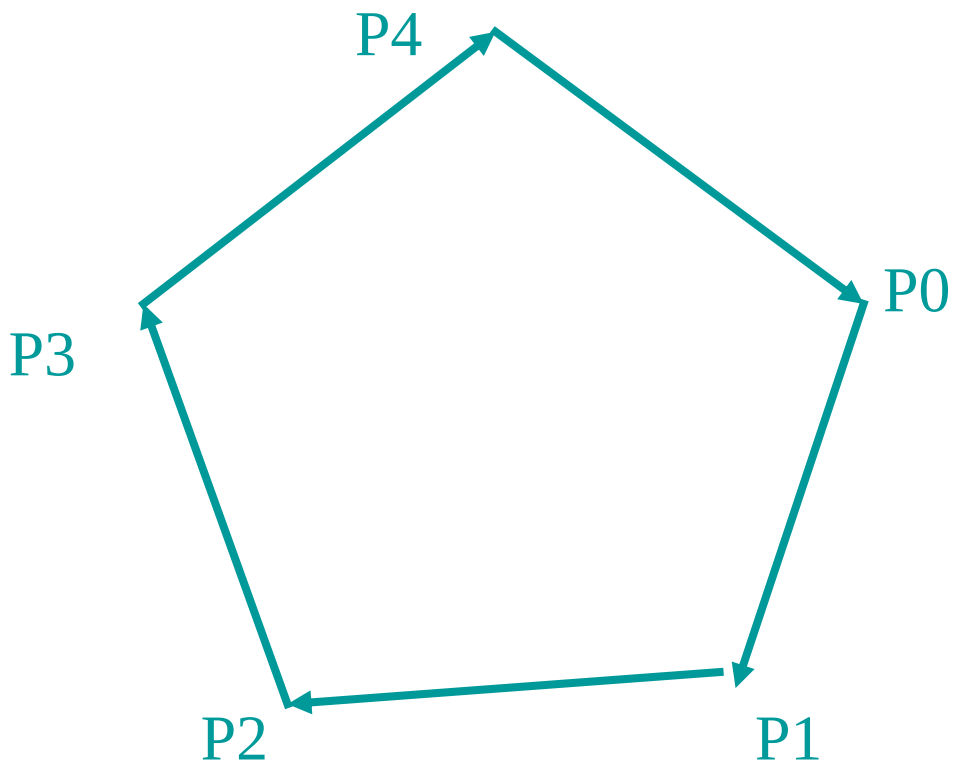


important

Caractérisation d'interblocage

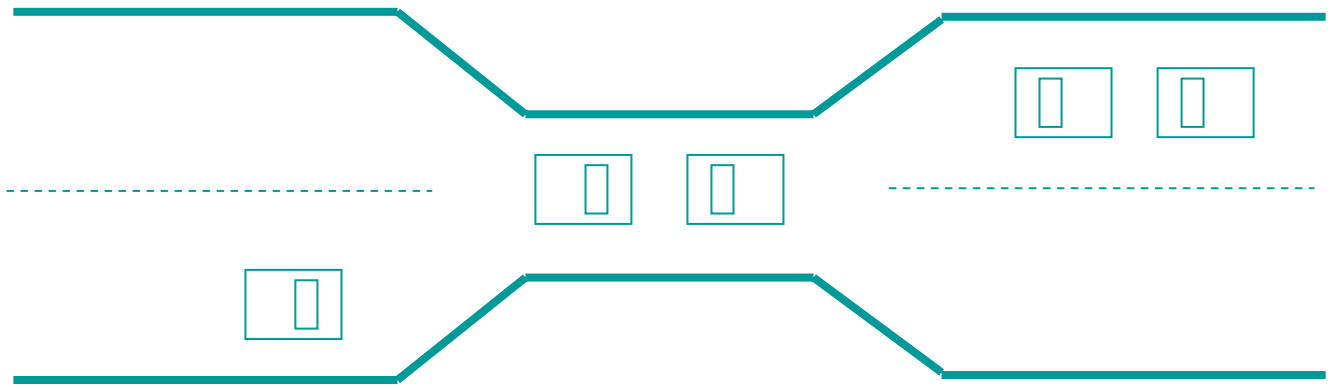
- **L'interblocage demande la présence simultanée de 4 conditions (conditions nécessaires et suffisantes)**
  - **Exclusion mutuelle:** le système a des ressources non partageables (1 seul proc à la fois peut s'en servir)
    - Ex.: un UCT, une zone de mémoire, une périphérique, mais aussi sémaphores, moniteurs, sections critiques
  - **Saisie et attente** (hold and wait): un processus a saisi une ressource non partageable et en attend des autres pour compléter sa tâche
  - **Pas de préemption:** un processus qui a saisi une ressource non partageable la garde jusqu'à ce qu'il aura complété sa tâche
  - **Attente circulaire:** il y a un cycle de processus tel que chaque processus pour compléter doit utiliser une ressource non partageable qui est utilisée par le suivant, et que le suivant gardera jusqu`à sa terminaison
    - **En présence des 3 premières conditions, une attente circulaire est un interblocage**
    - **Les 3 premières conditions n'impliquent pas nécessairement interblocage, car l'attente circulaire pourrait ne pas se vérifier**

Attente circulaire - aucun ne lâche - aucun processus ne peut terminer donc interblocage



Pour terminer, chaque processus doit saisir une ressource que le prochain ne lâchera pas → interblocage

Exercice



Réfléchissez à cet exemple dans lequel des voitures sont dans une situation d'interblocage sur un pont et voir comment les différentes conditions sont satisfaites.

V. aussi l'exemple à la page 1.

Exercice

- **Considérez un système dans lequel chaque processus n'a besoin que d'une seule ressource pendant toute son existence**

- **L'interblocage, est-il possible?**

Exercice

- **Vérifier que si dans un système il y a toujours suffisamment de ressources pour tous, il n'y aura jamais d'interblocages**

- **Cependant ceci est souvent impossible, surtout dans le cas de *sections critiques*, car les données partagées existent normalement dans un seul exemplaire**

En principe, un problème difficile

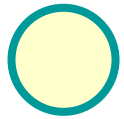
- **Le problème de déterminer s'il y a d'interblocage dans un système est connu comme problème *insoluble* en principe par ordinateur** **(résultat théorique)**
  - On ne pourrait pas écrire un programme S qui, étant donnée un autre programme X en entrée, pourrait déterminer à coup sûr si X contient la possibilité d'interblocage
- **Cependant, nous pouvons développer des critères pour trouver des réponses plausibles dans des cas particuliers**

Graphes d'allocation ressources

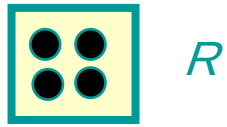
- **Un ensemble de sommets V et d'arêtes E**
- **V est partitionné dans:**
  - $P = \{P_1, P_2, \ldots, P_n\}$, l'ensemble qui consiste de tous les procs dans le système
  - $R = \{R_1, R_2, \ldots, R_m\}$, l'ensemble qui consiste de tous les types de ressources dans le système
- **arête requête – arête dirigée $P_i \rightarrow R_k$**
- **arête affectation – arête dirigée $R_i \rightarrow P_k$**

Graphe d'allocation ressources

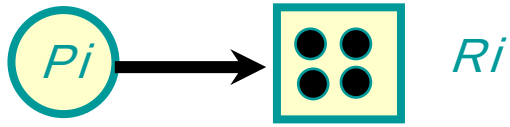
- **Processus**



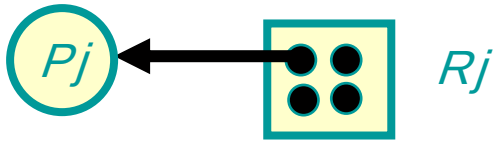
- **Ressource dont il y a 4 exemplaires (instances)**



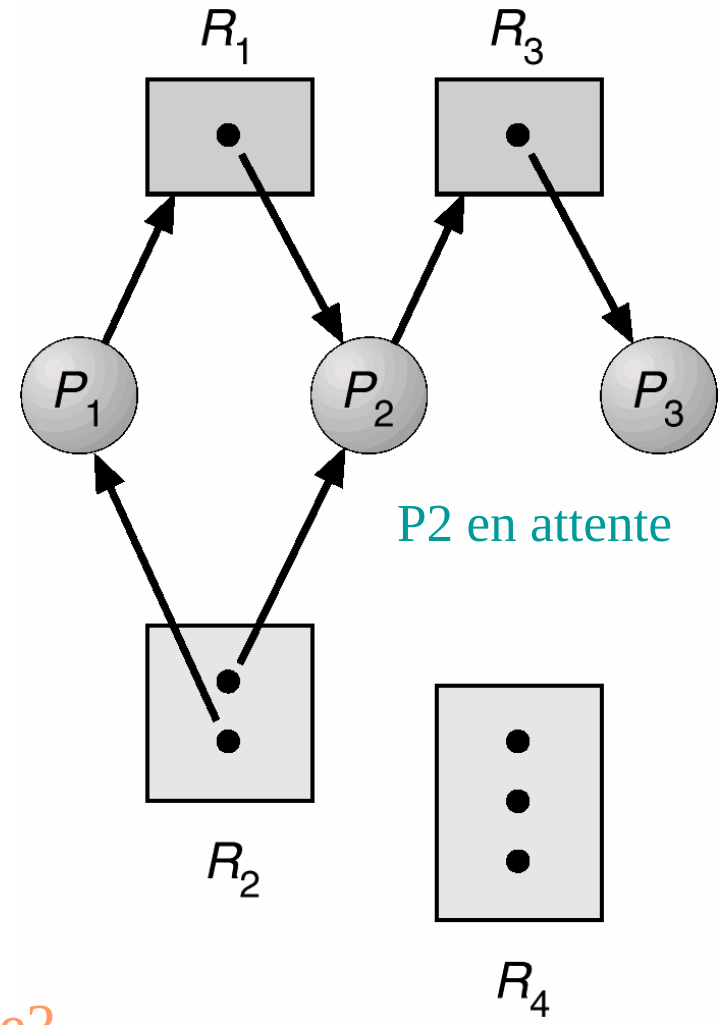
- **Pi a (ou aura) besoin pour terminer d'un exemplaire de Ri, dont il y en a 4**



- **Pj a saisi (et utilise) un exemplaire de Rj**



Exemple de graphe allocation ressources



P1 en attente

P3 pas en attente

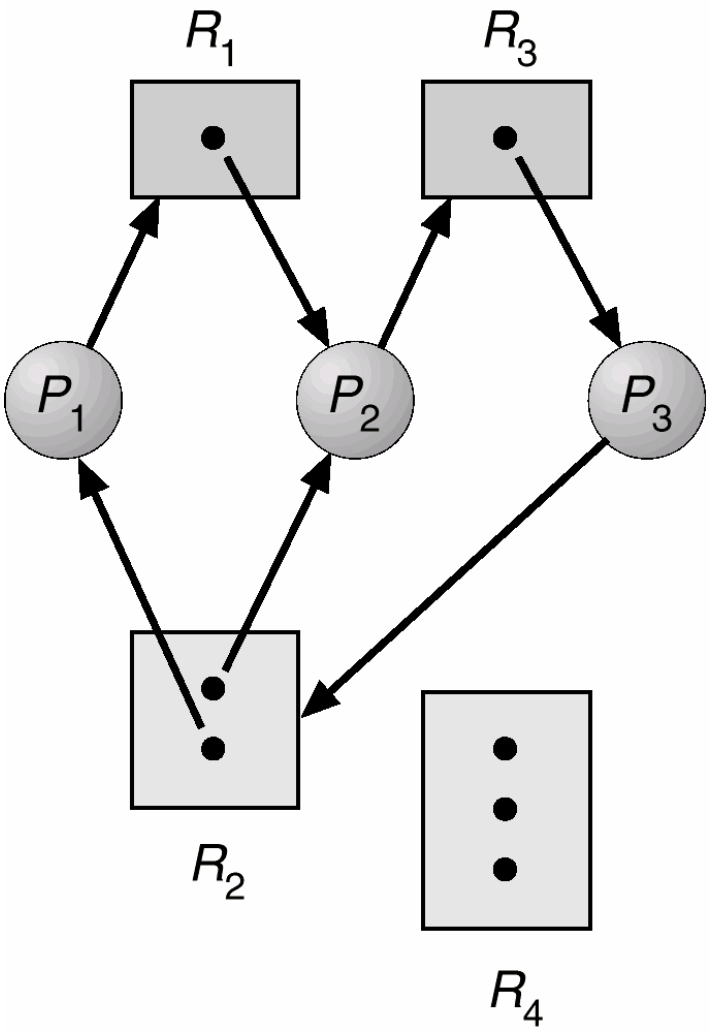
P2 en attente

Y-a-t-il interblocage?

Utilisation de ces graphes

- **Nous *supposons* l'existence des 3 premières conditions**
  - Excl. Mutuelle, saisie et attente, pas de préemption
- **Pour montrer qu'il *n'y a pas d'interblocage*, nous devons montrer qu'il *n'y a pas de cycle*, car il y a un processus qui peut terminer sans attendre aucun autre, et puis les autres de suite**
- **<P_3, P_2, P_1> est un ordre de terminaison de processus: tous peuvent terminer dans cet ordre**

Graphe allocation ressources avec interblocage



Nous avons deux cycles:

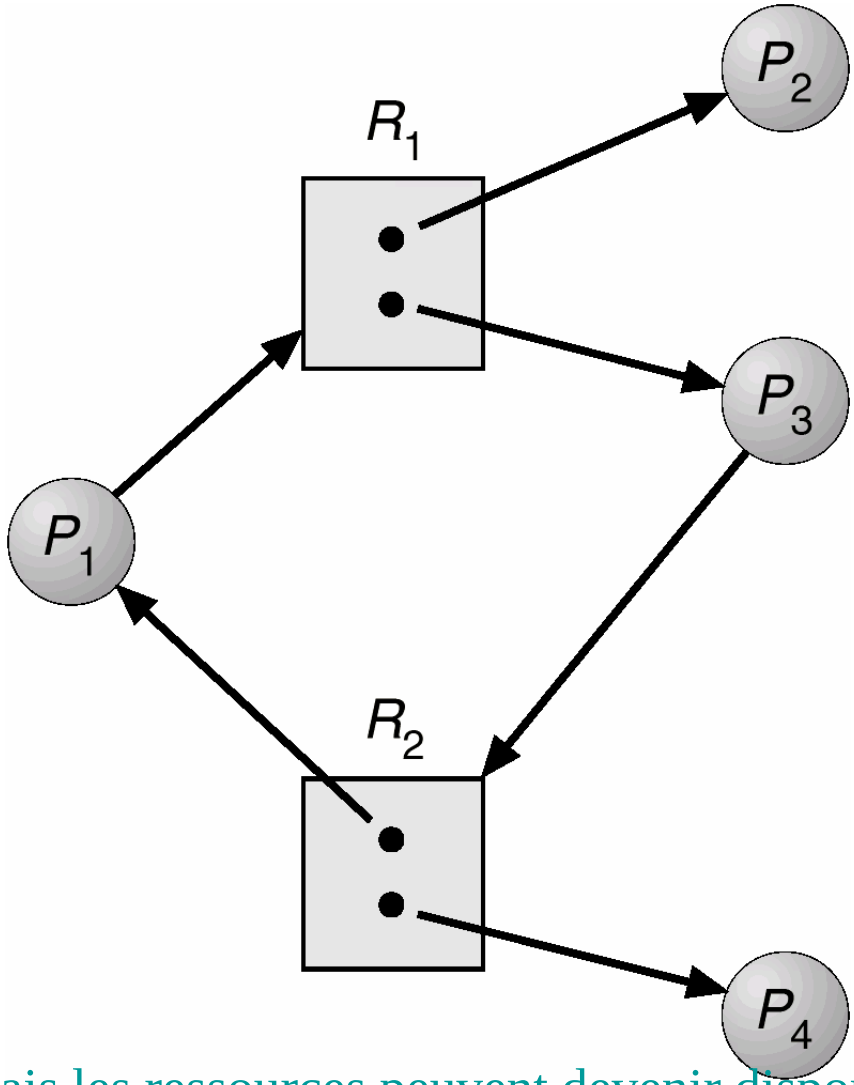
P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

aucun proc ne peut terminer

aucune possibilité d'en sortir

Graphe allocation ressources avec cycle, mais pas d' interblocage (pourquoi?)



Attente circulaire, mais les ressources peuvent devenir disponibles

Constatations

- **Les cycles dans le graphe alloc ressources ne signalent pas nécessairement une attente circulaire**
- **S ’il n`y a pas de cycles dans le graphe, aucun interblocage**
- **S ’il y a de cycles:**
  - Si seulement une ressource par type, interblocage
    - (pourquoi?!)
  - Si plusieurs ressources par type, possibilité d’interblocage
    - Il faut se poser la question: y-a-t-il un processus qui peut terminer et si oui, quels autres processus peuvent terminer en conséquence?

Hypothèse de terminaison

- **Un proc qui a toutes les ressources dont il a besoin, il s'en sert pour un temps fini, puis il les libère**
- **Nous disons que le processus termine, mais il pourrait aussi continuer, n'importe, l'important est qu'il laisse la ressource**

Méthodes pour traitement interblocage

- **Concevoir le système de façon qu`un interblocage soit impossible**
  - difficile, très contraignant
  - approprié dans le cas de systèmes critiques
- **Les interblocages sont possibles, mais sont évités (avoidance)**
- **Permettre les interblocages, en récupérer**
- **Ignorer le problème, qui donc doit être résolu par le gérant ou l 'usager**
  - malheureusement, méthode d 'utilisation générale!

Prévention d'interblocage: prévenir au moins une des 4 conditions nécessaires

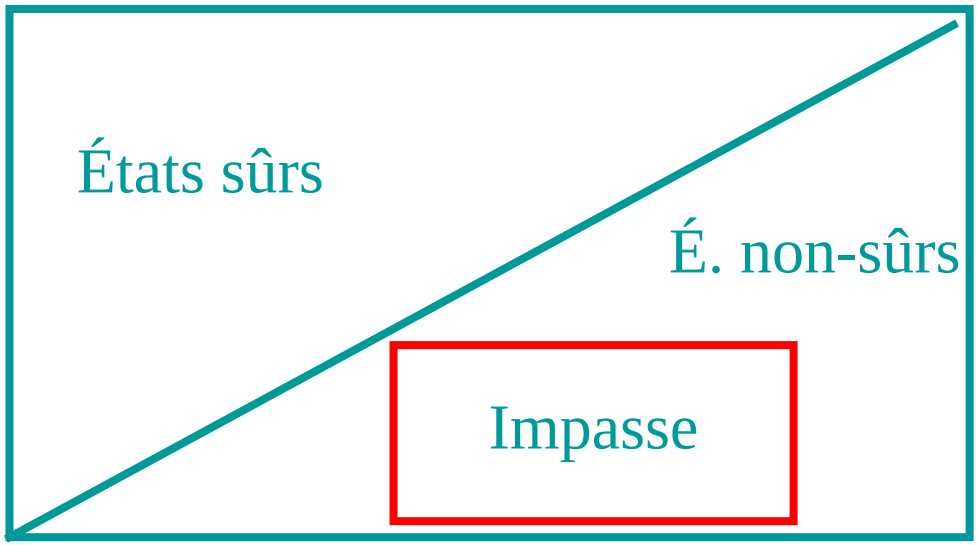
- Exclusion mutuelle: **réduire le plus possible l 'utilisation des ressources partagées et Sections Critiques**
  - Possible seulement dans le cas de procs totalement indépendants
- Saisie et attente **(hold and wait): un processus qui demande des nouvelles ressources ne devrait pas en retenir des autres (les demander toutes ensemble)**
  - Comment savoir?
- Préemption: **si un processus qui demande d'autres ressources ne peut pas les avoir, il doit être suspendu, ses ressources doivent êtres rendues disponibles**
  - OK, demande intervention du SE
- Attente circulaire: **imposer un ordre partiel sur les ressources, un processus doit demander les ressources dans cet ordre (p.ex. tout processus doit toujours demander une imprimante avant de demander une unité ruban)**
  - Difficile

Éviter les interblocages (deadlock avoidance)

- **Chaque processus doit déclarer le nombre max. de ressources dont il prévoit avoir besoin**
- **L'algorithme examine toutes les séquences d 'exécution possibles pour voir si une attente circulaire est possible**

État sûr (safe state)

- **Un état est sûr si le système peut en sortir sans interblocages**
- **Ne pas allouer une ressource à un processus si l'état qui en résulte n'est pas sûr**



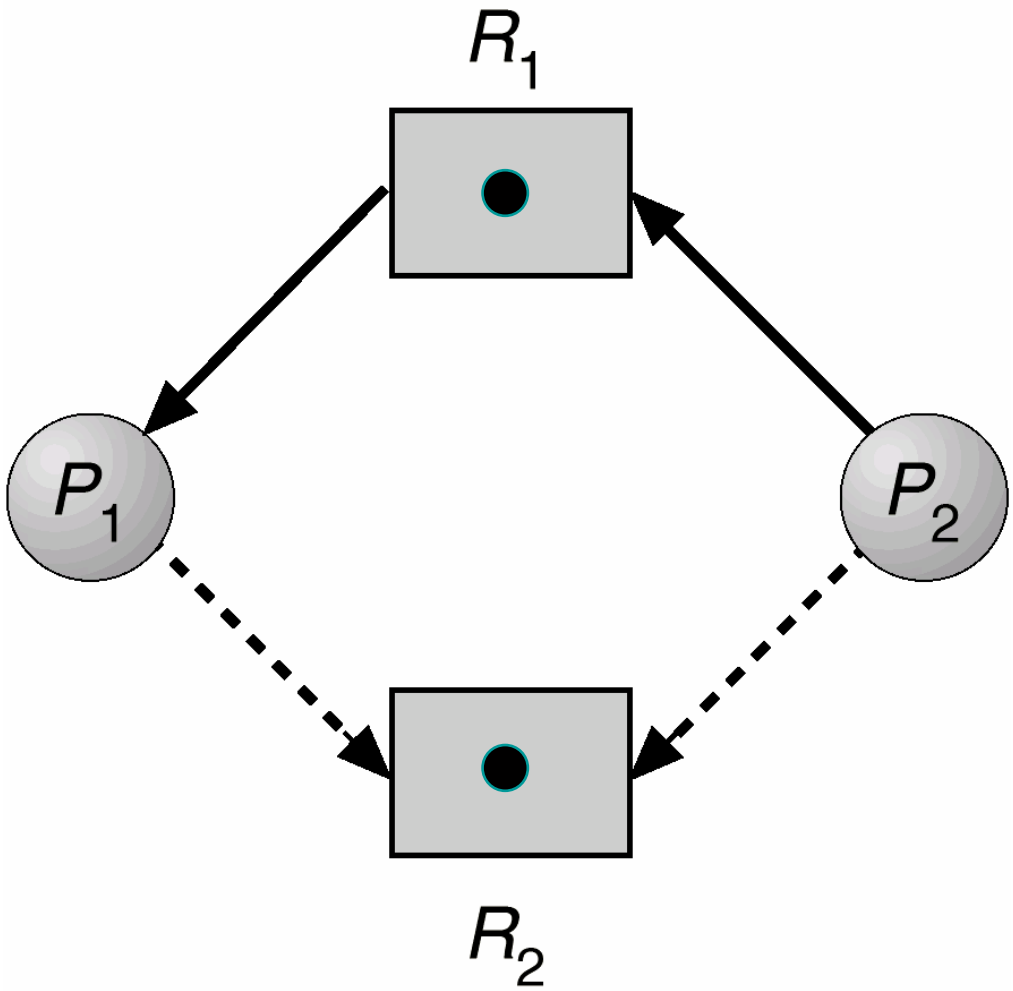
État sûr

- **Une séquence de proc <P_1, P_2, …, P_n> est sûre si pour chaque Pi, les ressources que Pi peut encore demander peuvent être satisfaites par les ressources couramment disponibles + ressources utilisées par *les Pj qui les précèdent*.**
  - Quand Pi aboutit, Pi+1 peut obtenir les ressources dont il a besoin, terminer, donc
- **<P_1, P_2, …, P_n> est un ordre de terminaison de processus: tous peuvent se terminer dans cet ordre**

Algorithme d'allocation de ressources

- **Il faut maintenant prendre en considération:**
  - les requêtes possibles dans le futur (chaque processus doit déclarer ça)
- **Arête demande P_i - - > R_j indique que le processus P_i peut demander la ressource R_j (ligne à tirets)**

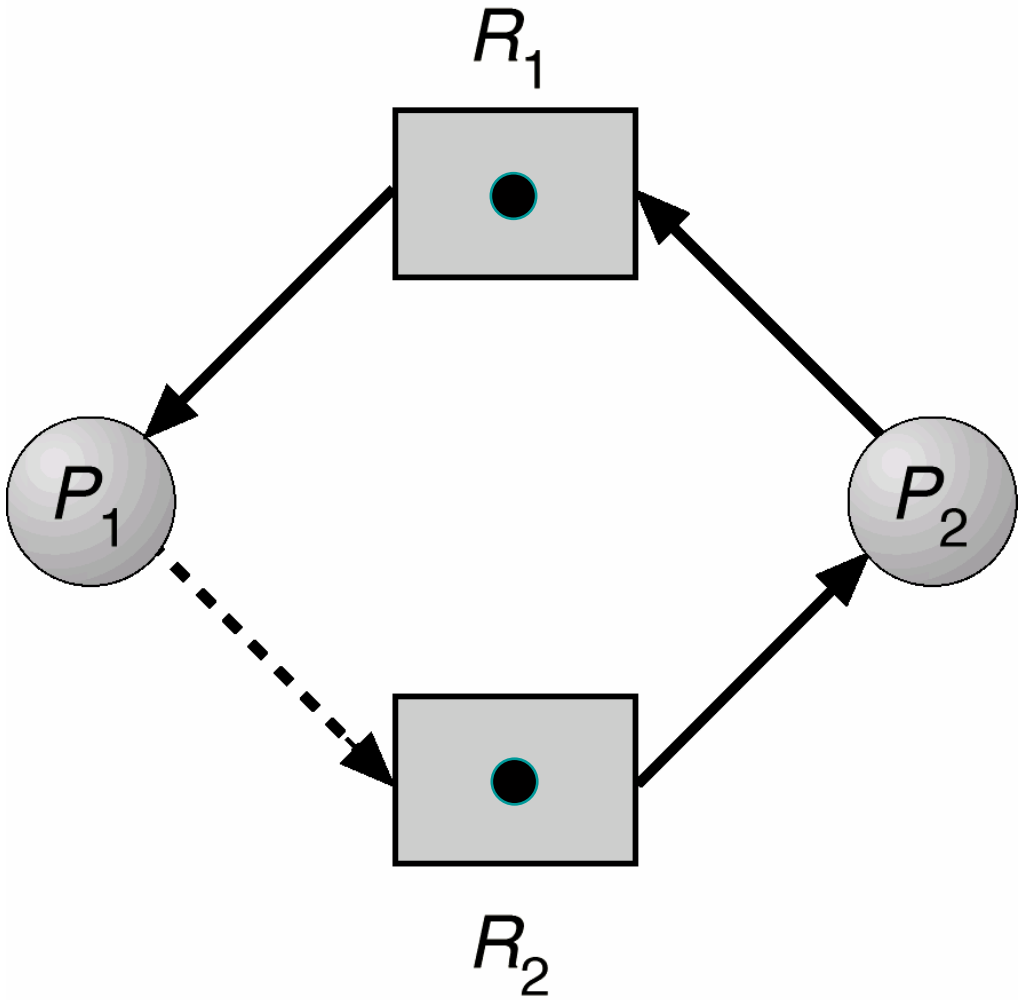
Graphe d`allocation ressources



Ligne continue: requête courante;

tirets: requête possible dans le futur

Un état pas sûr



Si P2 demande R2, ce dernier ne peut pas lui être donné, car ceci peut causer un cycle dans le graphe si P1 req R2. Mieux vaut attendre la fin de P1, puis faire finir P2

Détection d ’interblocage

- **On permet au système d’entrer dans un état d’interblocage**
- **L’interblocage est détecté**
- **On récupère de l’interblocage**

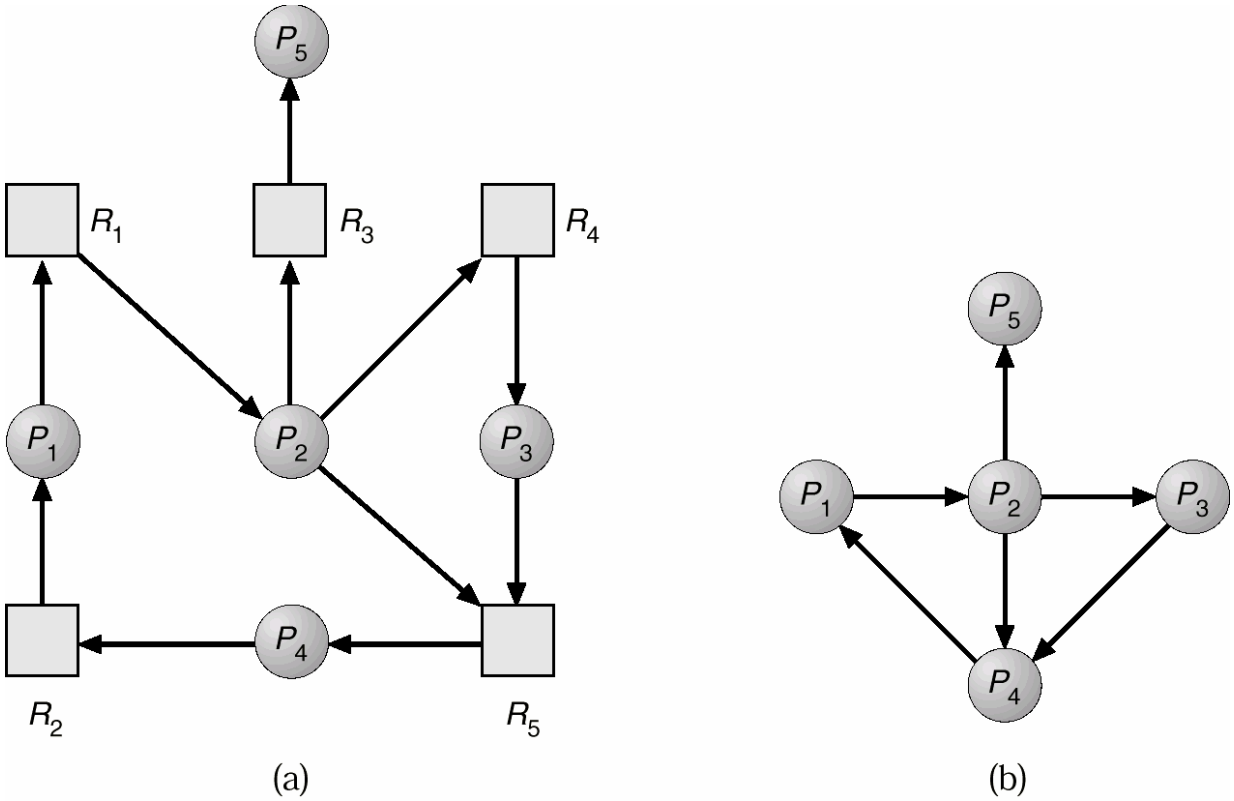
Différence entre attente et interblocage

- **Il est difficile de détecter s 'il y a effectivement une interblocage dans le système**
- **Nous pourrions voir qu'un certain nombre de processus est en attente de ressources**
  - ceci est normal!
- **Pour savoir qu'il y a interblocage, il faut savoir qu'aucun processus dans un groupe n'a de chance de recevoir la ressource**
  - car il y a attente circulaire!
- **Ceci implique une analyse additionnelle, que peu de SE se prennent la peine de faire...**

Méthode de détection d'interblocage dans le cas d'une ressource par type

- **Essentiellement, la méthode déjà décrite**
  - Construire un graphe d'allocation ressources et voir s'il y a une manière dont tous les proc peuvent terminer
- **Dans le cas d'une ressource par type, l`algorithme cherche des cycles dans le graphe (algorithme d'ordre n^2, si n=nombre de sommets)**
- **Plus difficile dans le cas de plus. ressources par type**
  - Pas discuté

Graphe allocation ressources et graphe d 'attente
(cas d'1 ressource par type)



Récupérer d'interblocages

- **Terminer tous les processus dans l'interblocage**
- **Terminer un processus à la fois, espérant d'éliminer le cycle d 'interblocages**
- **Dans quel ordre: différents critères:**
  - priorité
  - besoin de ressources: passé, futur
  - combien de temps il a exécuté, de combien de temps il a encore besoin
  - etc.

Récupération: préemption de ressources

- **Minimiser le coût de sélectionner la victime**
- **Rollback: retourner à un état sûr**
  - besoin d`établir régulièrement et garder des ‘points de reprise’, sortes de photos de l ’état courant du processus
  - p.ex. Word établit des points de reprise qu’il vous propose après un ‘accident’
- **Famine possible si le même processus est toujours sélectionné**

Combinaison d'approches

- **Combiner les différentes approches, si possible, en considération des contraintes pratiques**
  - prévenir
  - éviter
  - détecter
- **utiliser les techniques les plus appropriées pour chaque classe de ressource**

Importance du pb de l'interblocage

- **L'interblocage est quasiment ignoré dans la conception des systèmes d'aujourd'hui**
  - Avec l'exception des systèmes *critiques*
- **S'il se vérifie, l'usager verra une panne de système ou l'échec d'un processus**
- **Dans les systèmes à haut parallélisme du futur, il deviendra de plus en plus important de le prévenir et éviter**

Par rapport au manuel

- **Tout le chapitre, mais le code Java n'est pas important pour l'examen**

Interblocages: concepts importants

- **Caractérisation: les 4 conditions**
- **Graphes allocation ressources**
- **Séquences de terminaison**
- **États sûrs et non-sûrs**
- **Prévenir les interblocages**
- **Éviter les interblocages**
- **Détecter les interblocages**
- **Récuperer d'un interblocage**

Gestion de la mémoire

Chapitre 9

w3.uqo.ca/luigi

Dans ce chapitre nous verrons que, pour optimiser l'utilisation de la mémoire, les programmes sont éparpillés en mémoire selon des méthodes différentes:
Pagination, segmentation

Gestion de mémoire: objectifs

- **Optimisation de l 'utilisation de la mémoire principale = RAM**
- **Les plus grand nombre possible de processus actifs doit y être gardé, de façon à optimiser le fonctionnement du système en multiprogrammation**
  - garder le système le plus occupé possible, surtout l'UCT
  - s'adapter aux besoins de mémoire de l 'usager
    - allocation dynamique au besoin

Gestion de la mémoire: concepts dans ce chapitre

- **Adresse physique et adresse logique**
  - mémoire physique et mémoire logique
- **Remplacement**
- **Allocation contiguë**
  - partitions fixes
  - variables
- **Pagination**
- **Segmentation**
- **Segmentation et pagination combinées**
- **Groupes de paires (buddy systems)**

Application de ces concepts

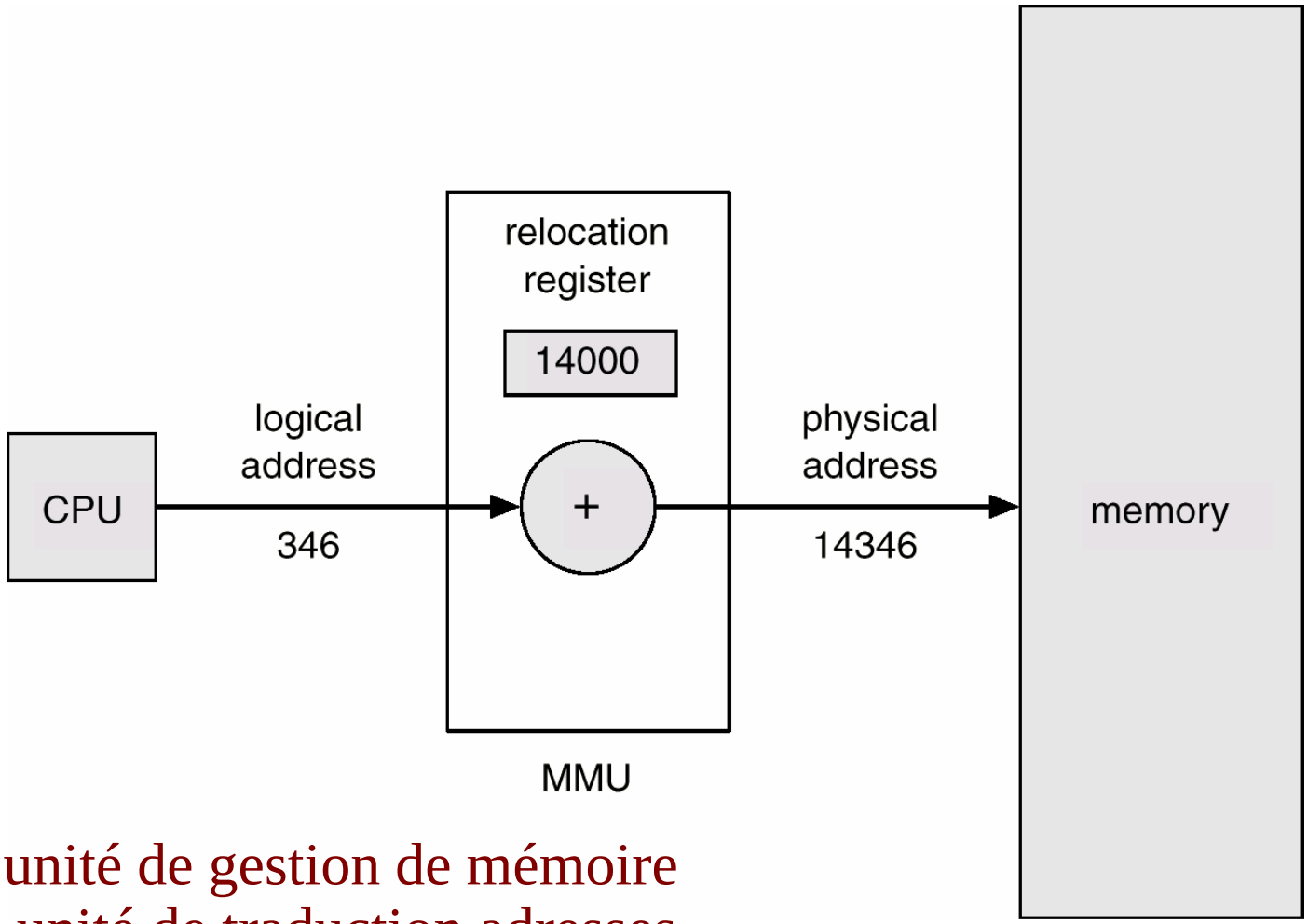
- **Pas tous les concepts de ce chapitre sont effectivement utilisés tels quels aujourd'hui dans la gestion de mémoire centrale**
- **Cependant plusieurs se retrouvent dans le domaine de la gestion de mémoires auxiliaires, surtout disques**

Mémoire/Adresses physiques et logiques

- **Mémoire physique:**
  - la mémoire principale RAM de la machine
- **Adresses physiques: les adresses de cette mémoire**
- **Mémoire logique: l'espace d`adressage d'un programme**
- **Adresses logiques: les adresses dans cet espace**

- **Il faut séparer ces concepts car normalement, les programmes sont chargés de fois en fois dans positions différentes de mémoire**
  - Donc adresse physique ≠ adresse logique

Traduction adresses logiques →adr. physiques



MMU: unité de gestion de mémoire
unité de traduction adresses
(memory management unit)

Définition des adresses logiques

- **Le manuel définit les adresses logiques comme étant les adresses générées par l'UCT**
  - Mais parfois l'MMU est partie de l 'UCT!
- **Je préfère la déf suivante:**
  - une adresse logique est une adresse à une location de programme
    - par rapport au programme lui-même seulement
    - indépendante de la position du programme en mémoire physique

Vue de l'usager

- **Normalement, nous avons plusieurs types d'adressages p.ex.**
  - les adresses du programmeur (noms symboliques) sont traduites au moment de la compilation dans des
  - adresses logiques
  - ces adresses sont traduites en adresses physiques par l'unité de traduction adresses (MMU)

- **Étant donné la grande variété de matériel et logiciel, il est impossible de donner des défs. plus précises.**

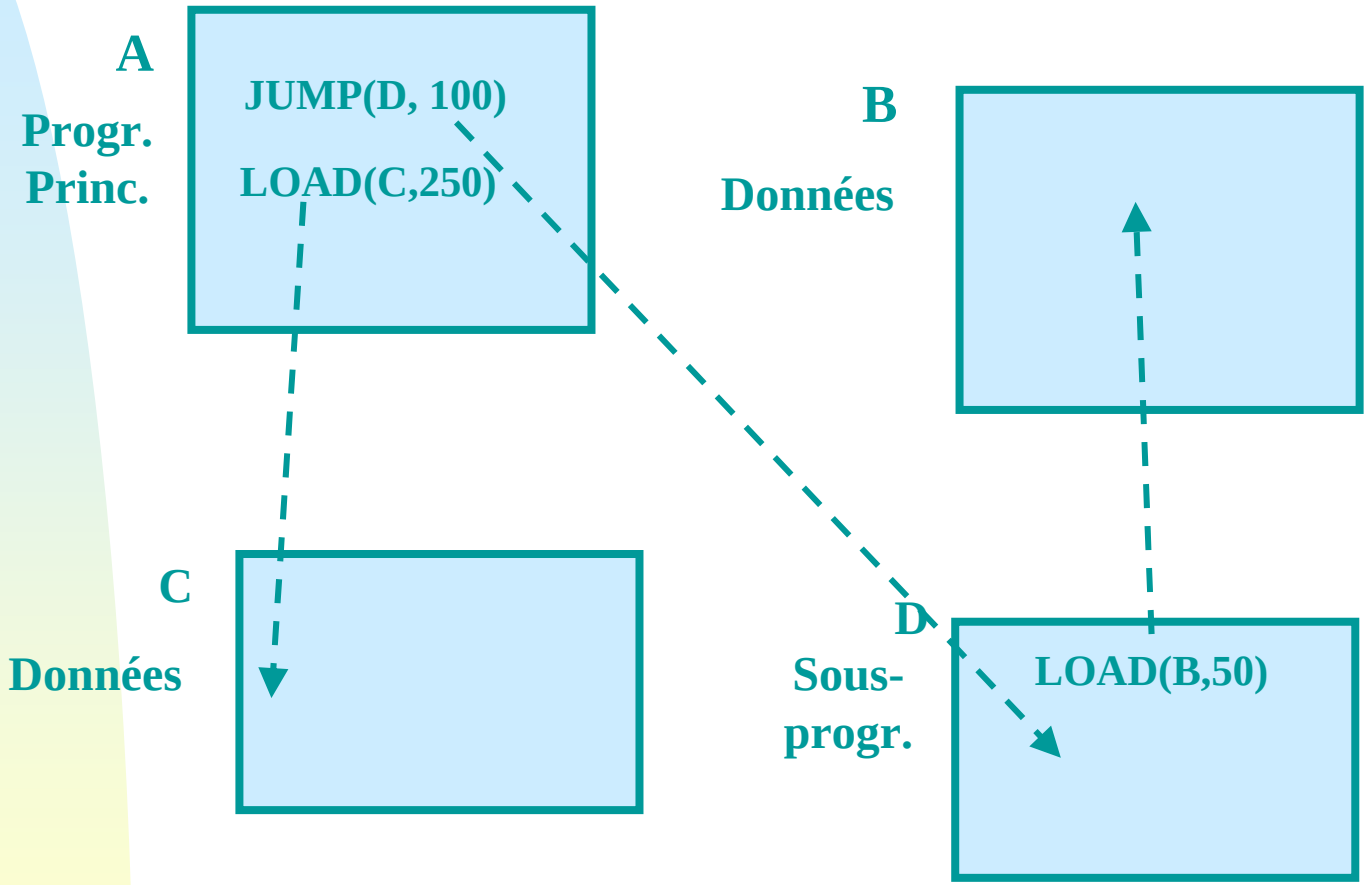
Liaison (Binding) d'adresses logiques et physiques (instructions et données)

- **La liaison des adresses logiques aux adresses physiques peut être effectuée en moments différents:**
  - Compilation: quand l'adresse physique est connue au moment de la compilation (rare)
    - p.ex. parties du SE
  - Chargement: quand l'adresse physique où le progr est chargé est connue, les adresses logiques peuvent être traduites (rare aujourd'hui)
  - Exécution: normalement, les adresses physiques ne sont connues qu'au moment de l 'exécution
    - p.ex. allocation dynamique

Deux concepts de base

- **Chargement = Loading. Le programme, ou une de ses parties, est chargé en mémoire physique, prêt à exécuter.**
  - Statique (avant l'exécution)
  - Dynamique (pendant l'exécution)
- **Édition de liens = Liaison (enchaînement) des différentes parties d'un programme pour en faire une entité exécutable.**
  - les références entre modules différents doivent être traduites
    - statique (avant l`exécution)
    - dynamique (sur demande pendant exécution)
      - N.B. parties du programme = modules = segments = sousprogrammes = objets, etc.

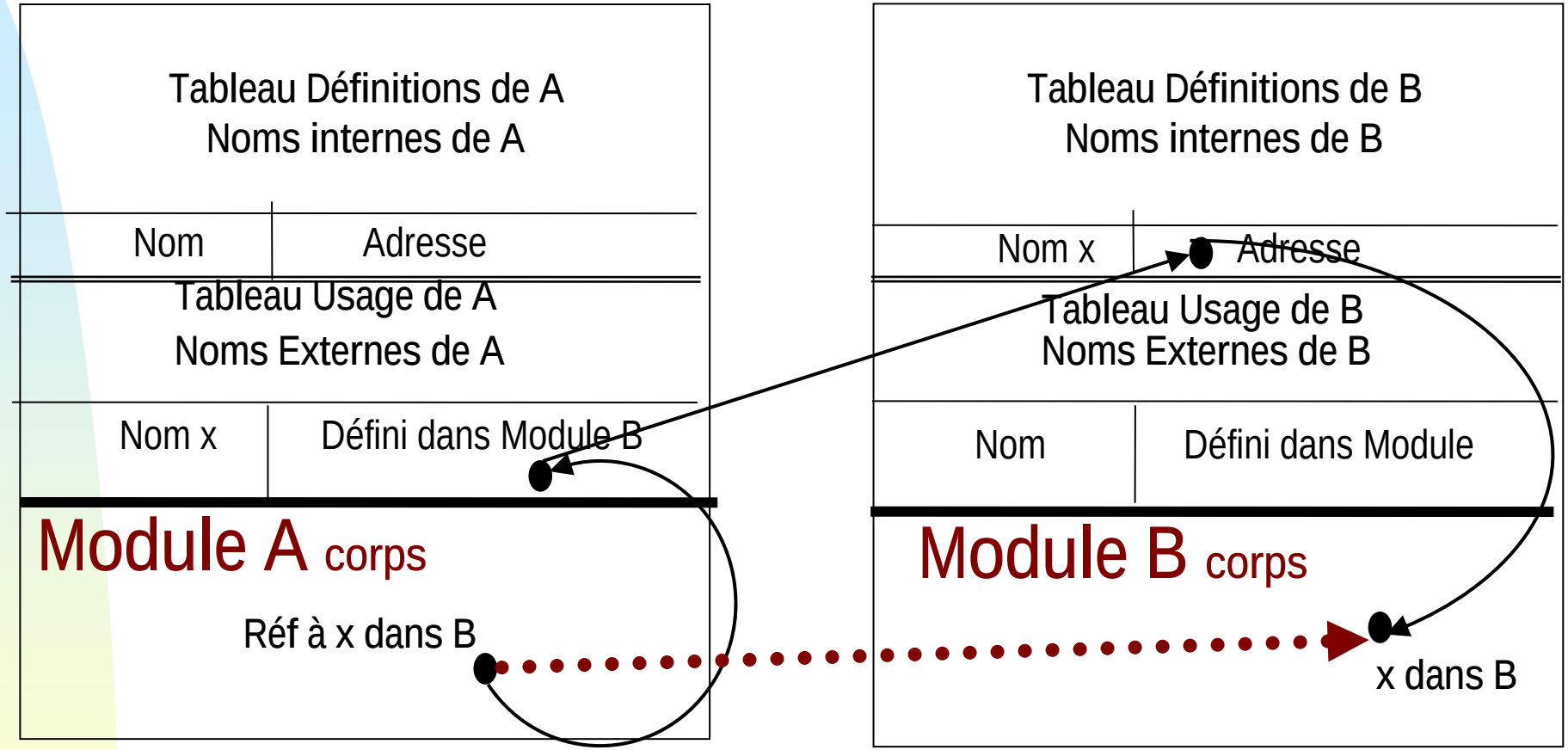
Édition de liens: adressage entre modules



Espace d'adressage logique

Les adresses sont en deux parties :
(No de module, Décalage dans le module)
doivent être traduites pour permettre adressage par l 'UCT

Édition de liens:
une méthode possible qui supporte la liaison dynamique

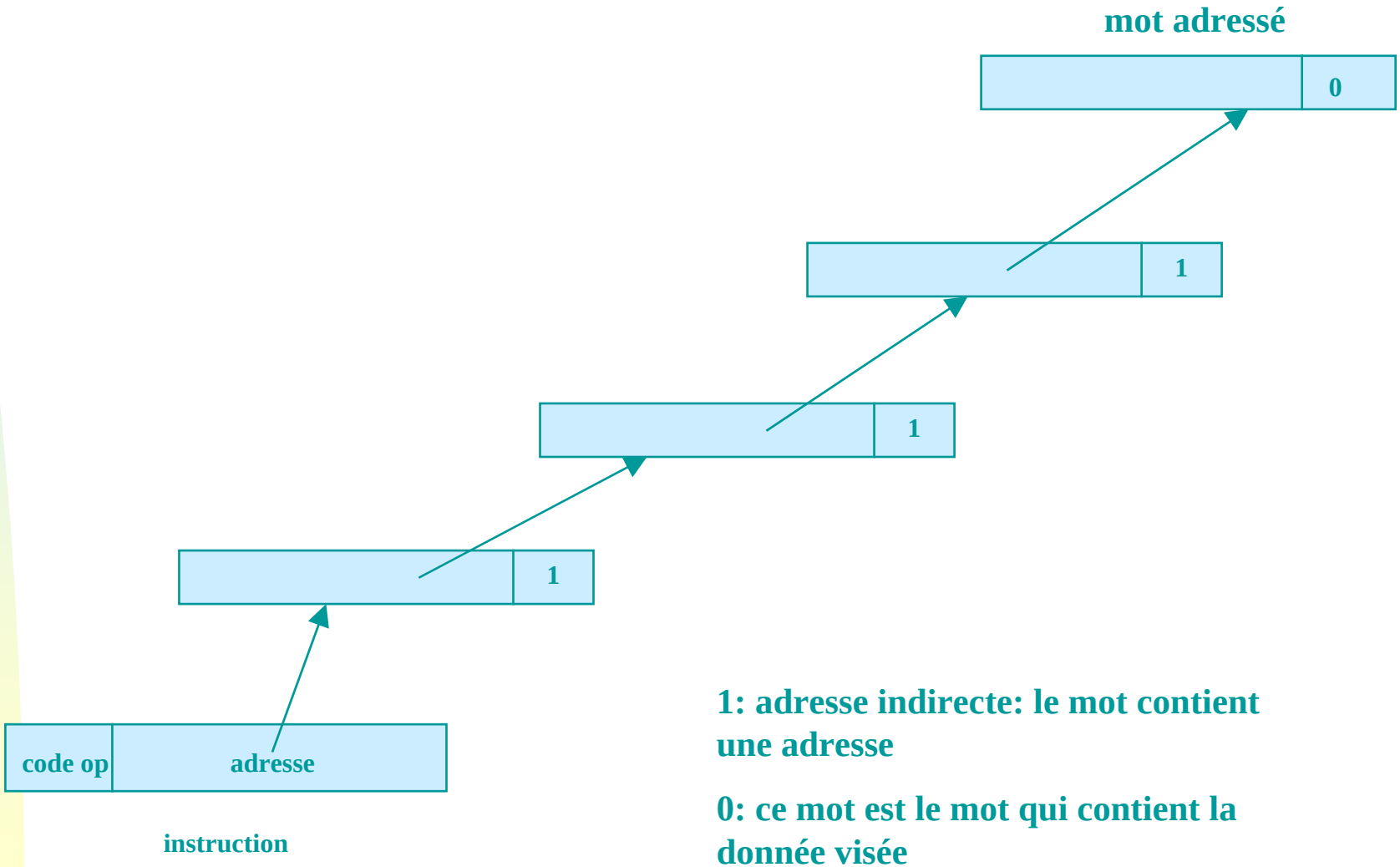


Chaque module contient deux tableaux

- un qui définit les **noms internes** qui peuvent être utilisés à l'extérieur
- un qui dit dans quels modules ses **noms externes** sont définis

Adressage indirecte
peut être utilisé dans cette méthode

(v. aussi chaînes d'adresses en C ou C++)

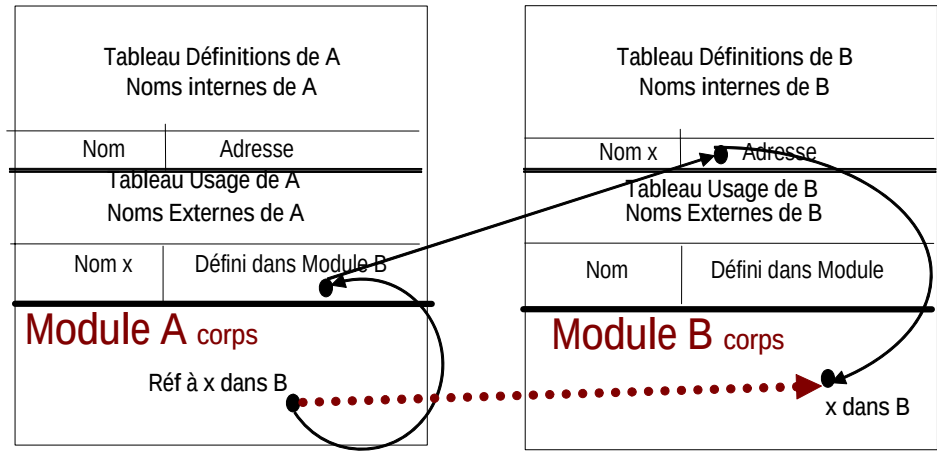


1: adresse indirecte: le mot contient une adresse

0: ce mot est le mot qui contient la donnée visée

Cette méthode supporte l'allocation dynamique

- Si le module B n'a pas encore été chargé, la référence à B dans le tableau des noms externes de A contiendra l 'information nécessaire pour trouver et charger B.
- Après avoir fait ça, cette référence sera traduite dans une adresse de mémoire physique.
- Si B reste toujours au même endroit, nous pouvons mettre dans A directement l'adresse finale (flèche pointillée), sinon nous pouvons continuer de rejoindre B par adressage indirecte

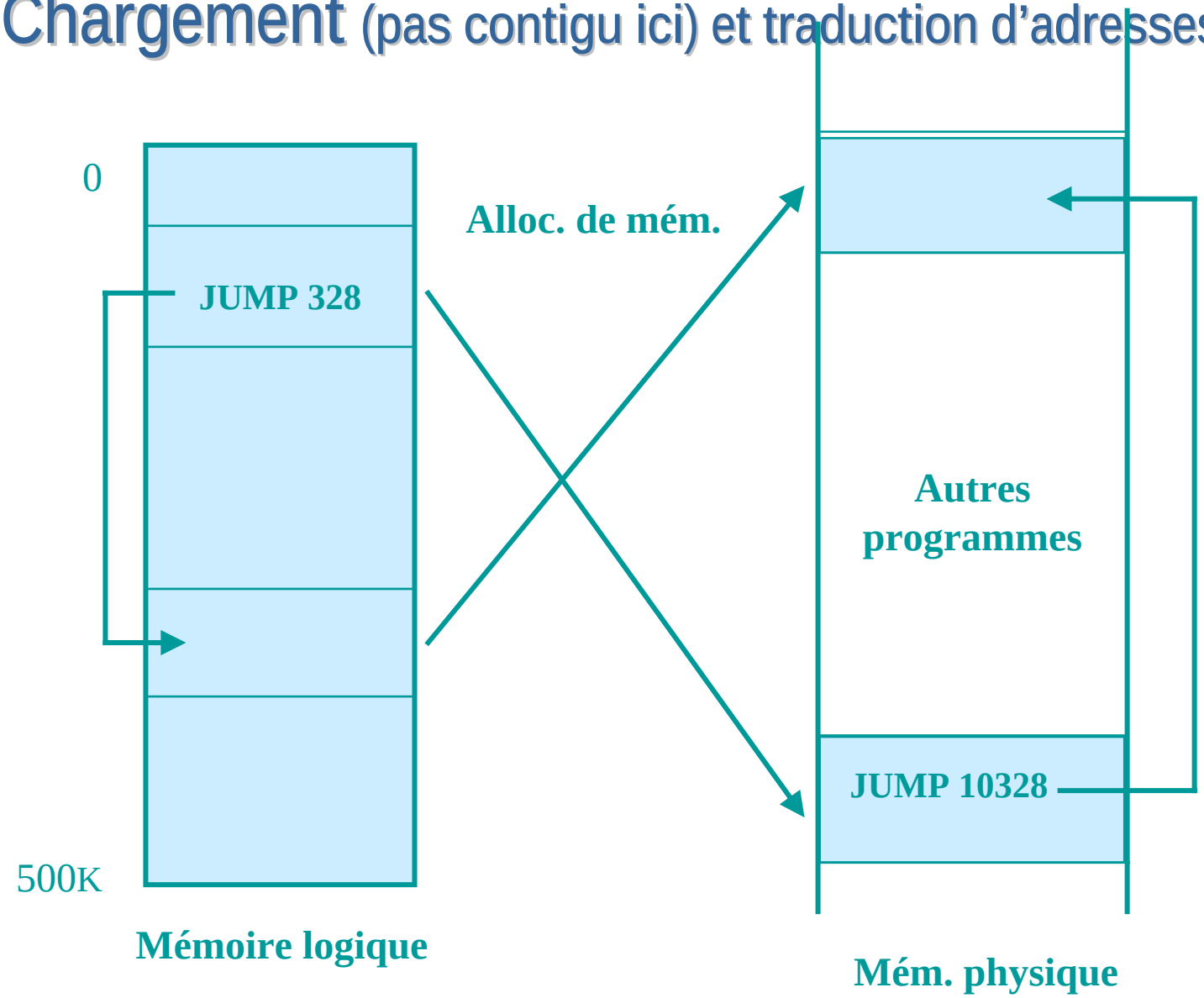


Aspects du chargement

- **Trouver de la mémoire libre pour un module de chargement: contiguë ou non**

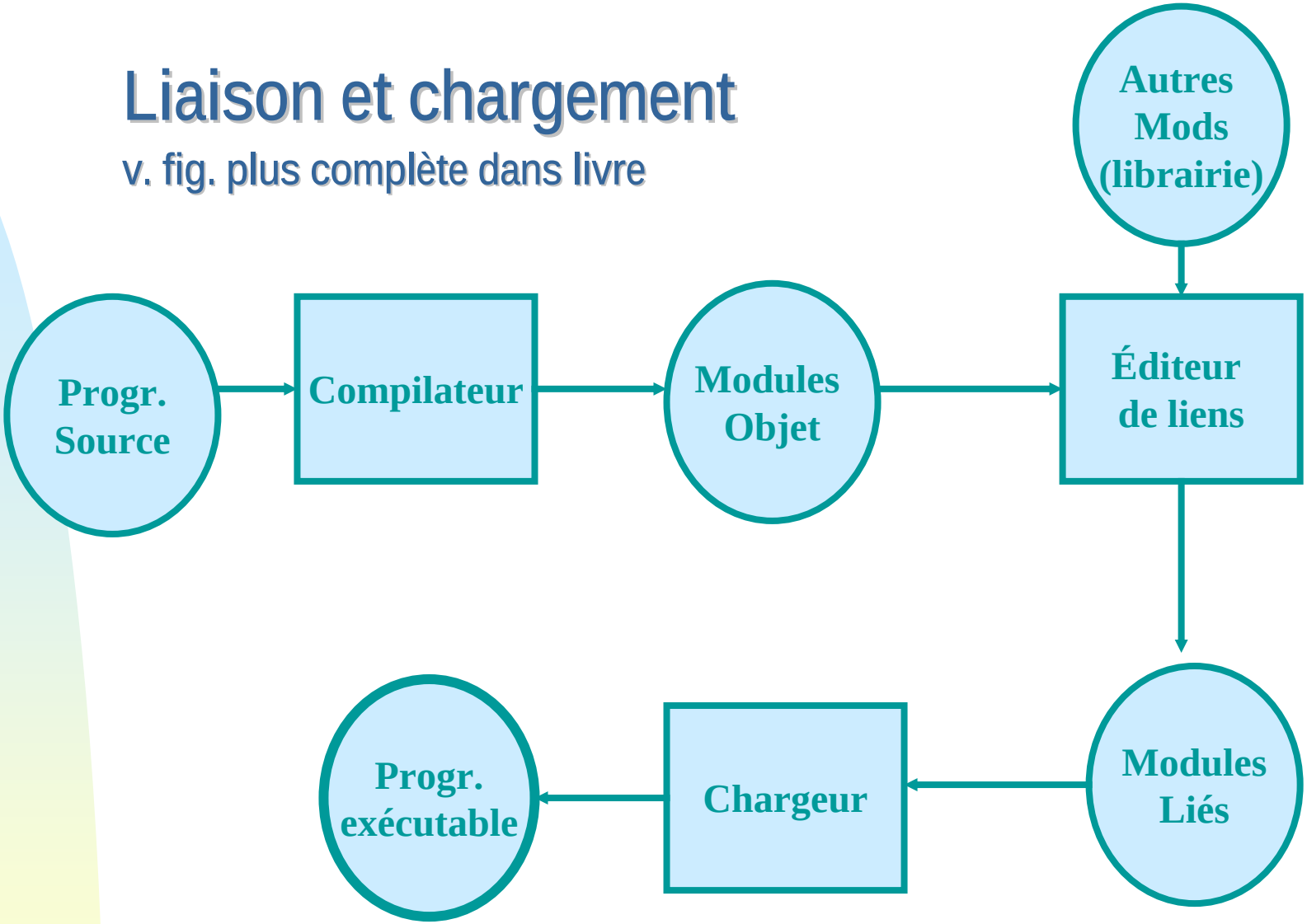
- **Traduire les adresses du programme et effectuer les liaisons par rapport aux adresses où le module est chargé**

Chargement (pas contigu ici) et traduction d'adresses



Liaison et chargement

v. fig. plus complète dans livre



NB: on fait l`hypothèse que tous les modules soient connus au début
Souvent, ce n'est pas le cas → chargement dynamique

Chargement et liaison dynamique

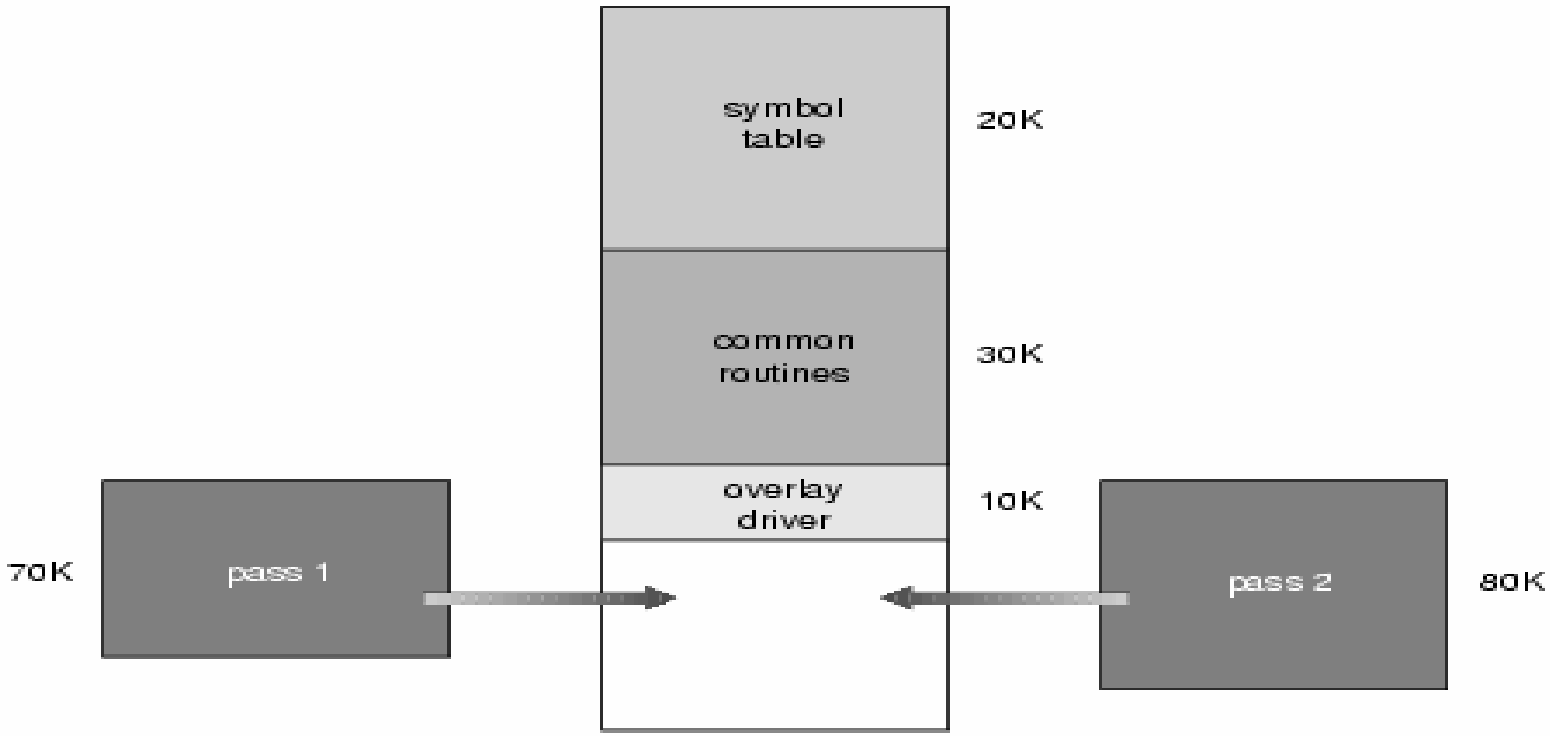
- **Un processus exécutant peut avoir besoin de différents modules du programme en différents moments**
- **Le chargement statique peut donc être inefficace**
- **Il est mieux de charger les modules sur demande = dynamique**
  - dll, dynamically linked libraries
- **Dans un programme qui peut avoir besoin de charger des modules dynamiquement, au début ces derniers sont représentés par des *stubs* qui indiquent comment arriver au modules (p.ex. où il se trouve: disque, www, autre...)**
- **À sa 1ère exéc. le stub cause le chargement du module en mémoire et sa liaison avec le reste du programme**
  - liaison dynamique
- **Les invocations successives du module ne doivent pas passer à travers ça, on saura l'adresse en mémoire**

Traduction d'adresses logique → physique

- **Dans les premiers systèmes, un programme était toujours chargé dans la même zone de mémoire**
- **La multiprogrammation et l'allocation dynamique ont engendré le besoin de charger un programme dans positions différentes**
- **Au début, ceci était fait par le chargeur (loader) qui changeait les adresses avant de lancer l 'exécution**
- **Aujourd'hui, ceci est fait par le MMU au fur et à mesure que le progr. est exécuté**
- **Ceci ne cause pas d'hausse de temps d 'exécution, car le MMU agit en parallèle avec autres fonctions d 'UCT**
  - P.ex. l 'MMU peut préparer l 'adresse d 'une instruction en même temps que l 'UCT exécute l 'instruction précédente

Recouvrement ou overlay

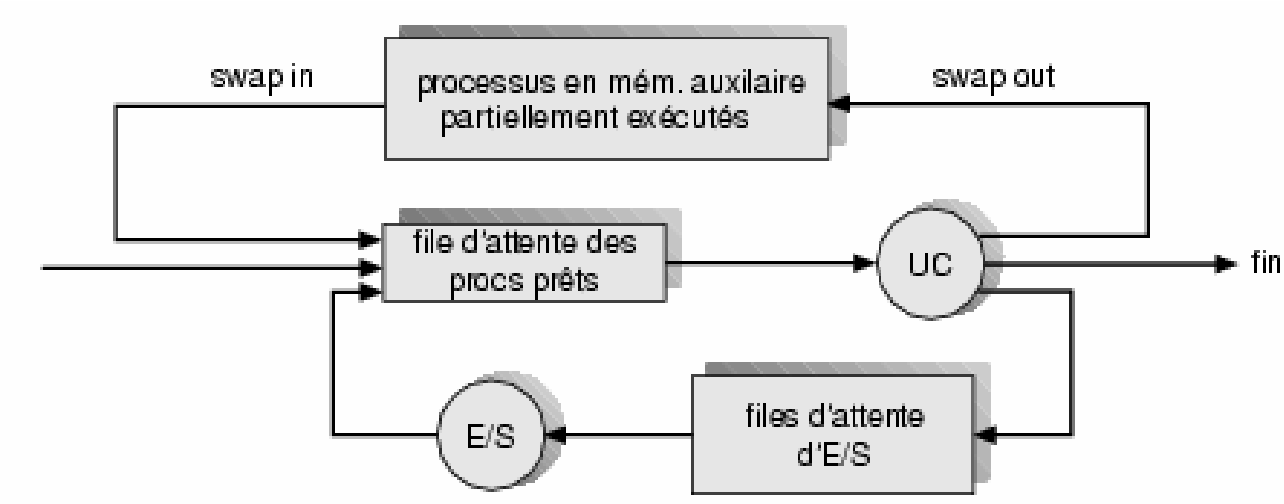
- **Dans quelques systèmes surtout dans le passé), la permutation de modules (swapping) d'un même programme pouvait être gérée par l 'usager**



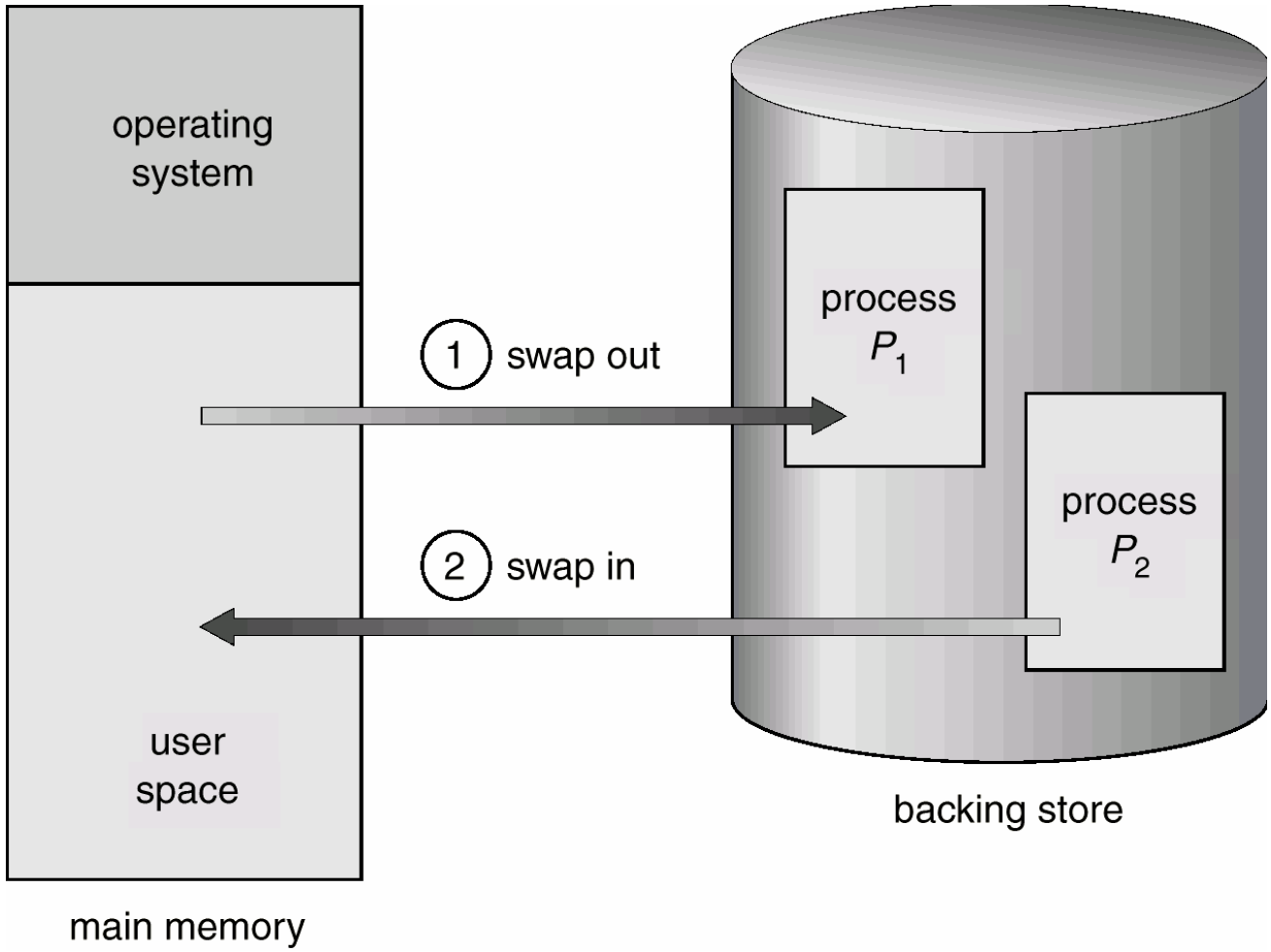
Deux parties d'un programme qui utilisent la même zone de mémoire

Permutation de programmes (swapping)

- **Un programme, ou une partie de programme, peut être temporairement enlevé de mémoire pour permettre l'exécution d'autres programmes (chap. 4)**
  - il est mis dans mémoire secondaire, normal. disque



Permutation de programmes (swapping)

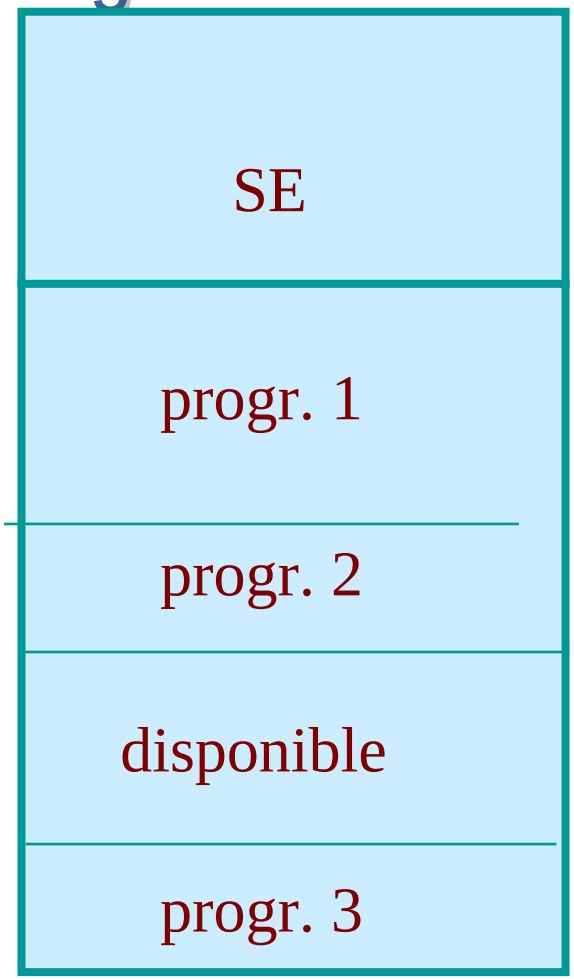


Affectation contiguë de mémoire

- **Nous avons plusieurs programmes à exécuter**
- **Nous pouvons les charger en mémoire les uns après les autres**
  - le lieu où un programme est lu n'est connu qu'au moment du chargement
- **Besoins de matériel: registres translation et registres bornes**

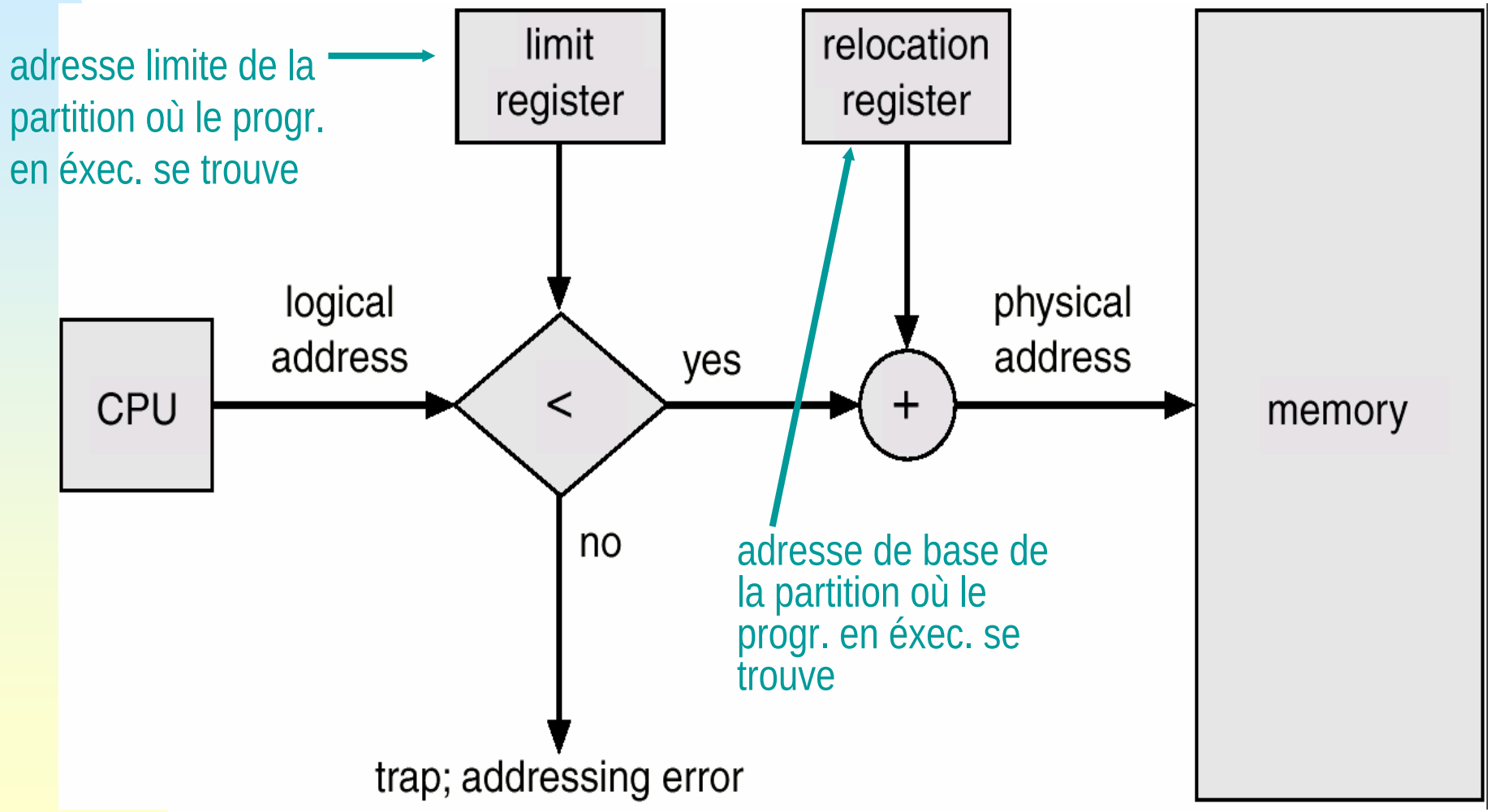
- **L'allocation contiguë n'est plus utilisée aujourd'hui pour la mémoire centrale, mais les concepts que nous verrons sont encore utilisés pour l'allocation de fichiers sur disques**

Affectation contiguë de mémoire



Nous avons ici 4 partitions pour des programmes - chacun est chargé dans une seule zone de mémoire

Registres bornes (ou limites) et translation dans MMU



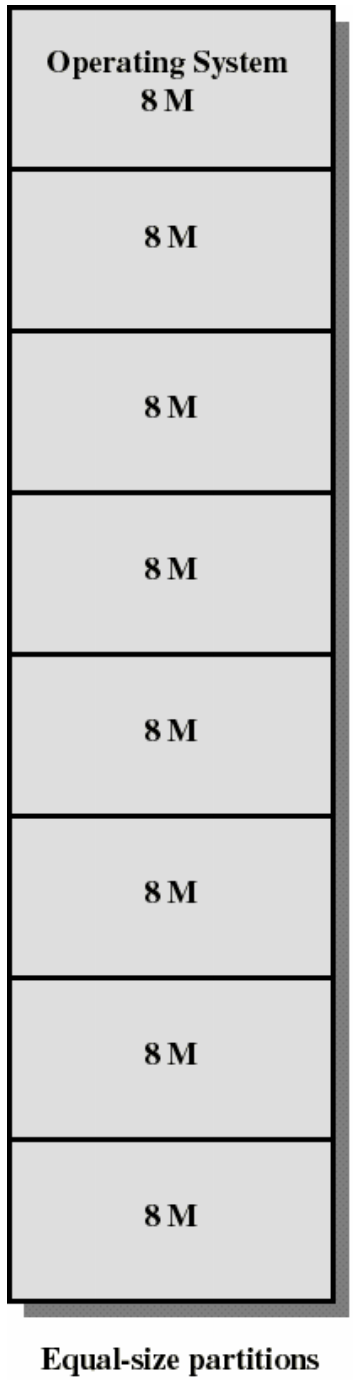
Fragmentation: mémoire non utilisée

- **Un problème majeur dans l`affectation contiguë:**
  - Il y a assez d ’espace pour exécuter un programme, mais il est fragmenté de façon non contiguë
    - externe: l`espace inutilisé est entre partitions
    - interne: l ’espace inutilisé est dans les partitions

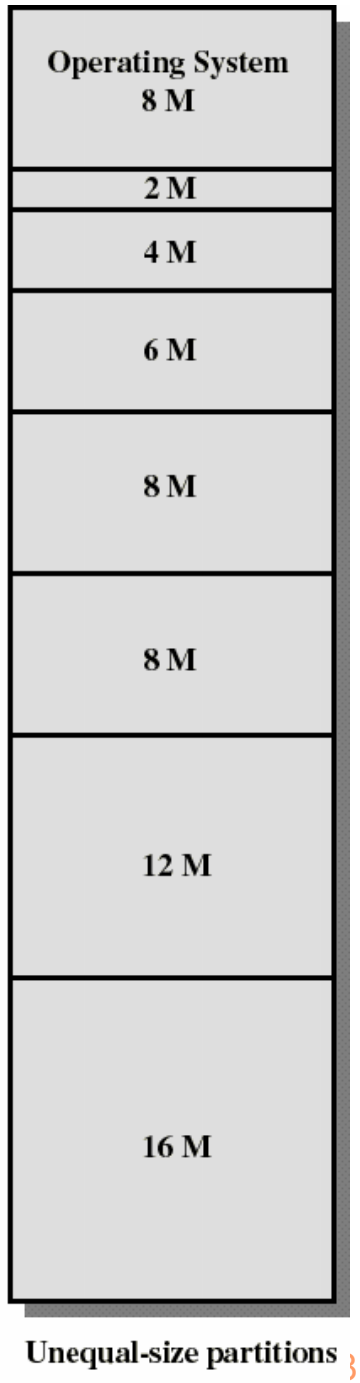
Partitions fixes

- Mémoire principale subdivisée en régions distinctes: partitions
- Les partitions sont soit de même taille ou de tailles inégales
- N'importe quel progr. peut être affecté à une partition qui soit suffisamment grande

(Stallings)



Equal-size partitions

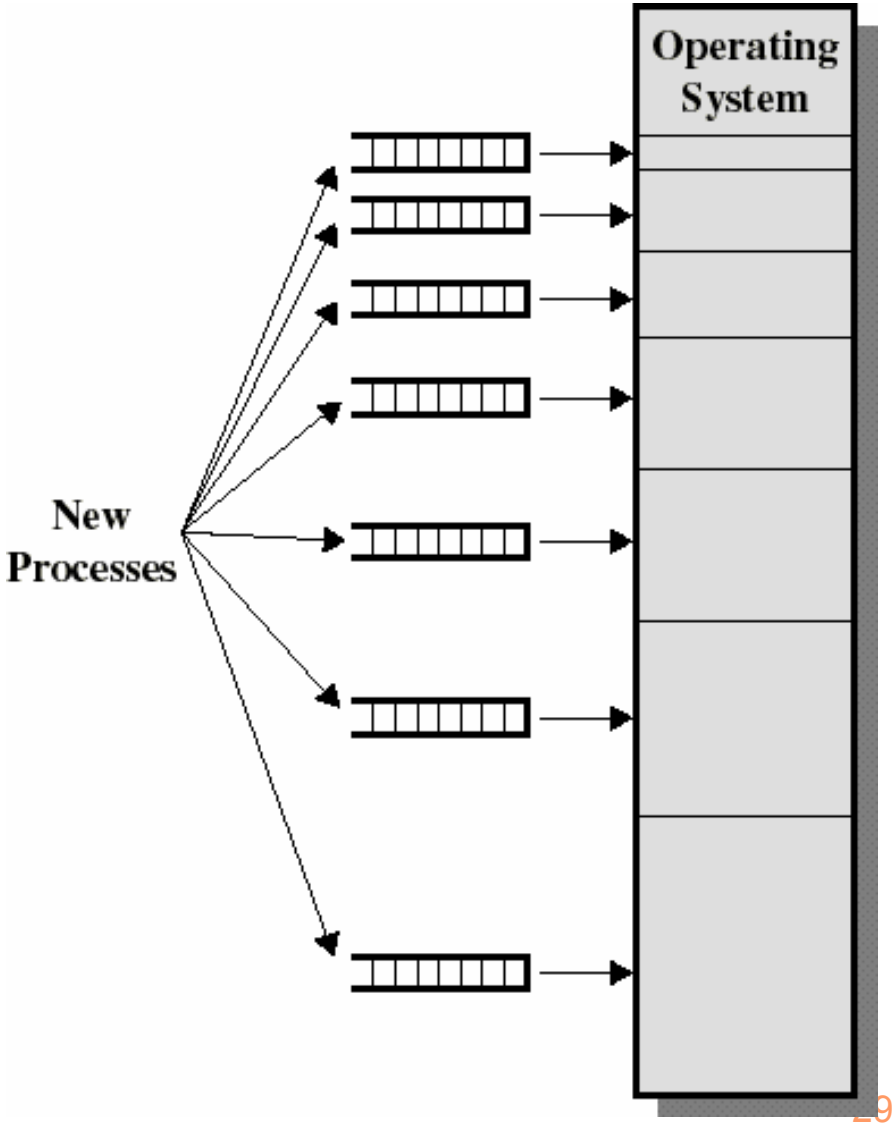


Unequal-size partitions

Algorithme de placement pour partitions fixes

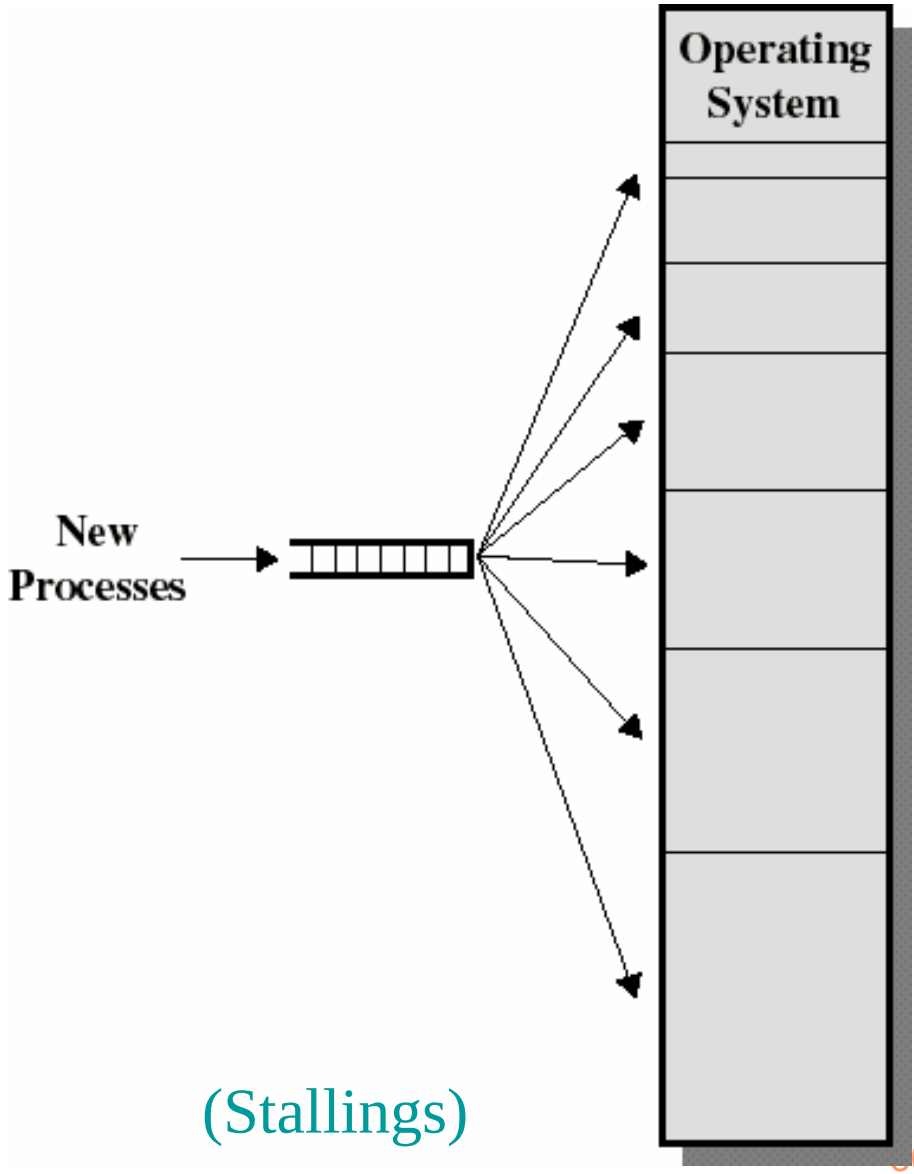
(Stallings)

- **Partitions de tailles inégales: utilisation de plusieurs files**
  - assigner chaque processus à la partition de la plus petite taille pouvant le contenir
  - 1 file par taille de partition
  - tente de minimiser la fragmentation interne
  - Problème: certaines files seront vides s'il n'y a pas de processus de cette taille (fr. externe)



Algorithme de placement pour partitions fixes

- **Partitions de tailles inégales: utilisation d'une seule file**
  - On choisit la plus petite partition libre pouvant contenir le prochain processus
  - le niveau de multiprogrammation augmente au profit de la fragmentation interne



(Stallings)

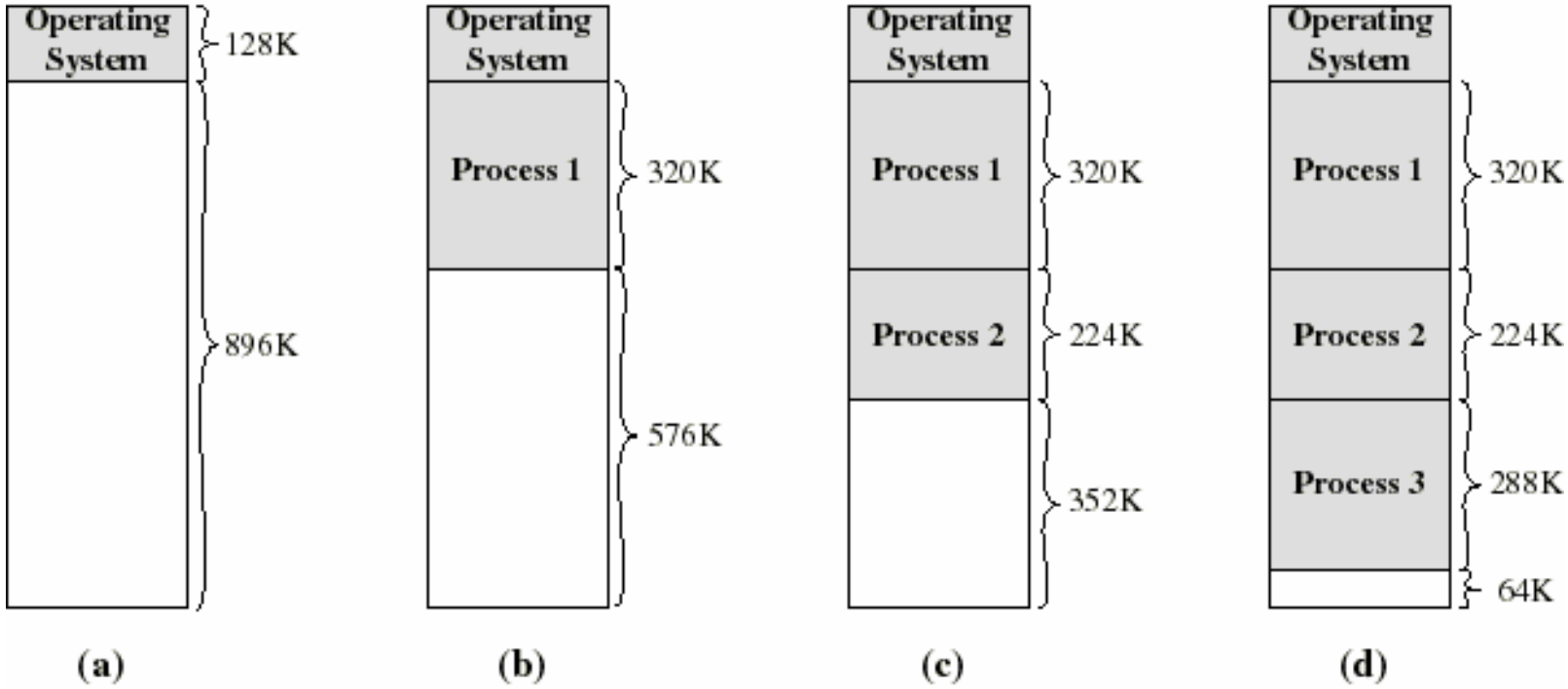
Partitions fixes

- **Simple, mais...**
- **Inefficacité de l'utilisation de la mémoire: tout programme, si petit soit-il, doit occuper une partition entière. Il y a fragmentation interne.**
- **Les partitions à tailles inégales atténue ces problèmes mais ils y demeurent...**

Partitions dynamiques

- **Partitions en nombre et tailles variables**
- **Chaque processus est alloué exactement la taille de mémoire requise**
- **Probablement des trous inutilisables se formeront dans la mémoire: c'est la fragmentation externe**

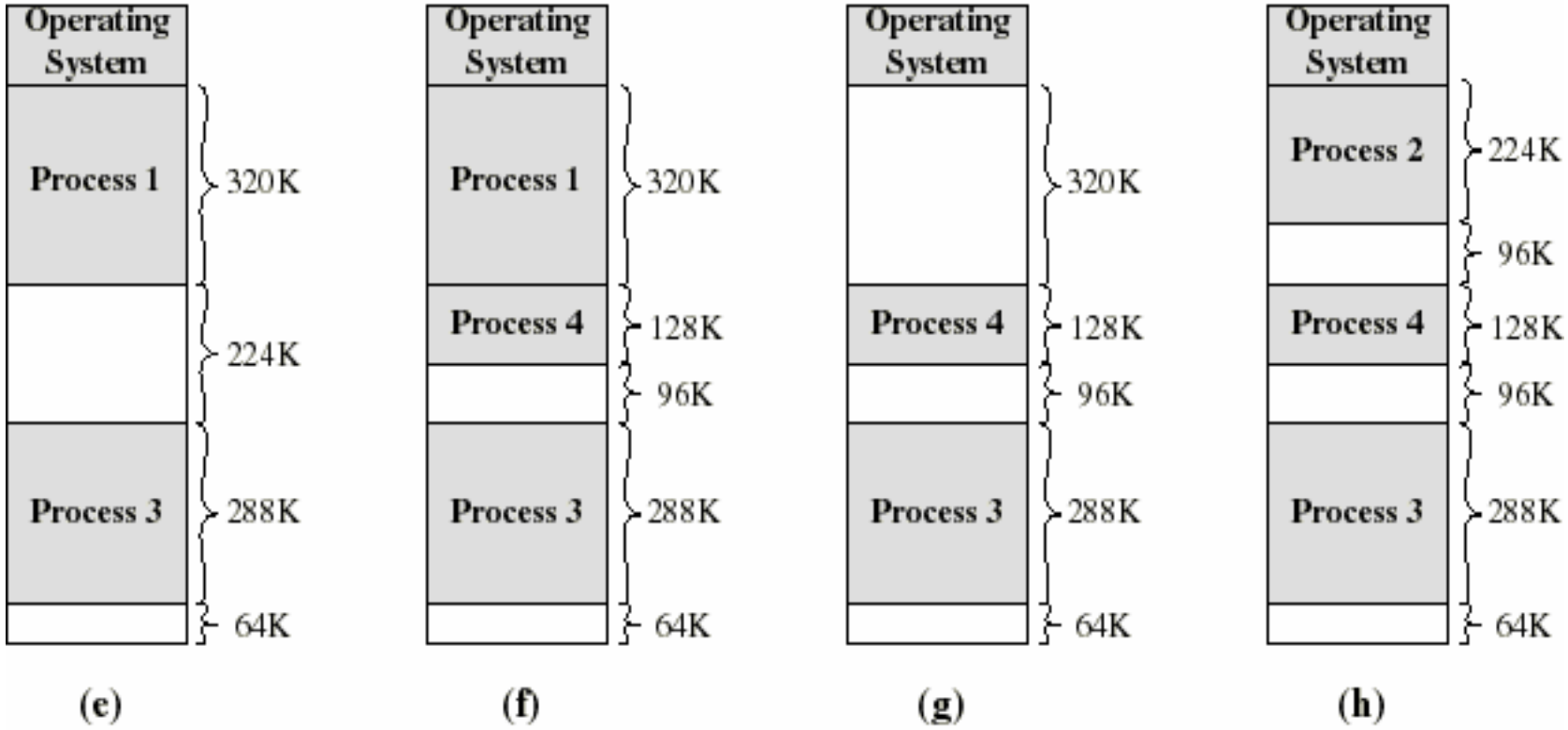
Partitions dynamiques: exemple (Stallings)



- **(d) Il y a un trou de 64K après avoir chargé 3 processus: pas assez d'espace pour autre processus**
- **Si tous les proc se bloquent (p.ex. attente d'un événement), P2 peut être permuté et P4=128K peut être chargé.**

Swapped out

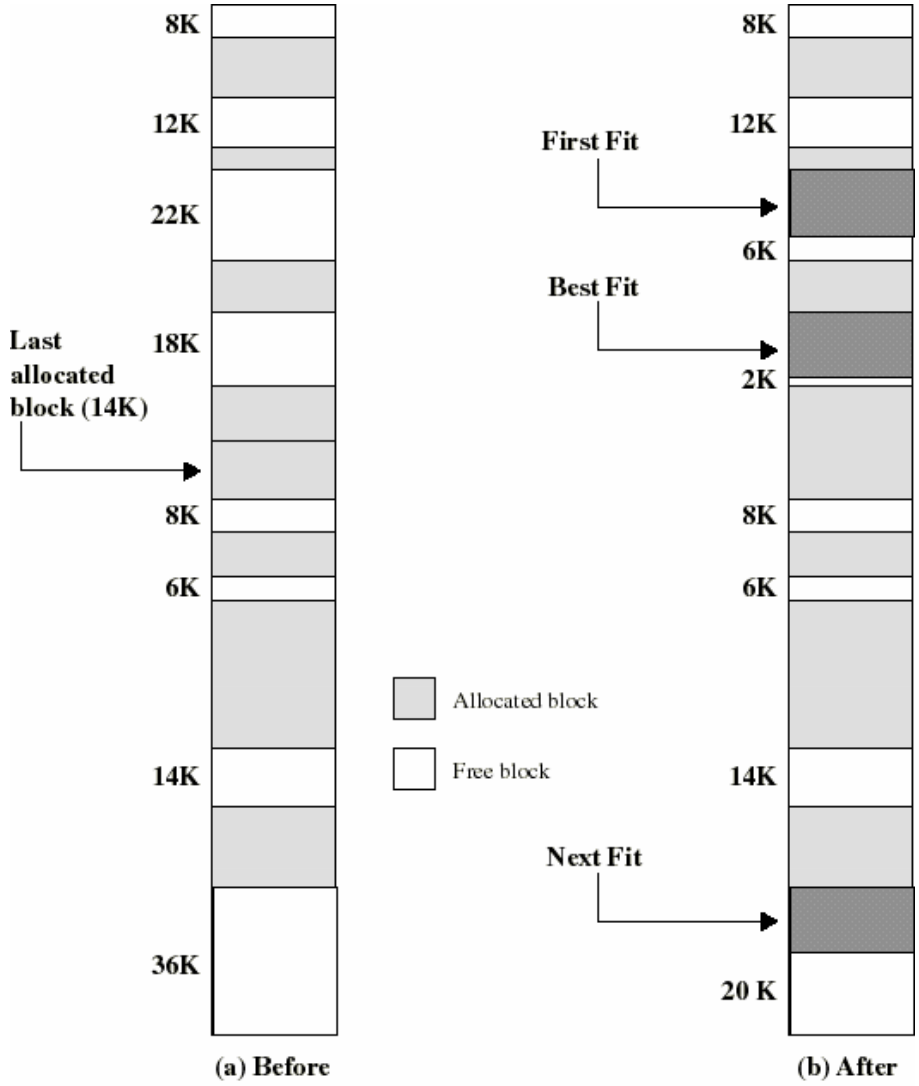
Partitions dynamiques: exemple (Stallings)



- (e-f) Progr. 2 est suspendu, Progr. 4 est chargé. Un trou de 224-128=96K est créé (*fragmentation externe*)
- (g-h) P1 se termine ou il est suspendu, P2 est repris à sa place: produisant un autre trou de 320-224=96K...
- Nous avons 3 trous petits et probabl. inutiles. 96+96+64=256K de fragmentation externe
- **COMPRESSION** pour en faire un seul trou de 256K

Algorithmes de Placement

- **pour décider de l'emplacement du prochain processus**
- **But: réduire l'utilisation de la compression (prend du temps...)**
- **Choix possibles:**
  - Best fit: choisir le plus petit trou (meilleur accès)
  - Worst fit: le plus grand (pire accès)
  - First-fit: choisir 1er trou à partir du début (premier accès)
  - Next-fit: choisir 1er trou à partir du dernier placement (prochain accès)



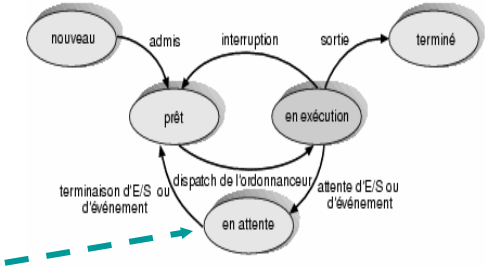
Example Memory Configuration Before and After Allocation of 16 Kbyte Block

(Stallings)

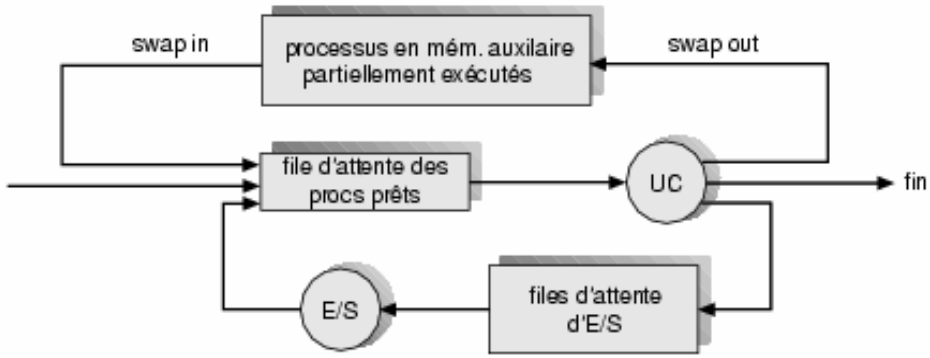
Algorithmes de placement: commentaires

- **Quel est le meilleur?**
  - critère principal: diminuer la probabilité de situations où un processus ne peut pas être servi, même s 'il y a assez de mémoire...
- **Best-fit: cherche le plus petit bloc possible: le trou créé est le plus petit possible**
  - la mémoire se remplit de trous trop petits pour contenir un programme
- **Worst-fit: le trous crées seront les plus grands possibles**
- **Next-fit: les allocations se feront souvent à la fin de la mémoire**
- **La simulation montre qu` il ne vaut pas la peine d 'utiliser les algo les plus complexes... donc first fit**

Suspension (v. chap 4)



- **Lorsque tous les programmes en mémoire sont bloqués, le SE peut en suspendre un (swap/suspend)**
  - On transfère au disque un des processus bloqués (en le mettant ainsi en état suspended) et le remplacer par un processus prêt à être exécuté
    - ce dernier processus exécute une transition d'état Nouveau ou Suspendu à état Ready



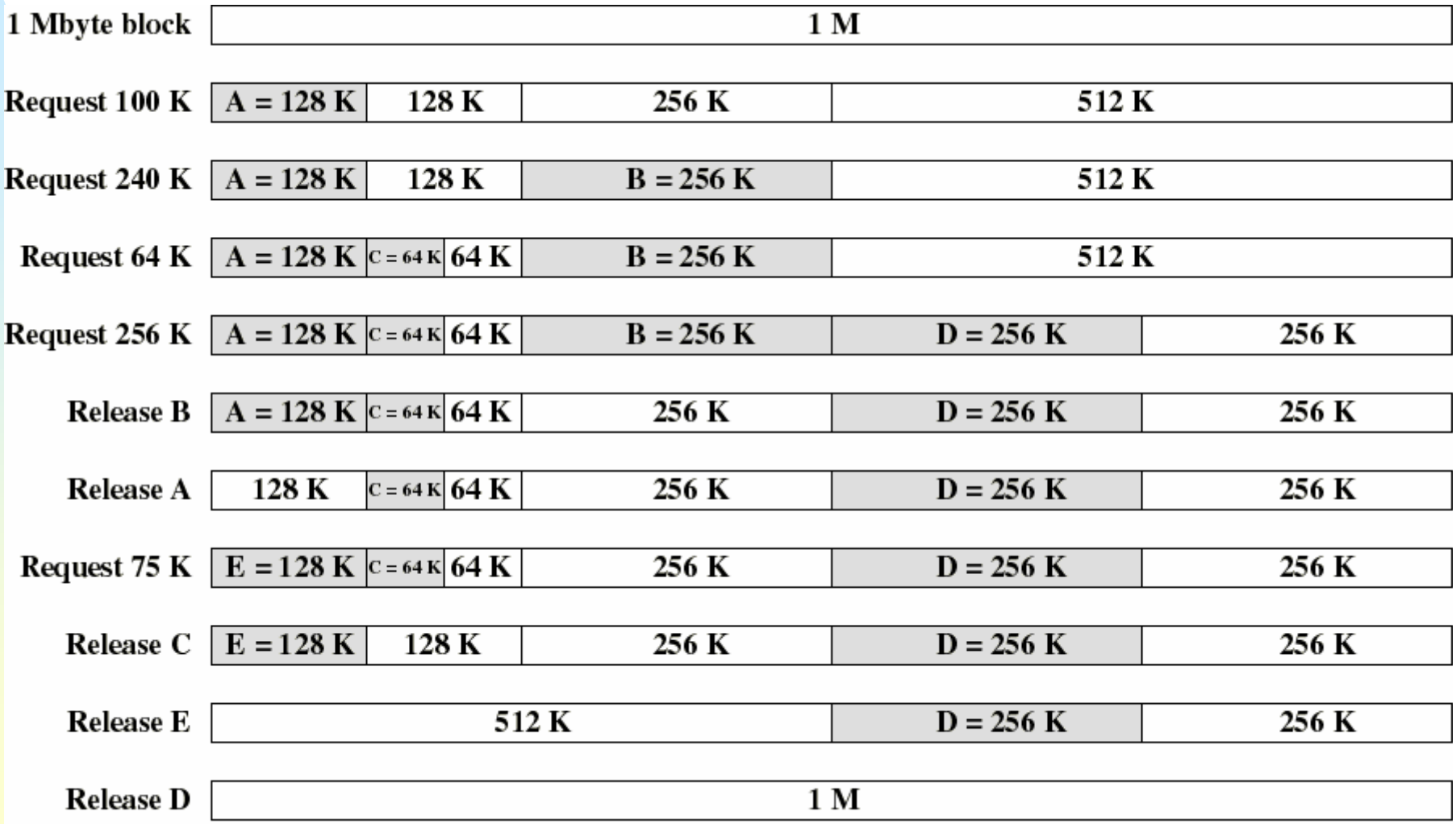
Compression (compaction)

- **Une solution pour la fragmentation externe**
- **Les programmes sont déplacés en mémoire de façon à réduire à 1 seul grand trou plusieurs petits trous disponibles**
- **Effectuée quand un programme qui demande d'être exécuté ne trouve pas une partition assez grande, mais sa taille est plus petite que la fragmentation externe existante**
- **Désavantages:**
  - temps de transfert programmes
  - besoin de rétablir tous les liens entre adresses de différents programmes

Système de «groupe de paires » (buddy systems): une approche complètement différente (Unix, Linux, sect. 21.6.1)

- **Débutons avec un seul gros block de taille 2^U**
- **Sur une requête pour un block de taille S:**
  - Si $2^{U-1} < S <= 2^U$ alors allouer le block entier de taille 2^U
  - Sinon, partager ce block en deux compagnons (buddies), chacun de taille 2^{U-1}
  - Si $2^{U-2} < S <= 2^{U-1}$ alors allouer un des deux compagnons
  - Sinon diviser un de ces 2 compagnons
- **Le processus est répété jusqu'à ce que le plus petit block pouvant contenir S soit généré**
- **Deux compagnons sont fusionnés lorsqu'ils deviennent tous deux non alloués**

Exemple d'un "Groupe de paires" (Stallings)



Système de groupes de paires (Buddy system)

- **Le SE maintient plusieurs listes de trous**
  - la i-liste est la liste des trous de taille 2^i
  - lorsqu'une paire de compagnons se trouvent dans une i-liste, ils sont enlevés de cette liste et deviennent un seul trou de la (i+1)-liste
- **Sur une requête pour une allocation d'un block de taille k tel que $2^{i-1} < k <= 2^i$:**
  - on examine d'abord la i-liste
  - si elle est vide, on tente de trouver un trou dans la (i+1)-liste, si trouvé, il sera divisé en 2

Comment trouver le compagnon

- **Étant donné**
  - L'adresse binaire d'un bloc
  - La longueur du bloc
  - Comment trouver son compagnon
- **Si la longueur du bloc est 2^n**
  - Si le bit n de l'adresse est 0, changer à 1 (compagnon est à droite)
  - Si le bit n de l'adresse est 1, changer à 0 (compagnon est à gauche)
- **Exemples:**
  - 011011110000 longueur 100: 011011110100
  - 011011110000 longueur 10000: 011011100000
- (Compter les bits à partir de la droite et de 0)

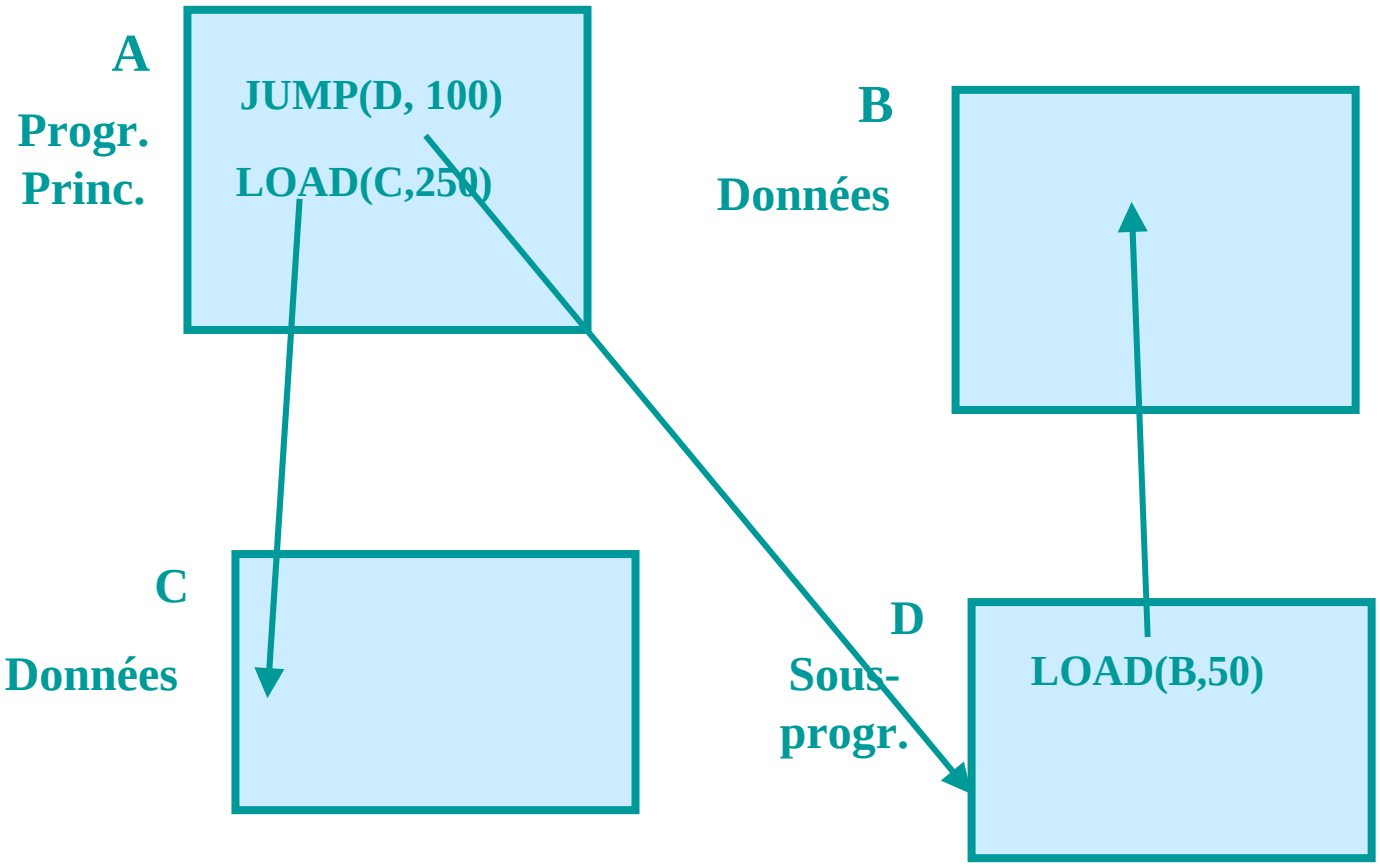
Groupes de paires: remarques

- **En moyenne, la fragmentation interne est de 25% par processus**
  - en moyenne pour chaque proc nous aurons une partition pleine de la grandeur d`une puissance de deux, plus une utilisée à moitié
  - + il pourra y avoir aussi des blocs non utilisés s'il n'y a pas de programmes en attente pouvant les utiliser.
- **Pas besoin de compression**
  - simplifie la gestion de la mémoire

Allocation non contiguë

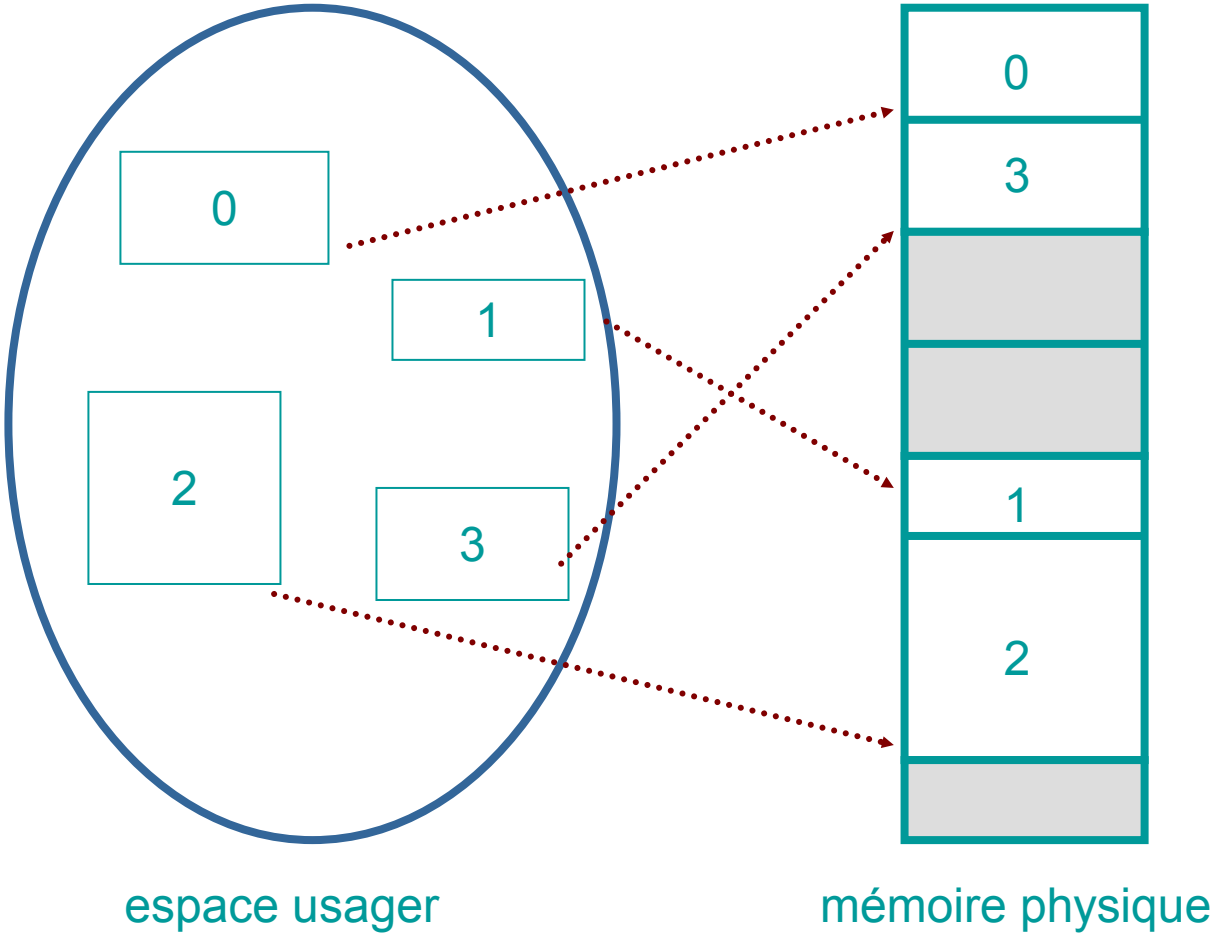
- **A fin de réduire la fragmentation, tous les ordis d'aujourd'hui utilisent l'allocation non contiguë**
  - diviser un programme en morceaux et permettre l`allocation séparée de chaque morceau
  - les morceaux sont beaucoup plus petits que le programme entier et donc permettent une utilisation plus efficace de la mémoire
    - les petits trous peuvent être utilisés plus facilement
- **Il y a deux techniques de base pour faire ceci: la pagination et la segmentation**
  - la segmentation utilise des parties de programme qui ont une valeur logique (des modules)
  - la pagination utilise des parties de programme arbitraires (morcellement du programmes en pages de longueur fixe).
  - elles peuvent être combinées
- **Je trouve que la segmentation est plus naturelle, donc je commence par celle-ci**

Les segments sont des parties logiques du progr.



4 segments: A, B, C, D

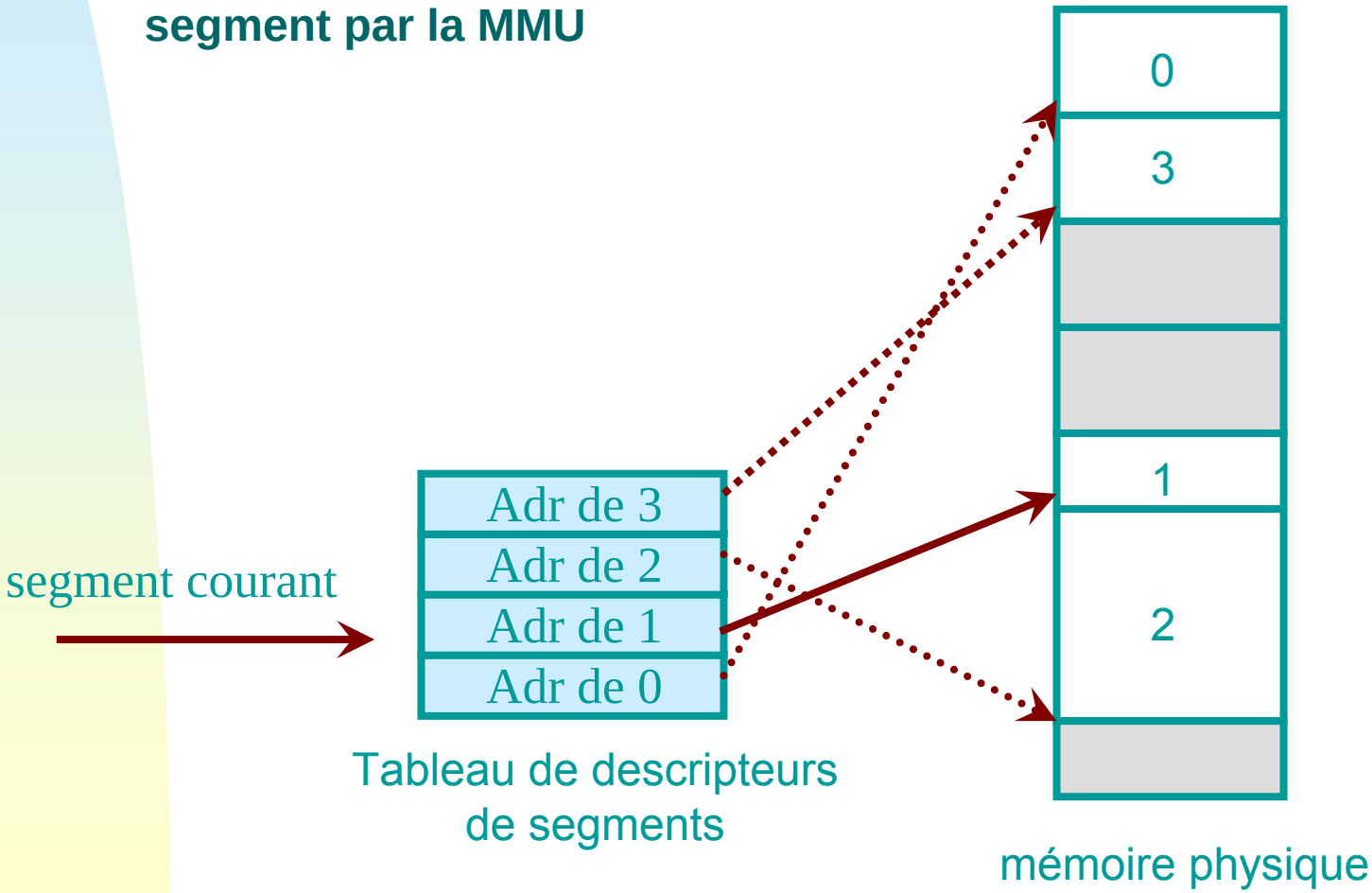
Les segments comme unités d'alloc mémoire



Étant donné que les segments sont plus petits que les programmes entiers, cette technique implique moins de fragmentation (qui est externe dans ce cas)

Mécanisme pour la segmentation

- **Un tableau contient l'adresse de début de tous les segments dans un processus**
- **Chaque adresse dans un segment est ajoutée à l 'adresse de début du segment par la MMU**

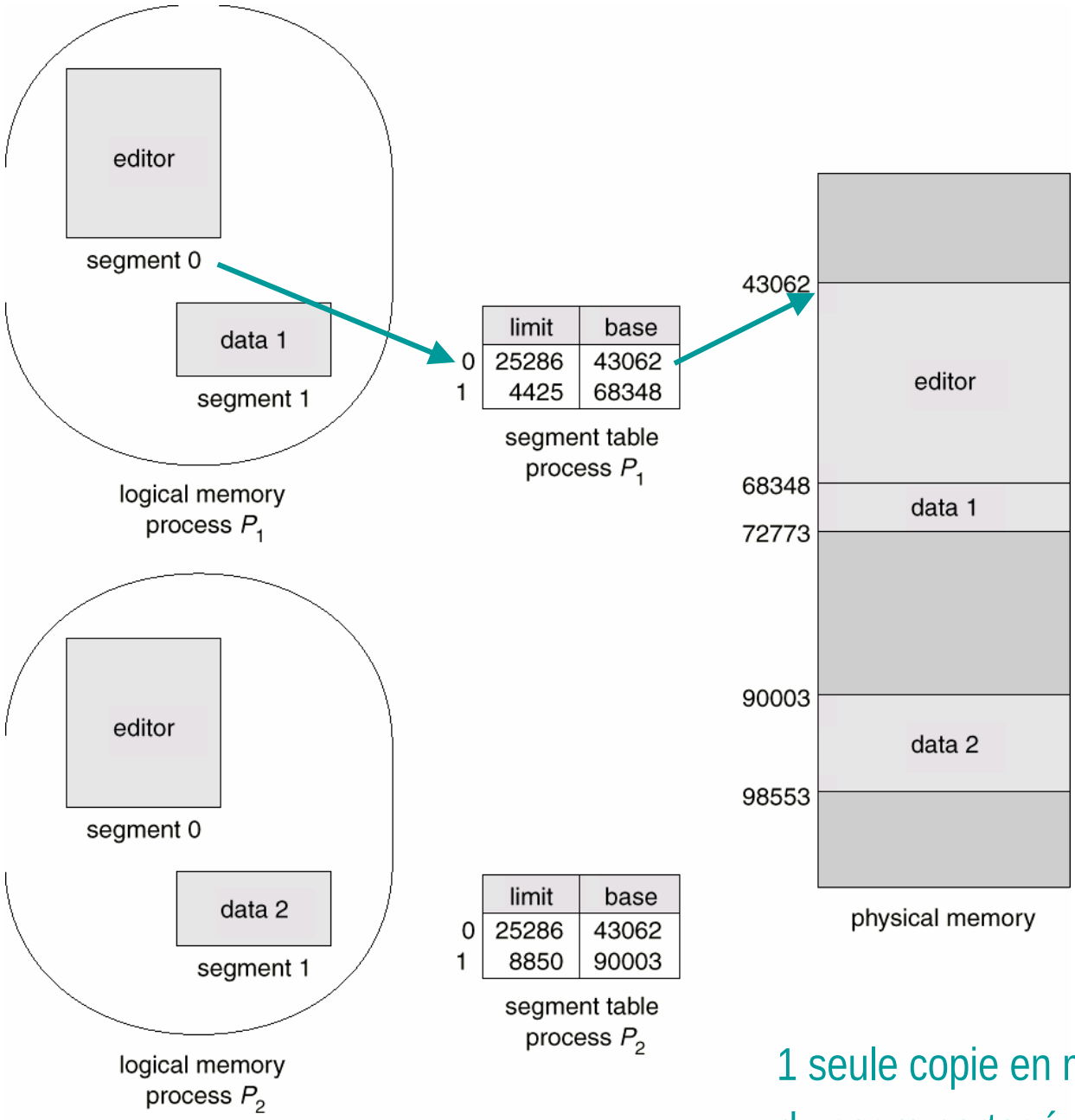


Détails

- **L'adresse logique consiste d 'une paire:**

  <No de segm, décalage>

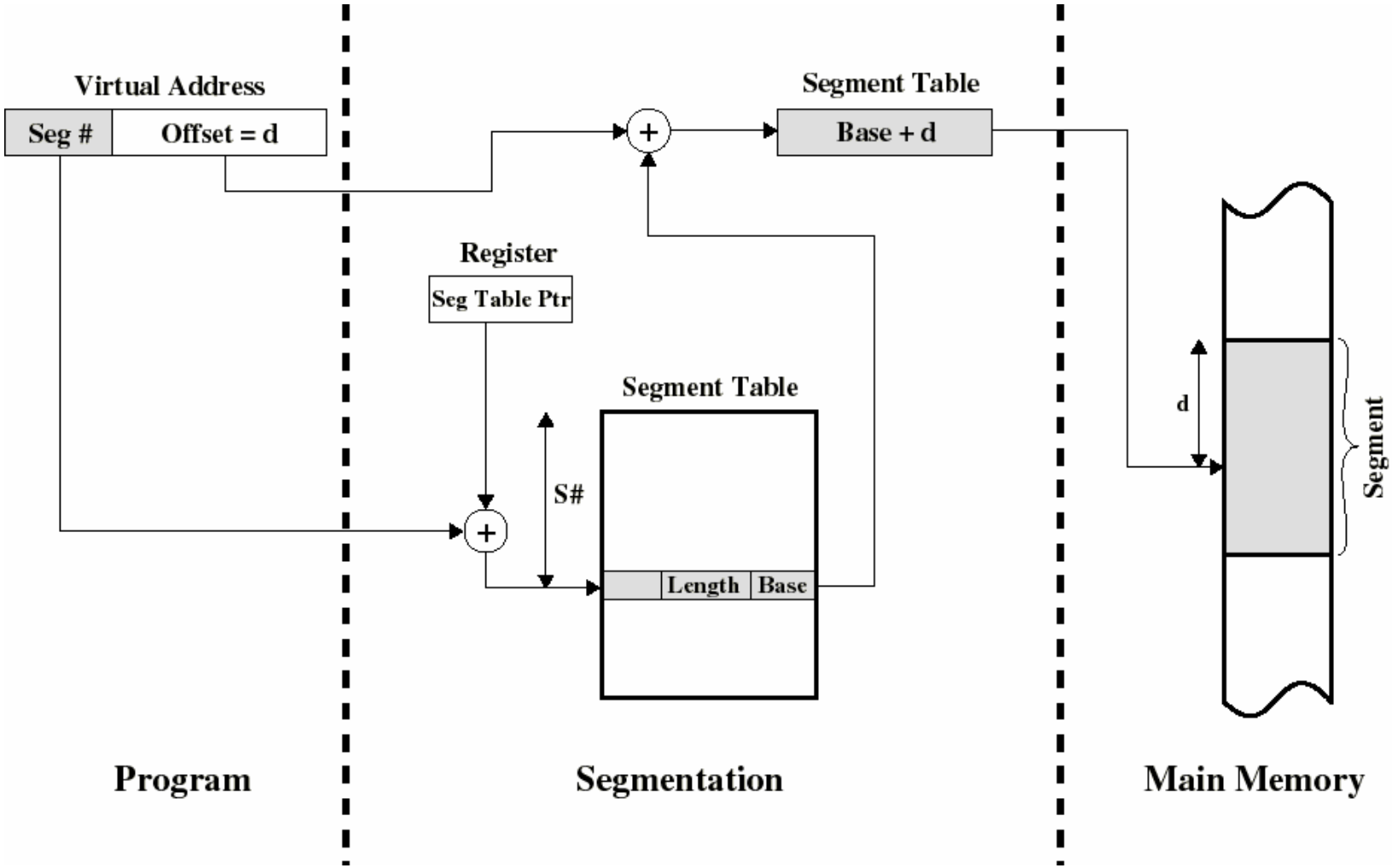
  où décalage est l 'adresse *dans* le segment
- **Le tableau des segments contient: descripteurs de segments**
  - adresse de base
  - longueur du segment
  - Infos de protection, on verra…
- **Dans le PCB du processus il y aura un pointeur à l 'adresse en mémoire du tableau des segments**
- **Il y aura aussi là dedans le nombre de segments dans le processus**
- **Au moment de la commutation de contexte, ces infos seront chargées dans les registres appropriés d'UCT**



1 seule copie en mémoire du segm partagé

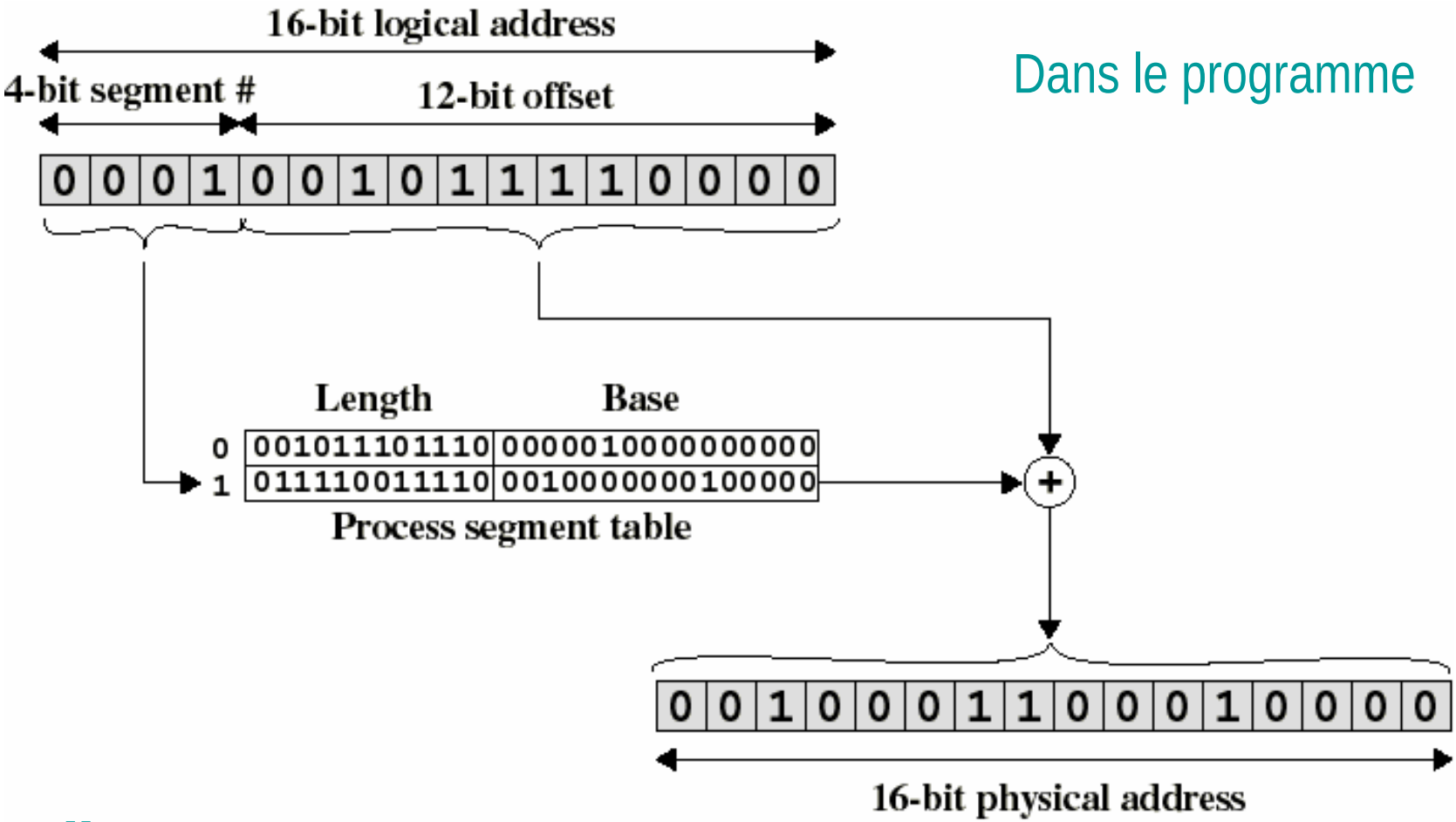
Traduction d`adresses dans la segmentation

(figure de Stallings: légère diff. par rapport à la terminologie de Silberschatz)



Aussi, si d > longueur: erreur!

Le mécanisme en détail (implanté dans le matériel)

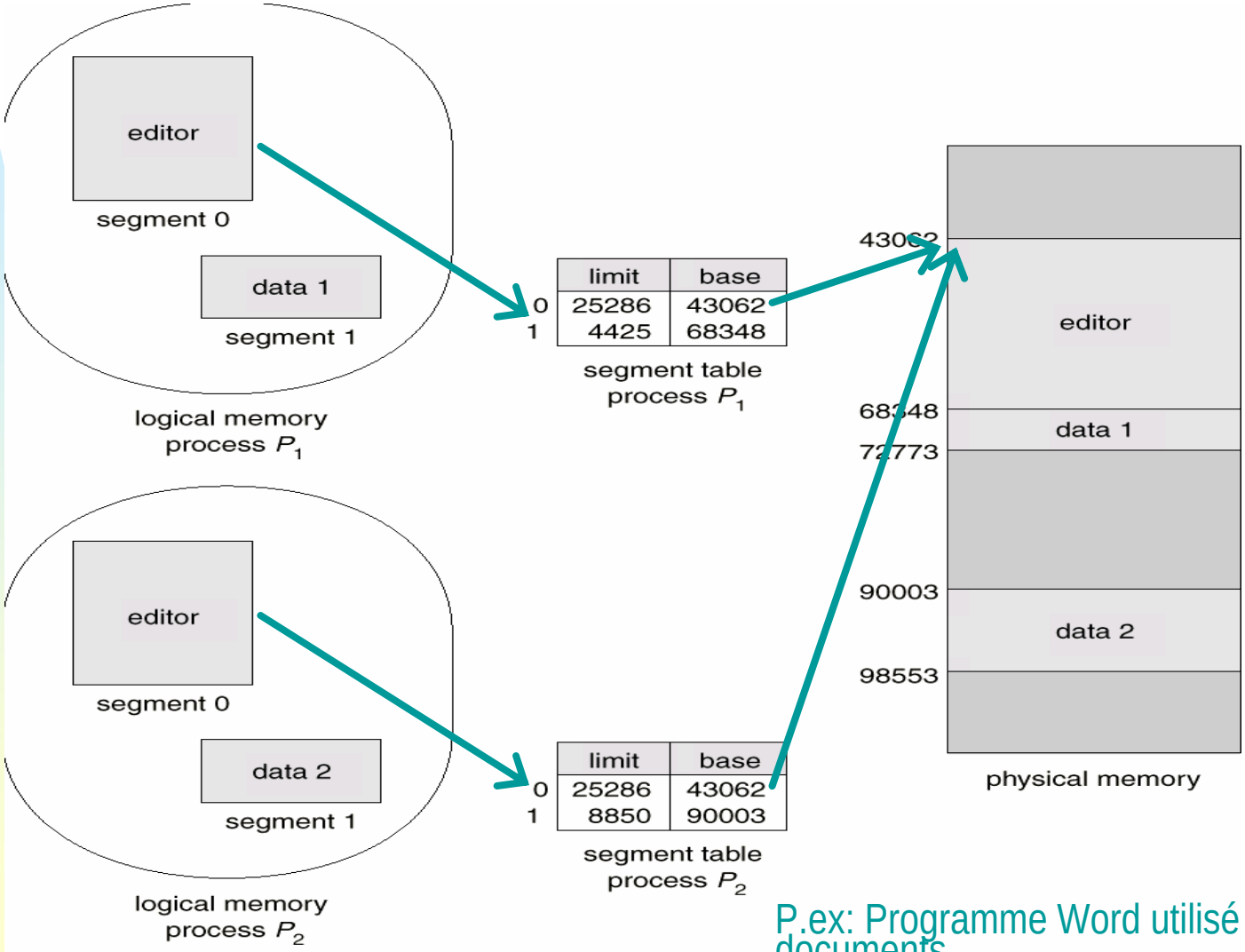


Dans le programme

Stallings

Adresse finale

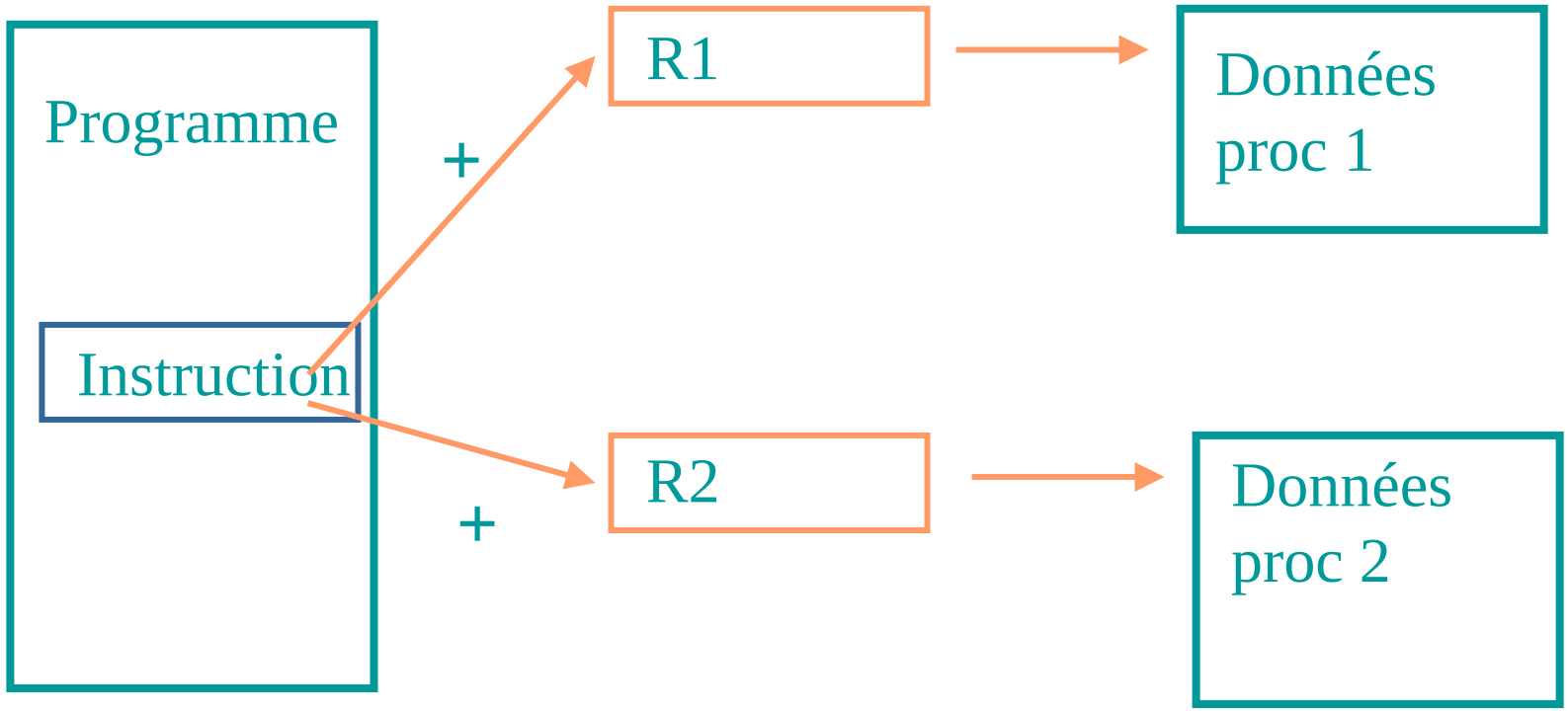
Partage de segments: le segment 0 est partagé



P.ex: Programme Word utilisé pour éditer différents documents

DLL utilisé par plus usagers

Mécanisme pour 2 processus qui exécutent un seul programme sur données différentes



La même instruction, si elle est exécutée

- par le proc 1, son adresse est modifiée par le contenu du registre 1
- par le proc 2, son adresse est modif par le contenu du registre 2

Ceci fonctionne même si l'instruction est exécutée par plus. UCT au même instant, si les registres se trouvent dans des UCT différentes

Segmentation et protection

- **Chaque *descripteur de segment* peut contenir des infos de protection:**
  - longueur du segment
  - privilèges de l`usager sur le segment: lecture, écriture, exécution
    - Si au moment du calcul de l'adresse on trouve que l'usager n'a pas droit d'accès→interruption
    - ces infos peuvent donc varier d 'usager à usager, par rapport au même segment!

Évaluation de la segmentation simple

- **Avantages: l'unité d'allocation de mémoire est**
  - plus petite que le programme entier
  - une entité logique connue par le programmeur
  - les segments peuvent changer de place en mémoire
  - la protection et le partage de segments sont aisés (en principe)
- **Désavantage: le problème des partitions dynamiques:**
  - La fragmentation externe n'est pas éliminée:
    - trous en mémoire, compression?
- **Une autre solution est d`essayer à simplifier le mécanisme en utilisant unités d`allocation mémoire de tailles égales**

    - **PAGINATION**

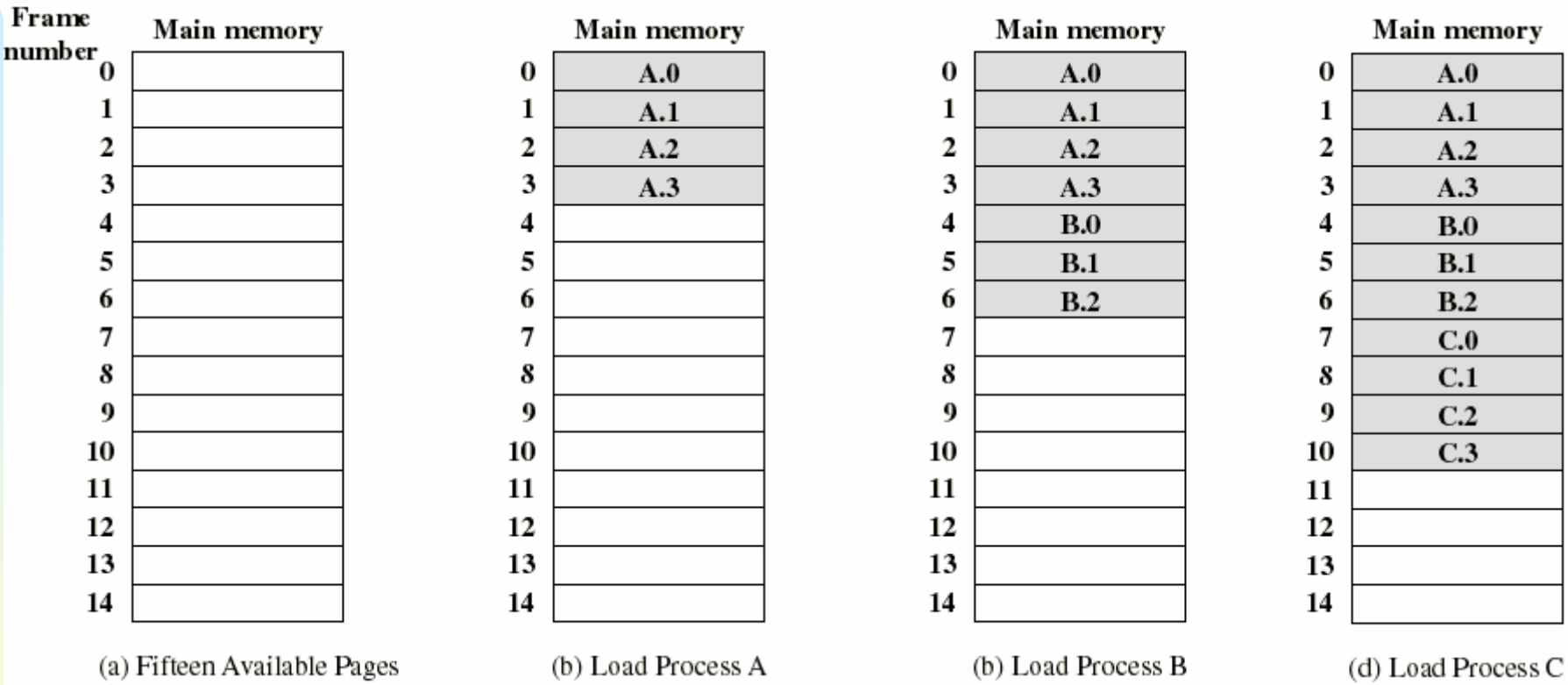
Segmentation contre pagination

- **Le pb avec la segmentation est que l'unité d'allocation de mémoire (le segment) est de longueur variable**
- **La pagination utilise des unités d'allocation de mémoire fixe, éliminant donc ce pb**

Pagination simple

- **La mémoire est partitionnée en petits morceaux de même taille: les pages physiques ou 'cadres' ou 'frames'**
- **Chaque processus est aussi partitionné en petits morceaux de même taille appelés pages (logiques)**
- **Les pages logiques d'un processus peuvent donc être assignés aux cadres disponibles n'importe où en mémoire principale**
- **Conséquences:**
  - un processus peut être éparpillé n'importe où dans la mémoire physique.
  - la fragmentation **externe** est éliminée

Exemple de chargement de processus

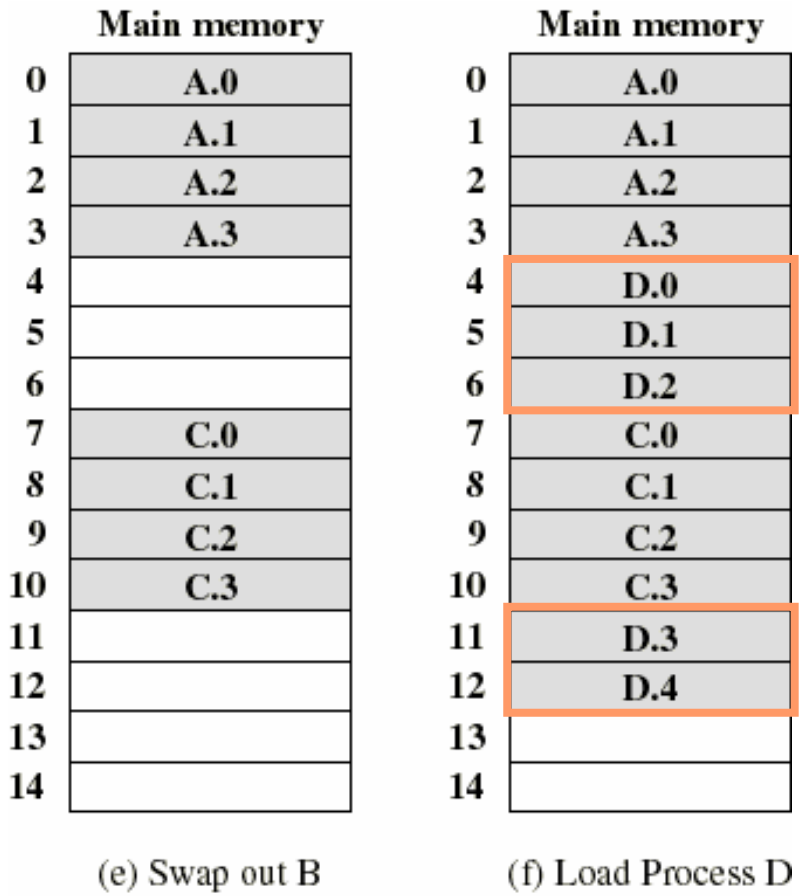


- **Supposons que le processus B se termine ou est suspendu**

Stallings

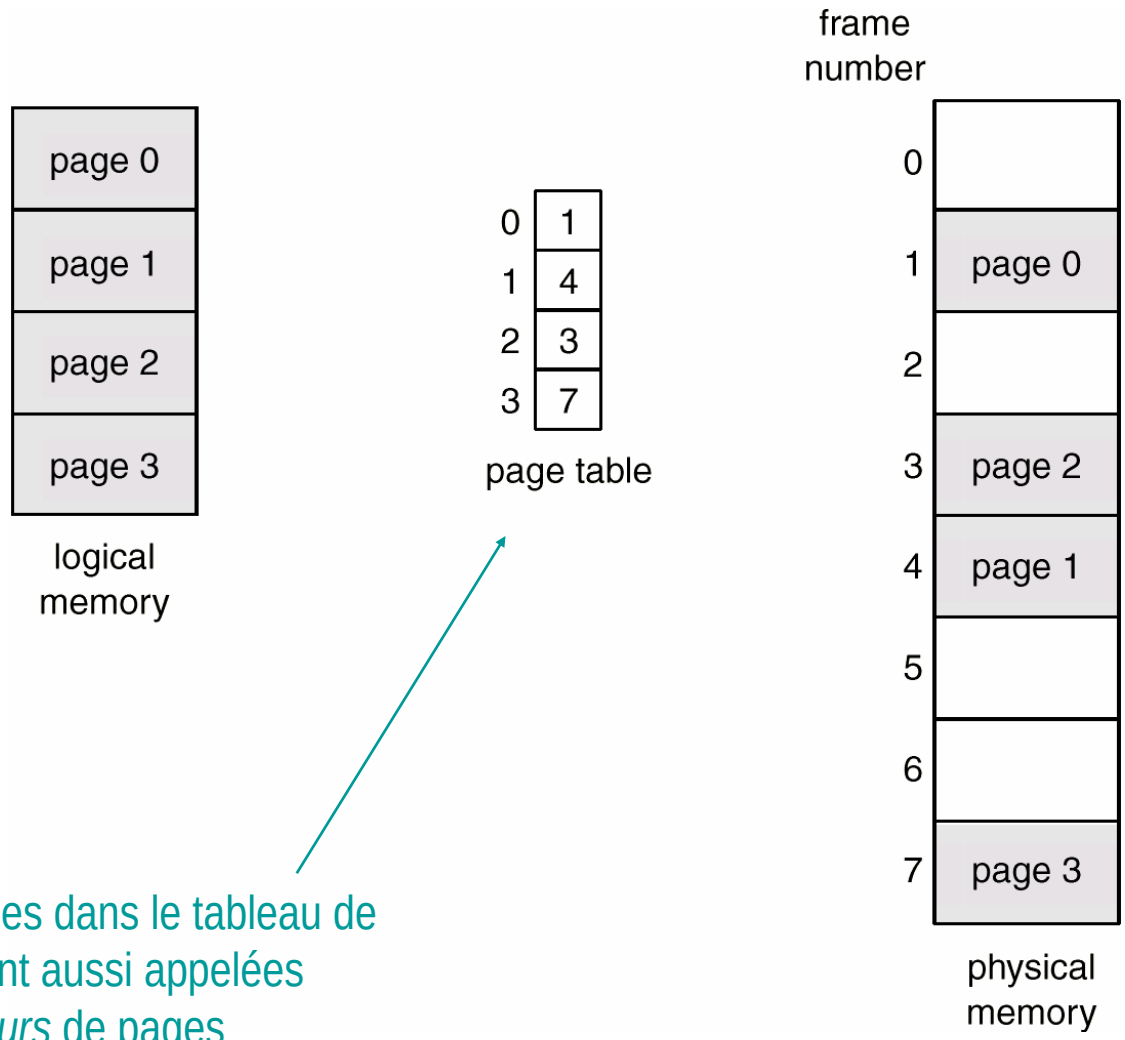
Exemple de chargement de processus (Stallings)

- **Nous pouvons maintenant transférer en mémoire un progr. D, qui demande 5 cadres**
  - bien qu`il n'y ait pas 5 cadres contigus disponibles
- **La fragmentation externe est limitée au cas que le nombre de pages disponibles n'est pas suffisant pour exécuter un programme en attente**
- **Seule la dernière page d'un progr peut souffrir de fragmentation interne (moy. 1/2 cadre par proc)**



(e) Swap out B (f) Load Process D

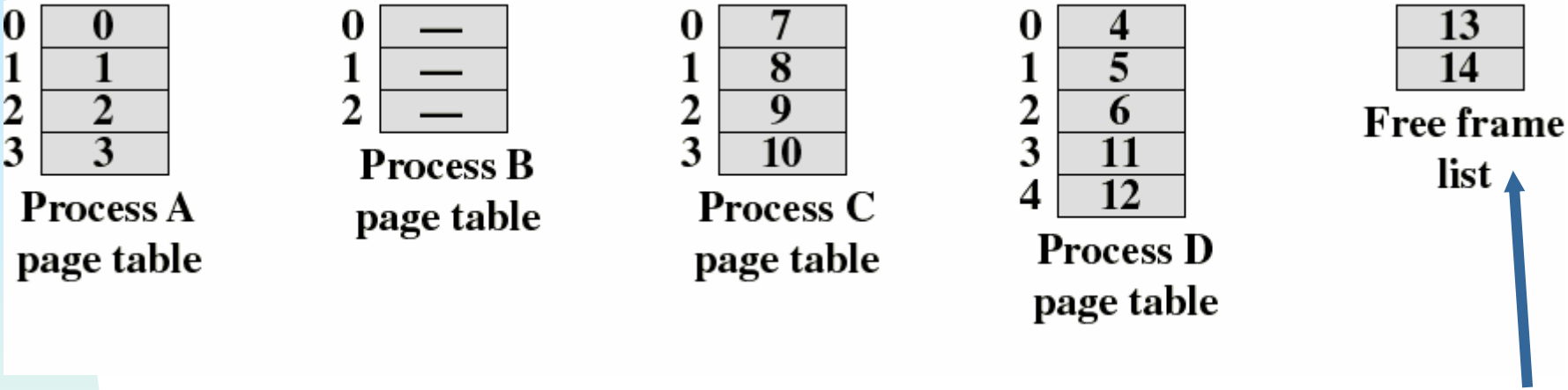
Tableaux de pages



Les entrées dans le tableau de pages sont aussi appelées *descripteurs* de pages

Tableaux de pages

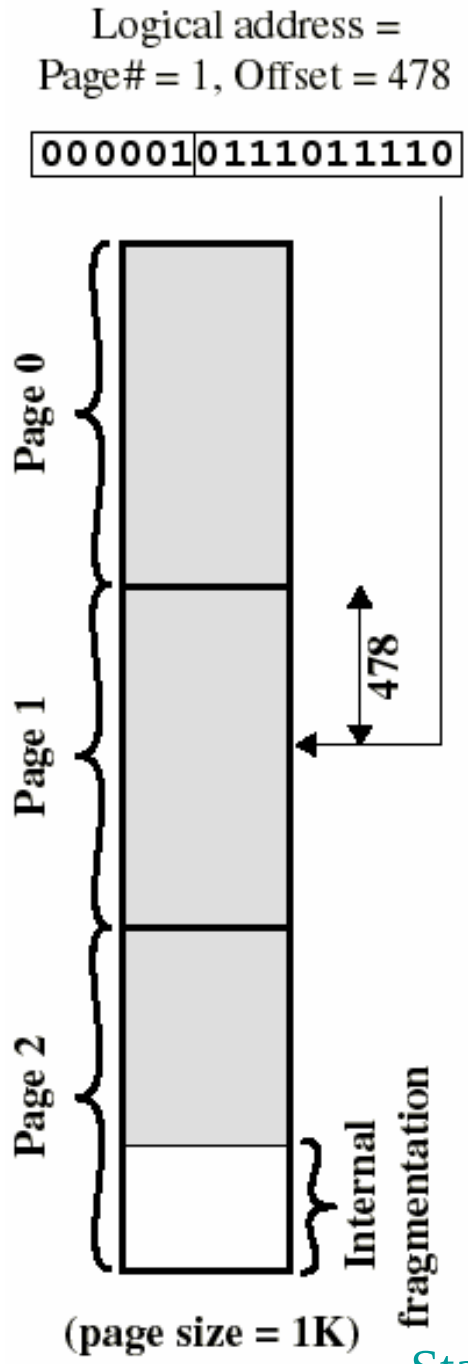
Stallings



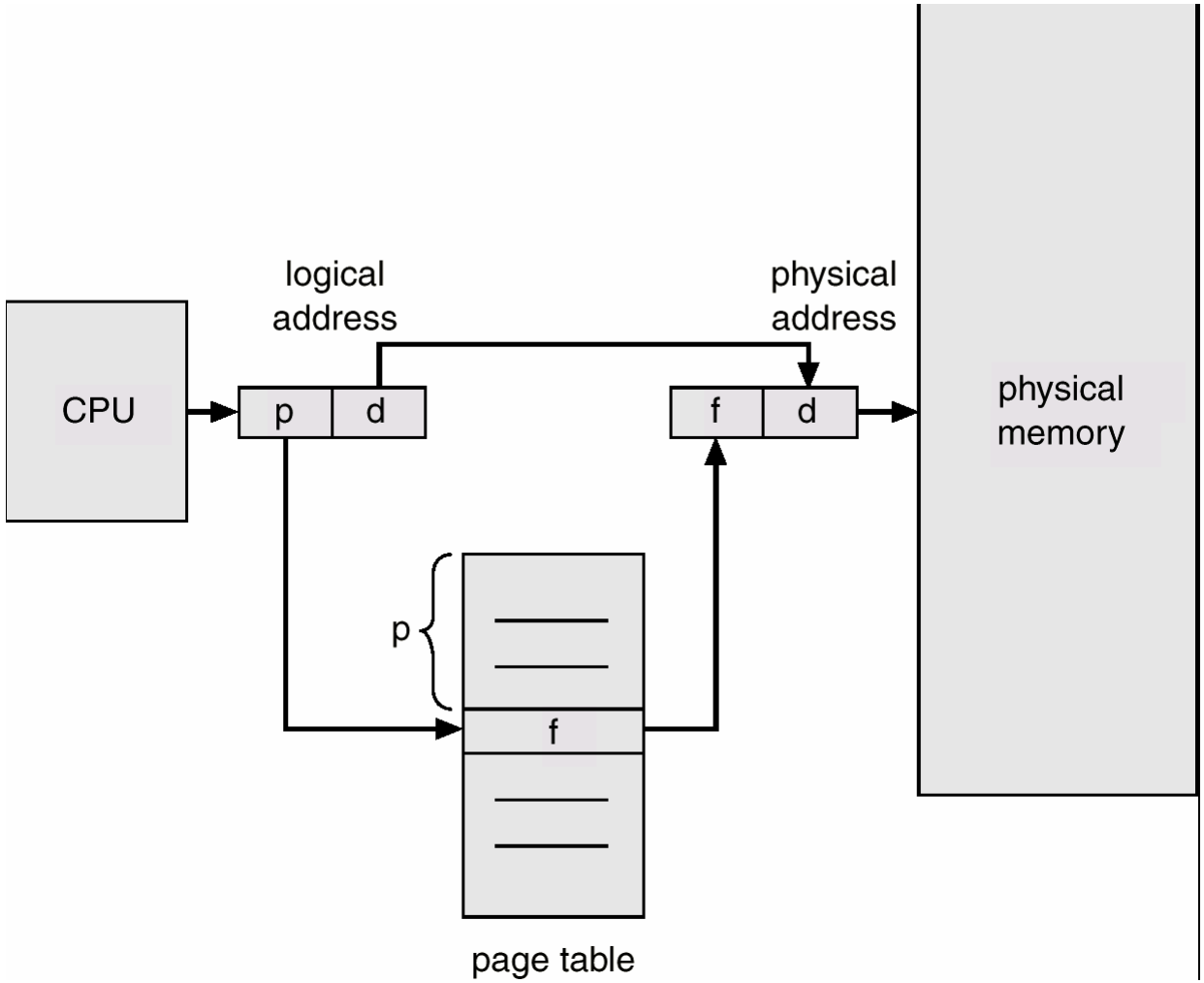
- **Le SE doit maintenir une table de pages pour chaque processus**
- **Chaque descripteur de pages contient le numéro de cadre où la page correspondante est physiquement localisée**
- **Une table de pages est indexée par le numéro de la page afin d'obtenir le numéro du cadre**
- **Une liste de cadres disponibles est également maintenue (free frame list)**

Traduction d'adresses

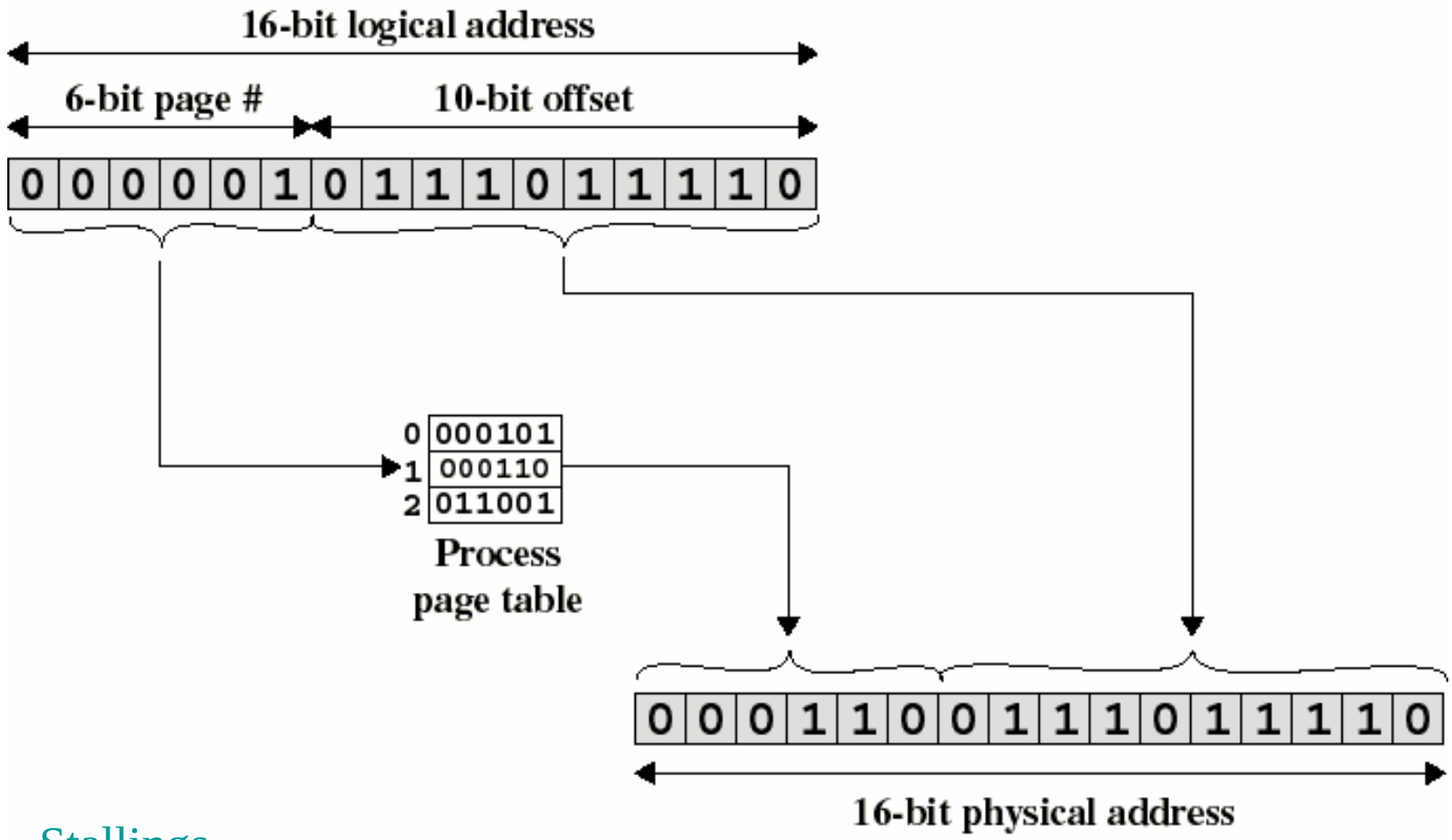
- L'adresse logique est facilement traduite en adresse physique
  - car la taille des pages est une puissance de 2
    - les pages débutent toujours à des adresses qui sont puissances de 2
    - qui ont autant de 0s à droite que la longueur de l'offset
    - donc ces 0s sont remplacés par l'offset
- Ex: si 16 bits sont utilisés pour les adresses et que la taille d'une page = 1K: on a besoins de 10 bits pour le décalage, laissant ainsi 6 bits pour le numéro de page
- L'adresse logique (n,m) est traduite à l 'adresse physique (k,m) en utilisant n comme index sur la table des pages et en le remplaçant par l 'adresse k trouvée
  - m ne change pas



Mécanisme: matériel



Traduction d'adresse (logique-physique) pour la pagination



Stallings

Trad. d'adresses: segmentation et pagination

- **Tant dans le cas de la segmentation, que dans le cas de la pagination, nous *ajoutons* le décalage à l'adresse du segment ou page.**
- **Cependant, dans la pagination, l'addition peut être faite par simple concaténation:**

11010000+1010
=
1101 1010

Deux petits problèmes

A) Considérez un système de 4 cadres ou pages physiques, chacune de 4 octets. Les adresses sont de 4 bits, deux pour le numéro de page, et 2 pour le décalage. Le tableau de pages du processus couramment en exécution est:

Numéro de page	Numéro de cadre
00	11
01	10
10	01
11	00

.

Considérez l'adresse logique 1010. Quelle sera l'adresse physique correspondante?

B) Considérez maintenant un système de segmentation, pas de pagination. Le tableau des segments du processus en exécution est comme suit:

Segment number	Base
00	110
01	100
10	000

Considérez l'adresse logique (no de seg, décalage)= (01, 01) , quelle est l'adresse physique?

Adresse logique (pagination)

- **Les pages sont invisibles au programmeur, compilateur ou assembleur (seule les adresses relatives sont employées)**
- **Un programme peut être exécuté sur différents matériels employant dimensions de pages différentes**
  - Ce qui change est la manière dont l'adresse est découpée

Problèmes d'efficacité

- **La traduction d`adresses, y compris la recherche des adresses des pages et de segments, est exécutée par des mécanismes de matériel**
- **Cependant, si la table des pages est en mémoire principale, chaque adresse logique occasionne au moins 2 références à la mémoire**
  - Une pour lire l'entrée de la table de pages
  - L'autre pour lire le mot référencé
- **Le temps d`accès mémoire est doublé...**

Pour améliorer l`efficacité

- **Où mettre les tables des pages (les mêmes idées s 'appliquent aussi aux tabl. de segm)**
- **Solution 1: dans des registres de UCT.**
  - avantage: vitesse
  - désavantage: nombre limité de pages par proc., la taille de la mém. logique est limitée
- **Solution 2: en mémoire principale**
  - avantage: taille de la mém. logique illimitée
  - désavantage: mentionné
- **Solution 3 (mixte): les tableaux de pages sont en mémoire principale, mais les adresses les plus utilisées sont aussi dans des registres d`UCT.**

Régistres associatifs TLB

TLB: Translation Lookaside Buffers, ou *caches* d'adressage

- **Recherche parallèle d 'une adresse:**
  - l 'adresse recherchée est cherchée dans la partie gauche de la table en parallèle (matériel spécial)
- **Traduction page → cadre**
  - Si la page recherchée a été utilisée récemment elle se trouvera dans les registres associatifs
    - recherche rapide

No Page	No Cadre
3	15
7	19
0	17
2	23

v. Content-addressable memory dans Wikipedia

Recherche *associative* dans TLB

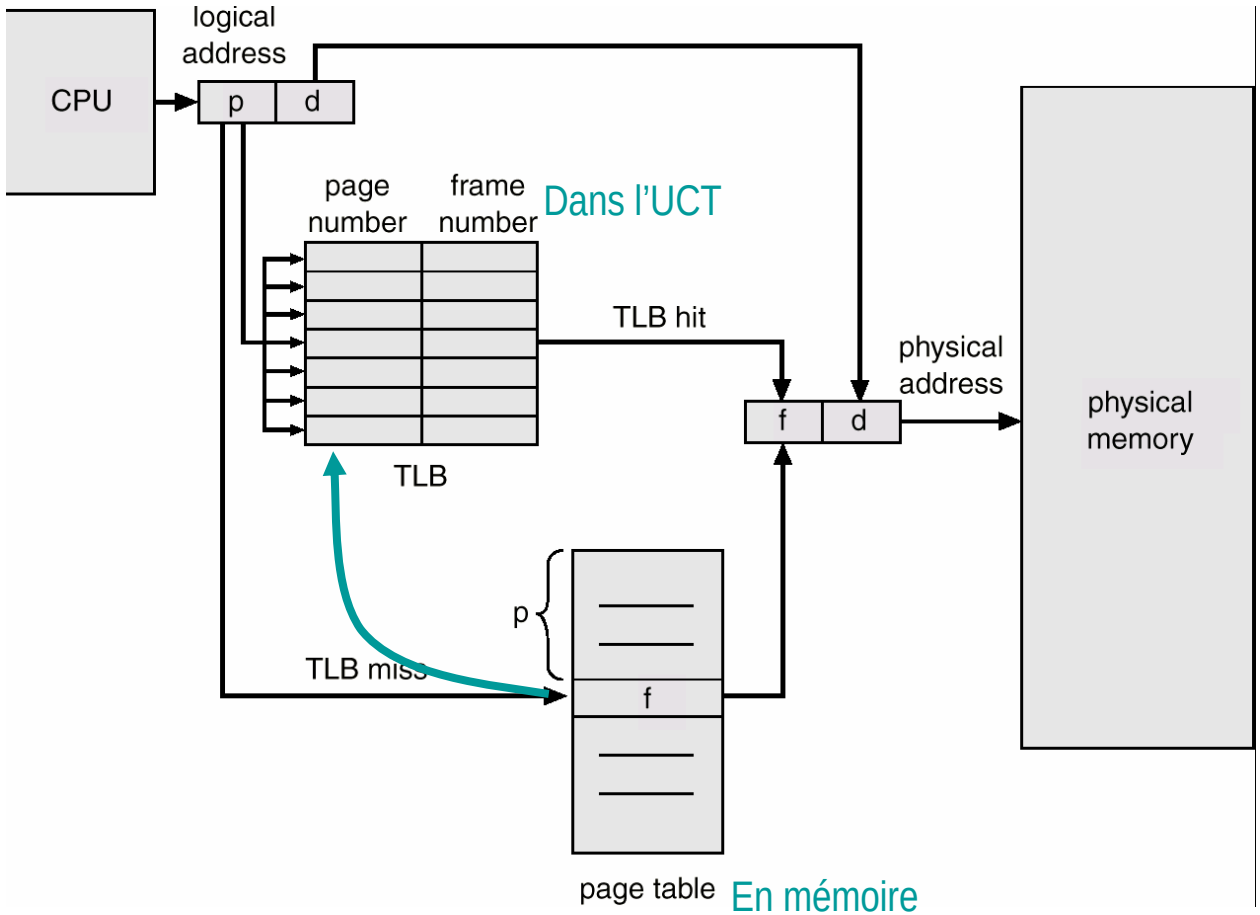
- **Le TLB est un petit tableau de registres de matériel où chaque ligne contient une paire:**
  - Numéro de page logique, Numéro de cadre
- **Le TLB utilise du matériel de *mémoire associative*: interrogation simultanée de tous les numéros logiques pour trouver le numéro physique recherché**
- **Chaque paire dans le TLB est fournie d 'un indice de référence pour savoir si cette paire a été utilisée récemment. Sinon, elle est remplacée par la dernière paire dont on a besoin**

Translation Lookaside Buffer (TLB)

- **Sur réception d'une adresse logique, le processeur examine le cache TLB**
- **Si cette entrée de page y est , le numéro de cadre en est extrait**
- **Sinon, le numéro de page indexe la table de page du processus (en mémoire)**
  - Cette nouvelle entrée de page est mise dans le TLB
  - Elle remplace une autre pas récemment utilisée
- **Le TLB est vidé quand l'UCT change de proc**

- **Les premières trois opérations sont faites par matériel**

Schéma d'utilisation TLB



Dans le cas de 'miss', f est trouvé en mémoire, puis il es mis dans le TLB

Temps d'accès réel avec TLB

- **Recherche dans TLB = ε unités de temps (normalement petit)**
- **Supposons que le cycle de mémoire soit 1 microseconde**
- **α = probabilité de touches (hit ratio) = probabilité qu'un numéro de page soit trouvé dans les registres associatifs (quantité entre 0 et 1)**
  - ceci est en relation avec le nombre de registres associatifs disponibles
- **Temps effectif d'accès tea:**

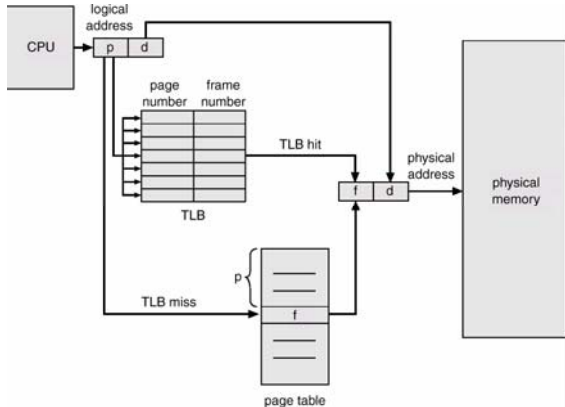
$$tea = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

si α est près de 1 et ε est petit, ce temps sera près de 1.

- **Dans plusieurs ordinateurs, il y a simultanéité entre ces opérations et d'autres opérations de l'UCT donc le temps d'accès réel est plus favorable**

- **Généralisation de la formule prenant m comme temps d'accès à la mémoire centrale:**

$$tea = (m+ \varepsilon)\,\alpha + (2m+ \varepsilon)(1- \alpha) = m\,\alpha + \varepsilon\alpha + 2m - 2m\,\alpha + \varepsilon - \varepsilon\alpha =$$
$$= 2m + \varepsilon - m\,\alpha$$



Cette formule peut être utilisée de différentes manières, p.ex. étant connu un tea désiré, déterminer α nécessaire

Tableaux de pages à deux niveaux

(quand les tableaux de pages sont très grands, ils peuvent être eux mêmes paginés)

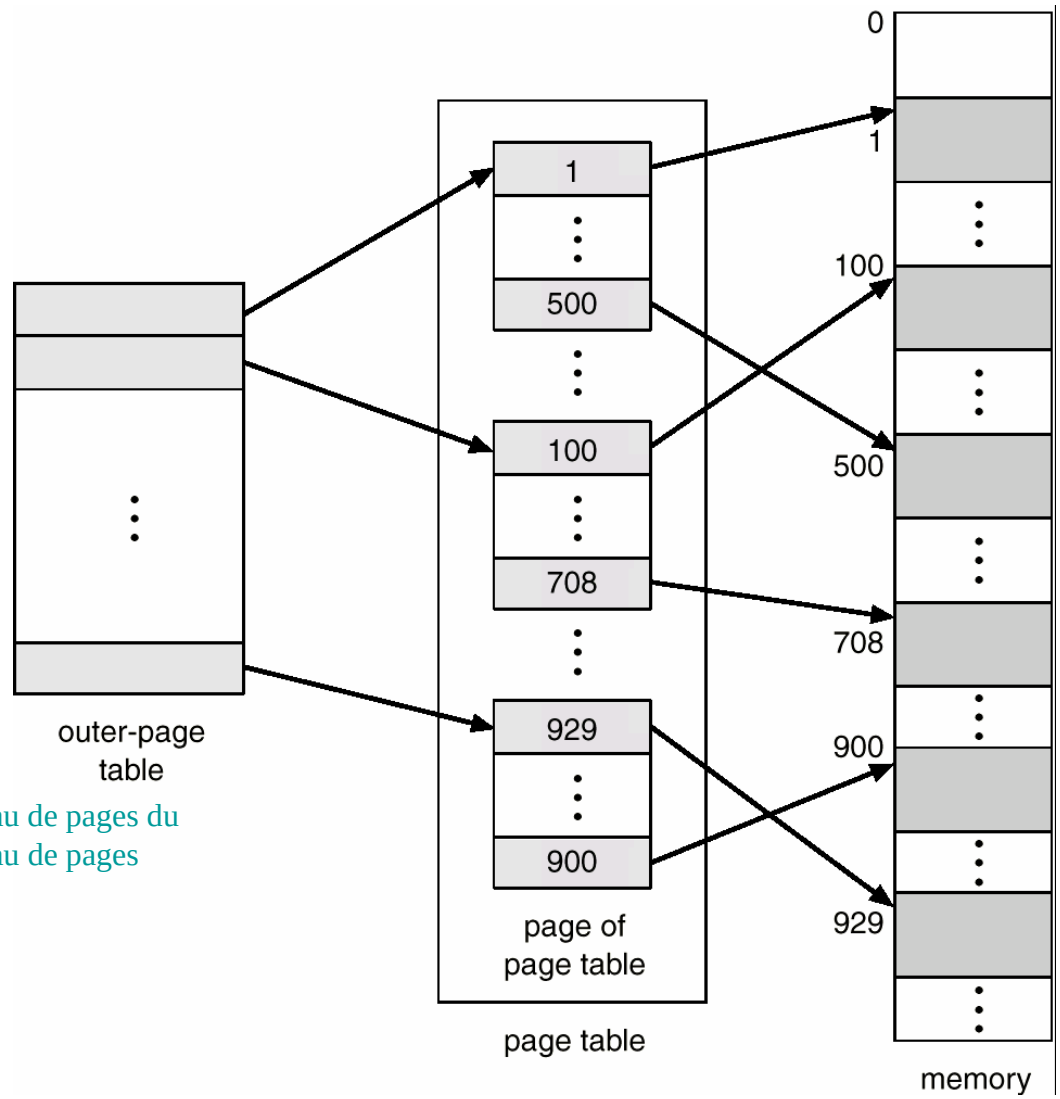
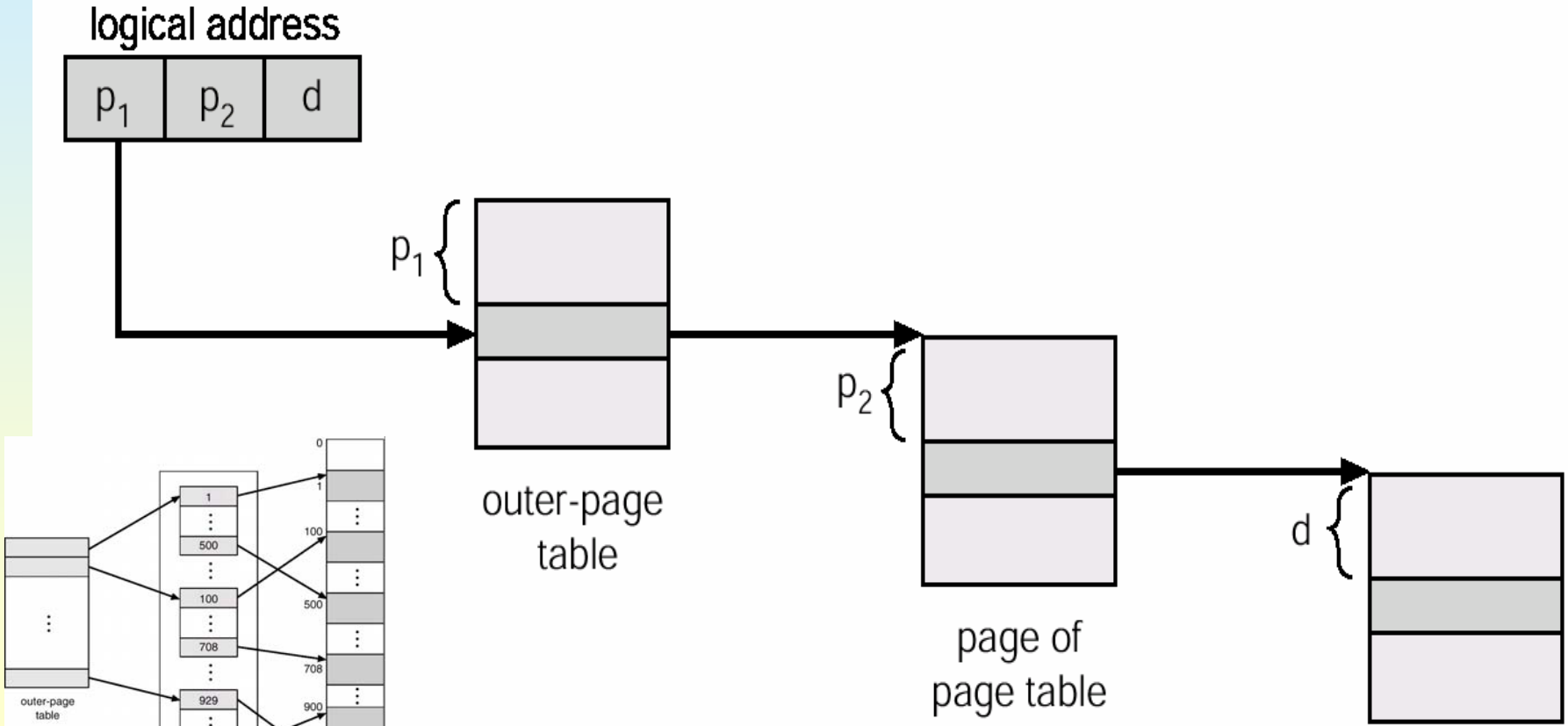


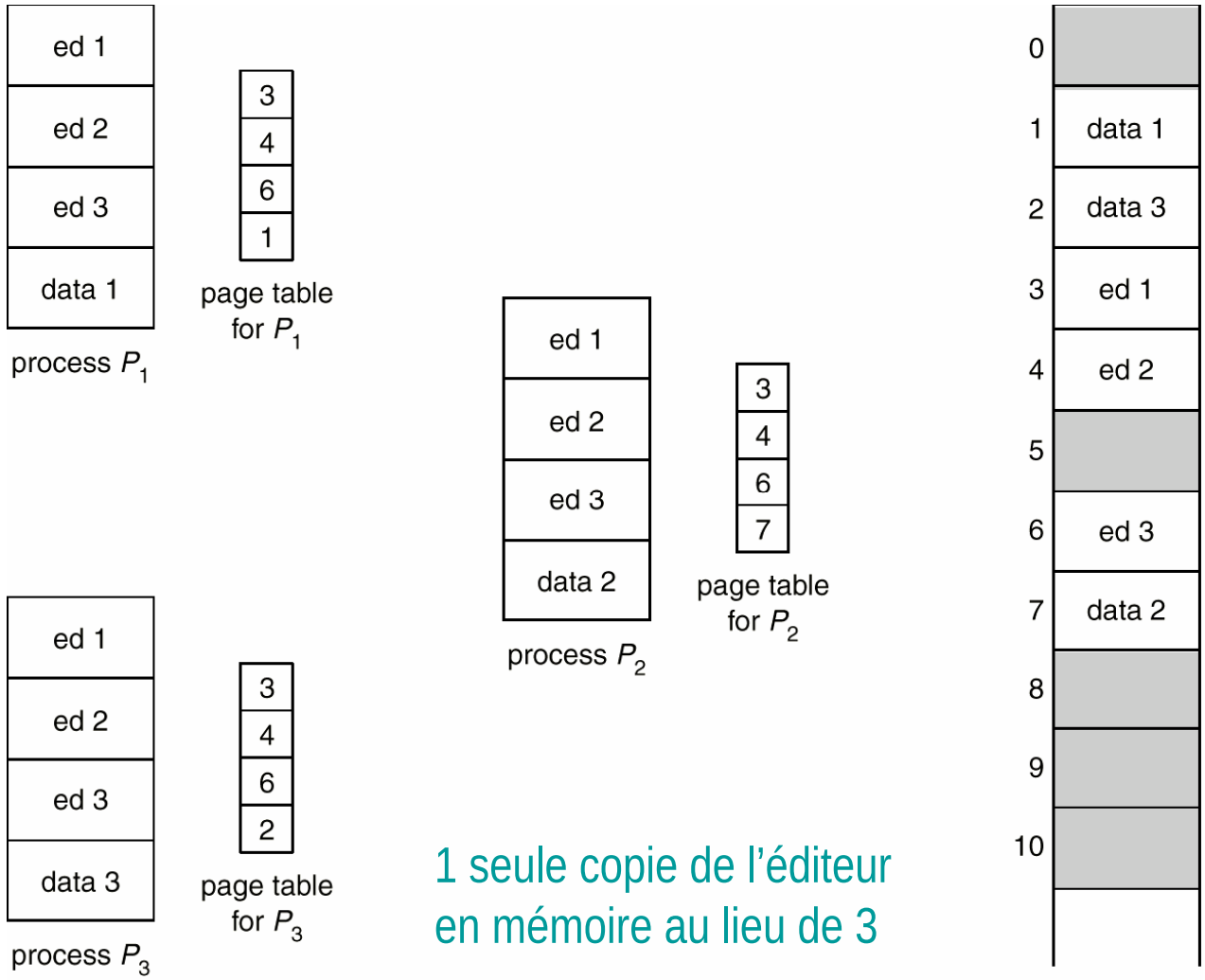
tableau de pages du tableau de pages

Tableaux de pages à deux niveaux

- **La partie de l ’adresse qui appartient au numéro de page est elle-même divisée en 2**



Partage de pages:
3 proc. partageant un éditeur, sur des données privées à chaque proc



1 seule copie de l'éditeur en mémoire au lieu de 3

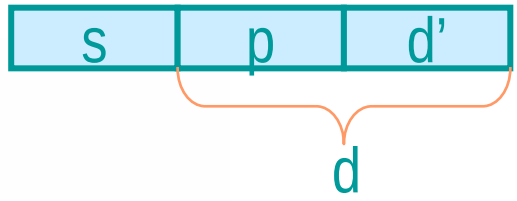
Segmentation simple vs Pagination simple

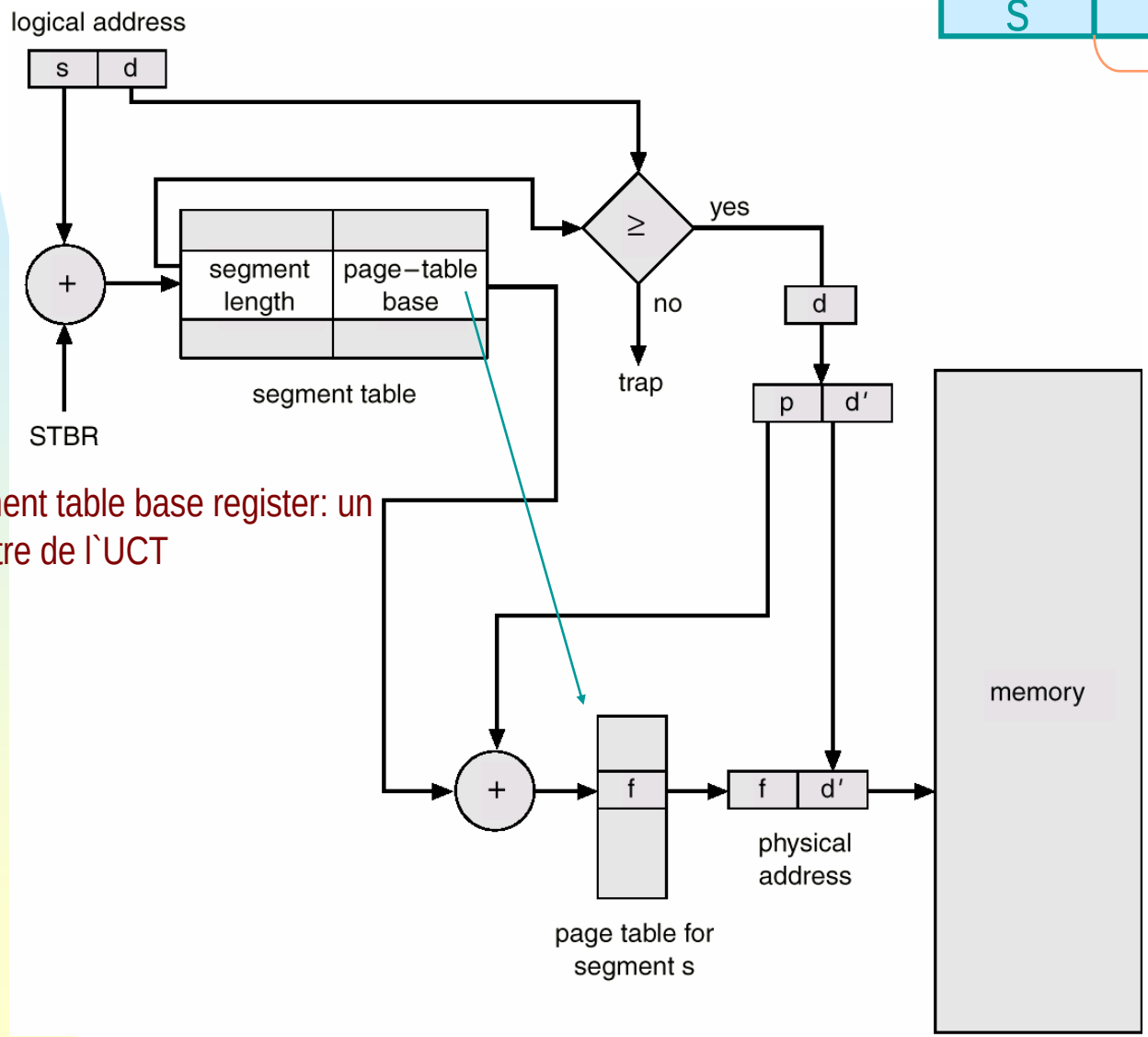
- **La pagination se préoccupe seulement du problème du chargement, tandis que**
- **La segmentation vise aussi le problème de la liaison**
- **La segmentation est visible au programmeur mais la pagination ne l'est pas**
- **Le segment est une unité logique de protection et partage, tandis que la page ne l'est pas**
  - Donc la protection et le partage sont plus aisés dans la segmentation
- **La segmentation requiert un matériel plus complexe pour la traduction d'adresses (addition au lieu d`enchaînement)**
- **La segmentation souffre de fragmentation *externe* (partitions dynamiques)**
- **La pagination produit de fragmentation *interne*, mais pas beaucoup (1/2 cadre par programme)**
- **Heureusement, la segmentation et la pagination peuvent être combinées**

Pagination et segmentation combinées

- **Les programmes sont divisés en segments et les segments sont paginés**
- **Donc chaque adresse de segment n`est pas une adresse de mémoire, mais une adresse au tableau de pages du segment**
- **Les tableaux de segments et de pages peuvent être eux-mêmes paginés**
  - Méthode inventée pour le système Multics de l'MIT, approx. 1965.

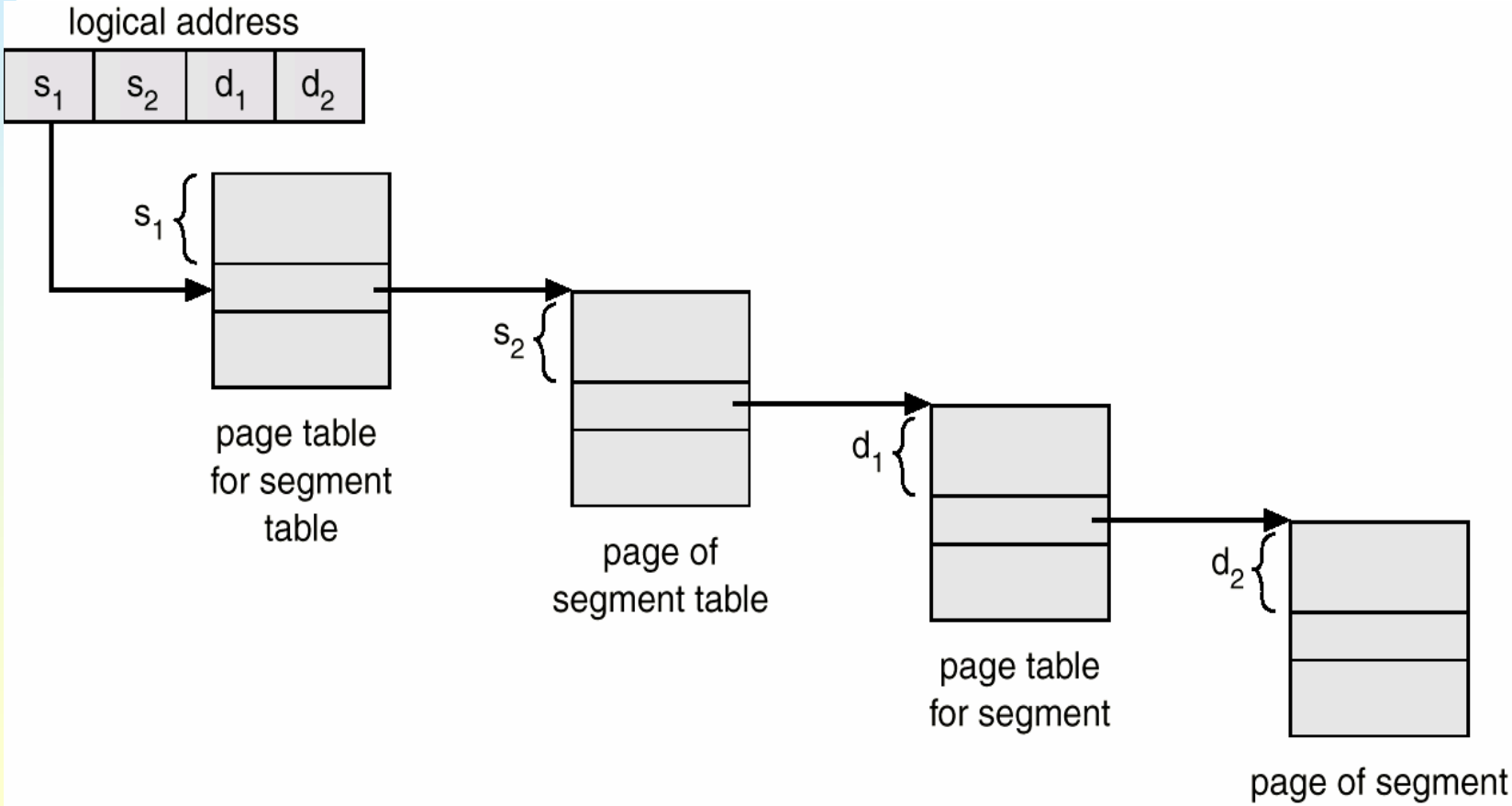
Adressage (sans considérer la pagination des tableaux de pages et de segments)





segment table base register: un registre de l`UCT

Segmentation et pagination combinées avec pagination des tableaux de pages et segments



Utilisation de Translation Lookaside Buffer

- **Dans le cas de systèmes de pagination à plusieurs niveaux, l'utilisation de TLB devient encore plus importante pour éviter multiples accès en mémoire pour calculer une adresse physique**
- **Les adresses les plus récemment utilisées sont trouvées directement dans la TLB.**

Conclusions sur Gestion Mémoire

- **Problèmes de:**
  - fragmentation (interne et externe)
  - complexité et efficacité des algorithmes
- **Méthodes**
  - Allocation contiguë
    - Partitions fixes
    - Partitions variables
    - Groupes de paires
  - Pagination
  - Segmentation
- **Problèmes en pagination et segmentation:**
  - taille des tableaux de segments et pages
  - pagination de ces tableaux
  - efficacité fournie par Translation Lookaside Buffer
- **Les différentes méthodes décrites dans ce chapitre, et dans le chapitre suivant, sont souvent utilisées conjointement, donnant lieu a des systèmes complexes**

Recapitulation sur la fragmentation

- **Partition fixes: fragmentation interne car les partitions ne peuvent pas être complèt. utilisées + fragm. externe s`il y a des partitions non utilisées**
- **Partitions dynamiques: fragmentation externe qui conduit au besoin de compression.**
- **Paires: fragmentation interne de 25% pour chaque processus + fragm. externe s 'il y a des partit. non utilisées**
- **Segmentation sans pagination: pas de fragmentation interne, mais fragmentation externe à cause de segments de longueur différentes, stockés de façon contiguë (comme dans les partitions dynamiques)**
- **Pagination:**
  - en moyenne, 1/2 cadre de fragm. interne par processus
  - dans le cas de mémoire virtuelle, aucune fragmentation externe (v. chap suivant)
- **Donc la pagination avec mémoire virtuelle offre la meilleure solution au pb de la fragmentation**

Par rapport au livre

- **Tout à l'exception de la section 9.4.4 (tables de pages inversées)**

DLL http://www.webopedia.com/TERM/D/DLL.html

- Short for *Dynamic Link Library,* a library of executable functions or data that can be used by a Windows application. Typically, a DLL provides one or more particular functions and a program accesses the functions by creating either a static or dynamic link to the DLL. A static link remains constant during program execution while a dynamic link is created by the program as needed. DLLs can also contain just data. DLL files usually end with the extension *.dll,.exe., drv,* or *.fon.*
- A DLL can be used by several applications at the same time. Some DLLs are provided with the Windows operating system and available for any Windows application. Other DLLs are written for a particular application and are loaded with the application. Short for *Dynamic Link Library,* a library of executable functions or data that can be used by a Windows application. Typically, a DLL provides one or more particular functions and a program accesses the functions by creating either a static or dynamic link to the DLL. A static link remains constant during program execution while a dynamic link is created by the program as needed. DLLs can also contain just data. DLL files usually end with the extension *.dll,.exe., drv,* or *.fon.*
- A DLL can be used by several applications at the same time. Some DLLs are provided with the Windows operating system and available for any Windows application. Other DLLs are written for a particular application and are loaded with the application.

Chapitre 10

Mémoire virtuelle

http://w3.uqo.ca/luigi/

Mémoire Virtuelle

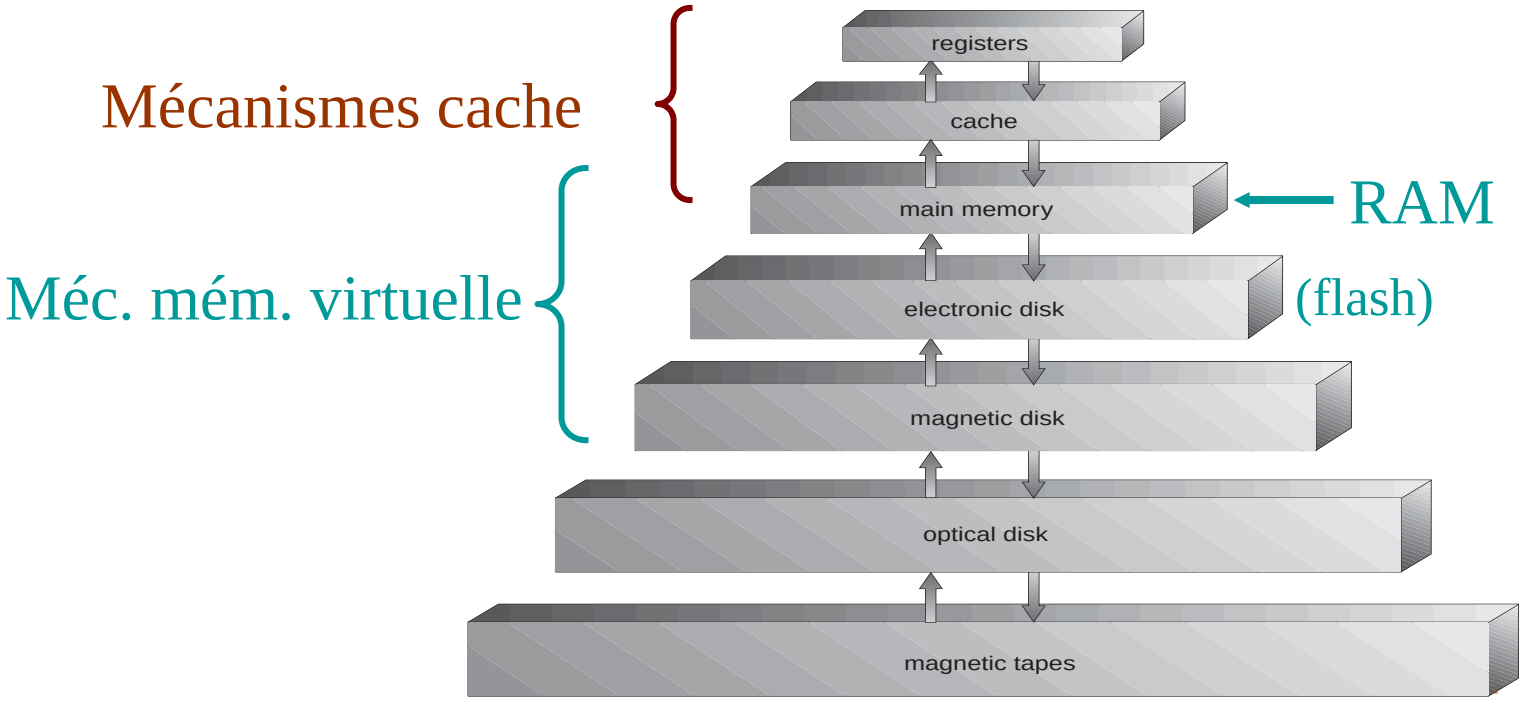
- **Pagination sur demande**
- **Problèmes de performance**
- **Remplacement de pages: algorithmes**
- **Allocation de cadres de mémoire**
- **Emballement**
- **Ensemble de travail**

Concepts importants du Chap. 10

- **Localité des références**
- **Mémoire virtuelle implémentée par va-et-vient des pages, mécanismes, défauts de pages**
- **Adresses physiques et adresses logiques**
- **Temps moyen d'accès à la mémoire**
  - Récriture ou non de pages sur mém secondaire
- **Algorithmes de remplacement pages:**
  - OPT, LRU, FIFO, Horloge
  - Fonctionnement, comparaison
- **Écroulement, causes**
- **Ensemble de travail (working set)**
- **Relation entre la mémoire allouée à un proc et le nombre d'interruptions**
- **Relation entre la dimension de pages et le nombre d'interruptions**
- **Prépagination, post-nettoyage**
- **Effets de l'organisation d'un programme sur l'efficacité de la pagination**

La mémoire virtuelle est une application du concept de hiérarchie de mémoire

- **C'est intéressant de savoir que des concepts très semblables s'appliquent aux mécanismes de la mémoire cache**
  - Cependant dans ce cas les mécanismes sont surtout de matériel



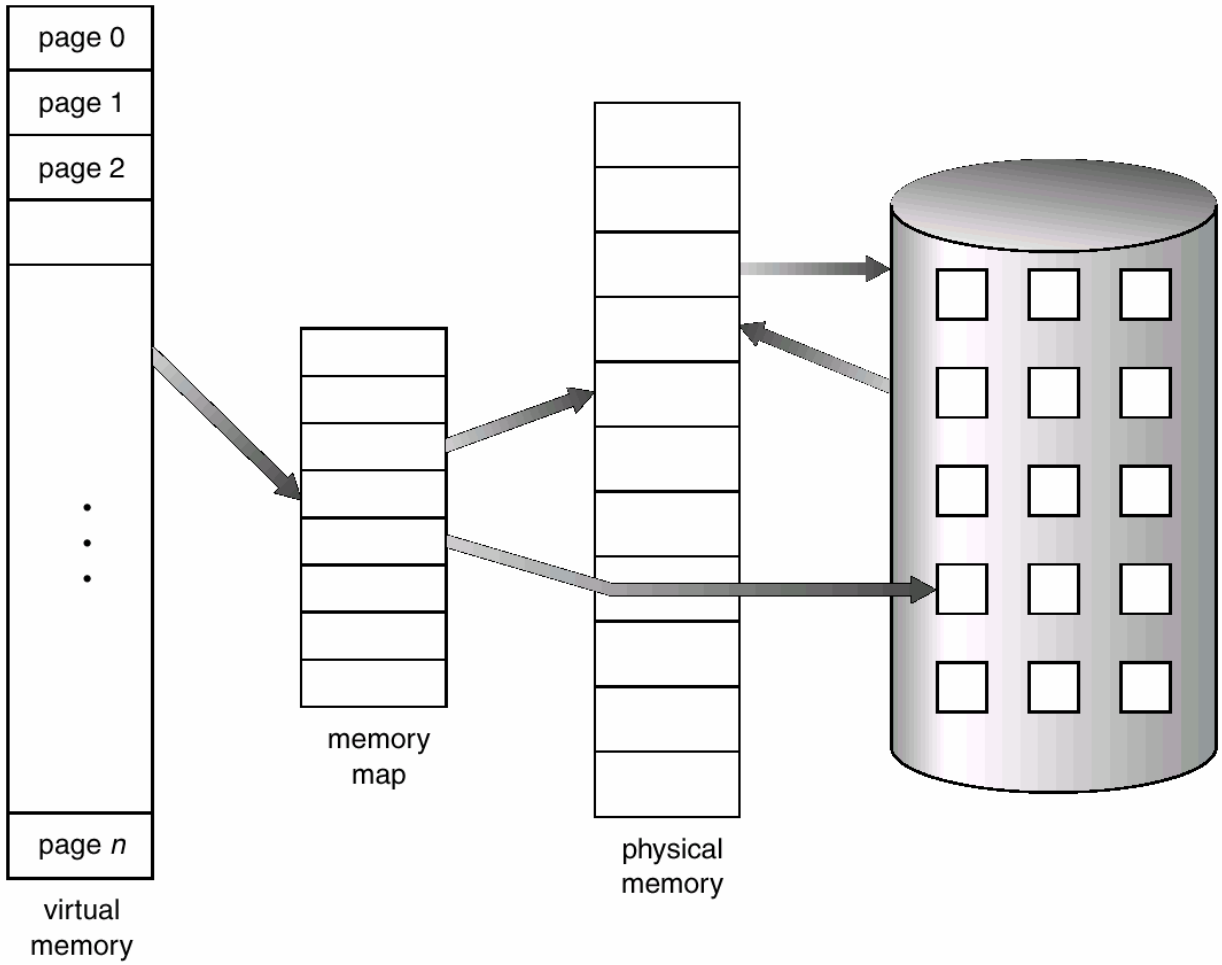
La mémoire virtuelle

- **À fin qu'un programme soit exécuté, il ne doit pas nécessairement être tout en mémoire centrale!**
- **Seulement les parties qui sont en exécution ont besoin d'être en mémoire centrale**
- **Les autres parties peuvent être sur mém secondaire (p.ex. disque), prêtes à être amenées en mémoire centrale sur demande**
  - Mécanisme de va-et-vient ou swapping
- **Ceci rend possible l'exécution de programmes beaucoup plus grands que la mémoire physique**
  - Réalisant une mémoire virtuelle qui est plus grande que la mémoire physique

De la pagination et segmentation à la mémoire virtuelle

- **Un processus est constitué de morceaux (pages ou segments) ne nécessitant pas d'occuper une région contiguë de la mémoire principale**
- **Références à la mémoire sont traduites en adresses physiques au moment d'exécution**
  - Un processus peut être déplacé à différentes régions de la mémoire, aussi mémoire secondaire!
- **Donc: tous les morceaux d'un processus ne nécessitent pas d'être en mémoire principale durant l'exécution**
  - L'exécution peut continuer à condition que la prochaine instruction (ou donnée) est dans un morceau se trouvant en mémoire principale
- **La somme des mémoires logiques des procs en exécution peut donc excéder la mémoire physique disponible**
  - Le concept de base de la mémoire virtuelle
- **Une image de tout l'espace d'adressage du processus est gardée en mémoire secondaire (normal. disque) d'où les pages manquantes pourront être prises au besoin**
  - Mécanisme de va-et-vien ou swapping

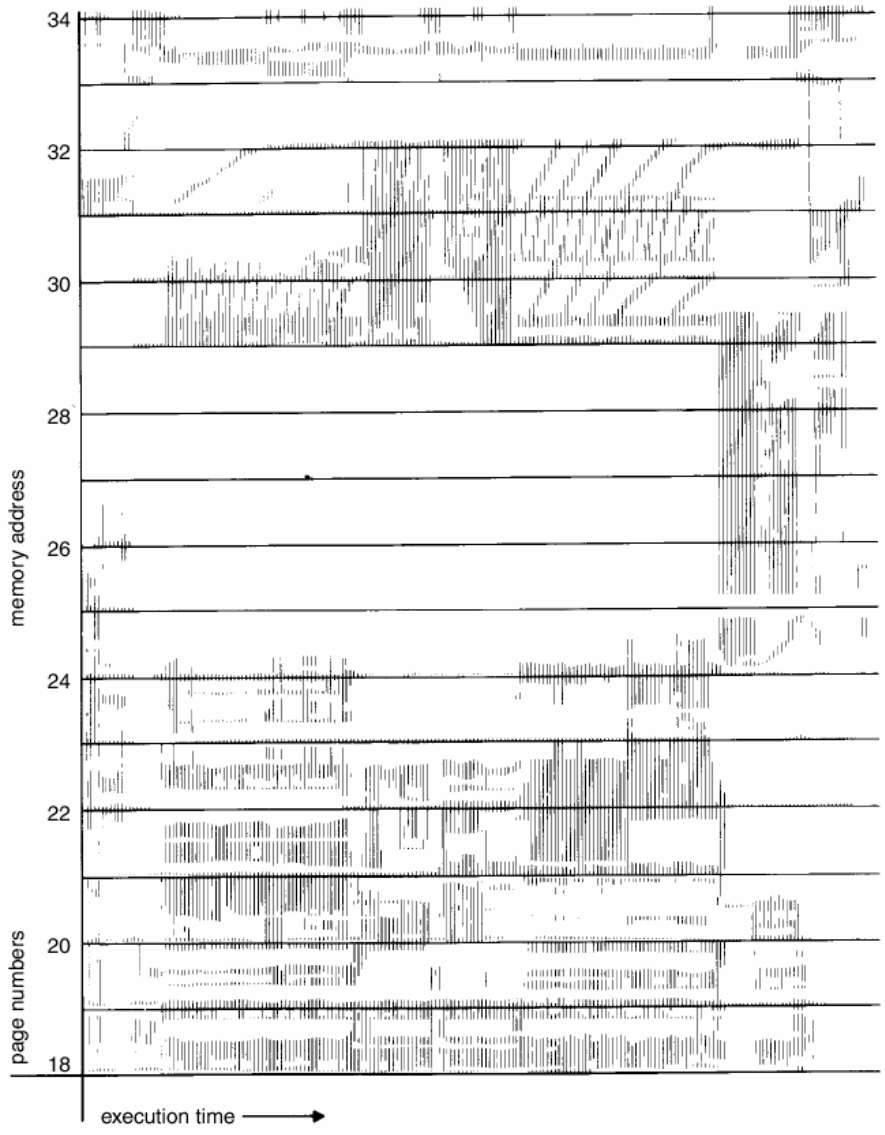
Mémoire virtuelle:
résultat d'un mécanisme qui combine la mémoire principale et les mémoires secondaires



Localité et mémoire virtuelle

- **Principe de localité des références: les références à la mémoire dans un processus tendent à se regrouper**
- **Donc: seule quelques pièces d'un processus seront utilisées durant une petite période de temps (pièces: pages ou segments)**
- **Il y a une bonne chance de "deviner" quelles seront les pièces demandées dans un avenir rapproché**

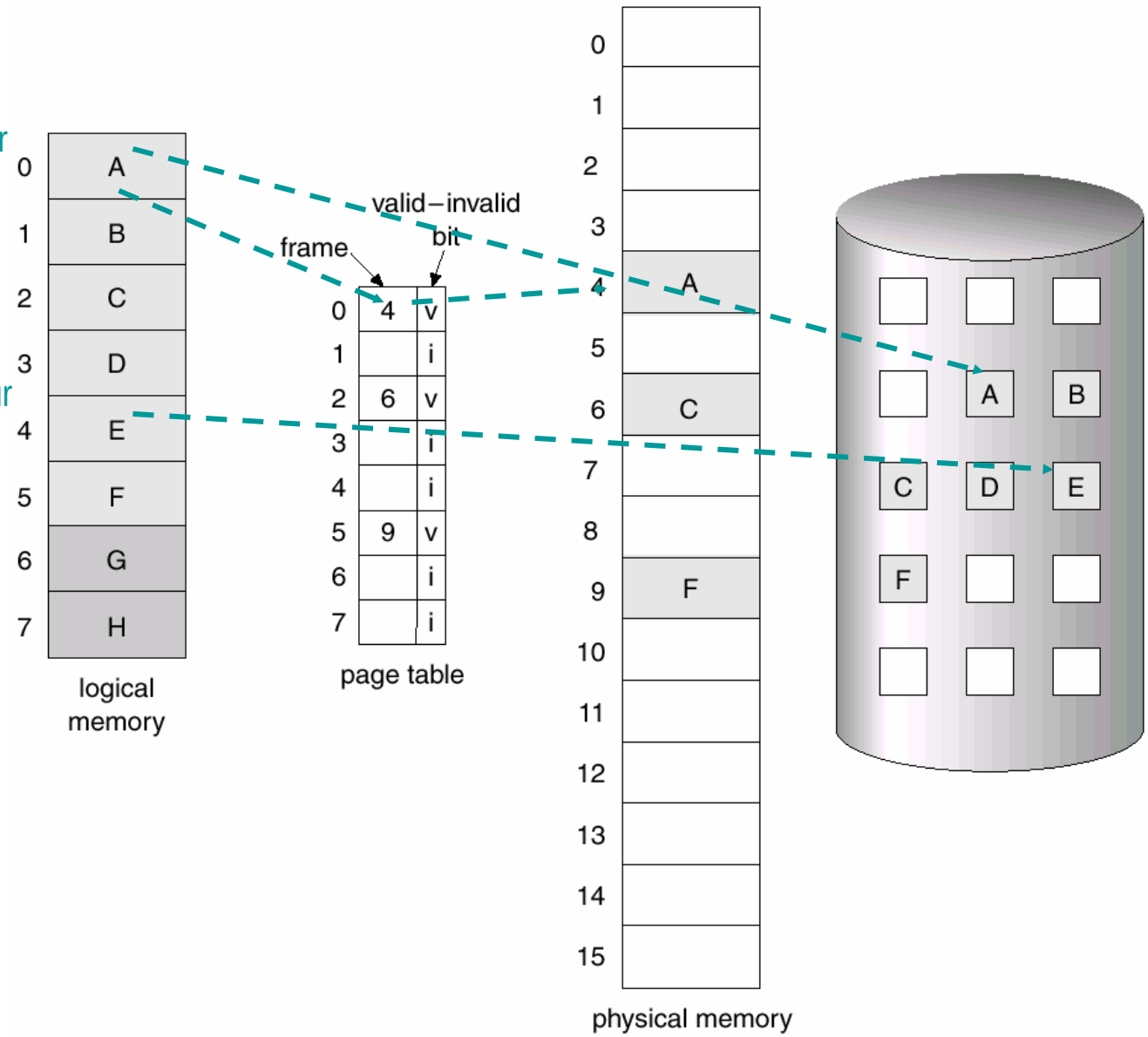
Visualisation de la localité des références



Pages en RAM ou sur disque

Page A en RAM et sur disque

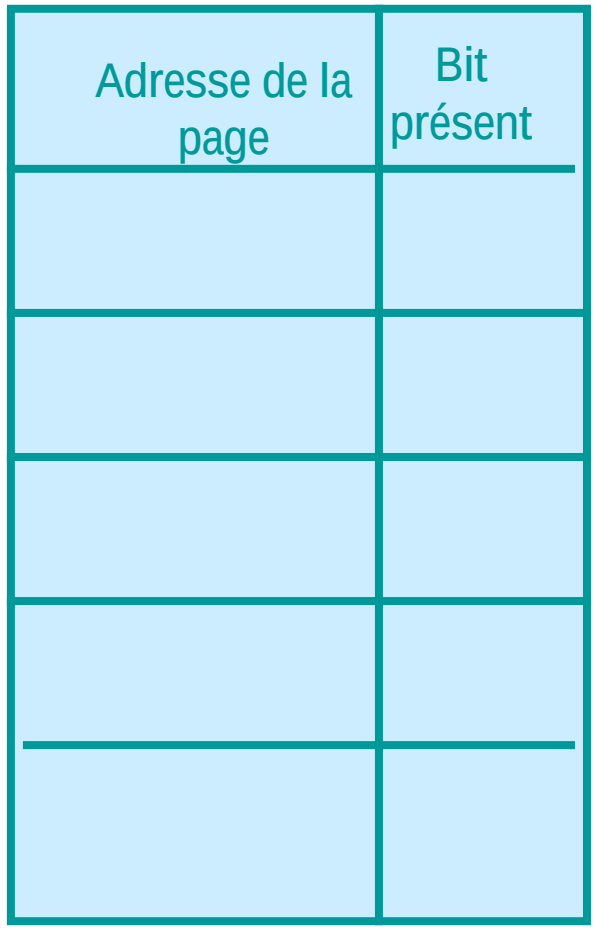
Page E seulement sur disque



Nouveau format du tableau des pages (la même idée peut être appliquée aux tableaux de segments)

Si la page est en mém. princ., ceci est une adr. de mém. Principale

sinon elle est une adresse de mémoire secondaire



bit *présent*
1 si en mém. princ.,
0 si en mém second.

Au début, bit présent = 0 pour toutes les pages

Avantages du chargement partiel

- **Plus de processus peuvent être maintenus en exécution en mémoire**
  - Car seules quelques pièces sont chargées pour chaque processus
  - L'usager est content, car il peut exécuter plusieurs processus et faire référence à des gros données sans avoir peur de remplir la mémoire centrale
  - Avec plus de processus en mémoire principale, il est plus probable d'avoir un processus dans l'état prêt, meilleure utilisation d'UCT
- **Plusieurs pages ou segments rarement utilisés n'auront peut être pas besoin d`être chargés du tout**
- **Il est maintenant possible d'exécuter un ensemble de processus lorsque leur taille excède celle de la mémoire principale**
  - Il est possible d'utiliser plus de bits pour l'adresse logique que le nombre de bits requis pour adresser la mémoire principale
  - Espace d 'adressage logique **> >** esp. d 'adressage physique

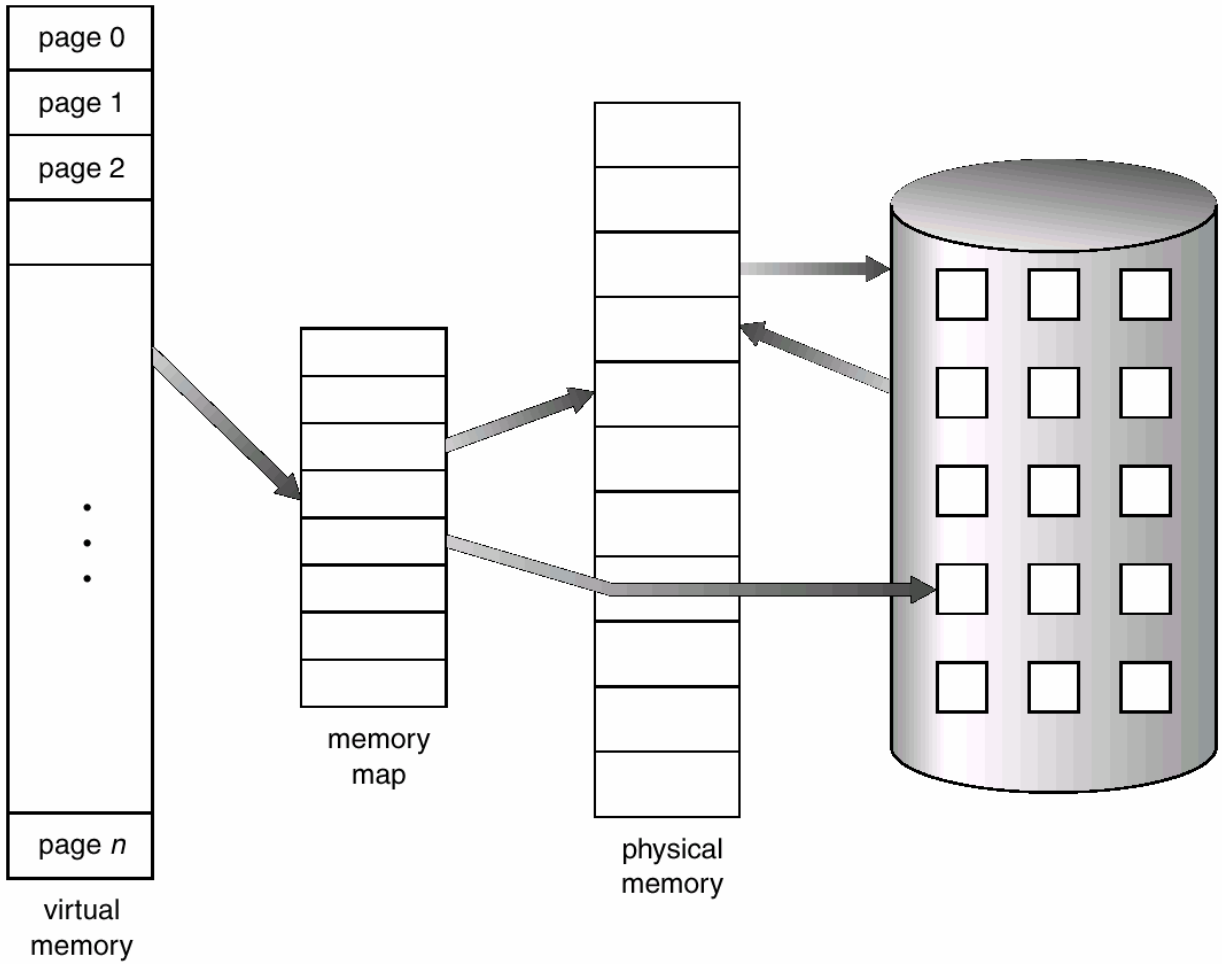
Mémoire Virtuelle: Pourrait Être Énorme!

- **Ex: 16 bits sont nécessaires pour adresser une mémoire physique de 64KB**
- **En utilisant des pages de 1KB, 10 bits sont requis pour le décalage**
- **Pour le *numéro de page* de l'adresse logique nous pouvons utiliser un nombre de bits qui excède 6, car toutes les pages ne doivent pas être en mémoire simultanément**
- **Donc la limite de la mémoire virtuelle est le nombre de bits qui peuvent être réservés pour l 'adresse**
  - Dans quelques architectures, ces bits peuvent être inclus dans des registres
- **La mémoire logique est donc appelée *mémoire virtuelle***
  - Est maintenue en mémoire secondaire
  - Les pièces sont amenées en mémoire principale seulement quand nécessaire, sur demande

Mémoire Virtuelle

- **Pour une meilleure performance, la mémoire virtuelle se trouve souvent dans une région du disque qui est n'est pas gérée par le système de fichiers**
  - Mémoire va-et-vient, swap memory
- **La mémoire physique est celle qui est référencée par une adresse physique**
  - Se trouve dans le RAM et cache
- **La traduction de l'adresse logique en adresse physique est effectuée en utilisant les mécanismes étudiés dans le chapitre précédent.**

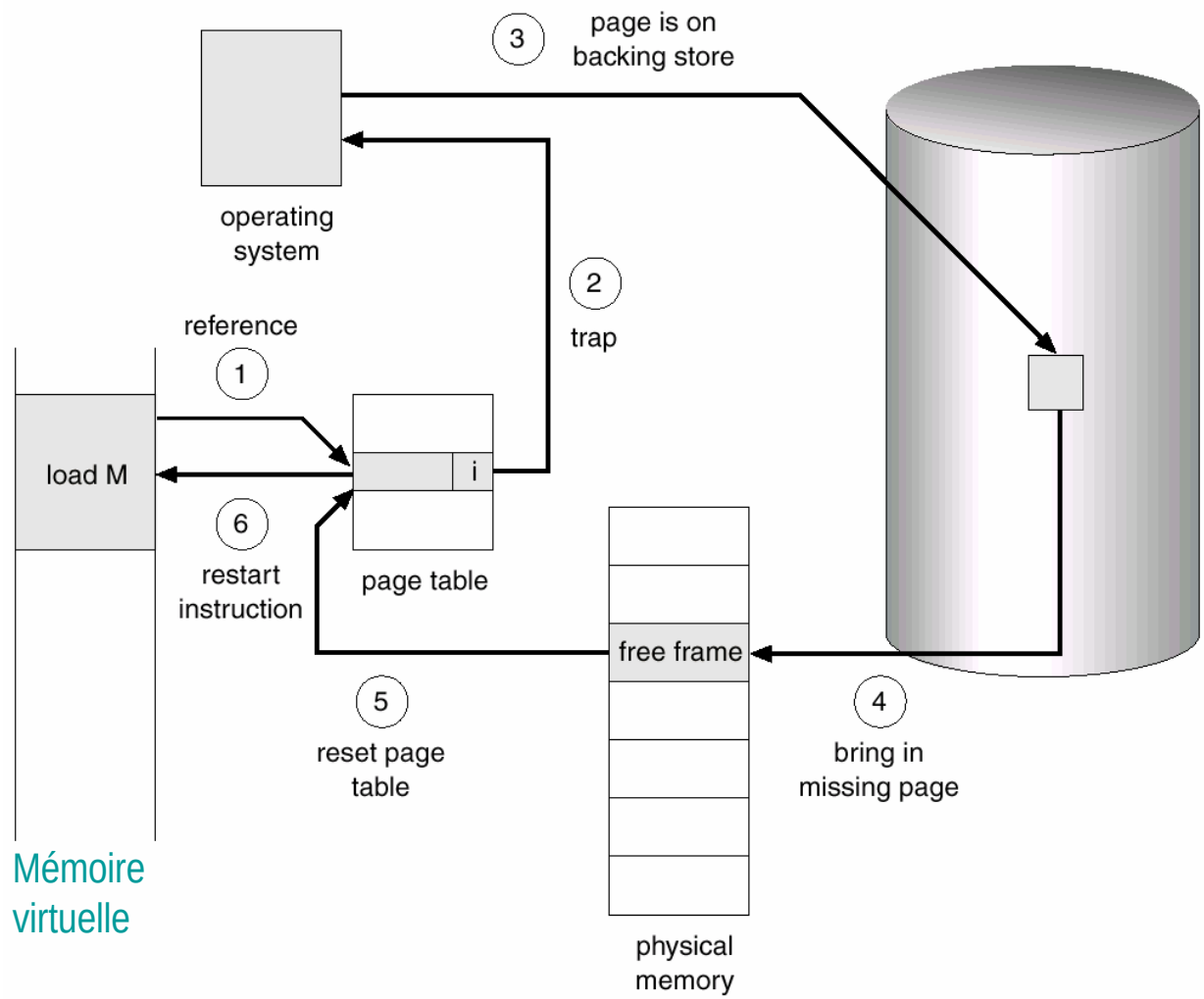
Mémoire virtuelle: le mécanisme de va-et-vient



Exécution d'un Processus

- **Le SE charge la mémoire principale de quelques pièces (seulement) du programme (incluant le point de départ)**
- **Chaque entrée de la table de pages (ou segments) possède un bit présent qui indique si la page ou segment se trouve en mémoire principale**
- **L'ensemble résident (résident set) est la portion du processus se trouvant en mémoire principale**
- **Une interruption est générée lorsque l'adresse logique réfère à une pièce qui n'est pas dans l'ensemble résident**
  - défaut de pagination, page fault

Exécution d'une défaut de page: va-et-vient plus en détail



Séquence d'événements pour défaut de page

- **Trappe au SE: page demandée pas en RAM**
- **Sauvegarder registres et état du proc dans PCB**
- **Un autre proc peut maintenant gagner l 'UCT**
- **SE détermine si la page demandée est légale**
  - sinon: terminaison du processus
- **et trouve la position de la page sur disque**
  - dans le descripteur de la page
- **lire la page de disque dans un cadre de mémoire libre (supposons qu`il y en a!)**
  - exécuter les ops disque nécessaires pour lire la page

Séquence d'événements pour défaut de page (ctn.)

- **L 'unité disque a complété le transfert et interrompt l'UCT**
  - sauvegarder les registres etc. du proc exécutant
- **SE met à jour le contenu du tableau des pages du proc. qui a causé le défaut de page**
- **Ce processus devient prêt=ready**
- **À un certain point, il retournera à exécuter**
  - la page désirée étant en mémoire, il pourra maintenant continuer

Temps moyen d'accès à la mémoire

Supposons que:

• accès en mémoire: 100 nanosecs

• temps de traitement de défaut de page: 25 millisecs = 25,000,000 nanosecs (inclut le temps de lecture-écriture disque)

• p: probabilité de trouver une page en mémoire (défaut) (quantité entre 0 et 1)

Temps moyen d'accès mémoire:

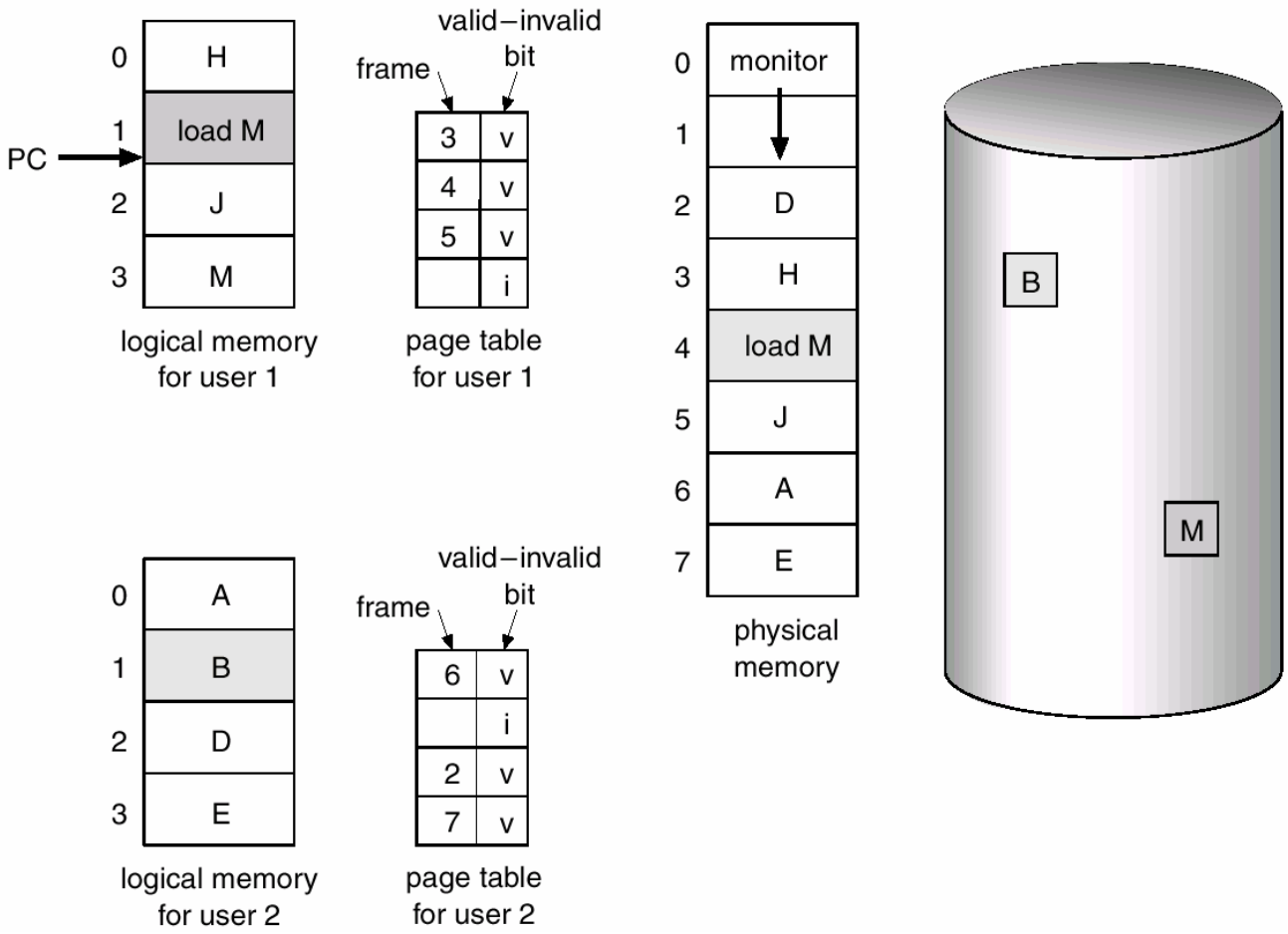
p x 100 + (1-p) x 25,000,000 (pas de défaut + défaut)

En utilisant la même formule, nous pouvons déterminer quel est le nombre de défauts que nous pouvons tolérer, si un certain niveau de performance est désiré (v. manuel).

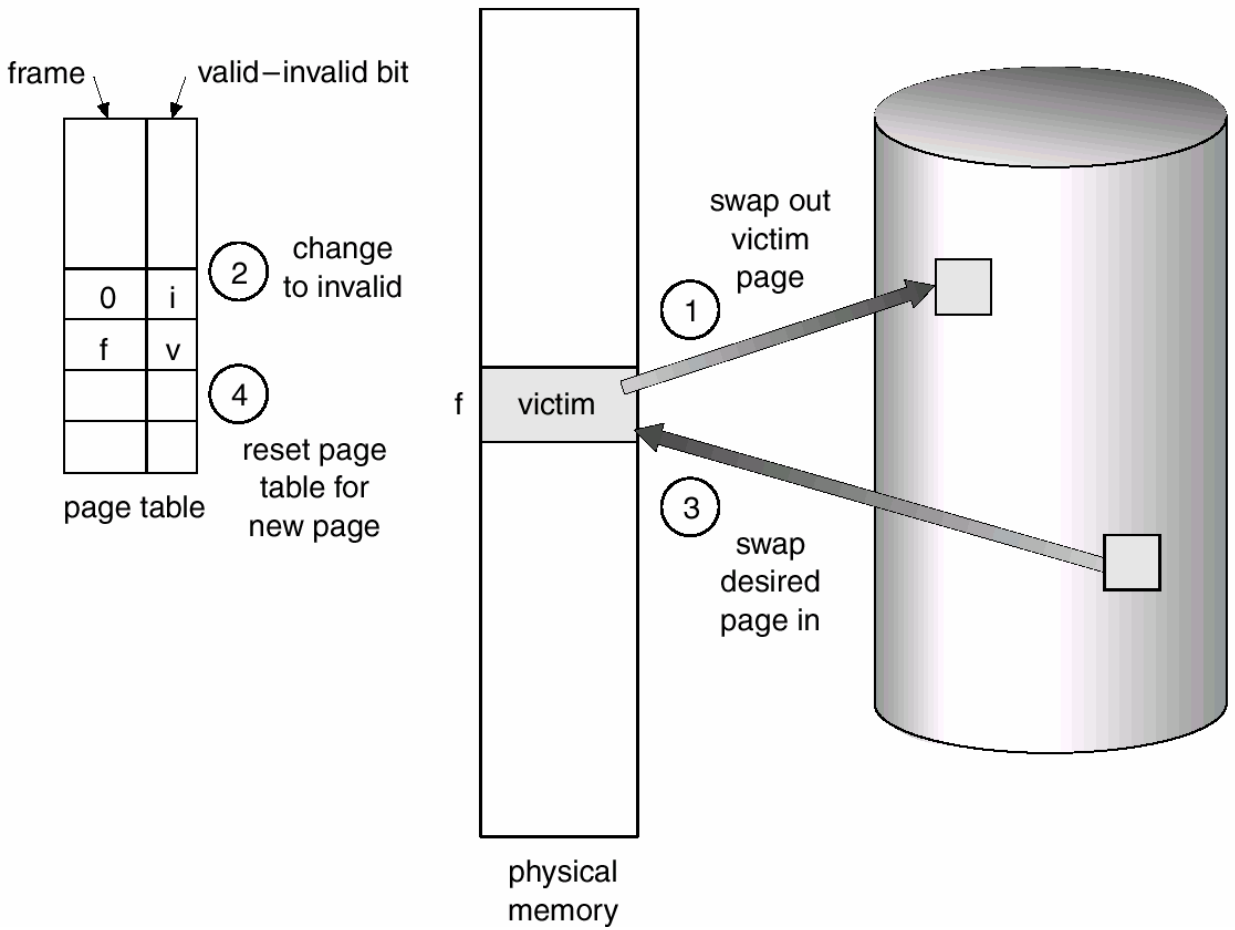
P.ex. avec ces params, si le ralentissement à cause de pagination ne peut pas excéder 10%, 1 seul défaut de pagination peut être toléré pour chaque 2,500,000 accès de mémoire.

(Temps d'accès disque réaliste aujourd'hui= autour de 10ms)

Quand la RAM est pleine mais nous avons besoin d`une page pas en RAM



La page victime...



Remplacement de pages

- **Quoi faire si un processus demande une nouvelle page et il n'y a pas de cadres libres en RAM?**
- **Il faudra choisir une page déjà en mémoire principale, appartenant au même ou à un autre processus, qu'il est possible d 'enlever de la mémoire principale**
  - la victime!
- **Un cadre de mémoire sera donc rendu disponible**
- **Évidemment, plusieurs cadres de mémoire ne peuvent pas être `victimisés`:**
  - p.ex. cadres contenant le noyau du SE, tampons d 'E/S...

Bit de modification , *dirty bit*

- **La ‘victime’ doit-elle être récrite en mémoire secondaire?**
- **Seulement si elle a été changée depuis qu`elle a été amenée en mémoire principale**
  - sinon, sa copie sur disque est encore fidèle
- **Bit de modif sur chaque descripteur de page indique si la page a été changée**
- **Donc pour calculer le coût en temps d’une référence à la mémoire il faut aussi considérer la probabilité qu’une page soit ‘sale’ et le temps de récriture dans ce cas**

Algorithmes de remplacement pages

- **Choisir la victime de façon à minimiser le taux de défaut de pages**
  - pas évident!!!
- **Page dont nous n`aurons pas besoin dans le futur? impossible à savoir!**
- **Page pas souvent utilisée?**
- **Page qui a été déjà longtemps en mémoire??**

- **etc. nous verrons...**

Critères d'évaluation des algorithmes

- **Les algorithmes de choix de pages à remplacer doivent être conçus de façon à minimiser le taux de défaut de pages à long terme**
- **Mais ils ne peuvent pas impliquer des inefficacités**
- **Ni l'utilisation de matériel dispendieux**

Critères d'efficacité

- **Il est intolérable d'exécuter un algorithme complexe chaque fois qu'une opération de mémoire est exécutée**
  - Ceci impliquerait des accès additionnels de mémoire
- **Cependant ceci peut se faire chaque fois qu'il y a une faute de pagination**
- **Les opérations qui doivent être faites à chaque accès de mémoire doivent être câblées dans le matériel**

Explication et évaluation des algorithmes

- **Nous allons expliquer et évaluer les algorithmes en utilisant la chaîne de référence pages suivante (prise du livre de Stallings):**

  2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

- **Attention: les séquences d'utilisation pages ne sont pas aléatoires...**
  - Localité de référence
- **Ces références proviendront de plusieurs processus**
- **L'évaluation sera faite sur la base de cet exemple, évidemment pas suffisant pour en tirer des conclusions générales**

Algorithmes pour la politique de remplacement

- **L'algorithme optimal (OPT) choisit pour page à remplacer celle qui sera référencée le plus tardivement**
  - produit le + petit nombre de défauts de page
  - impossible à réaliser (car il faut connaître le futur) mais sert de norme de comparaison pour les autres algorithmes:
    - Ordre chronologique d'utilisation (LRU)
    - Ordre chronologique de chargement (FIFO)
    - Deuxième chance ou Horloge (Clock)

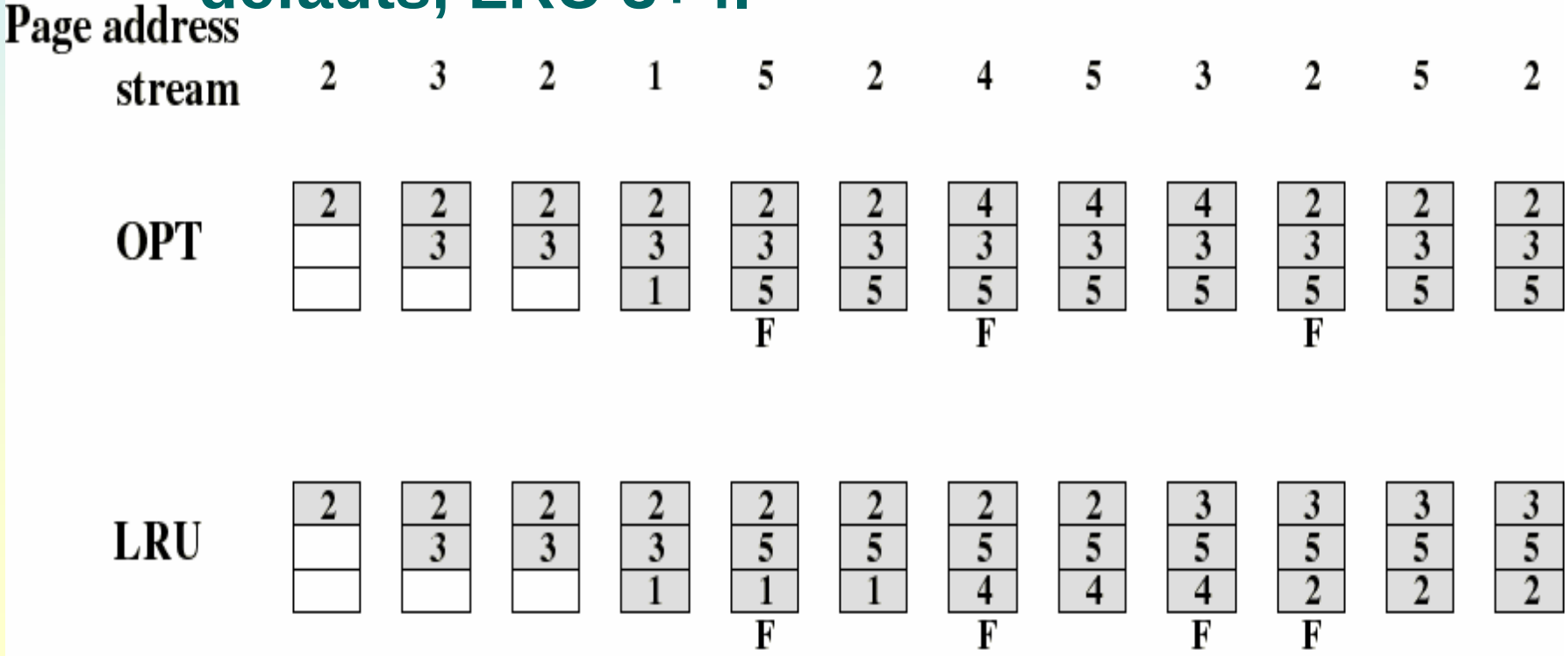
Algorithmes pour la politique de remplacement

- **Ordre chronologique d'utilisation (LRU)**
- **Remplace la page dont la dernière référence remonte au temps le plus lointain (le passé utilisé pour prédire le futur)**
  - En raison de la localité des références, il s'agit de la page qui a le moins de chance d'être référencée
  - performance presque aussi bonne que l'algo. OPT

Stallings

Comparaison OPT-LRU

- **Exemple: Un processus de 5 pages s'ìl n`y a que 3 pages physiques disponibles.**
- **Dans cet exemple, OPT occasionne 3+3 défauts, LRU 3+4.**



Note sur le comptage des défauts de page

- **Lorsque la mémoire principale est vide, chaque nouvelle page que nous ajoutons est le résultat d'un défaut de page**

- **Mais pour mieux comparer les algorithmes, il est utile de garder séparés ces défauts initiaux**
  - car leur nombre est le même pour tous les algorithmes

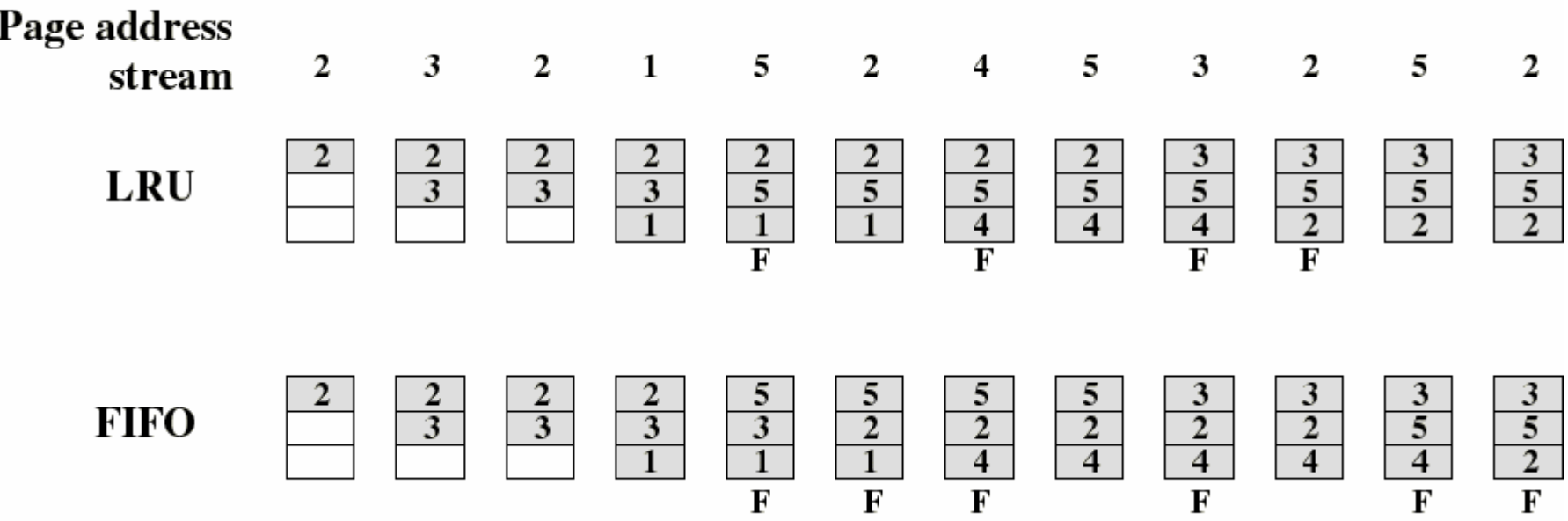
Implémentation problématique de LRU

- **Chaque page peut être marquée (dans le descripteur dans la table de pages) du temps de la dernière référence:**
  - besoin de matériel supplémentaire.
- **La page LRU est celle avec la + petite valeur de temps (nécessité d'une recherche à chaque défaut de page)**
- **On pourrait penser à utiliser une liste de pages dans l'ordre d 'utilisation: perte de temps à maintenir et consulter cette liste (elle change à chaque référence de mémoire!)**
- **D'autres algorithmes sont utilisés:**
  - LRU *approximations*

Premier arrivé, premier sorti (FIFO)

- **Logique: une page qui a été longtemps en mémoire a eu sa chance d ’exécuter**
- **Lorsque la mémoire est pleine, la plus vieille page est remplacée. Donc: “first-in, first-out”**
- **Simple à mettre en application**
- **Mais: Une page fréquemment utilisée est souvent la plus vielle, elle sera remplacée par FIFO!**

Comparaison de FIFO avec LRU (Stallings)



- **Contrairement à FIFO, LRU reconnaît que les pages 2 and 5 sont utilisées fréquemment**
- **Dans ce cas, la performance de FIFO est moins bonne:**
  - LRU = 3+4, FIFO = 3+6

Implantation de FIFO

- **Facilement implantable en utilisant un tampon circulaire de cadres de mémoire**
  - Qui ne doit être mis à jour que à chaque défaut de page
  - Exercice: concevoir l'implantation de ce tampon (v. exemple précédent)

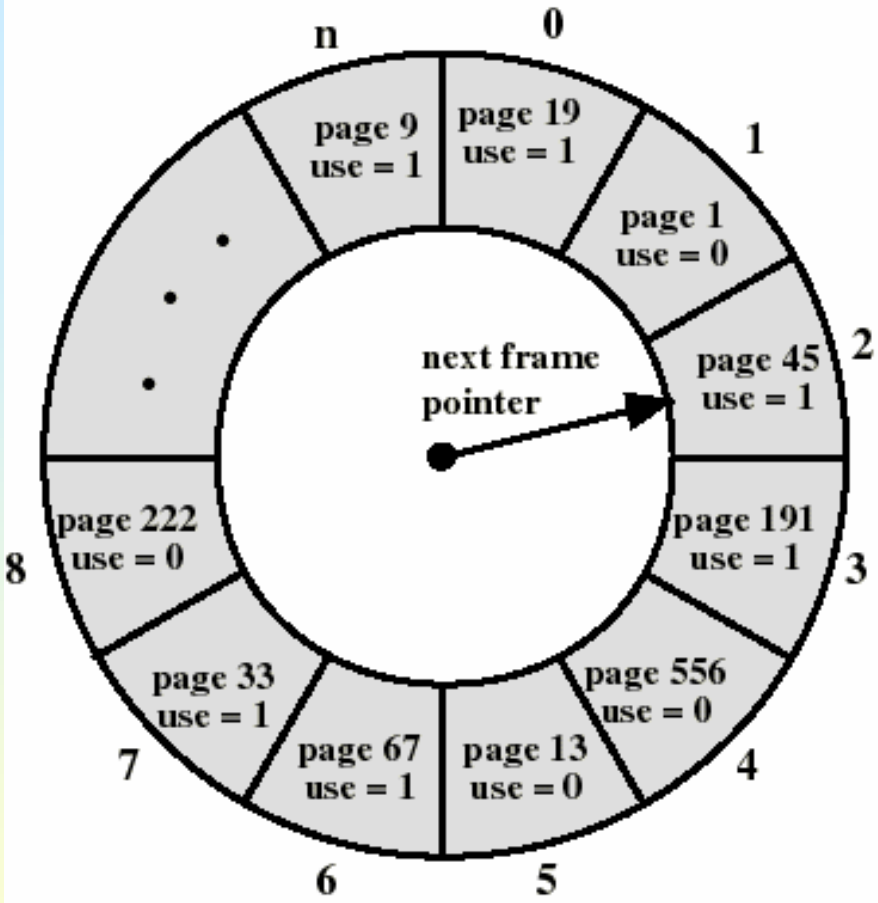
Problème conceptuel avec FIFO

- **Les premières pages amenées en mémoire sont souvent utiles pendant toute l'exécution d'un processus!**
  - variables globales, programme principal, etc.
- **Ce qui montre un problème avec notre façon de comparer les méthodes sur la base d 'une séquence aléatoire:**
  - les références aux pages dans un programme réel ne seront pas vraiment aléatoires

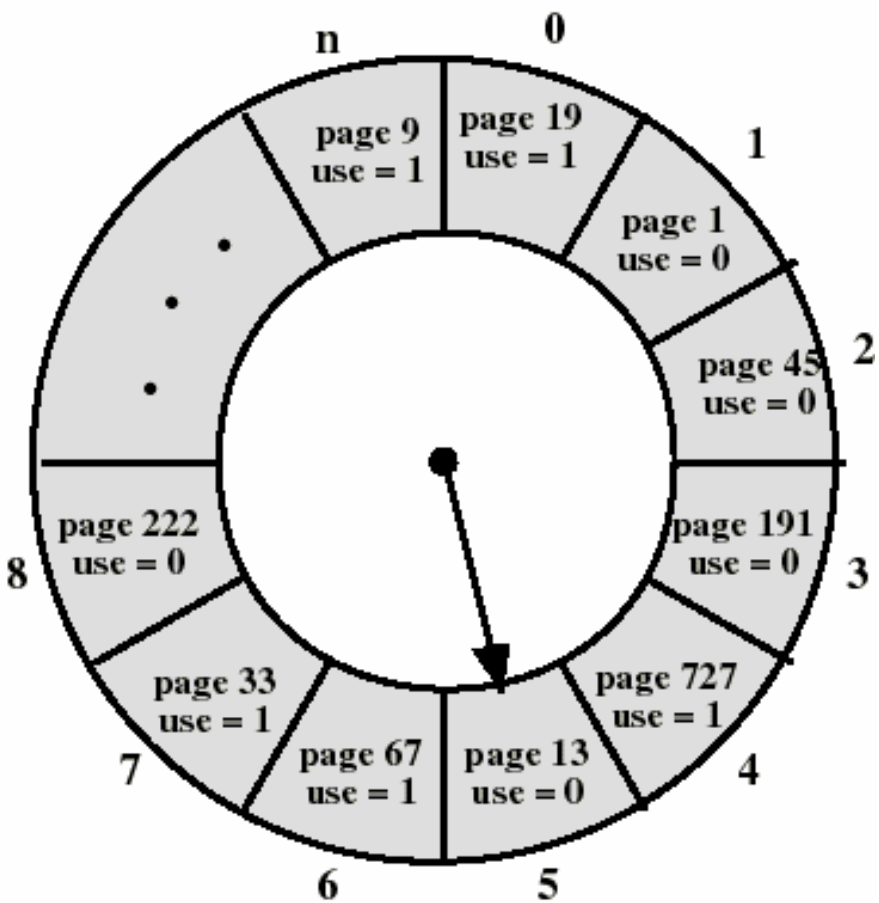
L'algorithme de l'horloge (deuxième chance)

- **Semblable à FIFO, mais il tient compte de l'utilisation récente de pages**
  - La structure à liste circulaire est celle de FIFO
- **Mais les cadres qui viennent d'être utilisés (bit=1) ne sont pas remplacées (deuxième chance)**
  - Les cadres forment conceptuellement un tampon circulaire
  - Lorsqu'une page est chargée dans un cadre, un pointeur pointe sur le prochain cadre du tampon
- **Pour chaque cadre du tampon, un bit "utilisé" est mis à 1 (par le matériel) lorsque:**
  - une page y est nouvellement chargée
  - sa page est utilisée
- **Le prochain cadre du tampon à être remplacé sera le premier rencontré qui aura son bit "utilisé" = 0.**
  - Durant cette recherche, tout bit "utilisé" = 1 rencontré sera mis à 0

Algorithme de l'horloge: un exemple (Stallings).



(a) State of buffer just prior to a page replacement

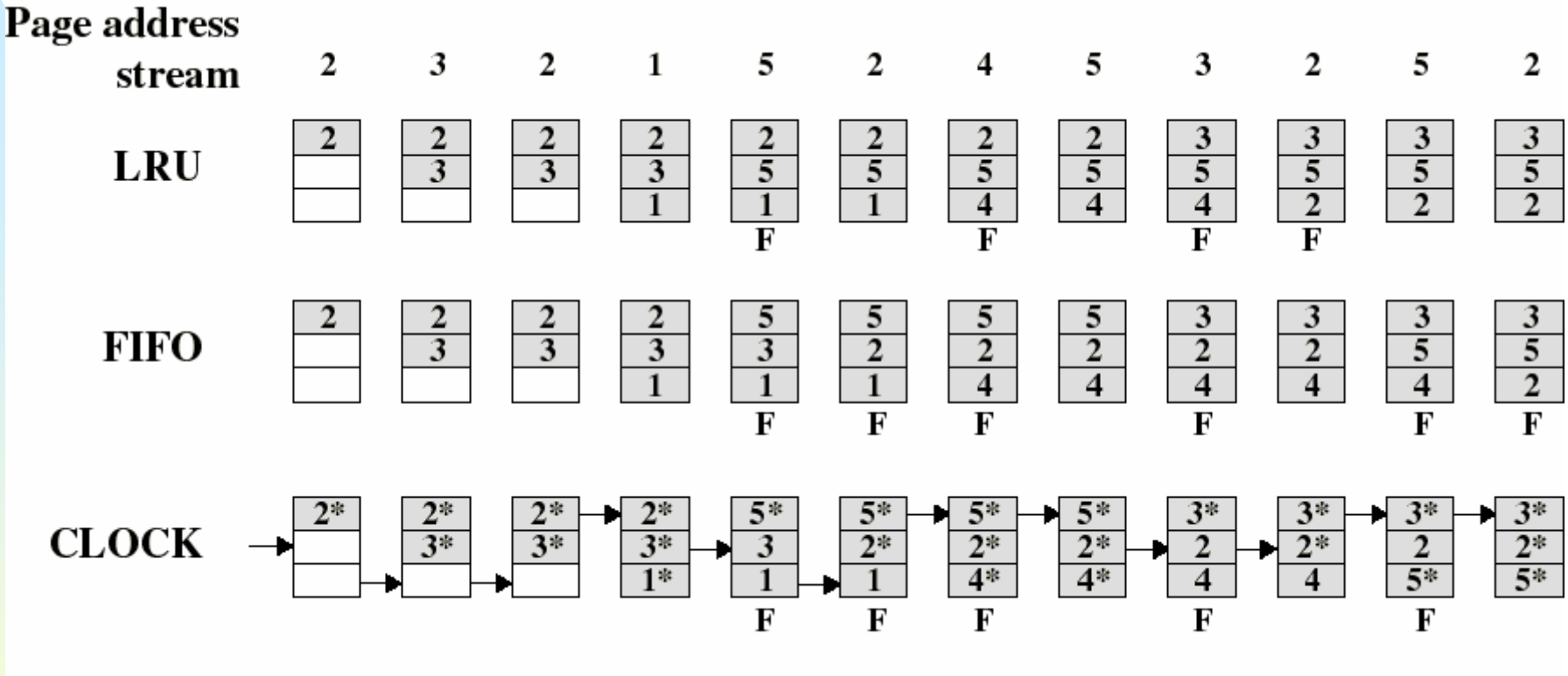


(b) State of buffer just after the next page replacement

La page 727 est chargée dans le cadre 4.

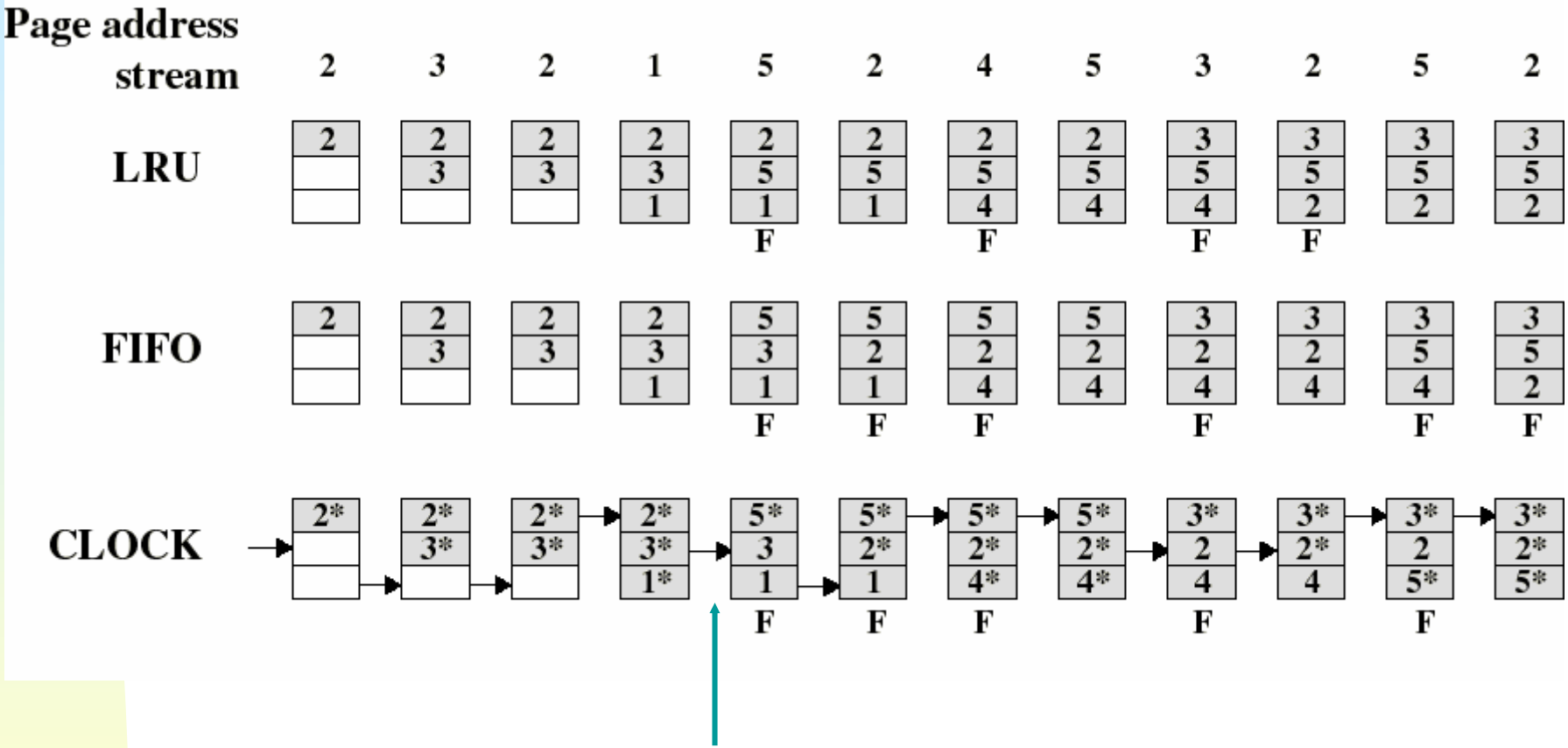
La proch. victime est 5, puis 8.

Comparaison: Horloge, FIFO et LRU (Stallings)



- Astérisque indique que le bit utilisé est 1
- L'horloge protège du remplacement les pages fréquemment utilisées en mettant à 1 le bit "utilisé" à chaque référence
- LRU = 3+4, FIFO = 3+6, Horloge = 3+5

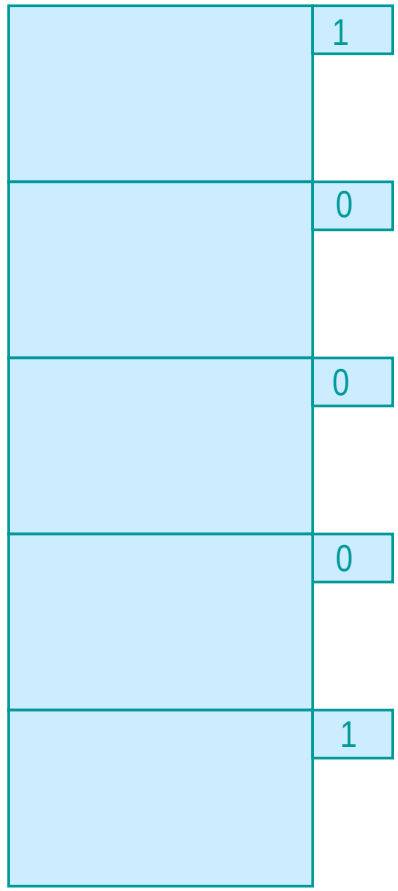
Détail sur le fonctionnement de l'horloge



Tous les bits étaient à 1. Nous avons fait tout le tour et donc nous avons changé le bit de toutes les pages à 0. Donc la 1ère page est réutilisée

Matériel additionnel pour l'algo CLOCK

- **Chaque bloc de mémoire a un bit 'touché' (use)**
- **Quand le contenu du bloc est utilisé, le bit est mis à 1 par le matériel**
- **Le SE regarde le bit**
  - S'il est 0, la page peut être remplacée
  - S'il est 1, il le met à 0



Résumé des algorithmes le plus importants

OPTIMAL	**Le meilleur en principe mais pas implantable, utilisé comme référence**
LRU	**Excellent en principe, mais demande du matériel dispendieux**
FIFO	**Facile à implanter, mais peut écarter des pages très utilisées**
Horloge	**Modification de FIFO vers LRU:** **évite d'écarter des pages récemment utilisées**

Les algorithmes utilisés en pratique sont des variations et combinaisons de ces concepts

Algorithmes *compteurs*

- **Garder un compteur pour les références à chaque page**
- **LFU: Least Frequently Used: remplacer la pages avec le plus petit compteur**
- **MFU: Most Frequently Used: remplacer les pages bien usées pour donner une chance aux nouvelles**
- **Ces algorithmes sont d'implantation dispendieuse et ne sont pas beaucoup utilisés**
  - Mise à jour de compteurs à chaque opération de mémoire!

Utilisation d'une pile (stack)

- **Quand une page est utilisée, est mise au sommet de la pile.**
  - donc la page la plus récemment utilisée est toujours au sommet,
  - la moins récemment utilisée est toujours au fond
- **Bonne implémentation du principe de localité, cependant…**
- **La pile doit être mise à jour chaque fois qu'une page est utilisée**
  - Inefficace, pas pratique

Anomalie de Belady

- **Pour quelques algorithmes, dans quelques cas il pourrait avoir plus de défauts avec plus de mémoire!**
  - p. ex. FIFO, mais pas LRU, OPT, CLOCK

Situation considérée normale

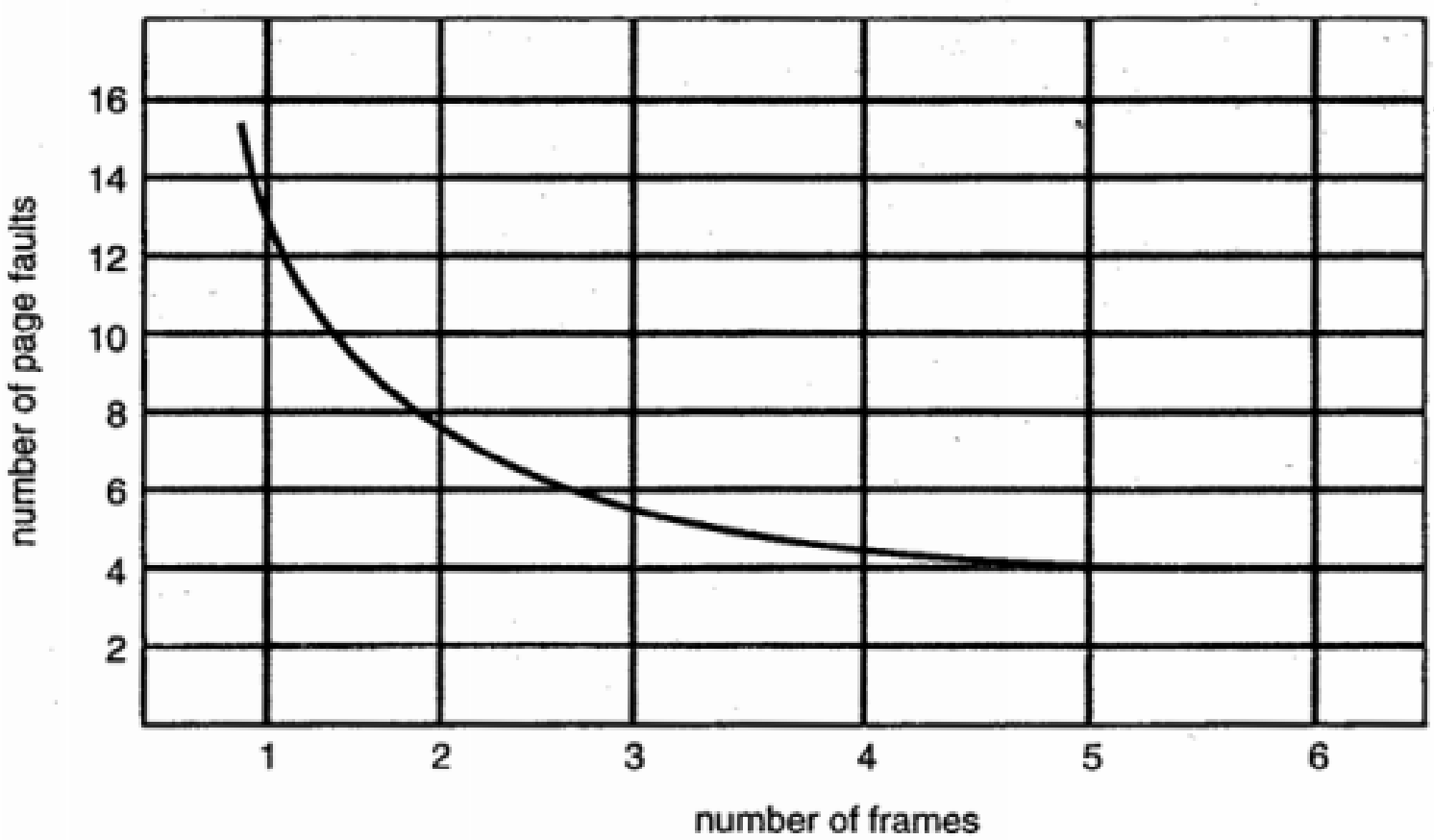
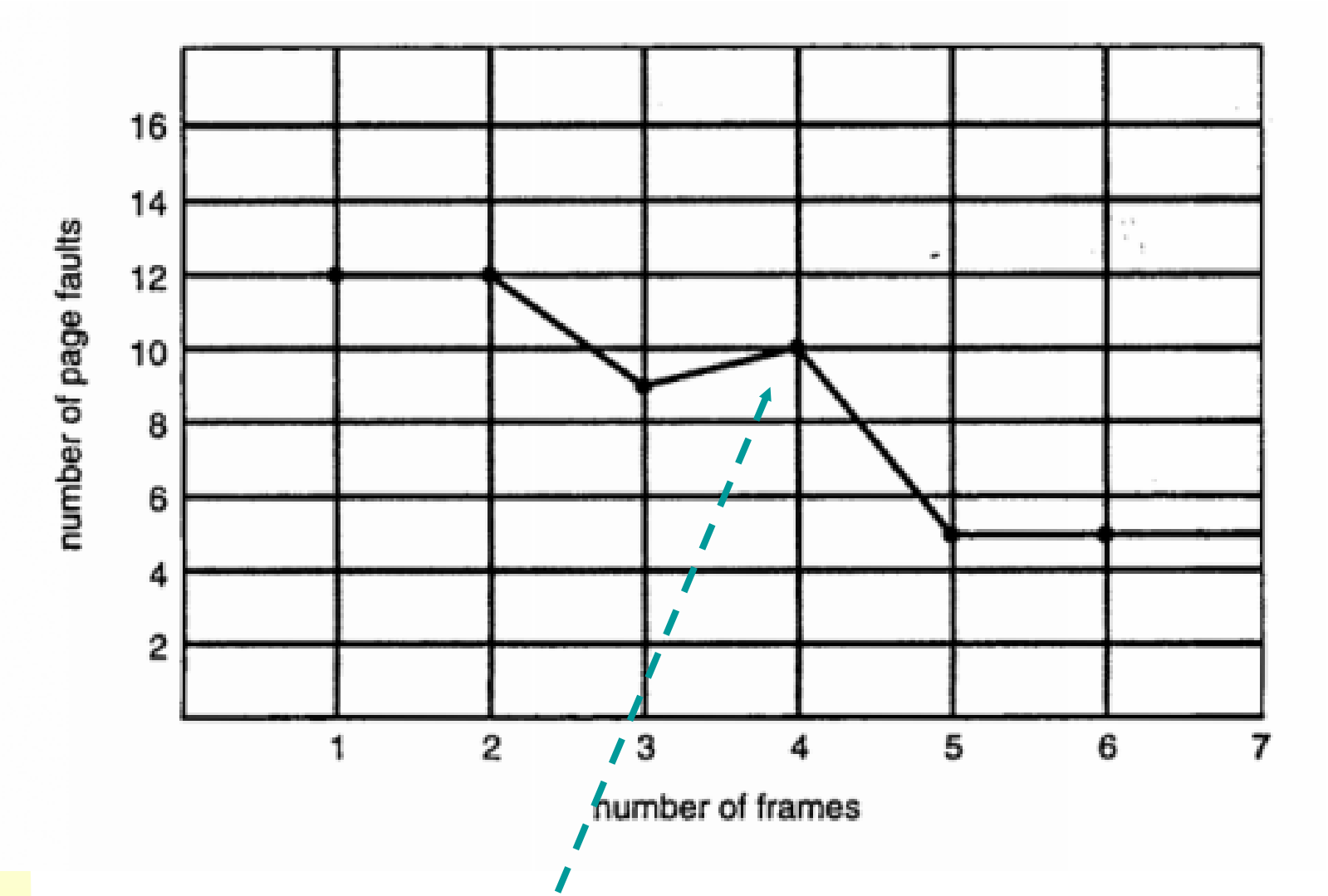


Figure 10.7 Graph of page faults versus the number of frames.

Anomalie de Belady (FIFO)



Cas d'intérêt théorique: + de mémoire, + de fautes (v. livre)

Le Chapitre 10 continue…

Chapitre 10: 2ème partie

Allocation de cadres RAM

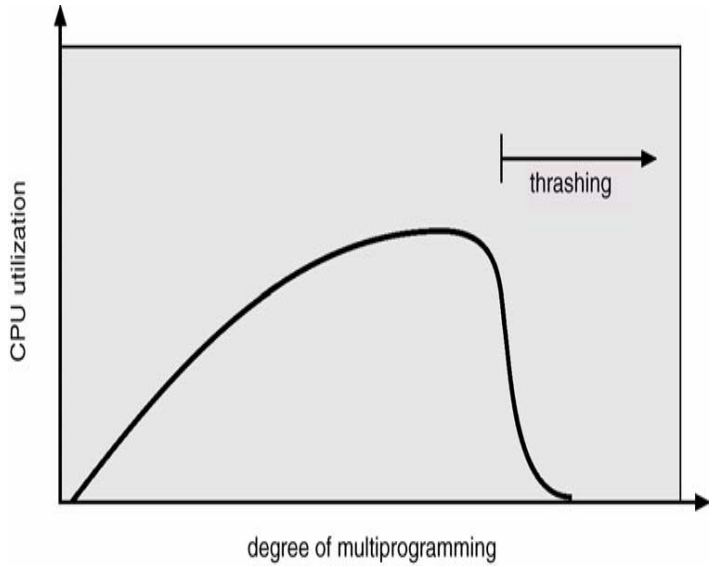
- **Pour exécuter, un processus a besoin d'un nombre minimal de cadres de mémoire RAM**
  - par exemple, quelques instructions pourraient avoir besoin de plusieurs pages simultanément pour exécuter!
- **Il est aussi facile de voir que un proc qui reçoit très peu de mémoire subira un nombre excessif de défauts de pagination, donc il sera excessivement ralenti**
- **Comment s`assurer qu'un proc soit alloué son minimum**
  - allocation égale: chaque processus a droit a une portion égale de la mémoire physique
  - allocation proportionnelle: chaque processus a droit à une portion proportionnelle à sa taille
    - le critère devrait plutôt être le besoin de pages: v. working set

Allocation globale ou locale

- globale: la `victime`est prise de n`importe quel processus
- locale: la `victime`est prise du processus qui a besoin de la page

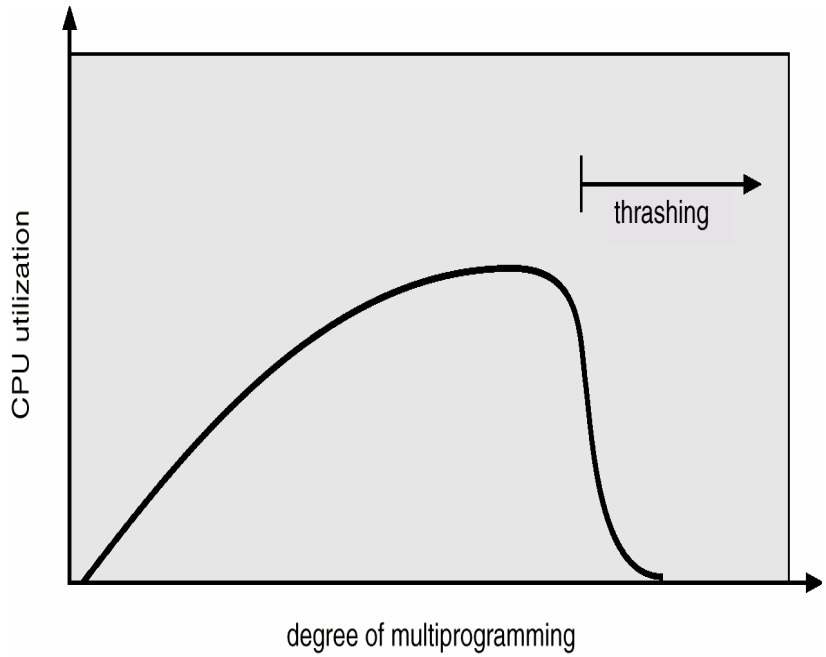
Écroulement ou thrashing (liter.: défaite)

- **S ’il n`y a pas assez de mémoire pour exécuter un proc sans trop de défauts de pagination, le proc finira pour passer trop de temps dans les files d`attente**
- **Si cette situation se généralise à plusieurs procs, l ’UCT se trouvera à être sous-utilisée**
- **Le SE pourra chercher de remédier à cette situation en augmentant le niveau de multiprogrammation**
  - plus de procs en mémoire!
  - moins de mém par proc!
  - plus de défauts de pagination!
- **Désastre: écroulement**
  - le système devient entièrement occupé à faire des E/S de pages, il ne réussit plus à faire de travail utile



La raison de l`écroulement

- **Chaque processus a besoin d ’un certain nombre de pages pour exécuter efficacement**
- **Le nombre de pages dont l’ensemble de processus a besoin à l’instant excède le nombre de cadres de mémoire RAM disponible**
  - défaite du concept de mémoire virtuelle



Ensemble de travail (working set)

- **L'ensemble de travail d'un proc donné à un moment d'exécution donné est l'ensemble des pages dont le proc a besoin pour exécuter sans trop de défauts de pagination**
  - Malheureusement, un concept flou

Chercher à prévoir les demandes de pages sur la base des demandes passées

- **Fixer un intervalle Δ**
- **Les pages intéressées par les dernières Δ operations de mémoire sont dans l ’ensemble de travail déterminé par Δ**
- **Comment choisir un Δ approprié?**

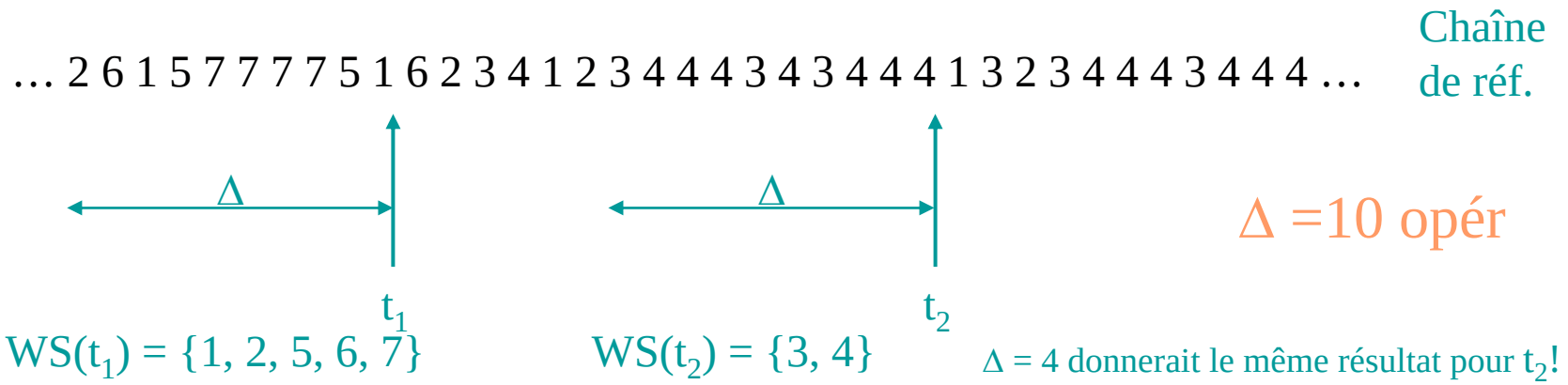


Figure 10.16 Working-set model.

Modèle de l'ensemble de travail

- Δ = **une fenêtre d'ensemble de travail**
  - = un nombre fixe de réf. de pages
  - p.ex. 10.000 opérations de mémoire
- **SI trop petit, il contiendra pas tout l 'ensemble de pages couramment utilisé par un proc**
- **Si trop grand, il contiendra plusieurs ensembles de pages**
- **WSS_i (ensemble de travail du proc. i)**
- ***D* = $\sum$ WSS_i nombre total de cadres demandés par tous les procs en exéc**
- **Si D > mémoire $\Rightarrow$ Risque d'écroulement**
- **S'assurer que ceci ne se vérifie pas**
  - si nécessaire, suspendre un des processus
- **Problème: choisir un bon Δ**
  - **peut être fait par le gérant du système**

Implémentation du concept de WS: difficile!

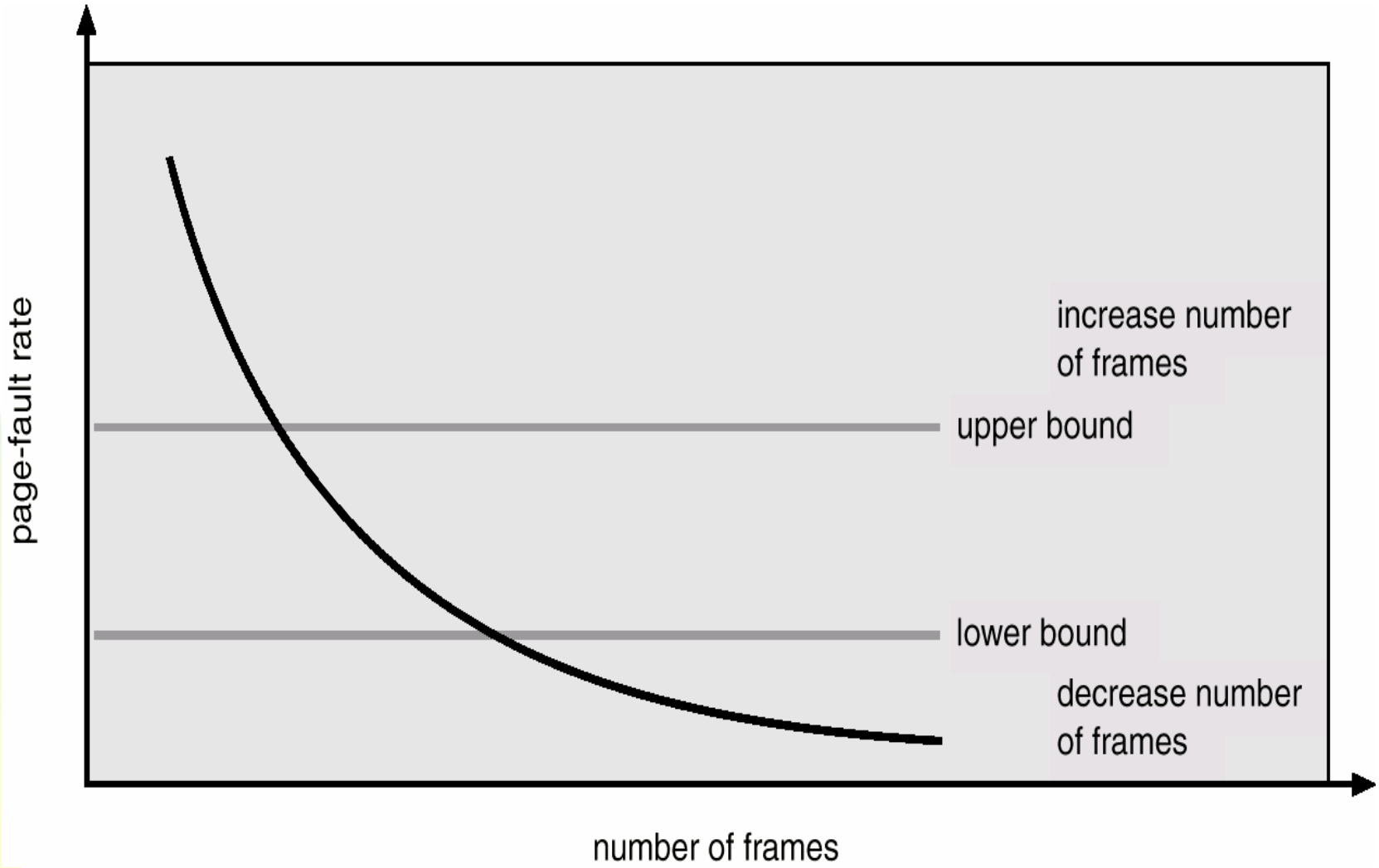
- **Minuterie et bits référence**
- **Bit de référence qui est mis à 1 chaque fois que une page est utilisée**
- **Minuterie qui interrompt régulièrement pour voir les pages qui ont été utilisées dans un intervalle de temps**

Le concept de WS en pratique

- **Deux types de difficultés:**
  - fixer le Δ de façon différente pour chaque processus, pour représenter ses besoins
- **Du matériel spécial est nécessaire pour suivre le WS d 'un proc à un moment donné**

Pour chaque processus, il existe une dimension de mémoire acceptable

ceci suggère une approche plus pratique



Une méthode plus facile à implanter que WS

- **Le gérant du système détermine quelles sont les nombres de défauts de pagination maximales et minimales tolérables dans le système, et pour chaque travail, selon ses caractéristiques**
- **Si un travail en produit plus que sa juste partie, lui donner plus de mémoire**
- **Si un travail en produit moins, lui donner moins de mémoire**
- **Suspendre si possible des travaux qu`on ne peut pas satisfaire**
- **Ou amorcer d'autres travaux si les ressources sont disponibles**

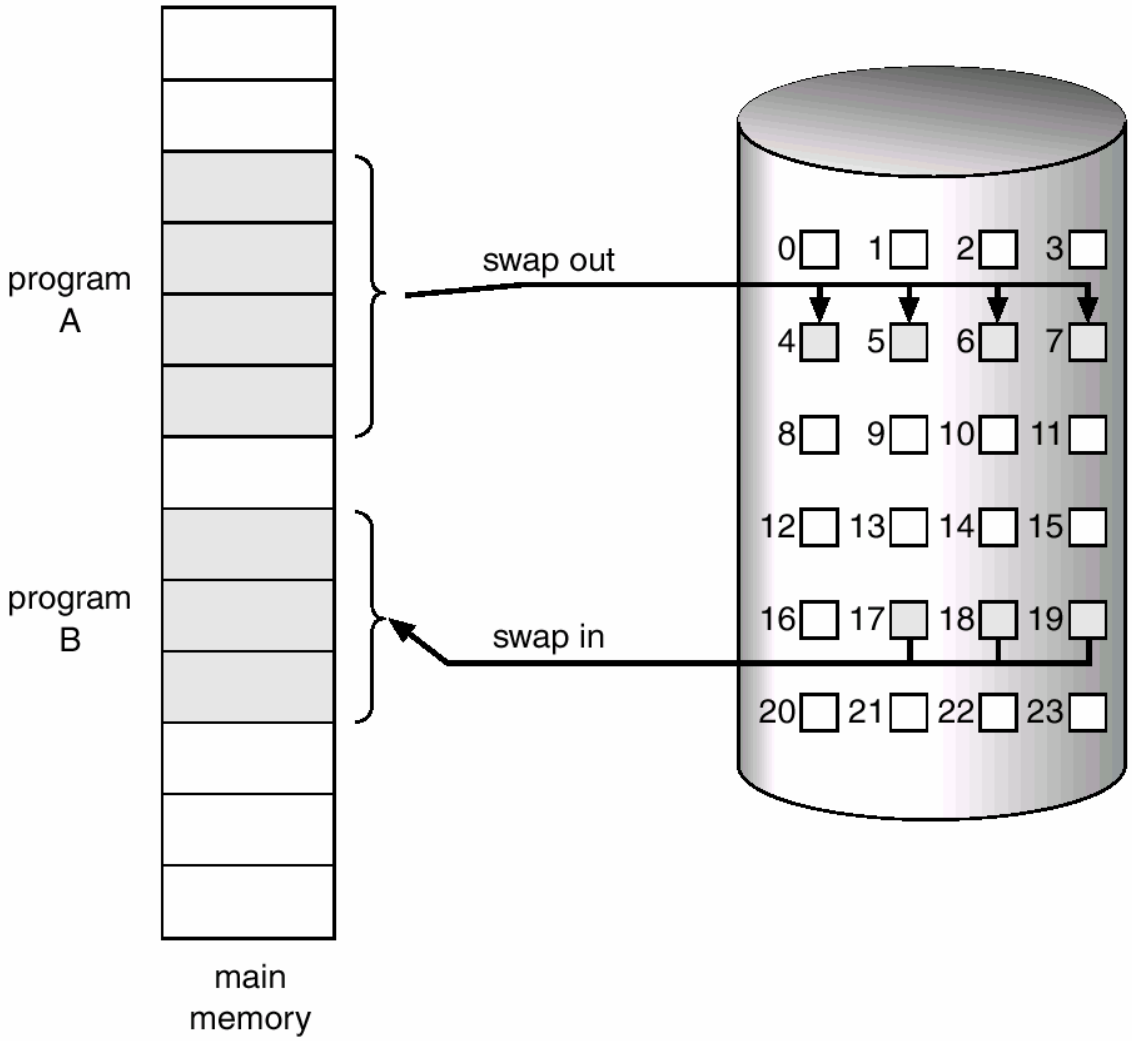
Optimisations

- **Un grand nombre de techniques d'optimisation ont été proposées et implantées**
  - Le manuel en discute plusieurs:
    - Prépagination, post-nettoyage
    - Stockage efficace
    - Taille optimale des pages

Prépagination, post-nettoyage

- **Prépagination**
  - noter quelles pages paraissent être reliées les unes aux autres
  - quand une de ces pages est amenée en mémoire RAM, y amener les autres en même temps
  - ne pas amener une seule page, s'il n`y a pas assez d 'espace pour les autres
  - éventuellement, stocker toutes ces pages dans des secteurs contigus de disque, de façon qu 'elles puissent être lues rapidement en séquence
- **Post-nettoyage (post-purging)**
  - enlever ensemble tout un groupe de pages reliées (même localité)

Stocker les pages de façon efficace

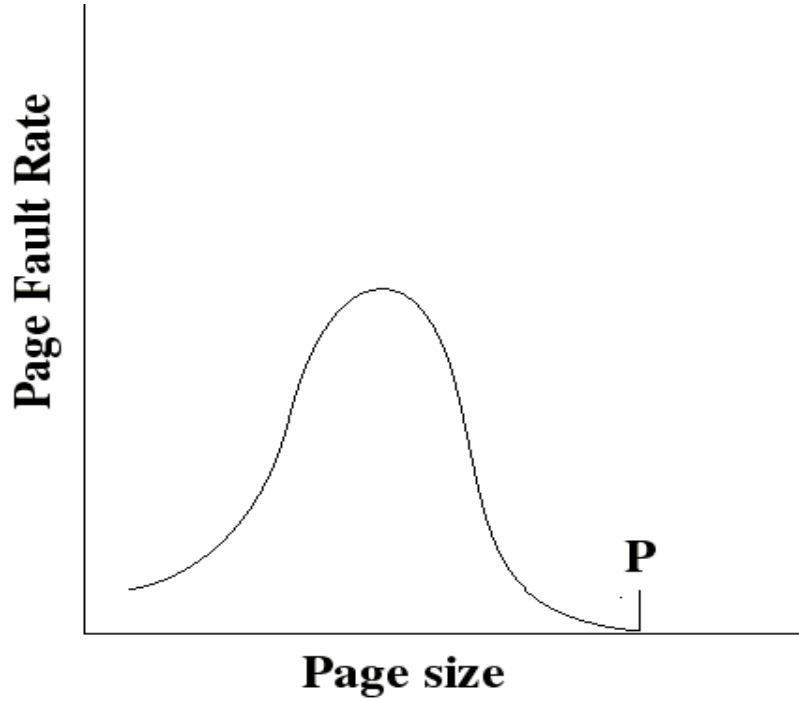


Tailles de pages (un pb de conception matériel)

- **Grande variété de tailles de pages aujourd'hui**
  - en pratique: de 512B à 16MB
- **Avantages des petites pages:**
  - moins de fragmentation interne
  - moins d'information inutile en mémoire
- **Avantages des grandes pages:**
  - tableaux de pages plus petits
  - le temps de disque le plus important est le temps de positionnement
    - une fois le disque positionné, il vaut la peine de lire une grande page
- **Tendance aujourd'hui vers grandes pages car nous avons des grands RAMs**
  - 4-8KB

Pages grandes ou petites

- **Avec une petite taille**
  - grand nombre de pages en mém. centrale
  - chaque page contient uniquement du code utilisé
  - peu de défauts de page une fois que toutes les pages utiles sont chargées
- **En augmentant la taille**
  - moins de pages peuvent être gardées dans RAM
  - chaque page contient plus de code qui n'est pas utilisé
  - plus de défauts de page
- **Mais ils diminuent lorsque nous approchons le point P où la taille d'une page est celle d'un programme entier**



Stallings

Différentes tailles de pages

- **Certains processeurs supportent plusieurs tailles. Ex:**
  - Pentium supporte 2 tailles: 4KB ou 4MB
  - R4000 supporte 7 tailles: 4KB à 16MB
  - dimensions différentes peuvent être en utilisation par des travaux différents, selon leur caractéristiques de localité
    - Travaux qui utilisent des grosses boucles et grosses structures de données pourront utiliser bien les grosses pages
    - Des gros travaux qui ‘sautent’ un peu partout sont dans la situation contraire
  - L’UCT (ou MMU) contient un registre qui dit la taille de page couramment utilisée

Effets de l'organisation du programme sur l'efficacité de la pagination

- **En principe, la pagination est censée être invisible au programmeur**
- **Mais le programmeur peut en voir les conséquences en termes de dégradation de performance**
  - p.ex. quand il change d 'un contexte à l'autre
- **Le programmeur peut chercher à éviter la dégradation en cherchant d 'augmenter la *localité des références* dans son programme**
- **En réalité, ce travail est fait normalement par le compilateur**

Effets de l'organisation du programme sur l'efficacité de la pagination

- **Structure de programme**
  - Array A[1024, 1024]
  - chaque ligne est stockée dans une page différente,
    - un cadre différent
  - Programme 1: **balayage par colonnes**:

    for *j* = 1 to 1024 **do**
      for *i* = 1 to 1024 **do**
        A[*i*,*j*] = 0;

    1024 x 1024 défauts de pagination
  - Programme 2: **balayage par lignes:**

    for *i* = 1 to 1024 **do**
      for *j* = 1 to 1024 **do**
        A[*i*,*j*] = 0;

    1024 défauts de pagination

Taille de pages et localité processus

- **Dans le cas de programmes qui exécutent du code qui ’saute’ beaucoup, les petites pages sont préférables (code OO est dans cette catégorie)**

Verrouillage de pages en mémoire

- **Certaines pages doivent être verrouillées en mémoire, p.ex. celles qui contiennent le noyau du SE**
- **Il est aussi essentiel de verrouiller en mémoire des pages sur lesquelles il y a exécution d 'E/S**
- **Ceci peut être obtenu avec un bit `verrou` sur le cadre de mémoire**
  - ce bit veut dire que ce cadre ne peut pas être sélectionné comme `victime`

Systèmes en temps réel

- **Avec la mémoire virtuelle, les temps d'exécution d'un processus deviennent moins prévisibles**
  - retards inattendus à cause de la pagination
- **Donc les systèmes en temps réel `durs` utilisent rarement la mémoire virtuelle**

Combinaison de techniques

- **Les SE réels utilisent les techniques que nous avons étudiées en *combinaison*, e.g.**
  - Linux utilise le buddy system en combinaison avec la pagination (la plus petite portion de mémoire allouable est une page)
  - d`autres systèmes utilisent les partitions fixes avec la pagination, ce qui peut être fait de plusieurs façons:
    - diviser la mémoire *réelle* en partitions fixes, assigner chaque partition à un ou plusieurs processus, puis paginer un processus dans la partitions qui lui a été assignée
    - diviser la mémoire *virtuelle* en partitions, assigner chaque partition à un ou plus. processus, puis utiliser la technique appropriée pour chaque processus dans sa partition
- **Les SE réels sont complexes et variés, mais les principes étudiés dans ce cours en constituent la base.**

Conclusions 1

- **Il est fortement désirable que l`espace d ’adressage de l ’usager puisse être beaucoup plus grand que l ’espace d ’adressage de la mémoire RAM**
- **Le programmeur sera donc libéré de la préoccupation de gérer son occupation de mémoire**
  - cependant, il devra chercher à maximiser la localité de son processus
- **La mémoire virtuelle aussi permet à plus de processus d ’être en exécution**
  - UCT, E/S plus occupées

Conclusions 2

- **Le problème de décider la page victime n’est pas facile.**
  - Les meilleurs algorithmes sont impossibles ou difficiles à implanter
  - Cependant en pratique l ’algorithme FIFO est acceptable

Conclusions 3

- **Il faut s'assurer que chaque processus ait assez de pages en mémoire physique pour exécuter efficacement**
  - risque d 'écroulement
- **Le modèle de l'ensemble de travail exprime bien les exigences, cependant il est difficile à implanter**
- **Solution plus pragmatique, où on décide de donner + ou - de mémoire aux processus selon leur débit de défauts de pagination**
- **À fin que ces mécanismes de gestion mémoire soient efficaces, plusieurs types de mécanismes sont désirables dans le matériel**

Dans un système réel

Une lecture disque sur un PC prend approx 10ms

- Donc la limite est autour de 100 fautes de pagination par seconde

Dans un serveur avec mémoire secondaire éléctronique (p.ex. flash memory)

- le taux de pagination peut arriver à milliers de pages par seconde
  - http://bsd7.starkhome.cs.sunysb.edu/~samson

Concepts importants du Chap. 10

- **Localité des références**
- **Mémoire virtuelle implémentée par va-et-vient des pages, mécanismes, défauts de pages**
- **Adresses physiques et adresses logiques**
- **Temps moyen d'accès à la mémoire**
  - Récriture ou non de pages sur mém secondaire
- **Algorithmes de remplacement pages:**
  - OPT, LRU, FIFO, Horloge
  - Fonctionnement, comparaison
- **Écroulement, causes**
- **Ensemble de travail (working set)**
- **Relation entre la mémoire allouée à un proc et le nombre d'interruptions**
- **Relation entre la dimension de pages et le nombre d'interruptions**
- **Prépagination, post-nettoyage**
- **Effets de l'organisation d'un programme sur l'efficacité de la pagination**

Par rapport au manuel…

- **Section 10.6: intéressante, mais pas sujet d'examen**

Chapitre 11

Systèmes de fichiers

http://w3.uqo.ca/luigi/

Concepts importants du chapitre

- **Systèmes fichiers**
- **Méthodes d’accès**
- **Structures Répertoires**
- **Protection**
- **Structures de systèmes fichiers**
- **Méthodes d’allocation**
- **Gestion de l’espace libre**
- **Implémentation de répertoires**
- **Questions d’efficacité**

Que c'est qu'un fichier

- **Collection nommée d'informations apparentées, enregistrée sur un stockage secondaire**
  - Nature permanente
- **Les données qui se trouvent sur un stockage secondaires doivent être dans un fichier**
- **Différents types:**
  - Données (binaire, numérique, caractères….)
  - Programmes

Structures de fichiers

- **Aucune – séquences d'octets…**
- **Texte: Lignes, pages, docs formatés**
- **Source: classes, méthodes, procédures…**
- **Etc.**

Attributs d'un fichier

- **Constituent les propriétés du fichiers et sont stockés dans un fichier spécial appelé répertoire (directory). Exemples d'attributs:**
  - Nom:
    - pour permet aux personnes d'accéder au fichier
  - Identificateur:
    - Un nombre permettant au SE d'identifier le fichier
  - Type:
    - Ex: binaire, ou texte; lorsque le SE supporte cela
  - Position:
    - Indique le disque et l'adresse du fichier sur disque
  - Taille:
    - En bytes ou en blocs
  - Protection:
    - Détermine qui peut écrire, lire, exécuter…
  - Date:
    - pour la dernière modification, ou dernière utilisation
  - Autres…

Opérations sur les fichiers: de base

- **Création**
- **Écriture**
  - Pointeur d'écriture qui donne la position d'écriture
- **Lecture**
  - Pointeur de lecture
- **Positionnement dans un fichier (temps de recherche)**
- **Suppression d'un fichier**
  - Libération d'espace
- **Troncature: remise de la taille à zéro tout en conservant les attributs**

Autres opérations

- **Ajout d'infos (p.ex. concaténation)**
- **Rénommage**
- **Copie**
  - peut être faite par rénommage: deux noms pour un seul fichier
- **Ouverture d'un fichier: le fichier devient associé à un processus qui en garde les attributs, position, etc.**
- **Fermeture**
- **Ouverture et fermeture peuvent être explicites (ops *open*, *close*)**
- **ou implicites**

Informations reliées à un fichier ouvert

- **Pointeurs de fichier**
  - Pour accès séquentiel
  - P.ex. pour read, write
- **Compteur d'ouvertures**
- **Emplacement**

Types de fichiers

- **Certains SE utilisent l'extension du nom du fichier pour identifier le type.**
  - Microsoft: Un fichier exécutable doit avoir l'extension .EXE, .COM, ou .BAT (sinon, le SE refusera de l'exécuter)
- **Le type n'est pas défini pour certains SE**
  - Unix: l'extension est utilisée (et reconnue) seulement par les applications
- **Pour certains SE le type est un attribut**
  - MAC-OS: le fichier a un attribut qui contient le nom du programme qui l'a généré (ex: document Word Perfect)

Types de fichiers

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Structure logique des fichiers

- **Le type d'un fichier spécifie sa structure**
  - Le SE peut alors supporter les différentes structures correspondant aux types de fichiers
    - Cela complexifie le SE mais simplifie les applications
- **Généralement, un fichier est un ensemble d'enregistrements (records)**
  - Chaque enregistrement est constitué d'un ensemble de champs (fields)
    - Un champ peut être numérique ou chaîne de chars.
  - Les enregistrements sont de longueur fixe ou variable (tout dépendant du type du fichier)
- **Mais pour Unix, MS-DOS et autres, un fichier est simplement une suite d'octets « byte stream »**
  - Donc ici, 1 enregistrement = 1 octet
  - C'est l'application qui interprète le contenu et spécifie une structure
  - Plus versatile mais plus de travail pour le programmeur

Méthodes d’accès

Séquentielle
Indexée Séquentielle
Indexée
Directe

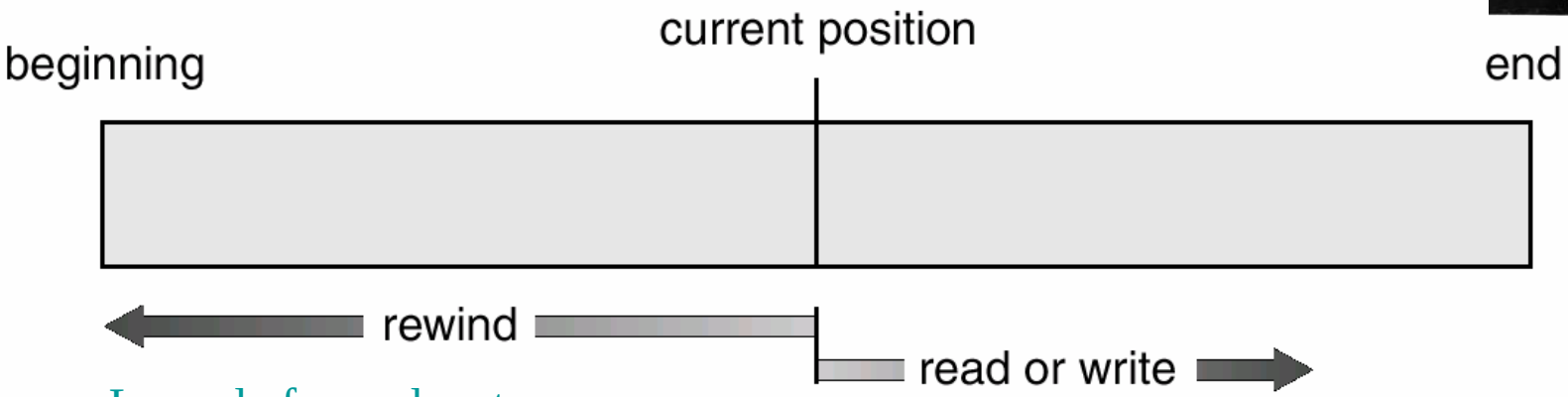
Méthodes d'accès: 4 de base

- **Séquentiel (rubans ou disques): lecture ou écriture des enregistrements dans un ordre fixe**
- **Indexé séquentiel (disques): accès séquentiel ou accès direct (aléatoire) par l'utilisation d'index**
- **Indexée: multiplicité d'index selon les besoins, accès direct par l'index**
- **Direct ou hachée: accès direct à travers tableau d'hachage**
- **Pas tous les SE supportent les méthodes d'accès**
  - Quand le SE ne les supporte pas, c'est aux librairies d'usager de les supporter

Méthodes d'accès aux fichiers

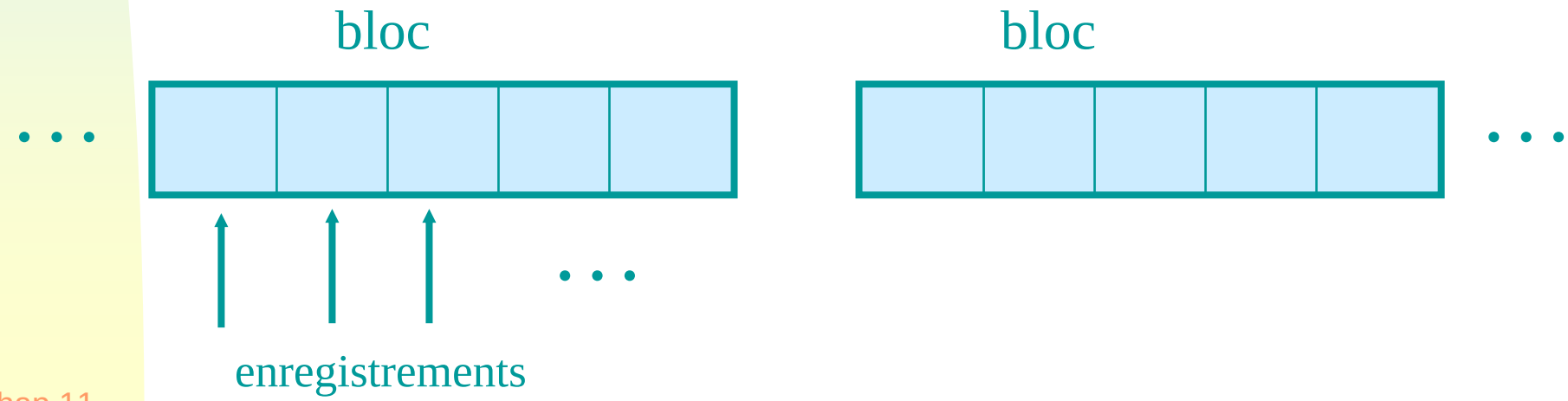
- **La structure logique d'un fichier détermine sa méthode d'accès**
- **Les SE sur les « mainframe » fournissent généralement plusieurs méthodes d'accès**
  - Car ils supportent plusieurs types de fichiers
- **Plusieurs SE modernes (Unix, Linux, MS-DOS…) fournissent une seule méthode d'accès (séquentielle) car les fichiers sont tous du même type (ex: séquence d'octets)**
  - Mais leur méthode d'allocation de fichiers (voir + loin) permet habituellement aux applications d'accéder aux fichiers de différentes manières
- **Ex: les systèmes de gestions de bases de données (DBMS) requièrent des méthodes d'accès plus efficaces que juste séquentielle**
  - Un DBMS sur un « mainframe » peut utiliser une structure fournie par le SE pour accès efficace aux enregistrements.
  - Un DBMS sur un SE qui ne fournit qu'un accès séquentiel doit donc « ajouter » une structure aux fichiers de bases de données pour accès directs plus rapides.

Fichiers à accès séquentiel (archétype: rubans)



La seule façon de retourner en arrière est de retourner au début (rébobiner, rewind)

En avant seulement, 1 seul enreg. à la fois



Lecture physique et lecture logique dans un fichier séquentiel

- **Un fichier séquentiel consiste en blocs d'octets enregistrés sur un support tel que ruban, disque…**
- **La dimension de ces blocs est dictée par les caractéristiques du support**
- **Ces blocs sont lus (lecture physique) dans un tampon en mémoire**
- **Un bloc contient un certain nombre d'enregistrements (records) qui sont des unités d'information logiques pour l'application (un étudiant, un client, un produit…)**
  - Souvent de longueur et contenu uniformes
  - Triés par une *clé,* normalement un code (code d'étudiant, numéro produit…)
- **Une lecture dans un programme lit le prochain enregistrement**
- **Cette lecture peut être réalisée par**
  - La simple mise à jour d'un pointeur si la lecture logique précédente ne s'était pas rendue à la fin du tampon
  - la lecture du proch. bloc (dans un tampon d'E/S en mémoire) si la lecture logique précédente s'était rendue à la fin du tampon
    - Dans ce cas le pointeur est remis à 0

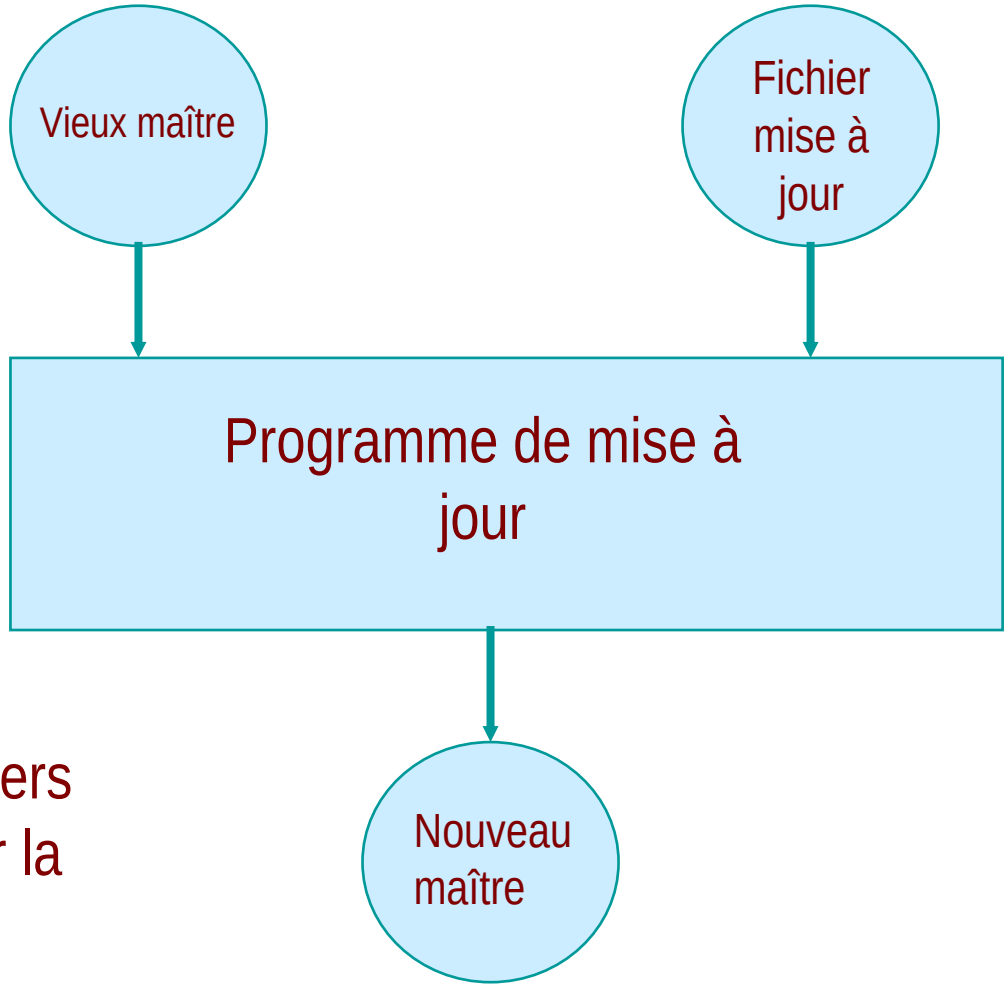
Autres propriétés des fichiers séquentiels

- **Pour l'écriture, la même idée: une instruction d'écriture dans un programme cause l'ajout d'un enregistrement à un tampon, quand le tampon est plein il y a une écriture de bloc**

- **Un fichier séquentiel qui a été ouvert en lecture ne peut pas être écrit et vice-versa (impossible de mélanger lectures et écritures)**

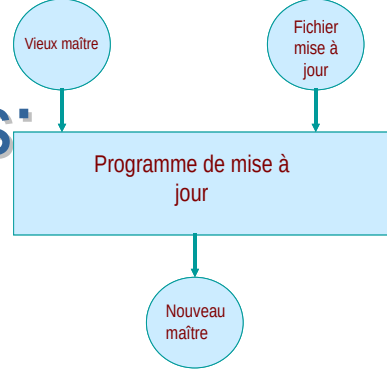
- **Les fichiers séquentiels ne peuvent être lus ou écrits qu'un enregistrement à la fois et seulement dans la direction 'en avant'**

Mise à jour de fichiers séquentiels



Tous les fichiers sont triés par la même **clé**

Mise à jour de fichiers séquentiels triés: exemple



02
05
12
17
21
26

Retirer 5
Modif 12
Ajout 20
Ajout 27

02
12
17
20
21
26
27

(12 a été modifié)

Vieux maître + Mises à jour = Nouveau maître

L'algorithme fonctionne lisant un enregistrement à la fois, en séquence, du vieux maître et du fichier des mises à jour

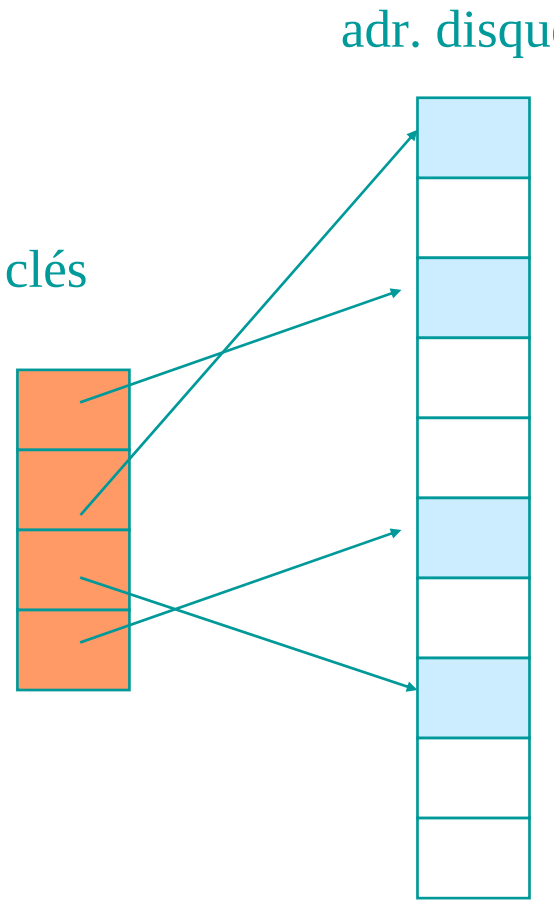
À un moment donné il n'y a que trois enregistrements en mémoire, un par fichier

Accès direct ou haché ou aléatoire:
accès direct à travers tableau d'hachage

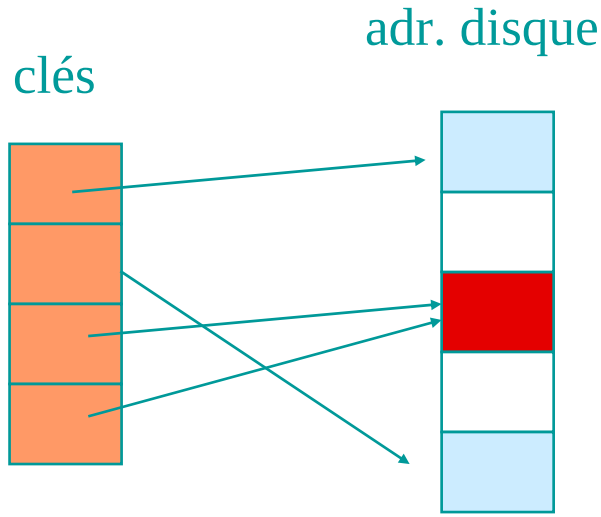
- **Une fonction d'hachage est une fonction qui traduit une clé dans adresse,**
  - P.ex. Matricule étudiant → adresse disque

- **Rapide mais:**
  - Si les adresses générées sont trop éparpillées, gaspillage d'espace
  - Si les adresses ne sont pas assez éparpillées, risque que deux clés soient renvoyées à la même adresse
    - Dans ce cas, il faut de quelques façon renvoyer une des clés à une autre adresse

Problème avec les fonctions d'hachage



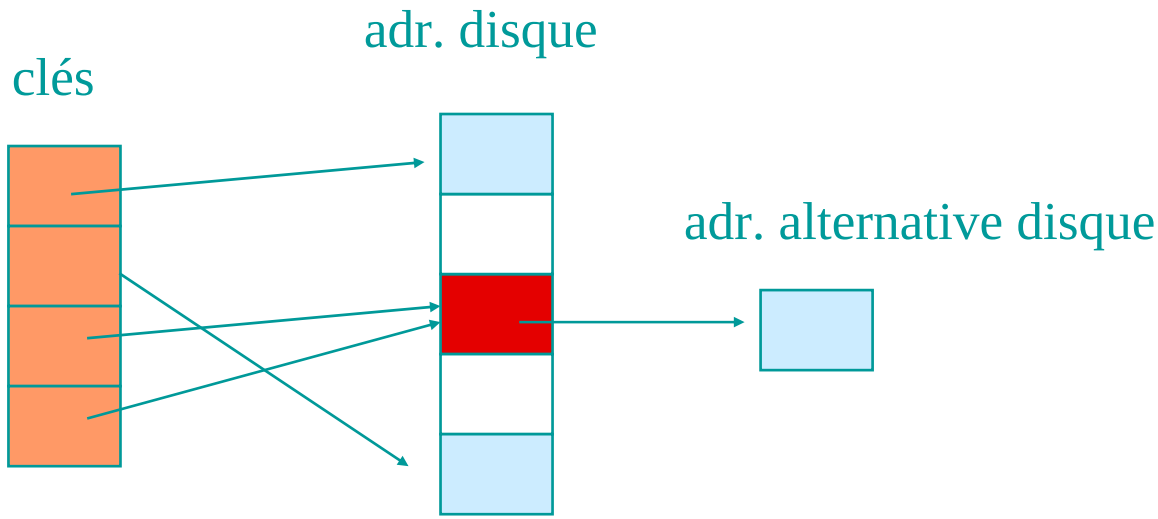
Fonction d'hachage dispersée qui n'utilise pas bien l'espace disponible



Fonction d'hachage concentrée qui utilise mieux l'espace mais introduit des doubles affectations

Hachage:Traitement des doubles affectations

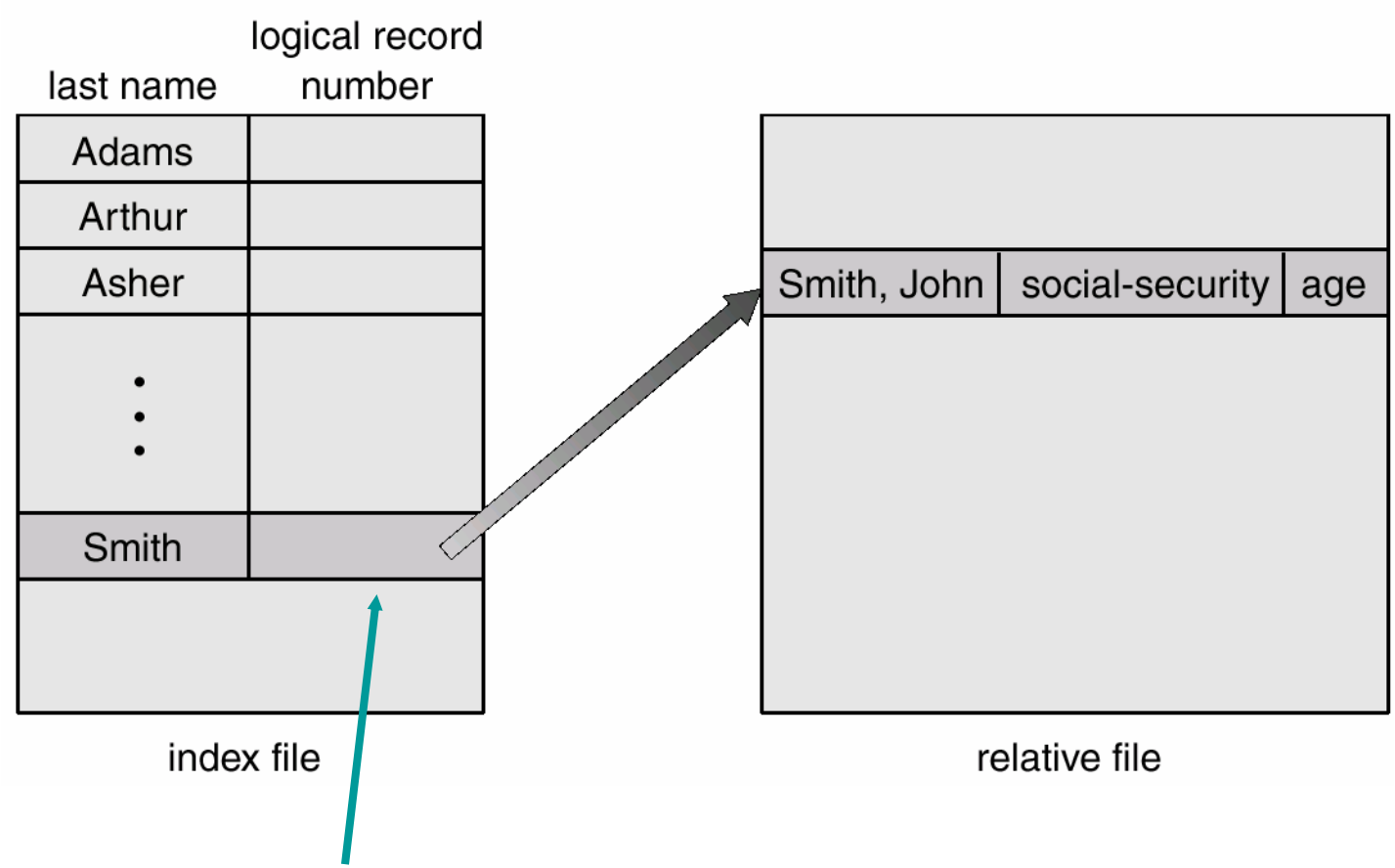
- **On doit détecter les doubles affectations, et s'il y en a, un des deux enregistrements doit être mis ailleurs**
  - ce qui complique l'algorithme



Adressage Indexé séquentiel (index sequential)

- **Un index permet d'arriver directement à l'enregistrement désiré, ou en sa proximité**
  - Chaque enregistrement contient un champ clé
  - Un fichier index contient des repères (pointeurs) à certain points importants dans le fichier principal (p.ex. début de la lettre S, début des Lalande, début des matricules qui commencent par 8)
  - Le fichier index pourra être organisé en niveaux (p.ex. dans l'index de S on trouve l'index de tous ceux qui commencent par S)
  - Le fichier index permet d'arriver au point de repère dans le fichier principal, puis la recherche est séquentielle

Exemples d'index et fichiers relatifs



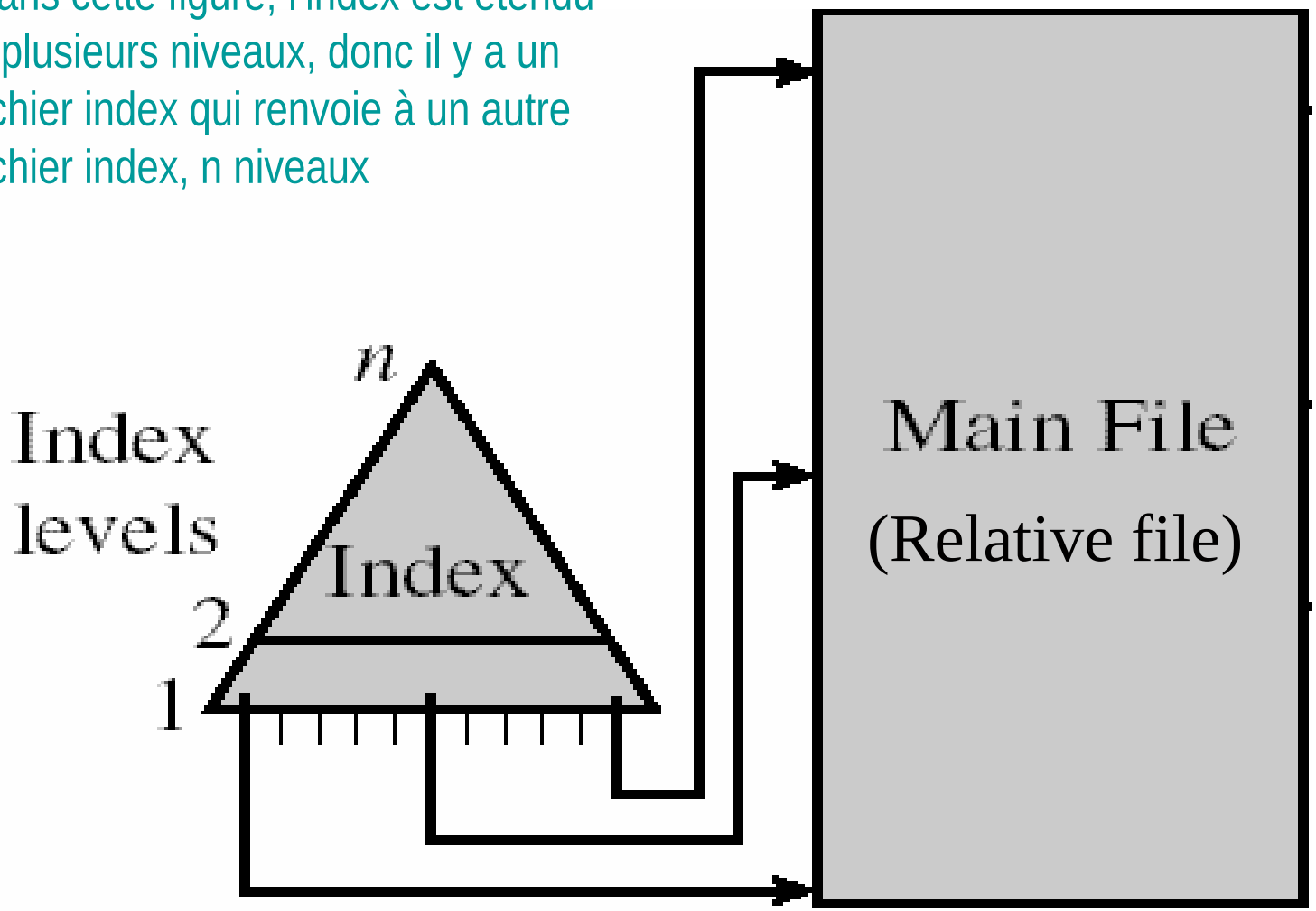
Pointe au début des Smiths (il y en aura plusieurs)

Le fichier index est à accès direct, le fichier relatif est à accès séquentiel

Accès direct: voir ci-dessous

Index et fichier principal (Stallings)

Dans cette figure, l'index est étendu à plusieurs niveaux, donc il y a un fichier index qui renvoie à un autre fichier index, n niveaux



Pourquoi plusieurs niveaux d'index

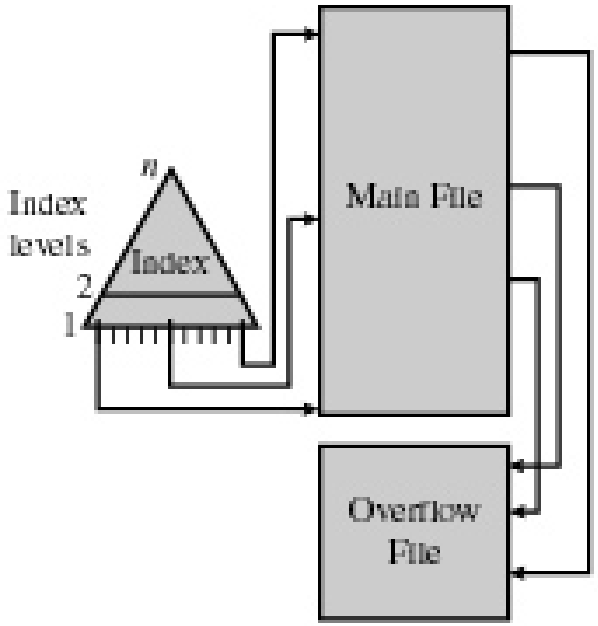
- **Un premier niveau d'index pourrait conduire au début de la lettre S**
- **Un deuxième niveau au début des Smith, etc.**
- **Donc dans le cas de fichiers volumineux plusieurs niveaux pourraient être justifiés.**

Séquentiel et index séquentiel: comparaison

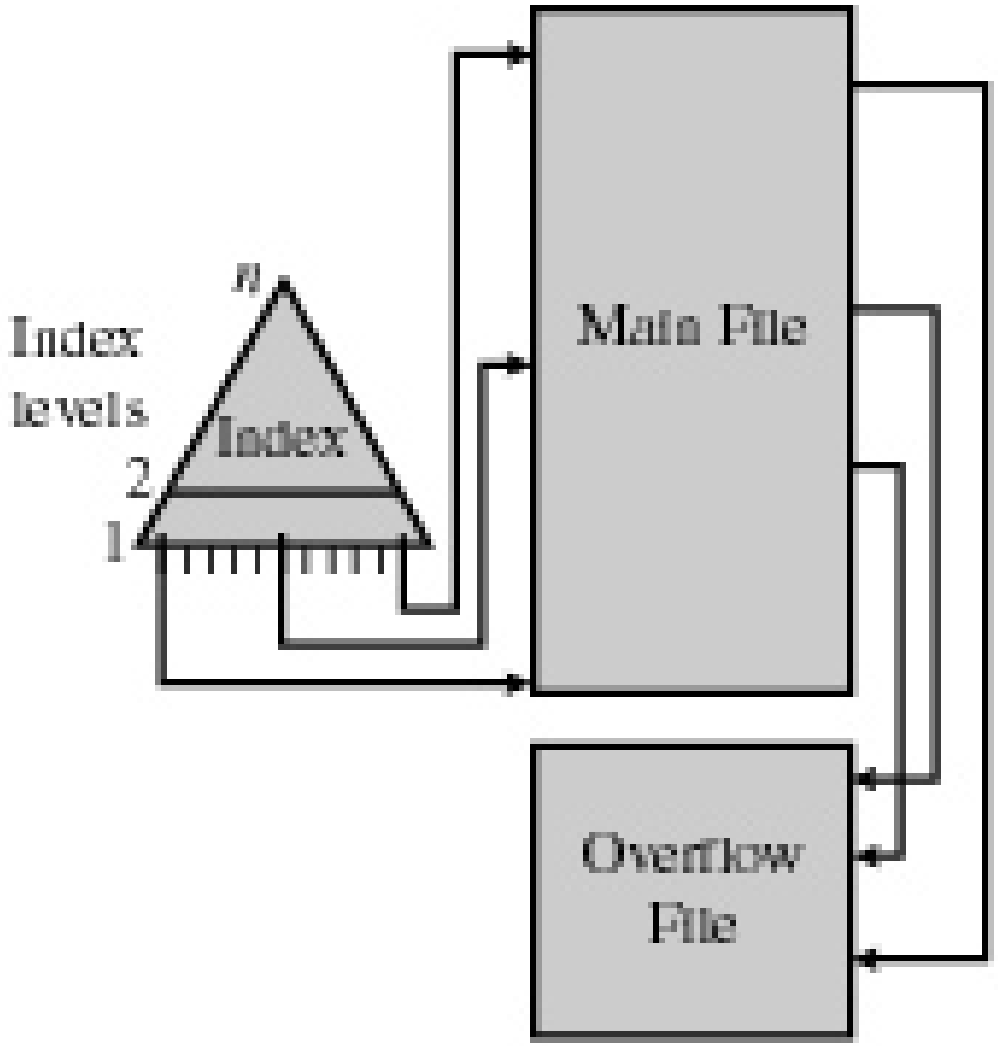
- **P.ex. Un fichier contient 1 million d'enregistrements**
- **En moyenne, 500.000 accès sont nécessaires pour trouver un enregistrement si l'accès est séquentiel!**
- **Mais dans un séquentiel indexé, s'il y a un seul niveau d'indices avec 1000 entrées (et chaque entrée pointe donc à 1000 autres),**
- ***En moyenne*, ça prend 1 accès pour trouver le repère approprié dans le fichier index**
- **Puis 500 accès pour trouver séquentiellement le bon enregistrement dans le fichier relatif**

Mais… besoin de fichier débordement

- **Les nouveaux enregistrements seront ajoutés à un fichier débordement**
- **Les enregistrements du fichier principal qui le précèdent dans l'ordre de tri seront mis à jour pour contenir un pointeur au nouveau enregistrement**
  - **Donc accès additionnels au fichiers débordement**
- **Périodiquement, le fichier principal sera fusionné avec le fichier débordement**



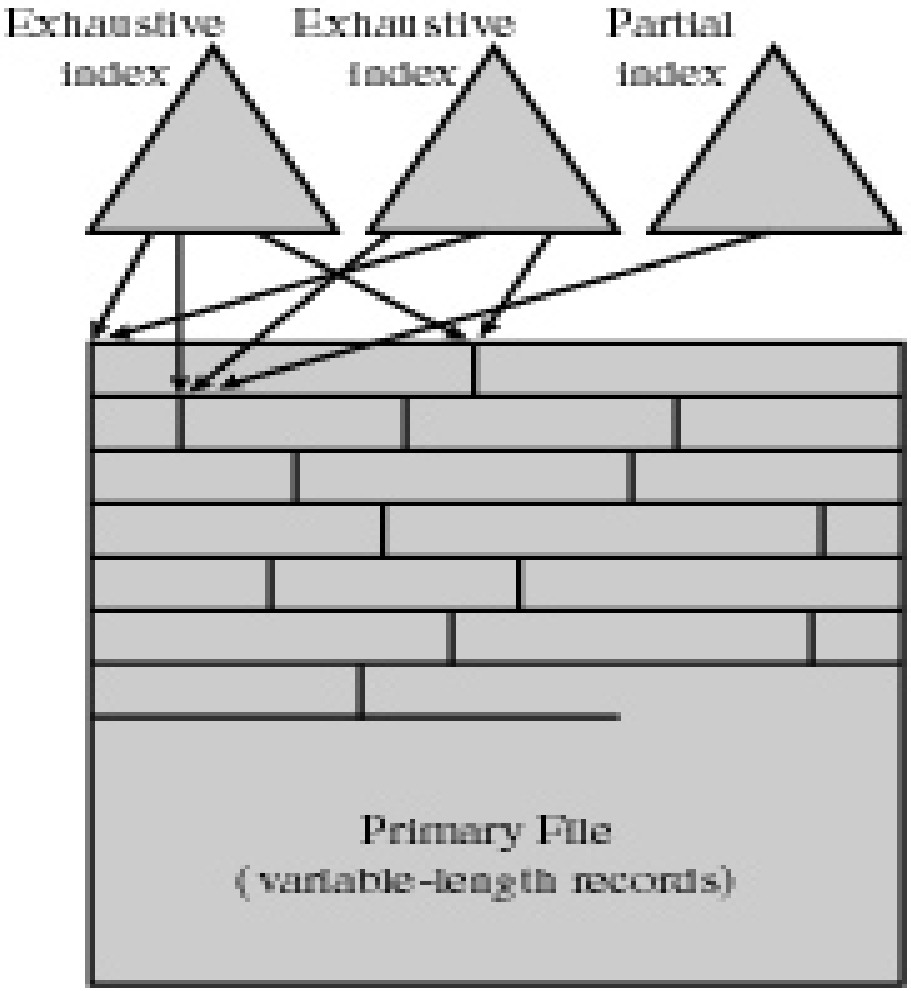
Indexed sequential (Stallings)



Fichier indexé

- **Utilise des index multiples pour différentes clés, selon les différents besoins de consultation**
- **Quelques uns pourraient être exhaustifs, quelques uns partiels, et organisés de façons différentes**

Indexed File (Stallings)



important

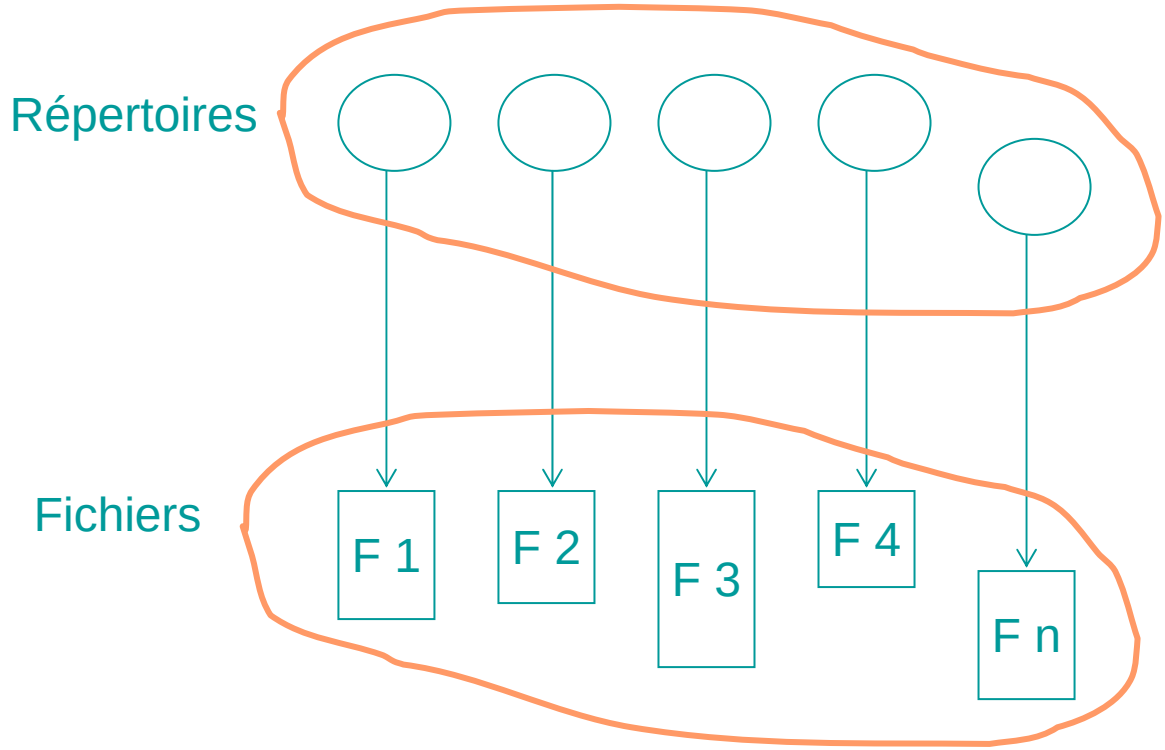
Utilisation des 4 méthodes

- **Séquentiel (rubans ou disques): applications périodiques qui demandent la lecture ou écriture des enregistrements dans un ordre fixe, ces applications sont appelées 'batch'='par lots'**
  - **Salaires, comptabilité périodique, sauvegarde périodique (backups)**
- **Indexé séquentiel (disques): accès séquentiel ou accès direct par l'utilisation d'index**
  - **Pour fichiers qui doivent être consultés parfois de façon séquentielle, parfois de façon directe (p.ex. par nom d'étudiant)**
- **Indexée: multiplicité d'index selon les besoins, accès direct par l'index**
  - **Pour fichiers qui doivent être consultés de façon directe selon des critères différents (p.ex. pouvoir accéder aux infos concernant les étudiants par la côte du cours auquel ils sont inscrits)**
- **Direct ou hachée: accès direct à travers tableau d'hachage**
  - **Pour fichiers qui doivent être consultés de façon directe par une clé uniforme (p.ex. accès aux information des étudiants par matricule)**

Répertoires

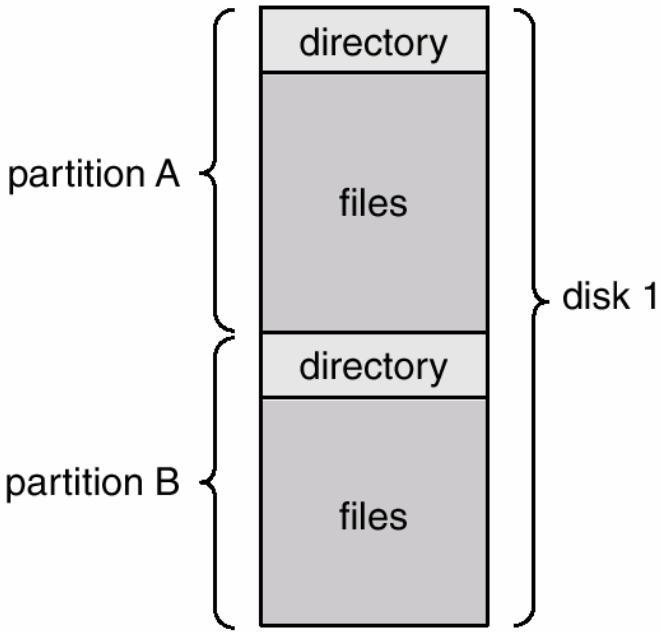
Structures de répertoires (directories)

- **Une collection de structures de données contenant infos sur les fichiers.**

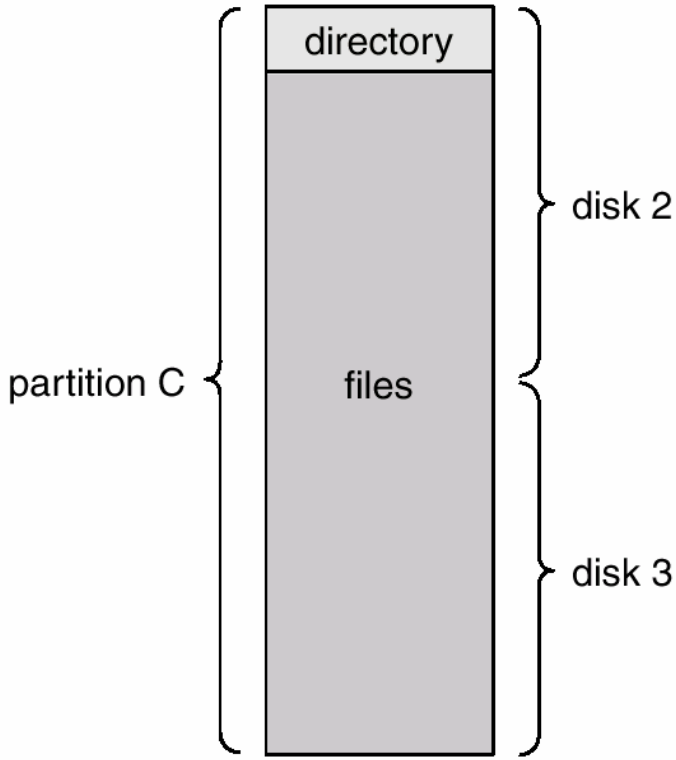


- **Tant les répertoires, que les fichiers, sont sur disques**
- **À l'exception d'un rép. racine en mém. centrale**

Organisation typique de système de fichiers



Deux répertoires dans un seul disque



Un répertoire dans deux disques

Information dans un répertoire

- **Nom du fichier**
- **Type**
- **Adresse sur disque, sur ruban...**
- **Longueur courante**
- **Longueur maximale**
- **Date de dernier accès**
- **Date de dernière mise à jour**
- **Propriétaire**
- **Protection**
- **etc**

Opérations sur répertoires

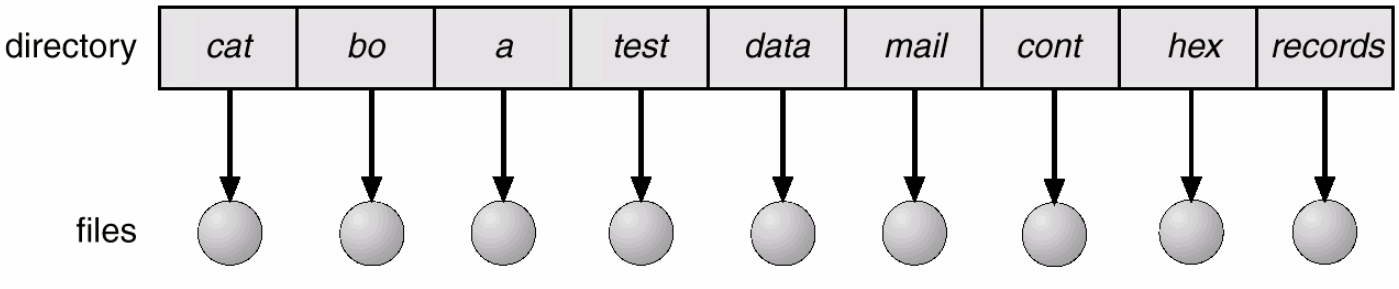
- **Recherche de fichier**
- **Création de fichier**
- **Suppression de fichier**
- **Lister un répertoire**
- **Renommer un fichier**
- **Traverser un système de fichier**

Organisation de répertoires

- **Efficacité: arriver rapidement à un enregistrement**
- **Structure de noms: convenable pour usager**
  - deux usagers peuvent avoir le même noms pour fichiers différents
  - Le même fichier peut avoir différents noms
- **Groupement de fichiers par type:**
  - tous les programmes Java
  - tous les programmes objet

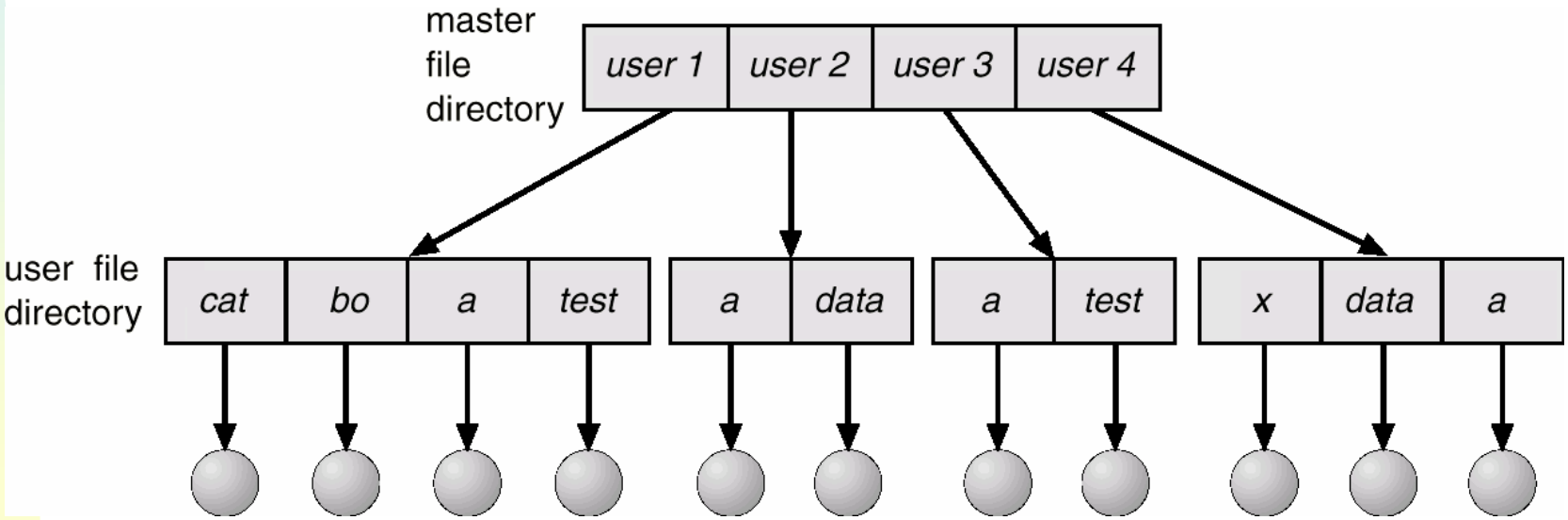
Structure à un niveau

- **Un seul rép. pour tous les usagers**
- **Ambiguïté de noms**
- **Pas de groupement**
- **Primitif, pas pratique**

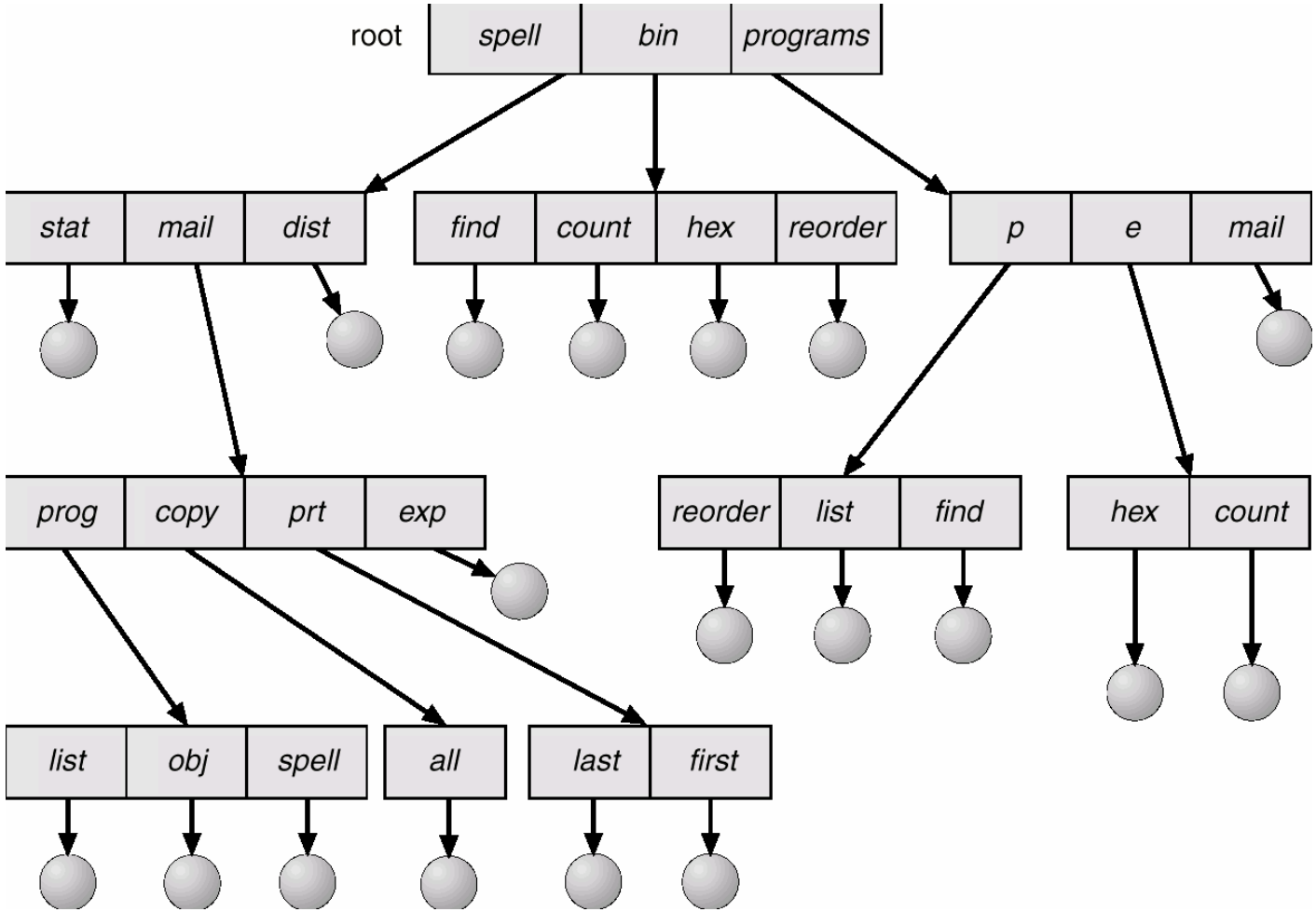


Répertoires à deux niveaux

- **Rép. séparé pour chaque usager**
- **`path name`, nom de chemin**
- **même nom de fichier pour usagers différents est permis**
- **recherche efficace**



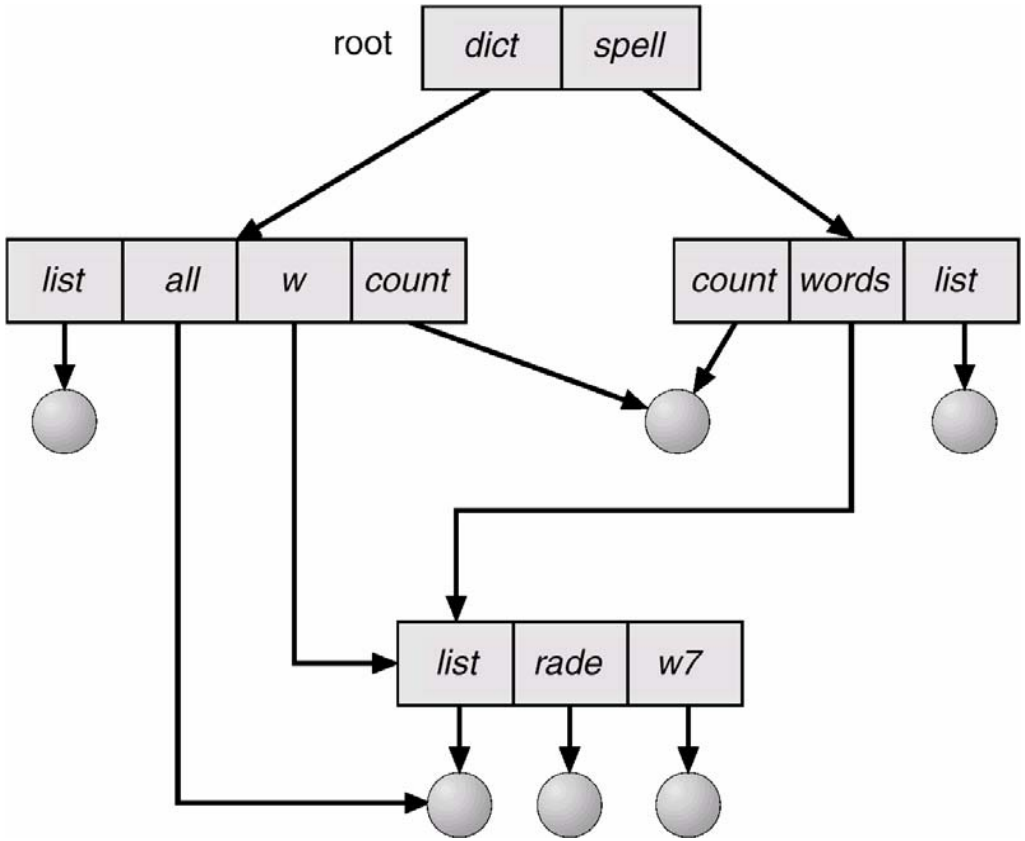
Répertoires à arbres (normal aujourd'hui)



Caractéristiques des répertoires à arbres

- **Recherche efficace**
- **Possibilité de grouper**
- **Repertoire courant (working directory)**
  - **cd** /spell/mail/prog

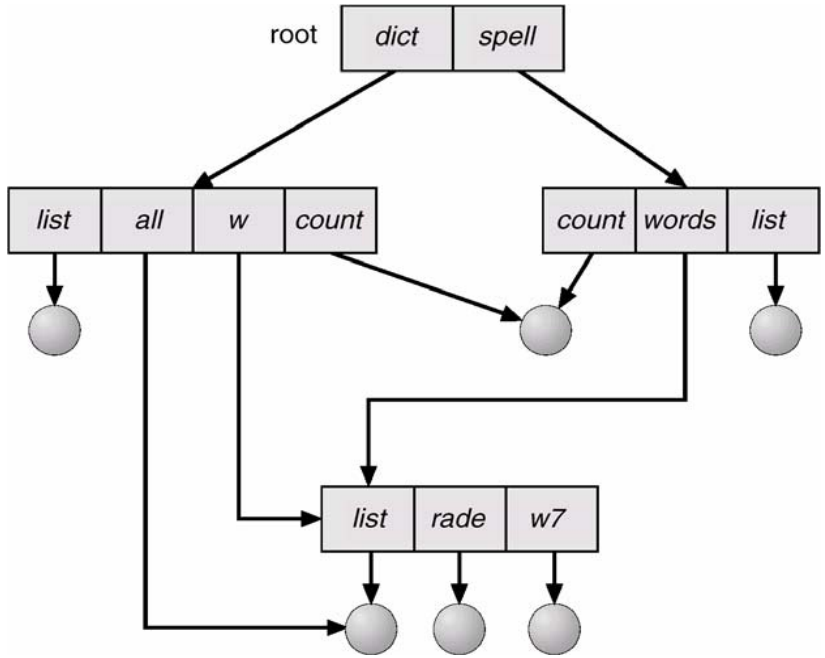
Graphes sans cycles: permettent de partager fichiers



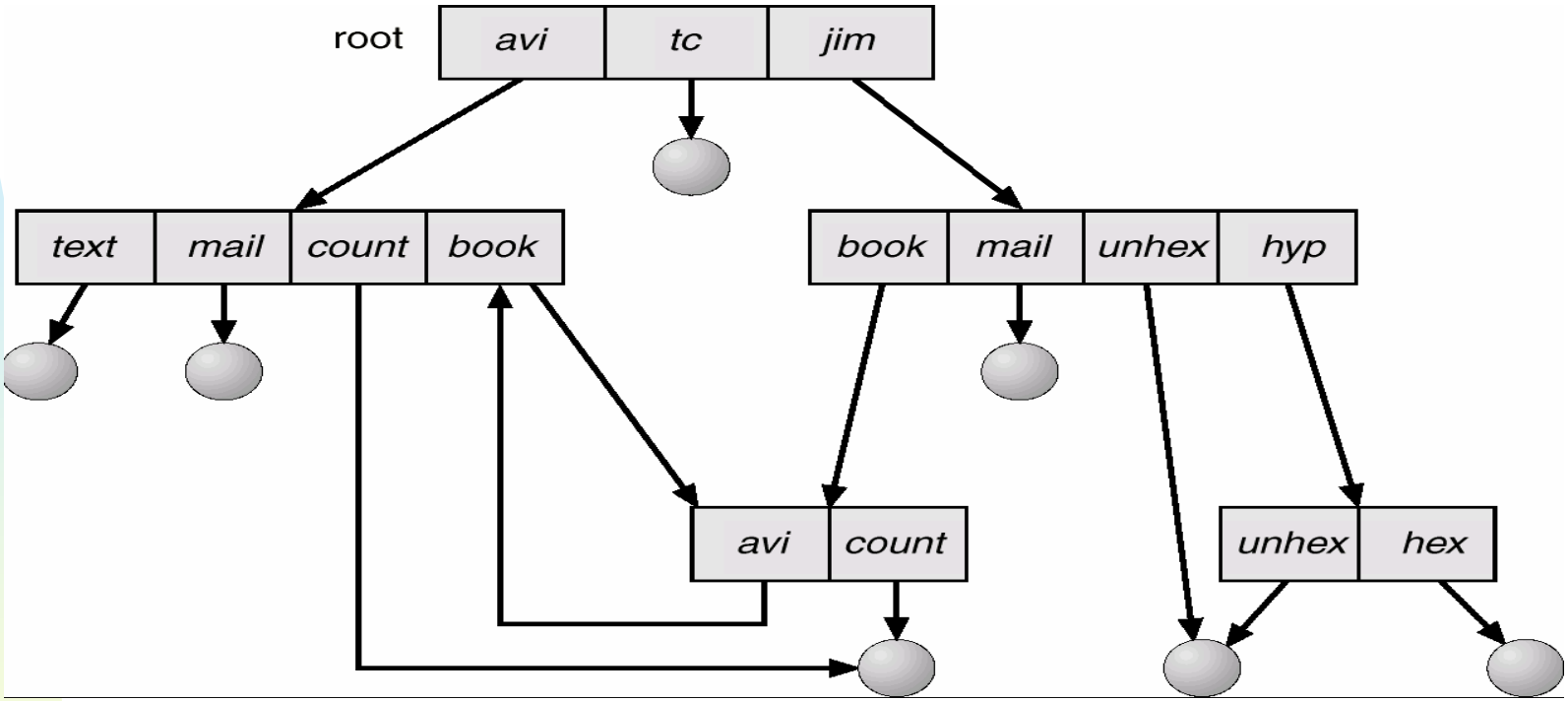
Unix: un *symbolic link* donne un chemin à un fichier ou sous-répertoire

Références multiples dans graphes acycliques

- **Un nœud peut avoir deux noms différents**
- **Si dict supprime list donc pointeur vers fichier inexistant (dangling pointer). Solutions:**
  - Pointeurs en arrière, effacent tous les pointeurs
  - Compteurs de références (s'il y a encore des refs au fichier, il ne sera pas effacé)
  - Ni Unix ni Microsoft n'implémentent ces politiques, donc messages d'erreur
  - Solutions impossibles à gérer dans un système fortement reparti (ex: www)



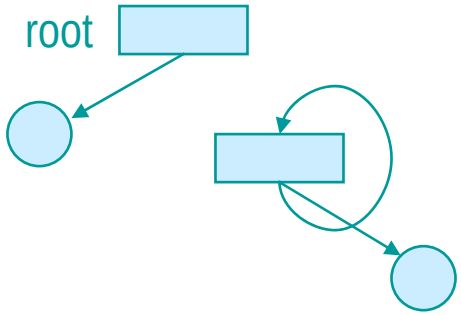
Graphes avec cycles (structure générale)



- Presque inévitables quand il est permis de pointer à un noeud arbitraire de la structure
- Pourraient être détectés avec des contrôles appropriés au moment de la création d 'un nouveau pointeur
- Contrôles qui ne sont pas faits dans les SE courants

Considérations dans le cas de cycles

- **En traversant le graphe, il est nécessaire de savoir si on retombe sur un noeud déjà visité**
- **Un noeud peut avoir compteur de ref != 0 en se trouvant dans une boucle de noeuds qui n 'est pas accessible!**
- **Des algorithmes existent pour permettre de traiter ces cas, cependant ils sont compliqués et ont des temps d 'exécution non négligeables, ce qui fait qu' ils ne sont pas toujours employés**
  - Ramasse-miettes = garbage collection



Un sous-arbre qui n'est pas accessible à partir de la racine mais il ne peut pas être effacé en utilisant le critère ref=0 car il fait ref à lui-meme!

Partage de fichiers

- **Désirable sur les réseaux**
- **Nécessite de protection**

Protection (détails dans un chap suivant)

- **Types d ’accès permis**
  - **lecture**
  - **écriture**
  - **exécution**
  - append (annexation)
  - effacement
  - listage: lister les noms et les attributs d ’un fichier
- **Par qui**

Listes et groupes d'accès - UNIX

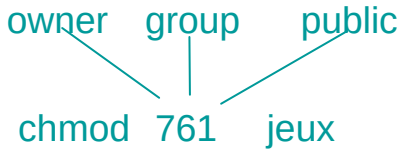
- **Modes d 'accès: R W E**
- **Trois classes d 'usager:**
  - propriétaire
  - groupe
  - public
- **demander à l 'administrateur de créer un nouveau groupe avec un certain usager et un certain propriétaire**
- **droit du propriétaire de régler les droits d 'accès et d 'ajouter des nouveaux usagers**

Listes et groupes d'accès

- **Mode d'accès: read, write, execute**
- **Trois catégories d'usagers:**

			RWX
a) owner access	**7**	⇒	**1 1 1**
b) group access	**6**	⇒	**1 1 0**
c) others access	**1**	⇒	**0 0 1**

- **Demander au gestionnaire de créer un groupe, disons G, et ajouter des usagers au groupe**
- **Pour un fichier particulier, disons jeux, définir un accès approprié**



Changer le groupe d'un fichier

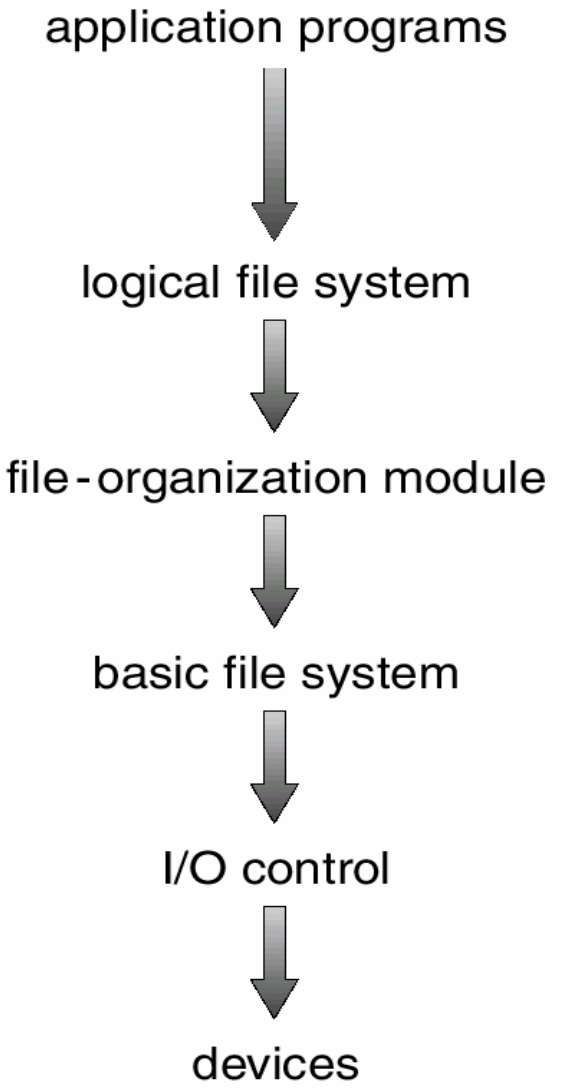
chgrp G *jeux*

Méthodes d’allocation

Structures de systèmes de fichiers

- **Structure de fichiers: deux façons de voir un fichier:**
  - unité d'allocation espace
  - collection d 'informations reliées
- **Le système de fichiers réside dans la mémoire secondaire: disques, rubans...**
- **File control block: structure de données contenant de l 'info sur un fichier**

Systèmes de fichiers à couches



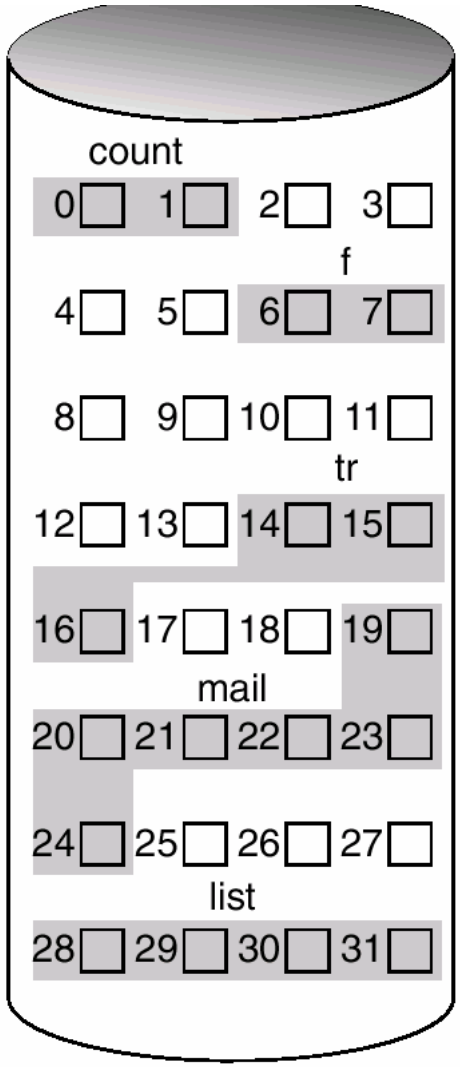
Structure physique des fichiers

- **La mémoire secondaire est subdivisée en blocs et chaque opération d'E /S s'effectue en unités de blocs**
  - Les blocs ruban sont de longueur variable, mais les blocs disque sont de longueur fixe
  - Sur disque, un bloc est constitué d'un multiple de secteurs contigus (ex: 1, 2, ou 4)
    - la taille d'un secteur est habituellement 512 bytes
- **Il faut donc insérer les enregistrements dans les blocs et les extraire par la suite**
  - Simple lorsque chaque octet est un enregistrement par lui-même
  - Plus complexe lorsque les enregistrements possèdent une structure (ex: « main-frame IBM »)
- **Les fichiers sont alloués en unité de blocs. Le dernier bloc est donc rarement rempli de données**
  - Fragmentation interne

Trois méthodes d'allocation de fichiers

- Allocation contiguë
- Allocation enchaînée
- Allocation indexée

Allocation contiguë sur disque



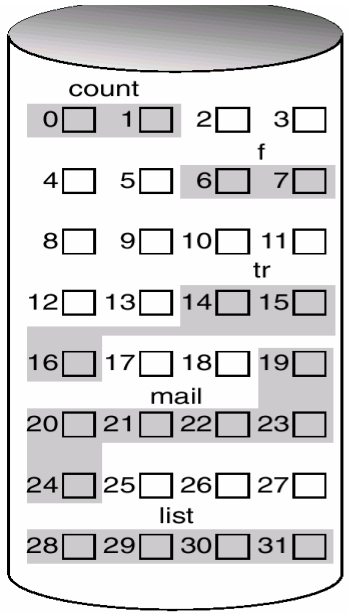
répertoire

directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation contiguë

- Chaque fichier occupe un ensemble de blocs contigu sur disque
- Simple: nous n'avons besoin que d'adresses de début et longueur
- Supporte tant l'accès séquentiel, que l'accès direct
- Moins pratique pour les autres méthodes

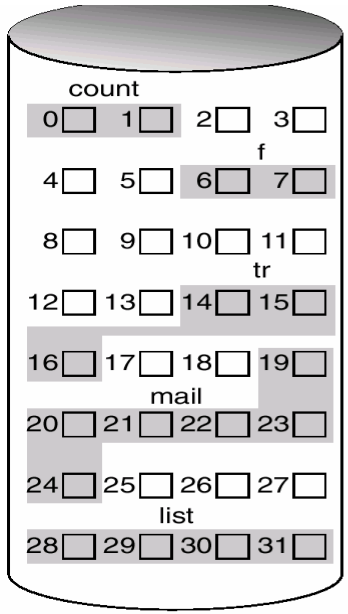


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation contiguë

- **Application des problèmes et méthodes vus dans le chapitre de l'alloc de mémoire contiguë**
- **Les fichiers ne peuvent pas grandir**
- **Impossible d'ajouter au milieu**
- **Exécution périodique d'une compression (compaction) pour récupérer l'espace libre**

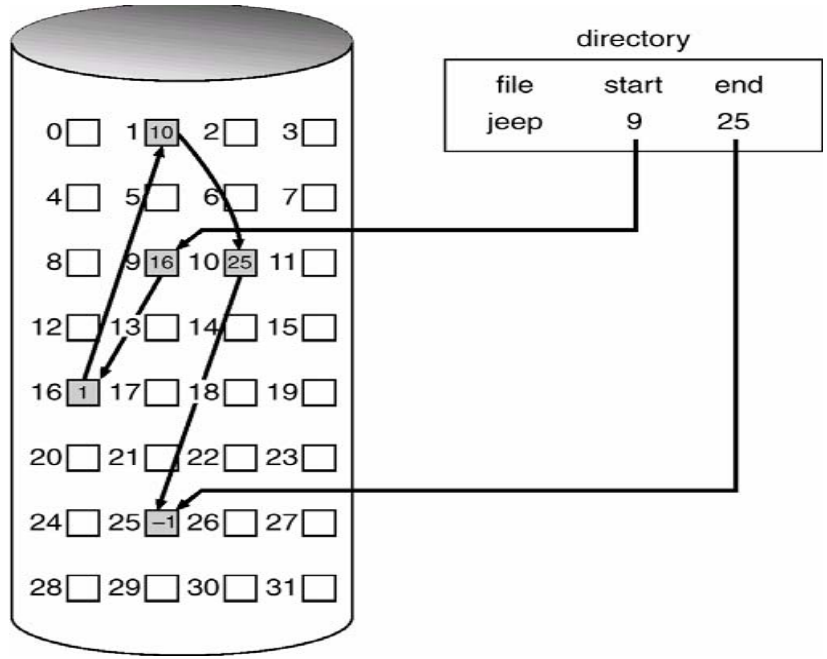


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation enchaînée

- **Le répertoire contient l ’adresse du premier et dernier bloc, possibl. le nombre de blocs**
- **Chaque bloc contient un pointeur à l’adresse du prochain bloc:**



Allocation enchaînée

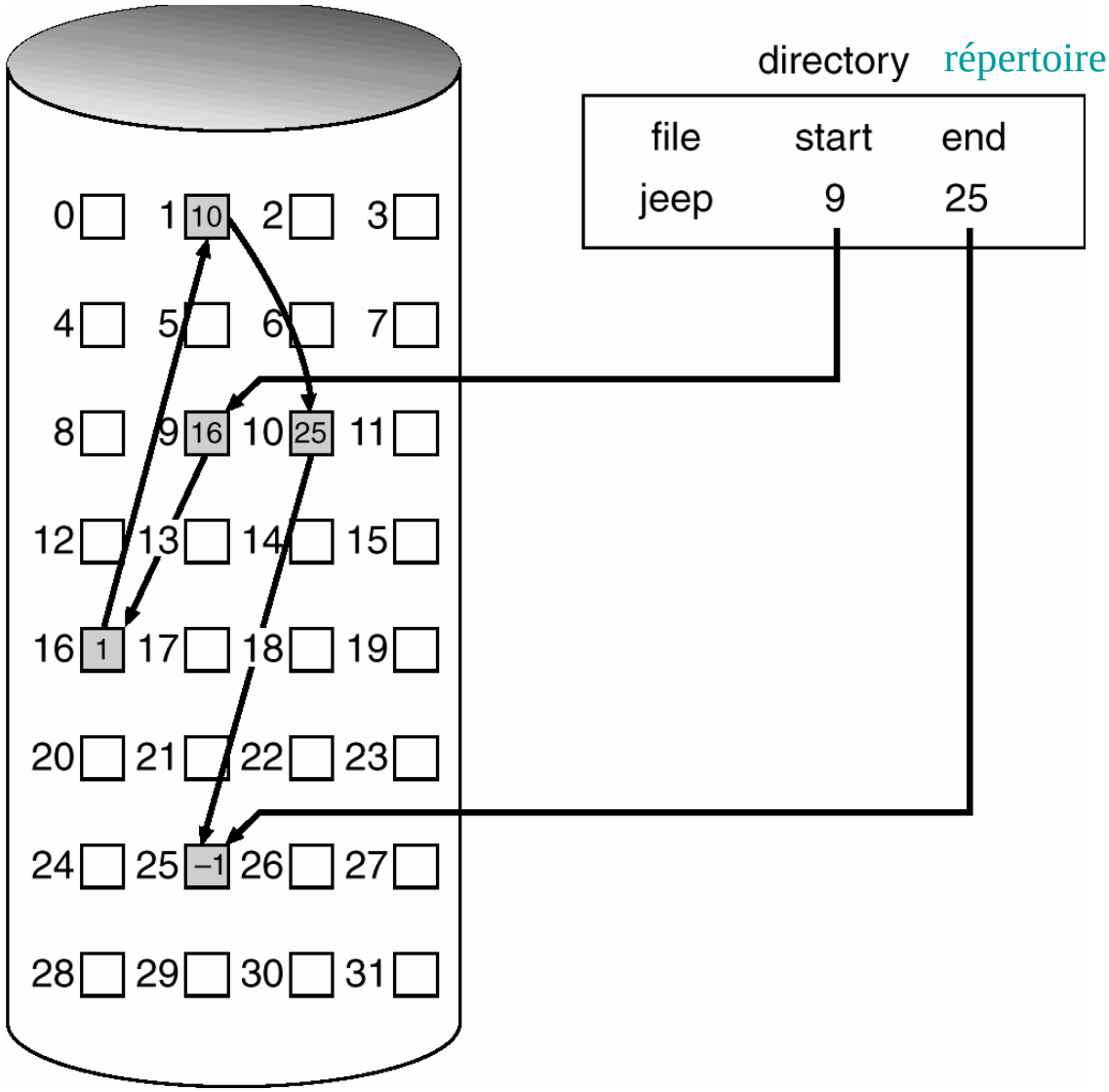
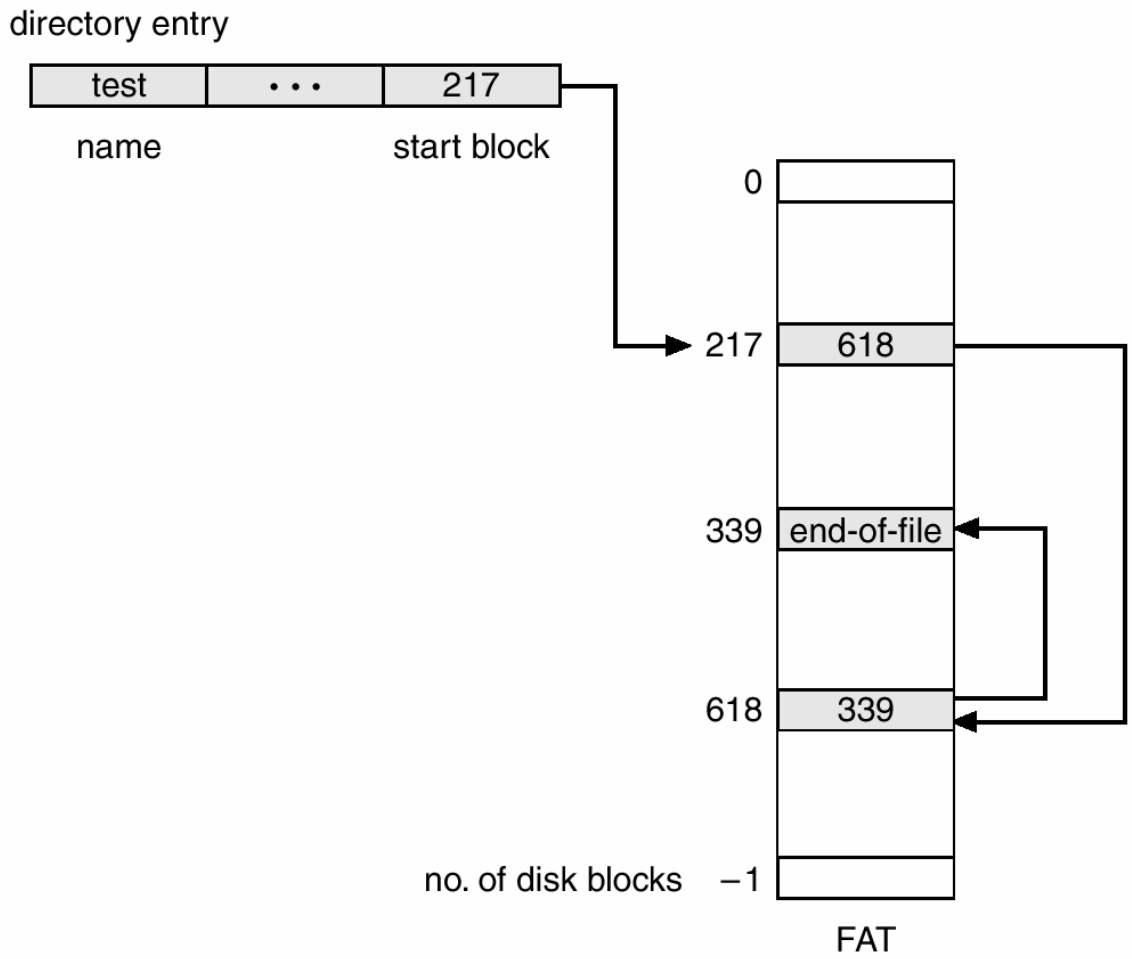


Tableau d'allocation de fichiers

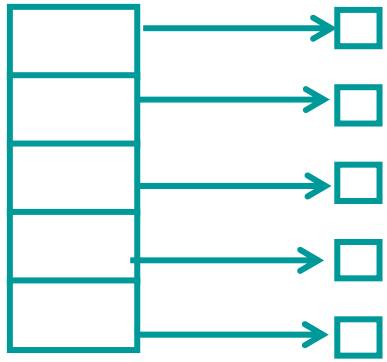


Avantages - désavantages

- **Pas de fragmentation externe - allocation de mémoire simple, pas besoin de compression**
- **L'accès à l 'intérieur d 'un fichier ne peut être que séquentiel**
  - Pas façon de trouver directement le 4ème enregistrement...
  - N'utilise pas la localité car les enregistrements seront éparpillés
- **L'intégrité des pointeurs est essentielle**
- **Les pointeurs gaspillent un peu d 'espace**

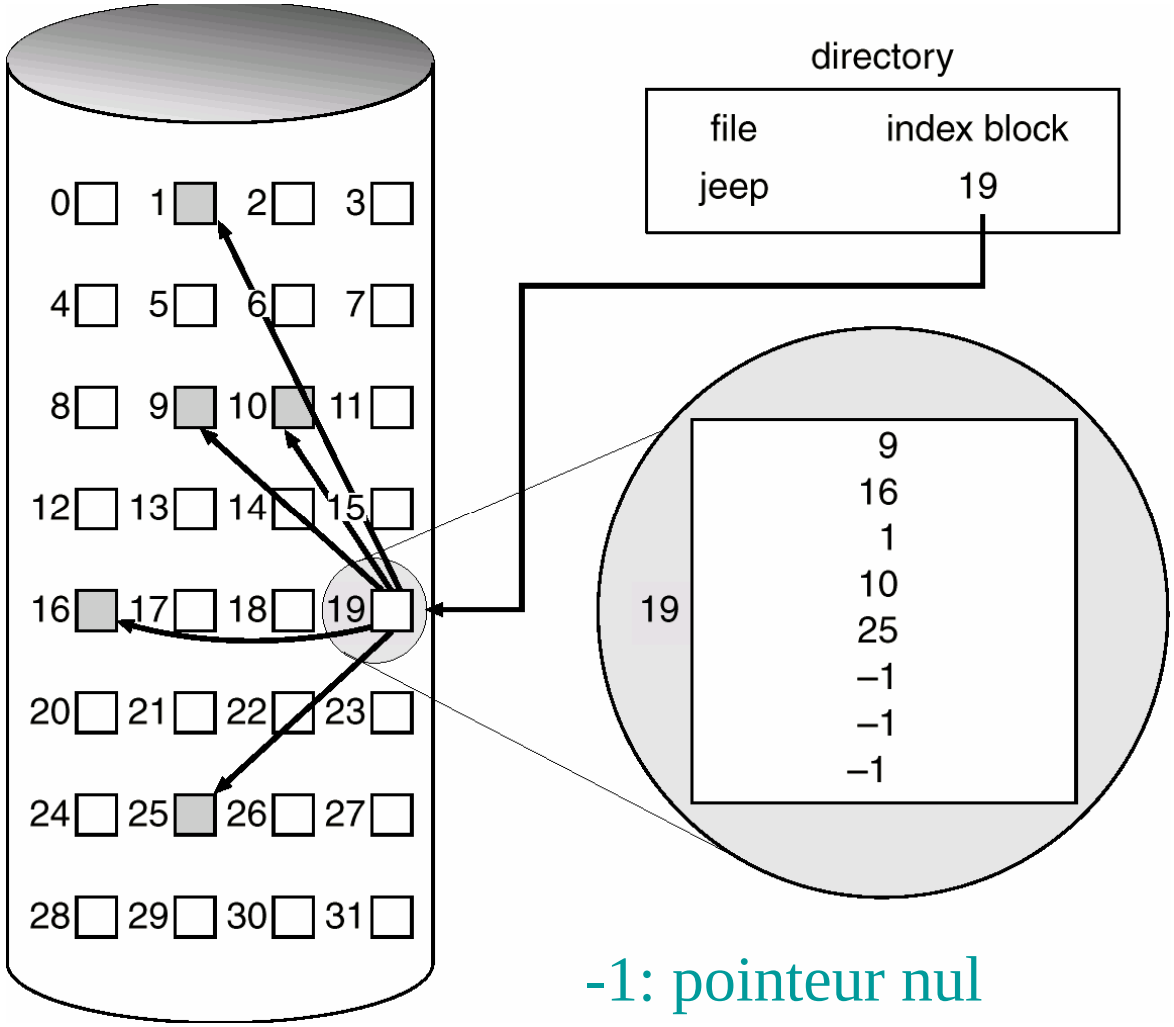
Allocation indexée: semblable à la pagination

- **Tous les pointeurs sont regroupés dans un tableau (index block)**



index table

Allocation indexée



-1: pointeur nul

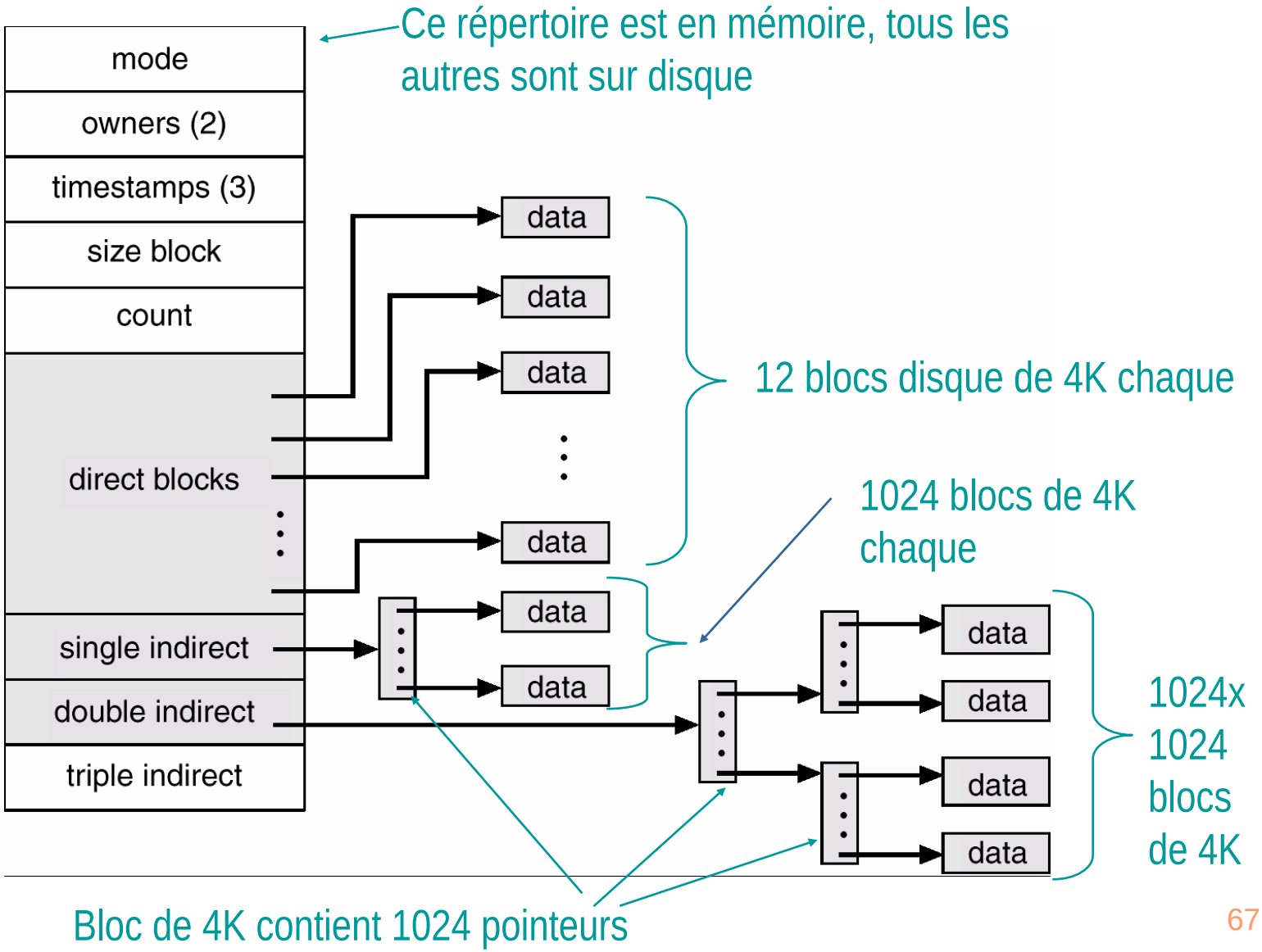
Allocation indexée

- **À la création d ’un fichier, tous les pointeurs dans le tableau sont *nil (-1)***
- **Chaque fois qu’un nouveau bloc doit être alloué, on trouve de l ’espace disponible et on ajoute un pointeur avec son adresse**

Allocation indexée

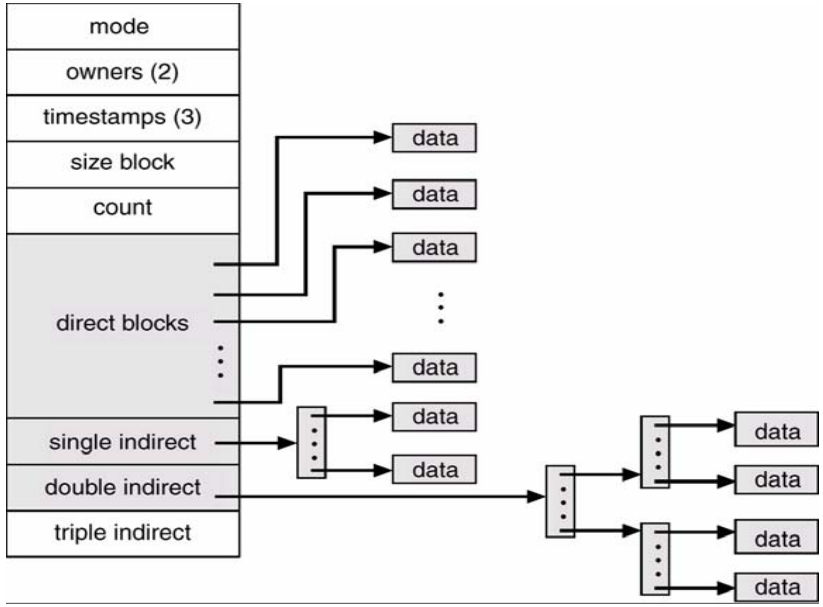
- **Pas de fragmentation externe, mais les index prennent de l'espace**
- **Permet accès direct (aléatoire)**
- **Taille de fichiers limitée par la taille de l'index block**
  - Mais nous pouvons avoir plusieurs niveaux d'index: Unix
- **Index block peut utiliser beaucoup de mémoire**

UNIX BSD: indexé à niveaux (config. possible)



UNIX BSD (v. manuel Section 20.7)

- Les premiers blocs d'un fichier sont accessibles directement
- Si le fichier contient des blocs additionnels, les premiers sont accessibles à travers un niveau d'indices
- Les suivants sont accessibles à travers 2 niveaux d'indices, etc.
- Donc le plus loin du début un enregistrement se trouve, le plus indirect est son accès
- Permet accès rapide à petits fichiers, et au début de tous les fich.
- Permet l'accès à des grands fichier avec un petit répertoire en mémoire



Gestion de l’espace libre

Gestion d'espace libre

Solution 1: vecteur de bits (solution Macintosh, Windows 2000)

- **Vecteur de bits (*n* blocs)**

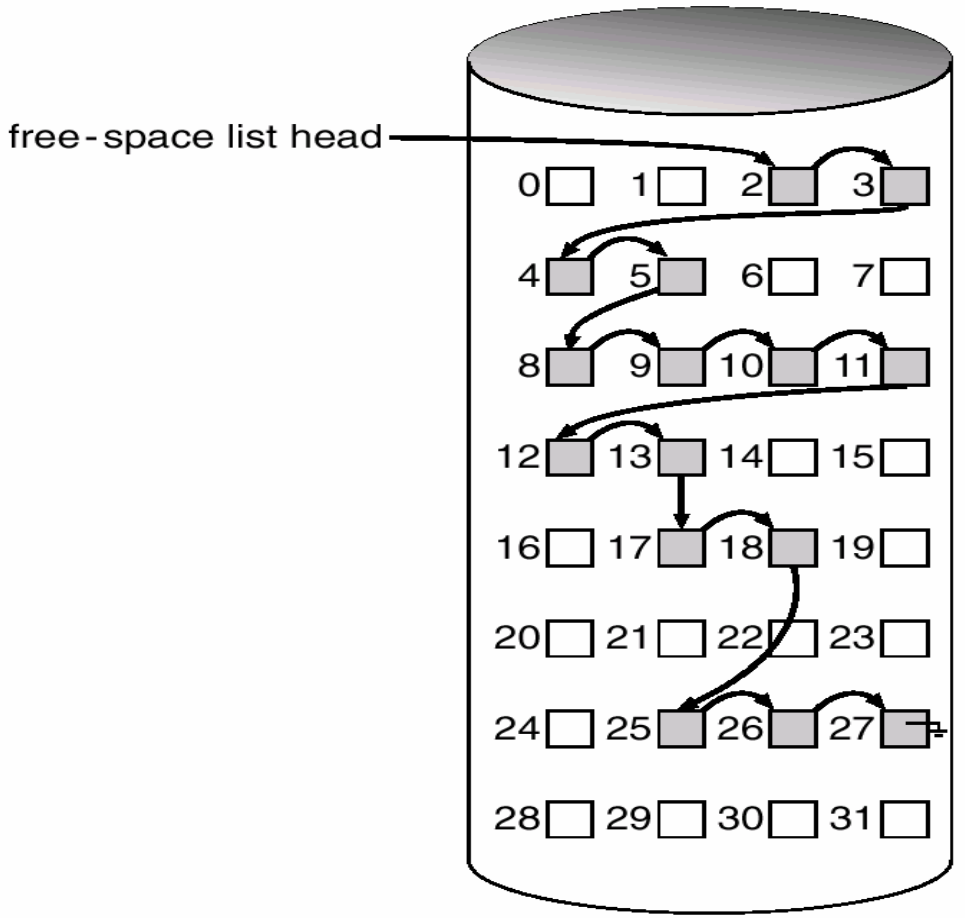
$$bit[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ libre} \\ 1 \Rightarrow \text{block}[i] \text{ occupé} \end{cases}$$

- **Exemple d'un vecteur de bits où les blocs 3, 4, 5, 9, 10, 15, 16 sont occupés:**
  - 0001110001100011…
- **L'adresse du premier bloc libre peut être trouvée par un simple calcul**

Gestion d'espace libre

Solution 2: Liste liée de mémoire libre (MS-DOS, Windows 9x)

Tous les blocs de mémoire libre sont liés ensemble par des pointeurs



Comparaison

- **Bitmap:**
  - si la bitmap de toute la mémoire secondaire est gardée en mémoire principale, la méthode est rapide mais demande de l'espace de mémoire principale
  - si les bitmaps sont gardées en mémoire secondaire, temps de lecture de mémoire secondaire...
    - Elles pourraient être paginées, p.ex.
- **Liste liée**
  - Pour trouver plusieurs blocs de mémoire libre, plus. accès de disque pourraient être demandés
  - Pour augmenter l'efficacité, nous pouvons garder en mémoire centrale l 'adresse du 1er bloc libre

Amélioration de performance

Implantation de répertoires (directories)

- **Liste linéaire de noms de fichiers avec pointeurs aux blocs de données**
  - accès séquentiel
  - simple à programmer
  - temps nécessaire pour parcourir la liste
- **Tableaux de hachage: tableaux calculés**
  - temps de recherche rapide
  - mais problèmes de collisions

Efficacité et performance

- **L ’efficacité dépend de:**
  - méthode d’allocation et d’organisation répertoires
- **Pour augmenter la performance:**
  - Rendre efficace l’accès aux blocs souvent visités
    - Dédier des tampons de mémoire qui contiennent l’image des infos plus souvent utilisées
  - Optimiser l’accès séquentiel s’il est souvent utilisé: free behind and read ahead

Disque RAM fonctionne comme mémoire cache pour le disque

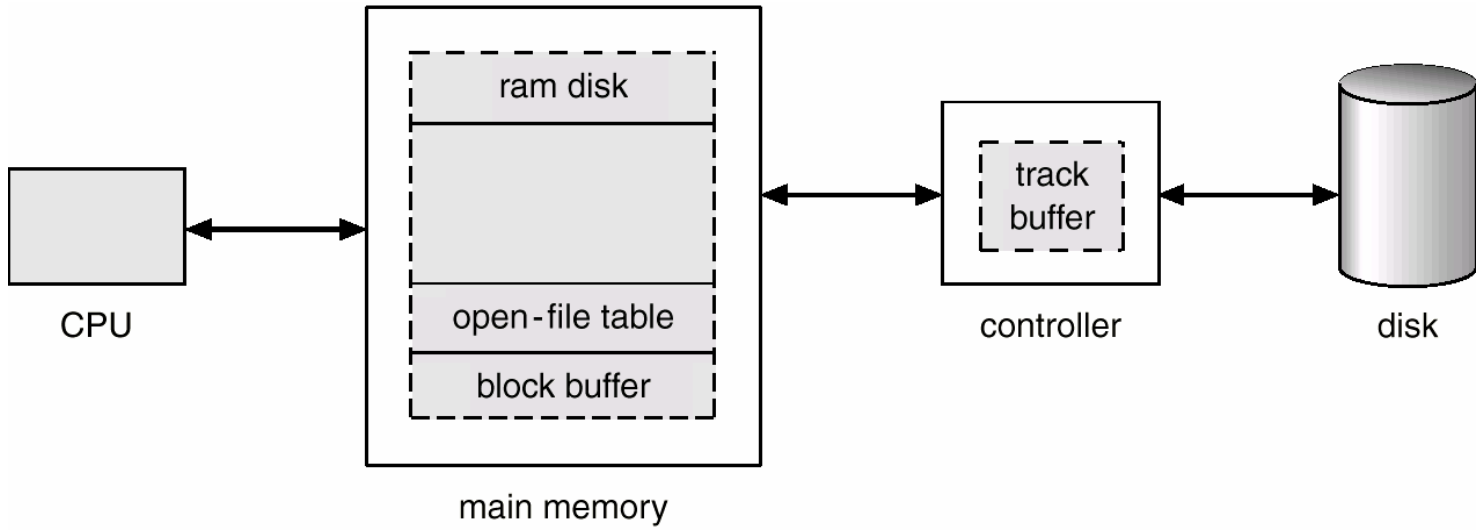


Image (partielle?) du contenu du disque en mémoire RAM

Le pilote de disque RAM accepte toutes les ops qui sont acceptées par le disque

Le problème de la récupération, ou cohérence

- **Parties des fichiers et des répertoires se trouvent tant en mémoire principale, que sur disque**
- **Il faut donc des mécanisme qui assurent leur cohérence, surtout après pannes dans le système**

Récupération: différentes méthodes

- **Contrôles de cohérence entre la structure de répertoires en mémoire centrale et le contenu des disques**
  - Essaye de réparer les incohérences
- **Sauvegarder les données sur disque dans autres supports auxiliaires (backups) (p.ex. autres disques, rubans)**
- **Restaurer les disques à partir de ces supports quand nécessaire**

Concepts importants du Chapitre 11

- **Fichiers: structures, attributs, opérations**
- **Méthodes d'accès: séquentiel, séquentiel indexée, indexée, direct ou hachée**
- **Répertoires et leur structures: repertoires arborescents, sans ou avec cycles**
- **Partage de fichiers, protection, liste d'accès**
- **Allocation d'espace: contiguë, enchaînée, indexée**
  - Application en UNIX
- **Gestion d'espace libre: bitmap, liste liée**

Par rapport au manuel…

- **Nous n’avons pas discuté:**
  - Section 11.1.6
  - Section 11.5.2
  - Section 11.6.4

Chapitre 12

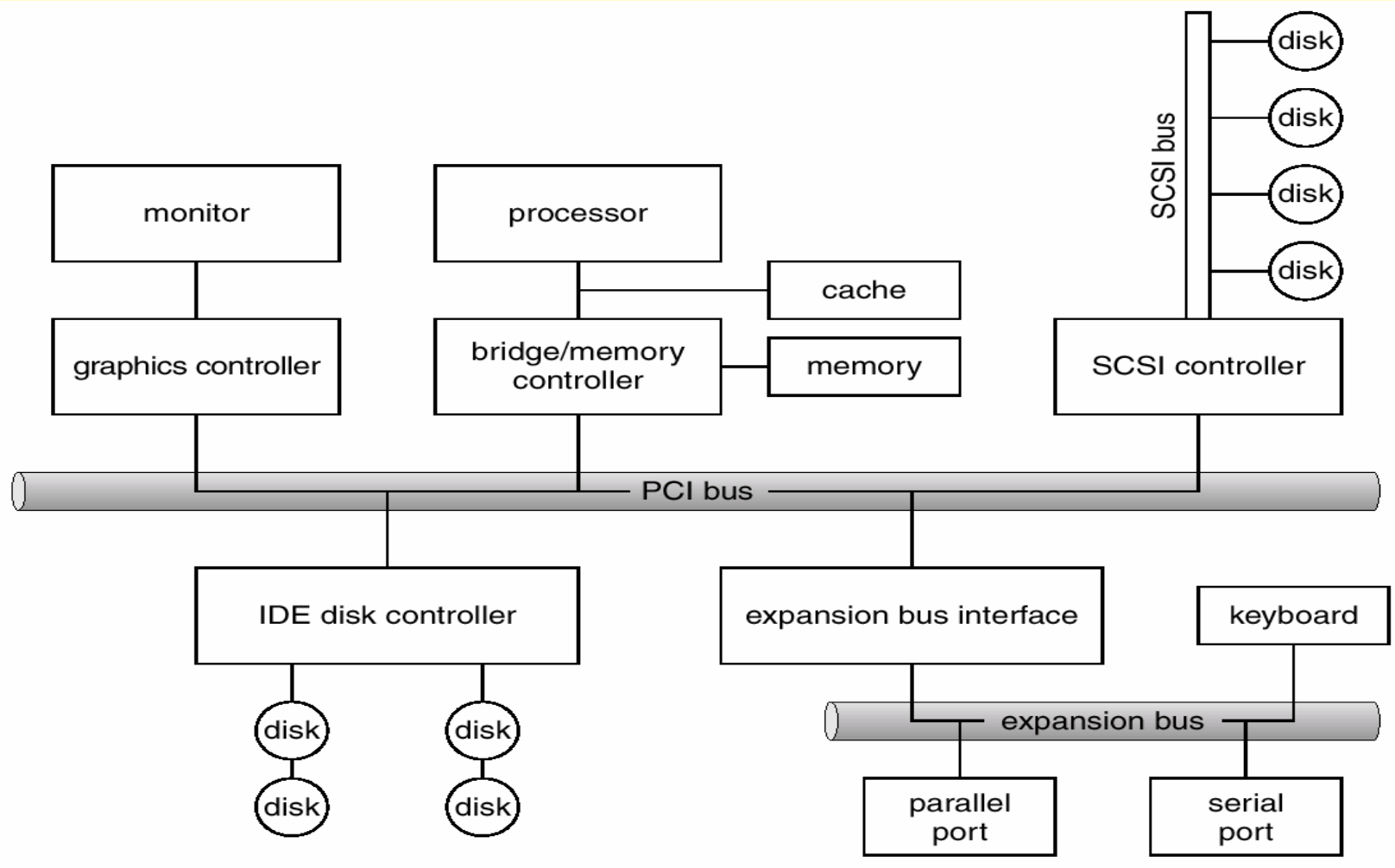
Systèmes d'entrée/sortie

http://w3.uqo.ca/luigi/

Concepts importants du chapitre

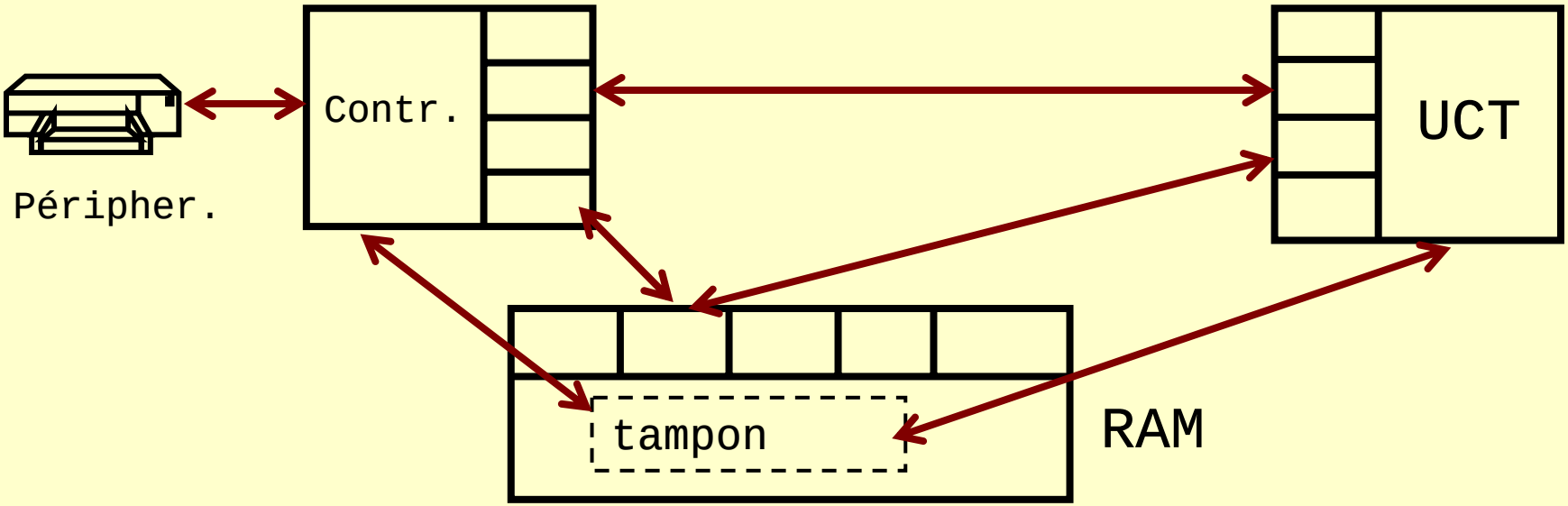
- Matériel E/S
- Communication entre UCT et contrôleurs périphériques
- Interruptions et scrutation
- DMA
- Pilotes et contrôleurs de périphs
- E/S bloquantes ou non, synchrones ou asynch
- Sous-système du noyau pour E/S
  - Tamponnage, cache, spoule

Structure typique de bus PC



Communication entre UCT et contrôleurs périphériques

- Différentes techniques:
  - UCT et contrôleurs communiquent directement par des registres
  - UCT et contrôleurs communiquent par des petites zones de mémoire centrale d'adresse fixe pour chaque contrôleur (ports)
  - Aussi, le SE utilise le RAM pour des tampons pour les données d'E/S
  - Et combinaisons de ces techniques



Adresses des ports d'E/S des périphériques PC

I/O address range (hexadecimal)	device
000-00F	DMA controller
020-021	interrupt controller
040-043	timer
200-20F	game controller
2F8-2FF	serial port (secondary)
320-32F	hard-disk controller
378-37F	parallel port
3D0-3DF	graphics controller
3F0-3F7	diskette-drive controller
3F8-3FF	serial port (primary)

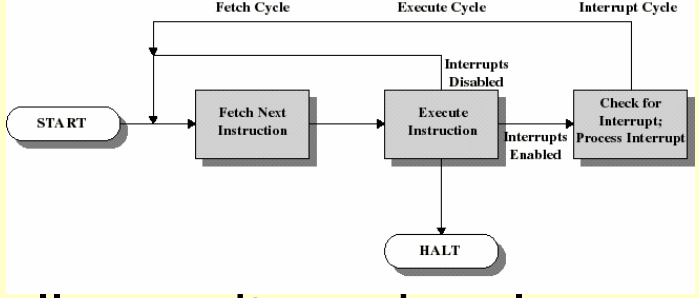
Quand communiquer entre programme et périphérique?

- Deux méthodes principales:
  - Scrutation *(polling):* l'initiative est au programme
  - Interruption: l'initiative est à la périphérique
- Grande variété d'implémentations

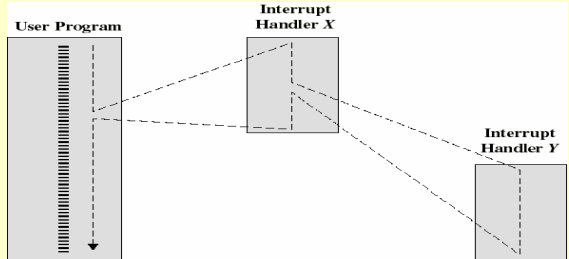
Scrutation (polling)

- Bit dans le contrôleur E/S:
  - 0 si si occupé à travailler
  - 1 si libre - prêt à accepter commande
- Bit dans l'UCT: commande prête (command ready)
  - 1 s'il y a une commande prête à exécuter par le contrôleur
- L'UCT continue de regarder le bit occupé du contrôleur jusqu'à ce qu'il soit libre (c'est la scrutation)
- Elle positionne alors le bit commande prête
- Le contrôleur voit ceci et devient occupé
- Inefficacité de cette méthode si (comme normal) l'UCT se trouve très souvent dans une boucle de scrutation
  - Attente occupée

Interruption



- L'UCT fait son travail, jusqu'à ce qu'elle reçoit un signal sur la ligne de requête d'interruption (interrupt request line)
  - Cette ligne est interrogée après l'exécution de chaque instruction
    - À moins que l'interruption ne soit masquée
    - Souvent deux lignes d'interruptions: masquable, non masquable
  - Transfert à une adresse de mémoire où il y a renvoi à la routine de traitement (dans le *vecteur d'interruption*)
  - Les interruptions ont différentes priorités
    - Si une interruption d'haute priorité se présente lors que le système est en train de traiter une int. de prio. faible, la première est traitée avant



Cycle d'E/S par interruption

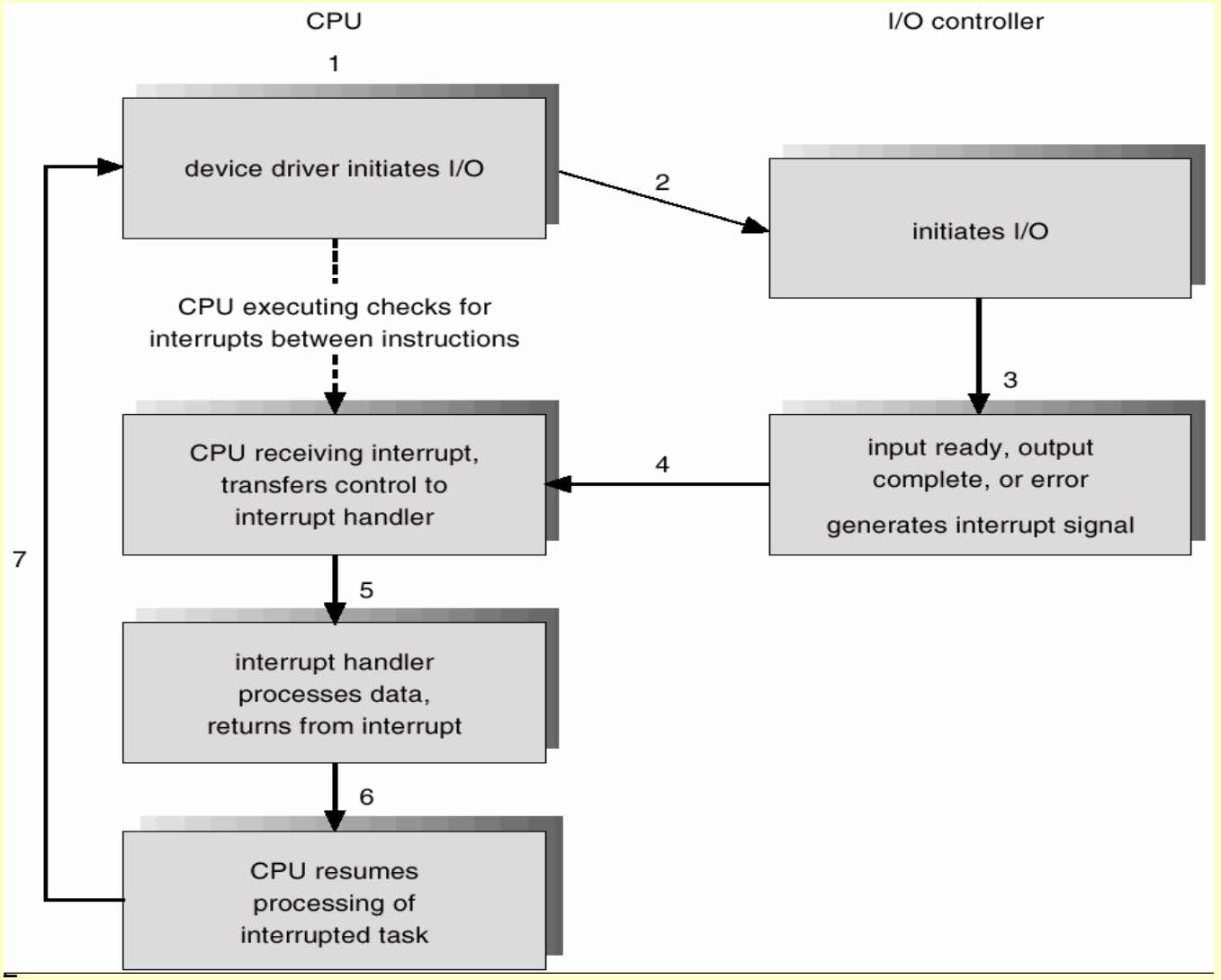


Table des vecteurs d'interruptions Intel Pentium (partie non-masquable jusqu'à 31)

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19Ð31	(Intel reserved, do not use)
32Ð255	maskable interrupts

Appels système

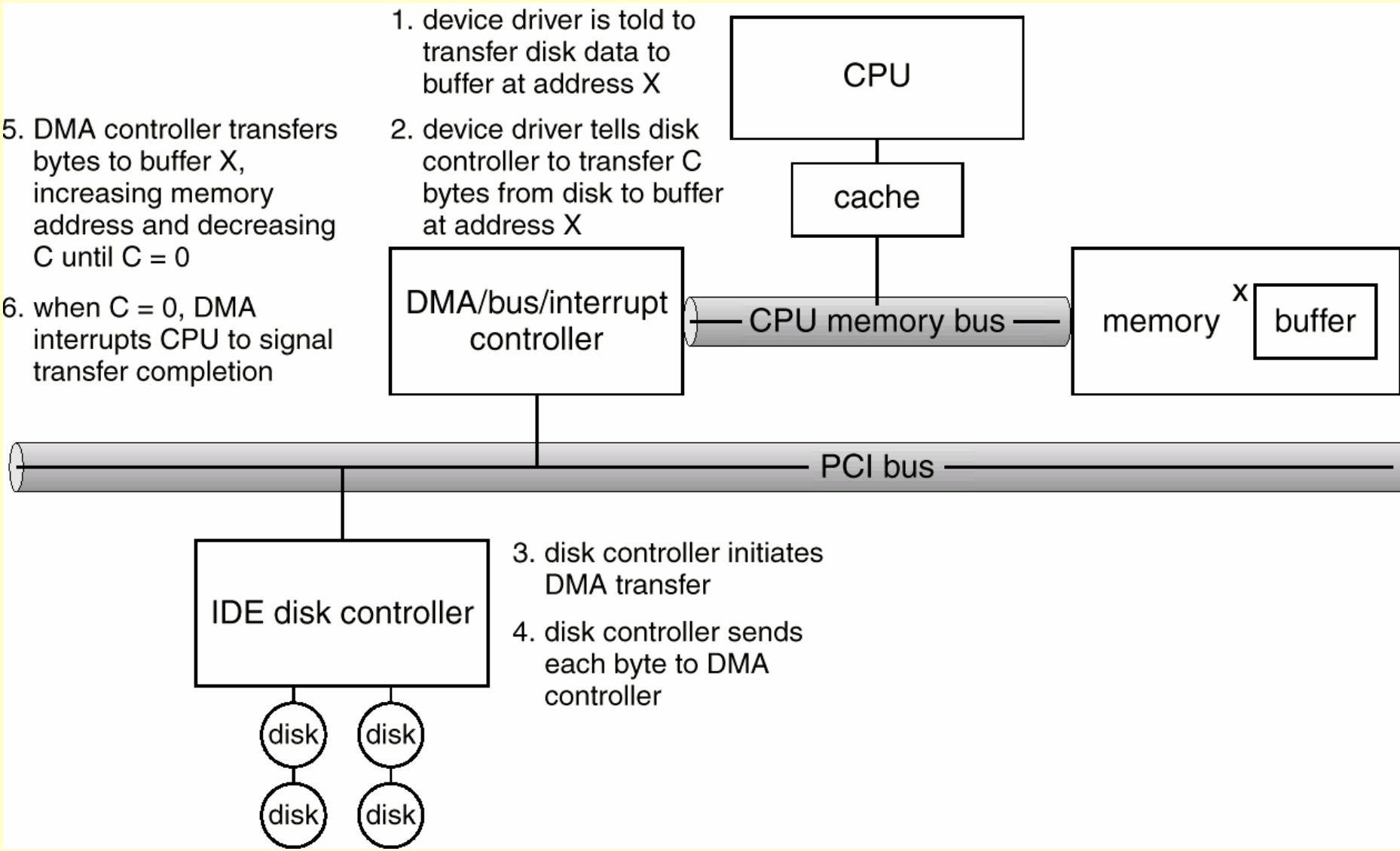
- Quand un usager demande un service du noyau, le SE exécute une instruction qui cause une *interruption logicielle ou déroutement* (trap)
- Le système passe en mode superviseur et trouve dans l'instruction les paramètres qui déterminent quoi faire
- Plus basse priorité que p.ex. E/S

Accès direct en mémoire (DMA)

- Dans les systèmes sans DMA, l'UCT est impliquée dans le transfert de chaque octet
- DMA est utile pour exclure l'implication de l'UCT surtout pour des E/S volumineuses
- Demande un contrôleur spécial
- Ce contrôleur a accès direct à la mémoire centrale

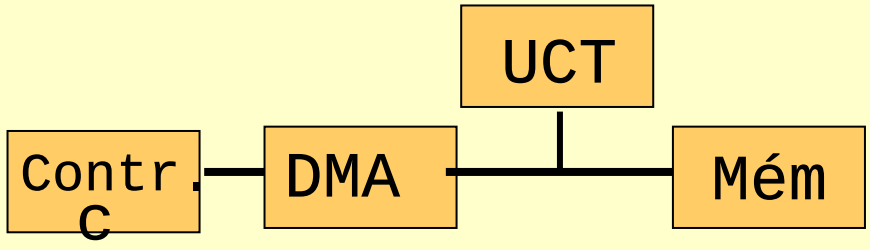
DMA: six étapes

Exemple: entrée disq→mém



Vol de cycles

- Le DMA ralentit encore le traitement d'UCT car quand le DMA utilise le bus mémoire, l'UCT ne peut pas s'en servir



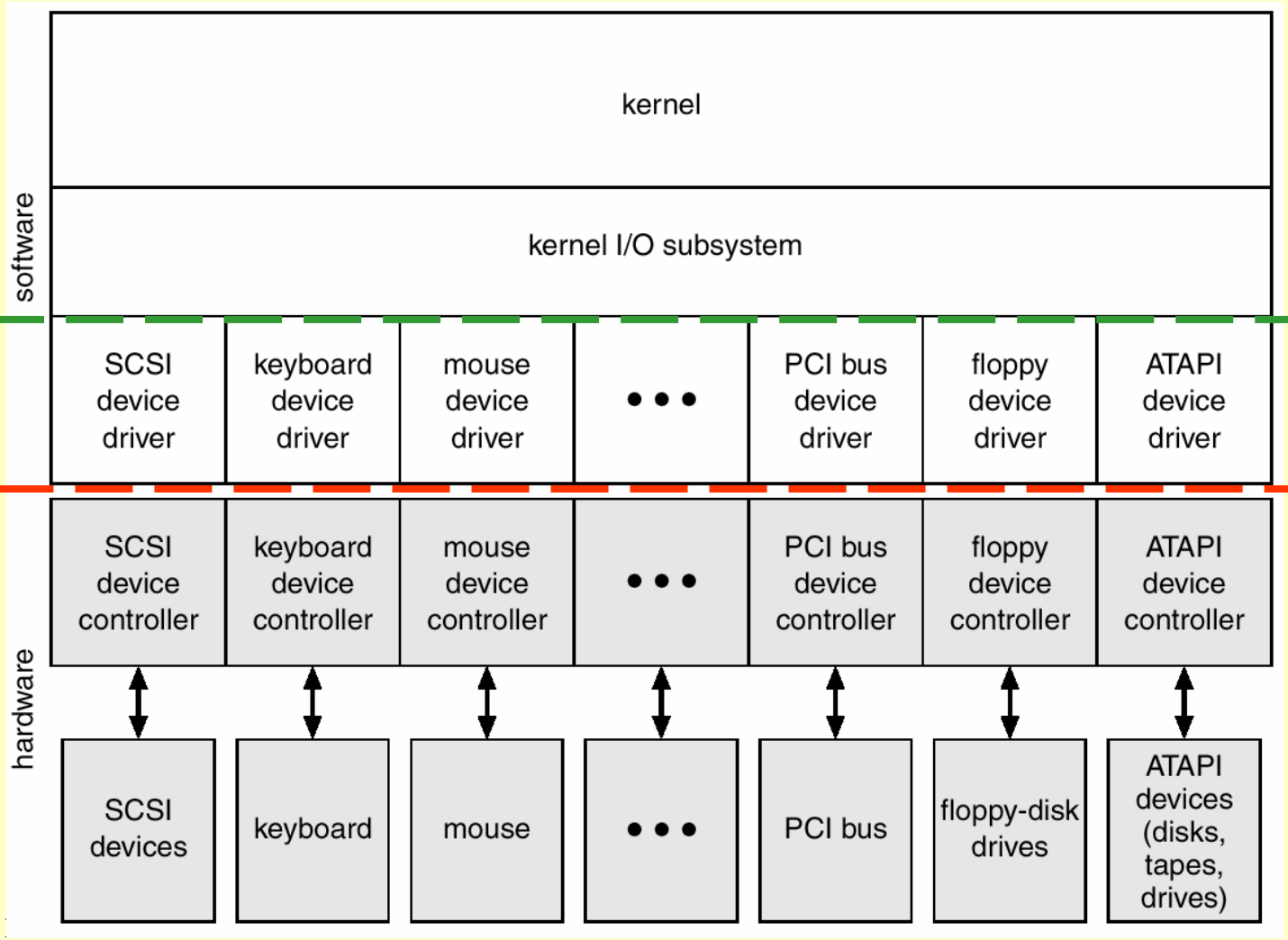
- Mais beaucoup moins que sans DMA, quand l'UCT doit s'occuper de gérer le transfert

Mémoire <-> Périphérique

Interface E/S d'application

- Le système d'E/S encapsule le comportement des périphériques dans classes génériques
- La couche *pilote* de périfs masque les différences existantes entre périphs spécifiques
- Ceci simplifie l'inter changement de différentes périphériques
- Mais le constructeur d'une périph doit créer autant de pilotes qu'il y a de SE qui peuvent utiliser la périph

Structure E/S d'un noyau (driver=pilote)



Caractéristiques des périphs E/S

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only readÐwrite	CD-ROM graphics controller disk

Périphériques par blocs ou caractères

- Périphériques par blocs: disques, rubans…
  - Commandes: read, write, seek
  - Accès brut (raw) ou à travers système fichiers
  - Accès représenté en mémoire (memory-mapped)
    - Semblable au concept de mémoire virtuelle ou cache:
      - une certaine partie du contenu de la périphérique est stocké en mémoire principale, donc quand un programme fait une lecture de disque, ceci pourrait être une lecture de mémoire principale
- Périphériques par caractère (clavier, écran)
  - Get, put traitent des caractères
  - Librairies au dessus peuvent permettre édition de lignes, etc.

Périphériques réseau

- Unix et Windows utilisent le concept de *socket*
  - Permet à l'application de faire abstraction du protocole
  - Fonctionnalité *select*: appel à *select* indique en retour quelles sont les sockets prêts à recevoir un paquet ou quelles peuvent accepter un paquet à transmettre
- Grand nombre de solutions différentes: pipes, FIFOs, streams, queues, mailboxes

Horloge et minuterie

- Fournit le temps courant, le temps écoulé, déclenche des minuteries
- Peuvent être utilisées pour interruptions périodiques, interruptions après des périodes données
  - Grand nombre d'applications

E/S bloquantes,synchrones et non-bloquantes

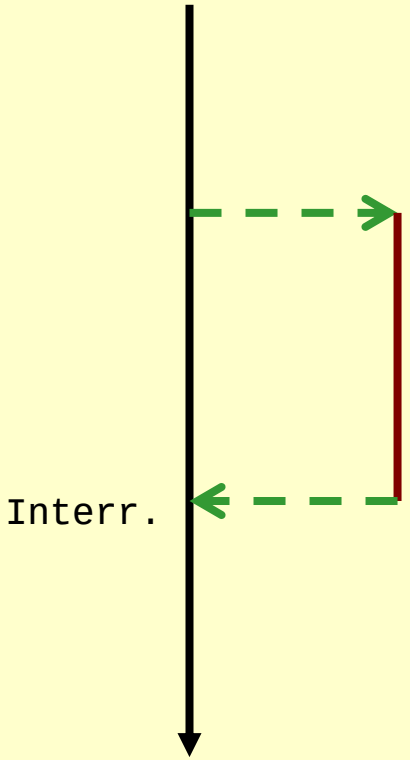
- Bloquante: le processus qui demande une E/S est suspendu jusqu'à la fin de l'E/S
  - Facile à comprendre
  - Mais le proc pourrait vouloir continuer pour faire d'autres choses urgentes

- Asynchrone: l'E/S est demandée, et le processus continue son exécution, il sera plus tard averti de la terminaison
  - Utilisée dans le passé, peu utilisée à présent car le temps de terminaison est imprévisible
  - Programmation difficile: quoi faire en attente de l'interruption?

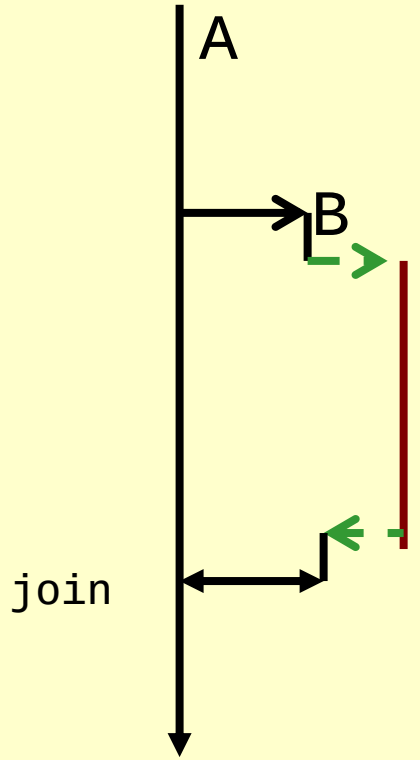
- Non-bloquante: dans ce cas, le proc peut créer des différentes tâches qui lui permettent de continuer avec autres activités
  - Les différentes tâches se réunissent à un point programmé par les deux

E/S synchrones, asynch et non-bloquantes

Synchrone:proc suspendu pendant E/S



Asynch:proc continue, sera interrompu quand E/S complétée



Non-bloquante:proc A crée proc B pour faire une E/S synchrone, continue pour faire autre chose puis les deux se réunissent

Sous-système E/S du noyau

- Fonctionnalités:
  - Ordonnancement E/S
  - Mise en tampon
  - Mise en cache
  - Mise en attente et réservation de périphérique
  - Gestion des erreurs

Sous-système E/S du noyau
Ordonnancement E/S

- Optimiser l'ordre dans lequel les E/S sont exécutées
  - Chapitre sur ordonnancement disque

Sous-système E/S du noyau

Mise en tampon

- Double tamponnage:
  - P.ex. en sortie: un processus écrit le prochain enregistrement sur un tampon en mémoire tant que l'enregistrement précédent est en train d'être écrit
  - Permet superposition traitement/E/S

Sous-système E/S du noyau
Mise en cache

- Quelques éléments couramment utilisés d'une mémoire secondaire sont gardés en mémoire centrale
- Donc quand un processus exécute une E/S, celle-ci pourrait ne pas être une E/S réelle:
  - Elle pourrait être un transfert en mémoire, une simple mise à jour d'un pointeur, etc.
    - V.disque RAM Chap. 11

Sous-système E/S du noyau
Mise en attente et réservation de périphérique: spoule

- Spoule ou Spooling est un mécanisme par lequel des travaux à faire sont stockés dans un fichier, pour être ordonnancés plus tard
- Pour optimiser l'utilisation des périphériques lentes, le SE pourrait diriger à un stockage temporaire les données destinés à la périphérique (ou provenant d'elle)
  - P.ex. chaque fois qu'un programmeur fait une impression, les données pourraient au lieu être envoyées à un disque, pour être imprimées dans leur ordre de priorité
  - Aussi les données en provenance d'un lecteur optique pourraient être stockées pour traitement plus tard

Sous-système E/S du noyau
Gestion des erreurs

- Exemples d'erreurs à être traités par le SE:
  - Erreurs de lecture/écriture, protection, périph non-disponible
- Les erreurs retournent un code 'raison'
- Traitement différent dans les différents cas…

Structures de données du noyau

- Le noyau garde toutes les informations concernant les fichiers ouverts, composants E/S, connections

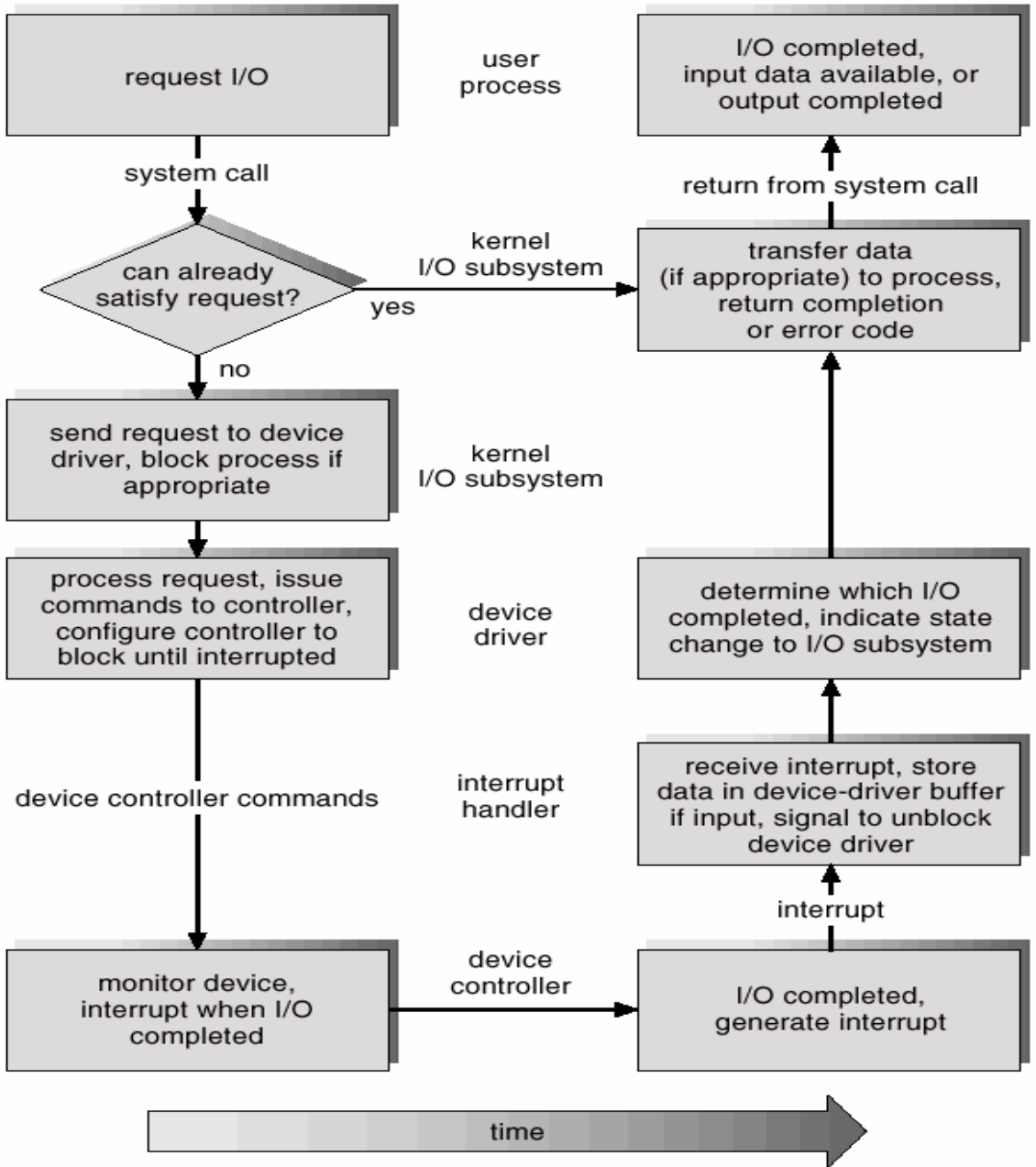
- Un grand nombre de structures de données complexes pour garder l'information sur les tampons, l'allocation de mémoire, etc.

Gestion de requêtes E/S

- P. ex. lecture d'un fichier de disque
  - Déterminer où se trouve le fichier
  - Traduire le nom du fichier en nom de périphérique et location dans périphérique
  - Lire physiquement le fichier dans le tampon
  - Rendre les données disponibles au processus
  - Retourner au processus

Cycle de vie d'une requête E/S

Si données déjà en RAM (cache)



Performance

- Ignorer cette section

Concepts importants du chapitre

- Matériel E/S
- Communication entre UCT et contrôleurs périfériques
- Interruptions et scrutation
- DMA
- Pilotes et contrôleurs de périfs
- E/S bloquantes ou non, asynchrones
- Sous-système du noyau pour E/S
  - Tamponnage, cache, spoule

Concept de Cache

- Le mot *cache* est utilisé souvent dans situations apparemment différentes, p.ex:
  - Entre l'UCT et la mémoire RAM nous pouvons avoir une mémoire cache
  - Le Translation Lookaside Buffer (TLB) utilisé pour contenir les adresses de mémoire virtuelle les plus récemment utilisés est souvent appelé cache
  - La technique de charger en RAM certains données disque qui sont beaucoup utilisées est aussi appelée 'caching'
- En général, caching veut dire transférer dans une mémoire plus rapide des informations normalement contenues dans une mémoire plus lente, quand le SE croit qu'elles seront demandées bientôt

Chapitre 13 (Seul. 13.1, 13.2, 13.4)

Structure de mémoire de masse (disques)

http://w3.uqo.ca/luigi/

Concepts importants du chapitre 13

- **Fonctionnement et structure des unités disque**
- **Calcul du temps d'exécution d'une séquence d'opérations**
- **Différents algorithmes d'ordonnancement**
  - Fonctionnement, rendement
- **Gestion de l'espace de permutation**
  - Unix

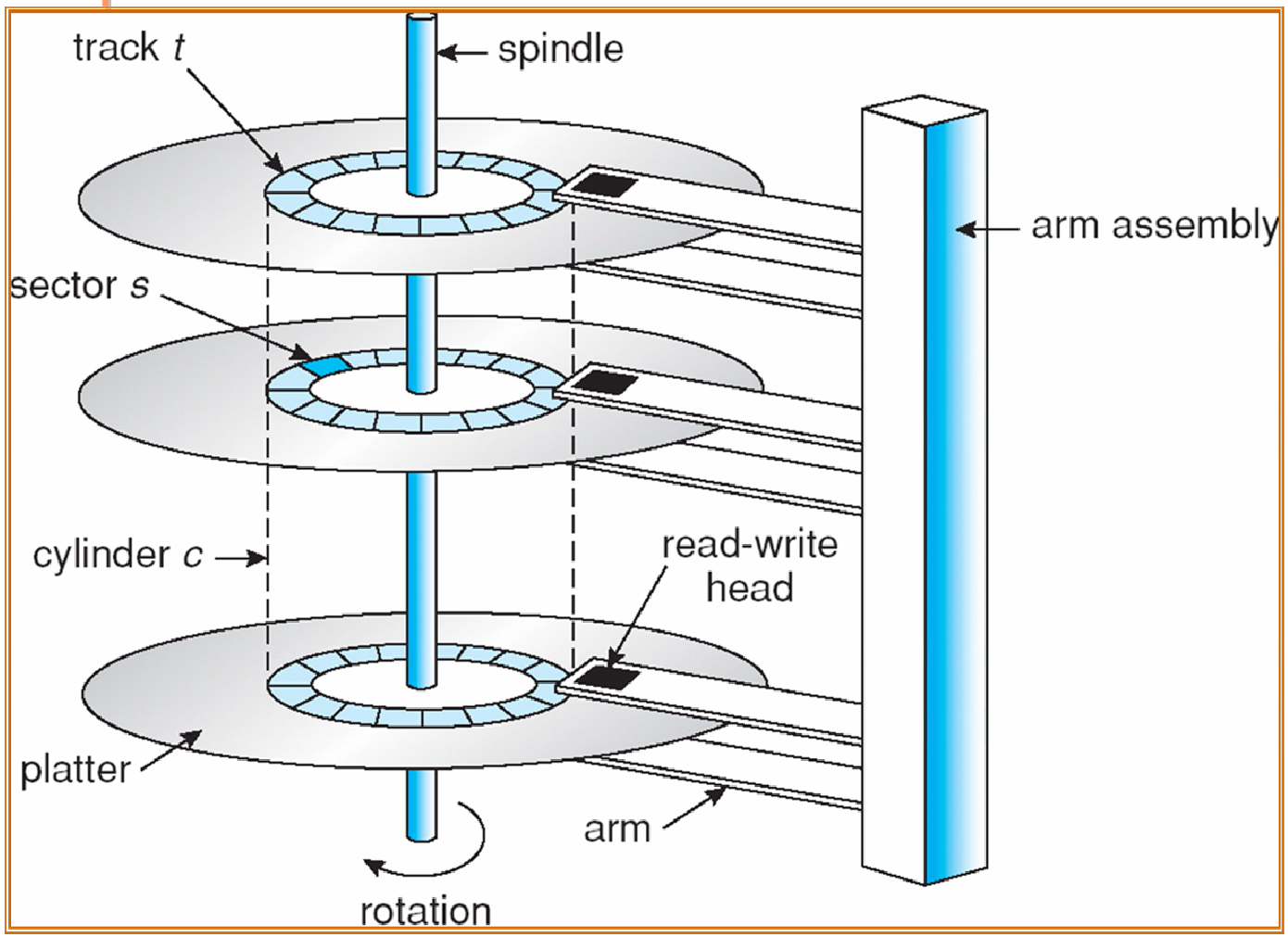
Disques magnétiques

- **Plats rigides couverts de matériaux d ’enregistrement magnétique**
  - surface du disque divisée en pistes (tracks) qui sont divisées en secteurs
  - le contrôleur disque détermine l`interaction logique entre l ’unité et l ’ordinateur
  - Beaucoup d’infos utiles dans:

http://www.storagereview.com/guide2000/ref/hdd/index.html

Nomenclature -

cylindre: l'ensemble de pistes qui se trouvent dans la même position du bras de lecture/écriture



Un disque à plusieurs surfaces

http://computer.howstuffworks.com

Disques éléctroniques

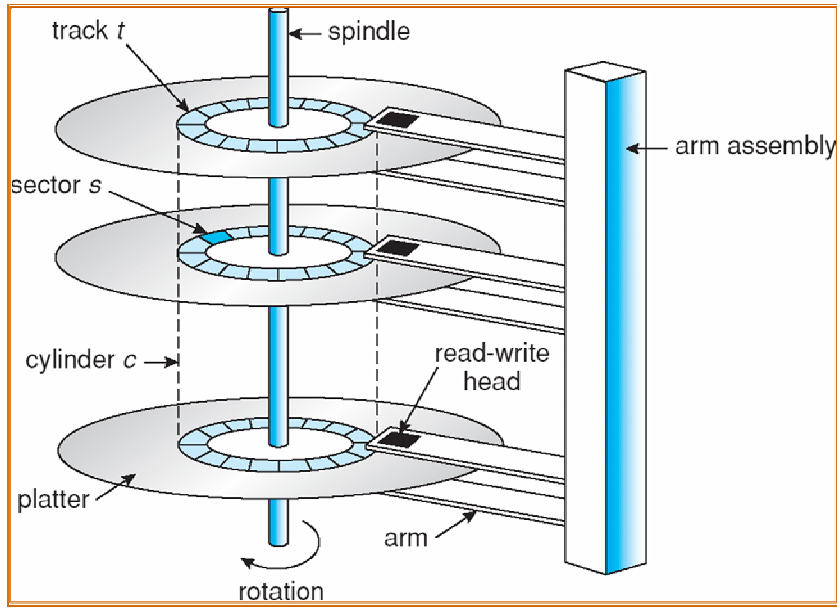
- **Aujourd'hui nous trouvons de plus en plus des types de mémoires qui sont adressées comme si elle étaient des disques, mais sont complètement électroniques**
  - P. ex. flash memory
  - Il n'y aura pas les temps de positionnement, latence, etc. discutés plus tard

Ordonnancement disques

- **Problème: utilisation optimale du matériel**
- **Réduction du temps total de lecture disque**
  - étant donné une file de requêtes de lecture disque, dans quel ordre les exécuter?

Paramètres à prendre en considération

- **Temps de positionnement (seek time):**
  - le temps pris par l`unité disque pour se positionner sur le cylindre désiré
- **Temps de latence de rotation**
  - le temps pris par l 'unité de disque qui est sur le bon cylindre pour se positionner sur le secteur désirée
- **Temps de lecture**
  - temps nécessaire pour lire la piste
- **Le temps de positionnement est normalement le plus important, donc il est celui que nous chercherons à minimiser**

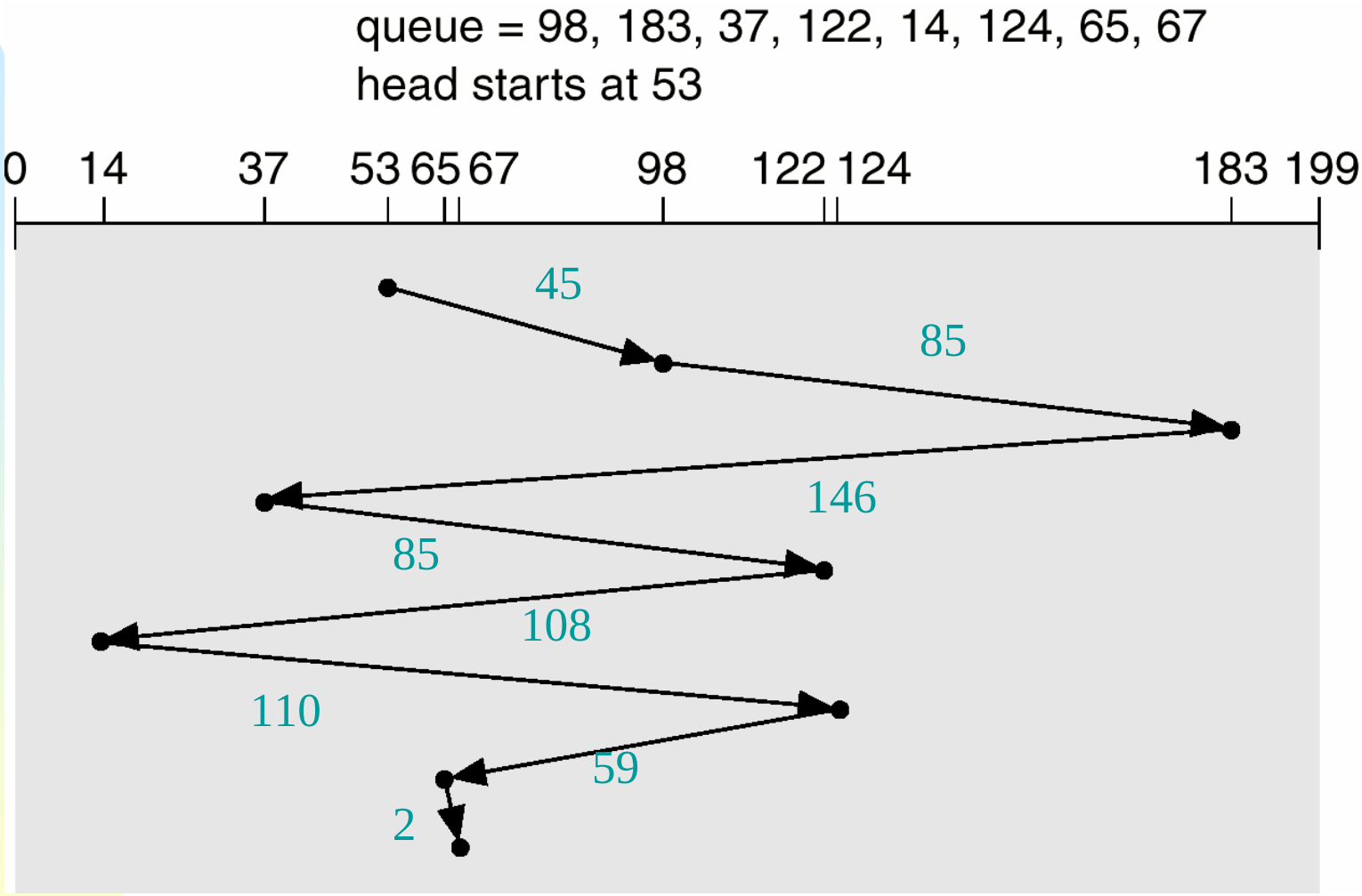


File d’attente disque

- Dans un système multiprogrammé avec mémoire virtuelle, il y aura normalement une file d’attente pour l ’unité disque
- Dans quel ordre choisir les requêtes d ’opérations disques de façon à minimiser les temps de recherche totaux
- Nous étudierons différents méthodes par rapport à une file d ’attente arbitraire:

  98, 183, 37, 122, 14, 124, 65, 67
- Chaque chiffre est un numéro séquentiel de cylindre
- Il faut aussi prendre en considération le cylindre de départ: 53
- Dans quel ordre exécuter les requêtes de lecture de façon à minimiser les temps totaux de positionnement cylindre
- Hypothèse simpliste: un déplacement d`1 cylindre coûte 1 unité de temps

Premier entré, premier sorti: FIFO



axe de rotation

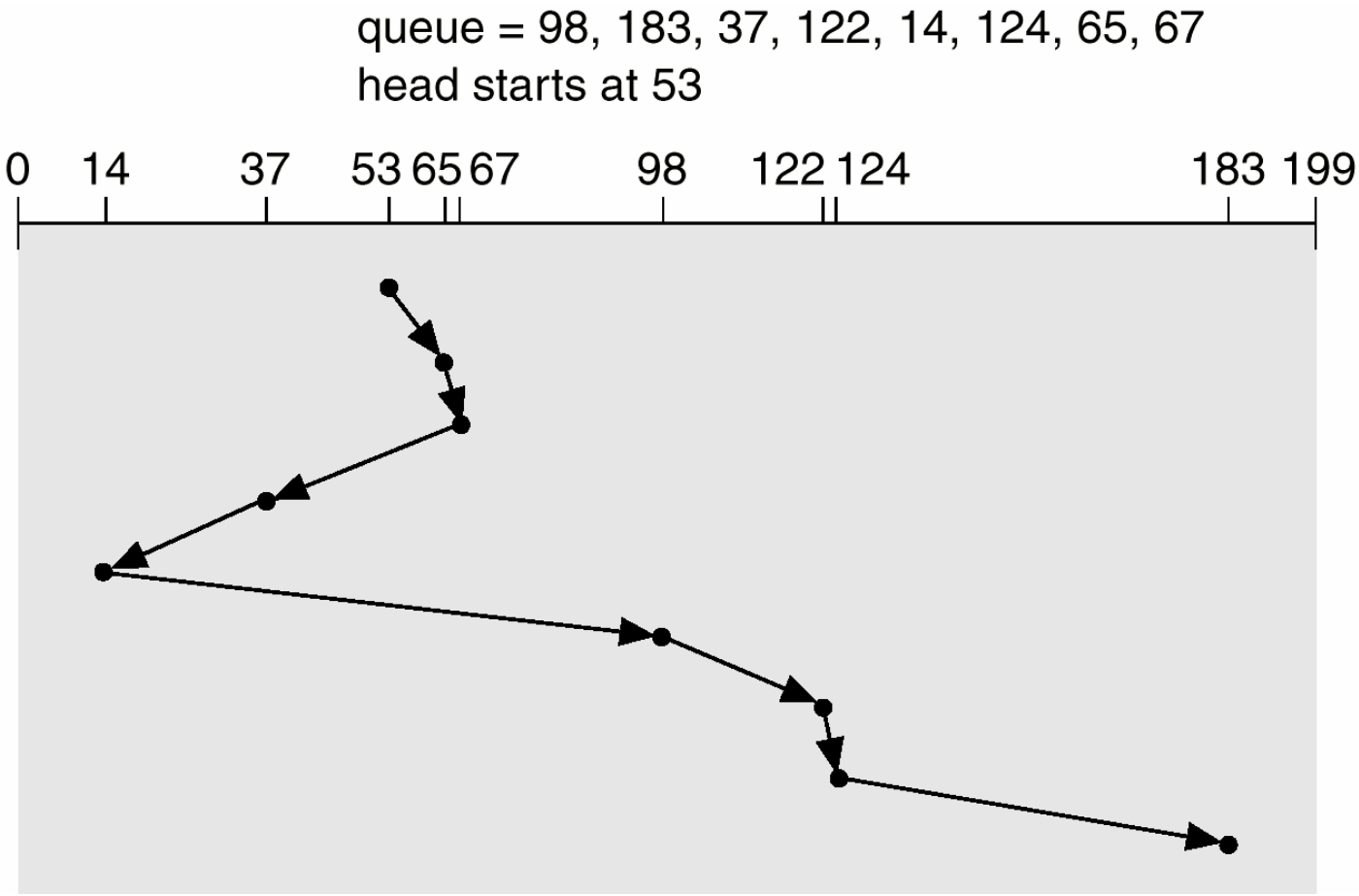
Mouvement total: 640 cylindres = (98-53) + (183-98)+...

En moyenne: 640/8 = 80

SSTF: Shortest Seek Time First
Plus court d'abord

- **À chaque moment, choisir la requête avec le temps de recherche le plus court à partir du cylindre courant**
- **Clairement meilleur que le précédent**
- **Mais pas nécessairement optimal! (v. manuel)**
- **Peut causer famine**

SSTF: Plus court servi



Mouvement total: 236 cylindres (680 pour le précédent)

En moyenne: 236/8 = 29.5 (80 pour le précédent)

Problèmes commun à tous les algorithmes qui sélectionnent toujours le plus voisin:

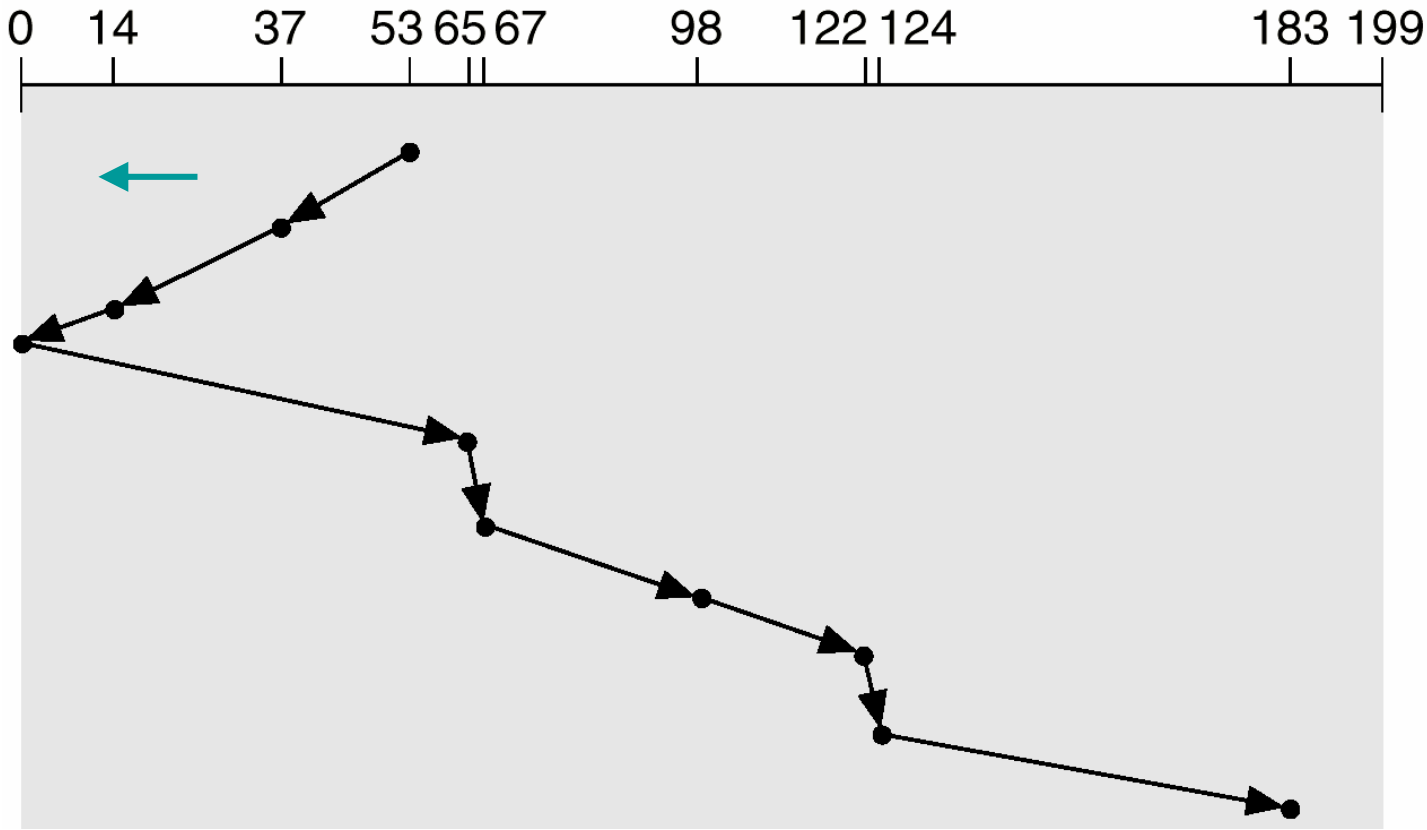
- **Balaient efficacement un voisinage, puis quand ils ont fini là dedans doivent faire des déplacement plus importants pour traiter ce qui reste**
- **Famine pour les autres s'il y a apport continu d'éléments dans le voisinage**

SCAN: l'algorithme de l'ascenseur

- **La tête balaye le disque dans une direction, puis dans la direction opposée, etc., en desservant les requêtes quand il passe sur le cylindre désiré**
  - Pas de famine

SCAN: l 'ascenseur

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53 direction ←



Mouvement total: 208 cylindres
En moyenne: 208/8 = 26 (29.5 pour SSTF)

Problèmes du SCAN

- **Peu de travail à faire après le renversement de direction**
- **Les requêtes seront plus denses à l'autre extrémité**
- **Arrive inutilement jusqu'à 0**

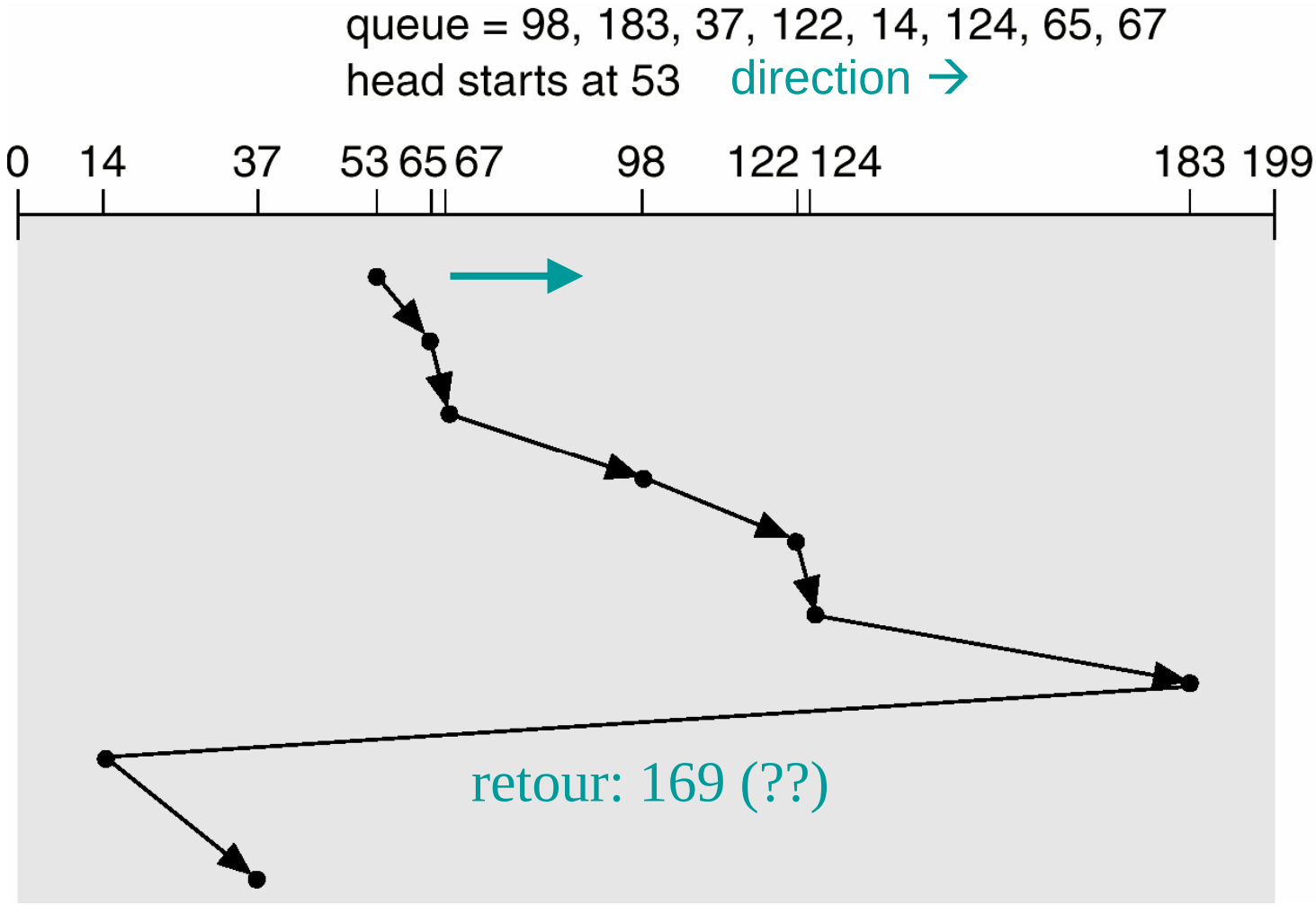
C-SCAN

- **Retour rapide au début (cylindre 0) du disque au lieu de renverser la direction**
- **Hypothèse: le mécanisme de retour est beaucoup plus rapide que le temps de visiter les cylindres**
  - Comme si les disques étaient en forme de beignes!

C-LOOK

- **La même idée, mais au lieu de retourner au cylindre 0, retourner au premier cylindre qui a une requête**

C-LOOK

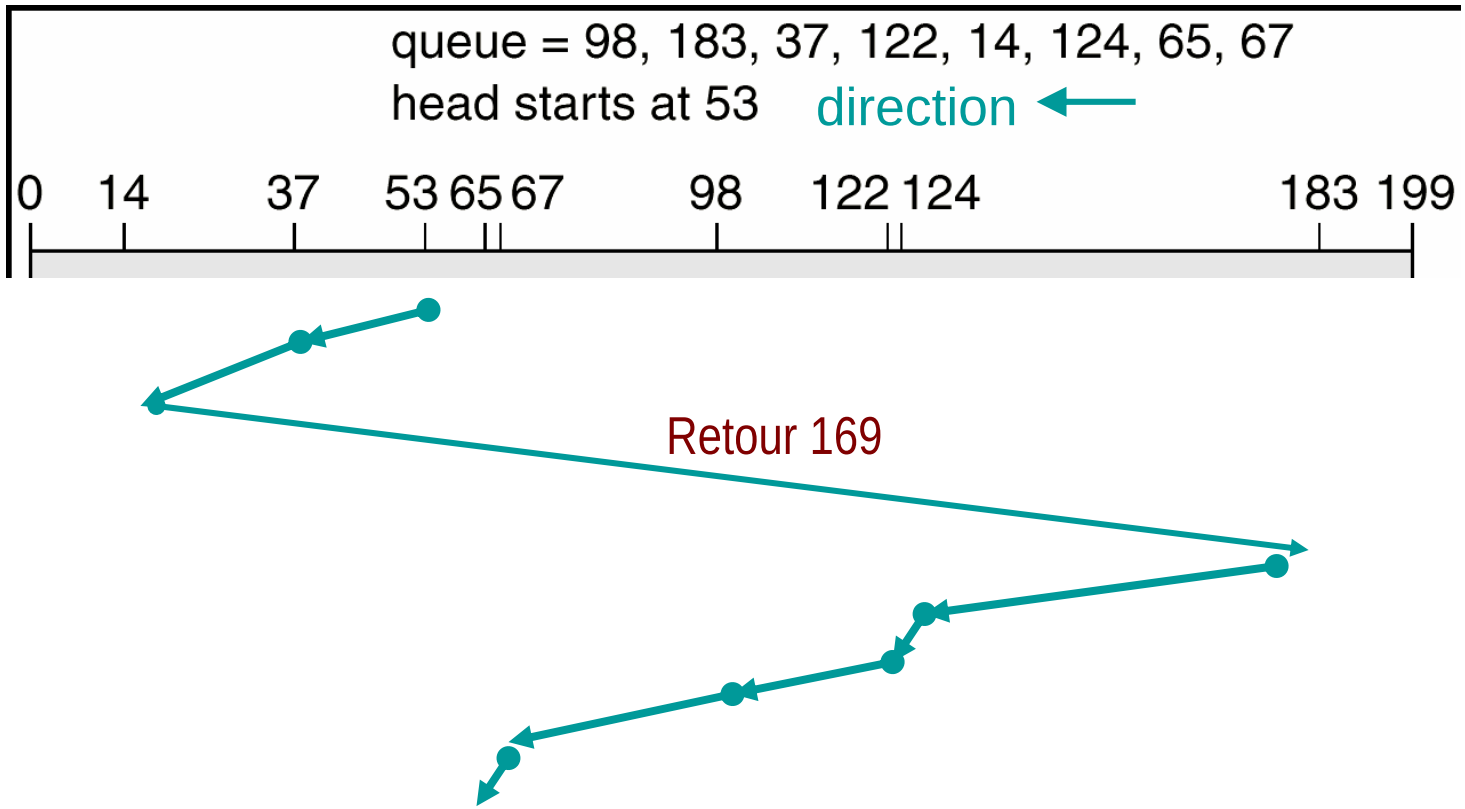


153 sans considérer le retour (19.1 en moyenne) (26 pour SCAN)

MAIS 322 avec retour (40.25 en moyenne)

Normalement le retour sera rapide donc le coût réel sera entre les deux

C-LOOK avec direction initiale opposée



Résultats très semblables:
157 sans considérer le retour, 326 avec le retour

Exemple pratique…

- **Si on doit ramasser des gens de Gatineau Est à Aylmer, il pourrait être plus rapide de faire le tour à Gatineau, puis prendre l'autoroute jusqu'à Aylmer et ramasser le reste, au lieu d'arriver à Aylmer par rues régulières tout en ramassant des gens…**

Comparaison

- **Si la file souvent ne contient que très peu d'éléments, l'algorithme du 'premier servi ' devrait être préféré (simplicité)**
- **Sinon, SSTF ou SCAN ou C-SCAN?**
- **En pratique, il faut prendre en considération:**
  - Les temps réels de déplacement et retour au début
  - L`organisation des fichiers et des répertoires
    - Les répertoires sont sur disque aussi…
  - La longueur moyenne de la file
  - Le débit d 'arrivée des requêtes

Gestion de l'espace de permutation en mémoire virtuelle (swap space) (13.4)

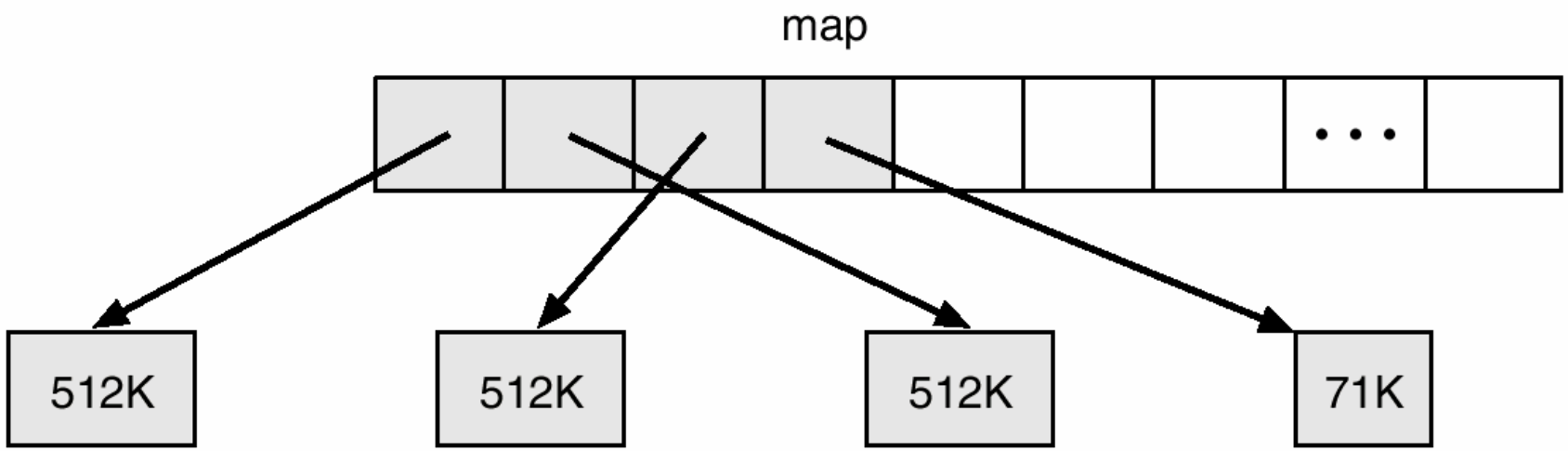
- **Nous avons vu comment les systèmes de mém virtuelle utilisent la mém secondaire**
- **Grande variété d'implémentations de systèmes d'espace de permutation dans différents SE**
- **L'esp permutation (swap)**
  - peut être des fichiers normaux dans l'espace disque utilisé par les autres fichiers,
  - ou peut avoir sa propre partition disque
- **Peut être mis dans des disques plus efficaces**

Gestion d'espace de permutation (disque) en Unix 4.3BSD

- **Pour chaque processus, il y a**
  - Un segment texte = le programme
    - Ne change pas pendant exécution
  - Et il y a aussi un segment données
    - Sa taille peut changer pendant exéc

Unix 4.3BSD: Tableau d'allocation du segment de texte=programme

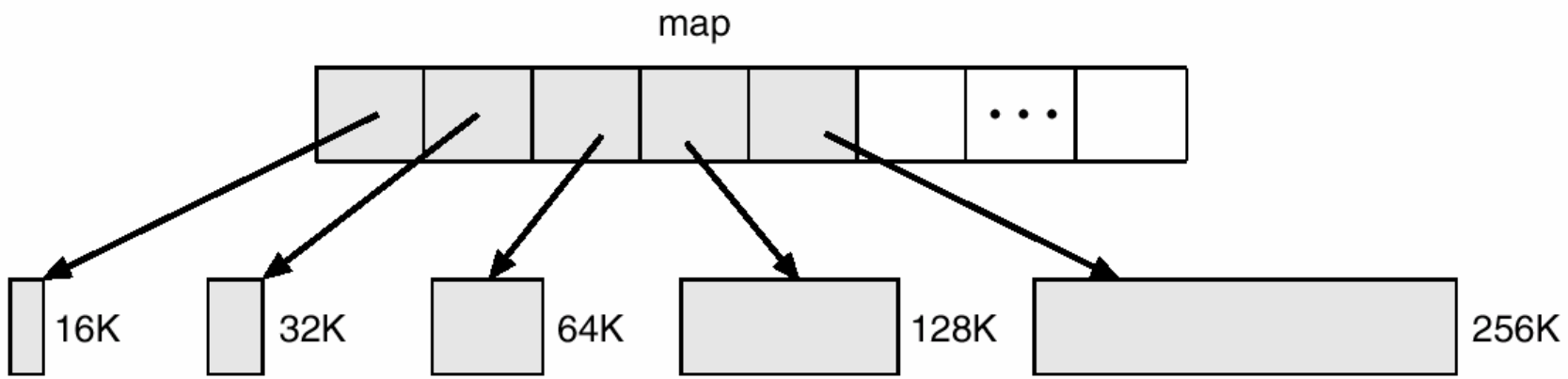
- **L'espace disque est alloué en morceaux fixes de 512K**



Dernier morceau de programme plus court

Unix 4.3BSD: Tableau d'allocation du segment données sur disque

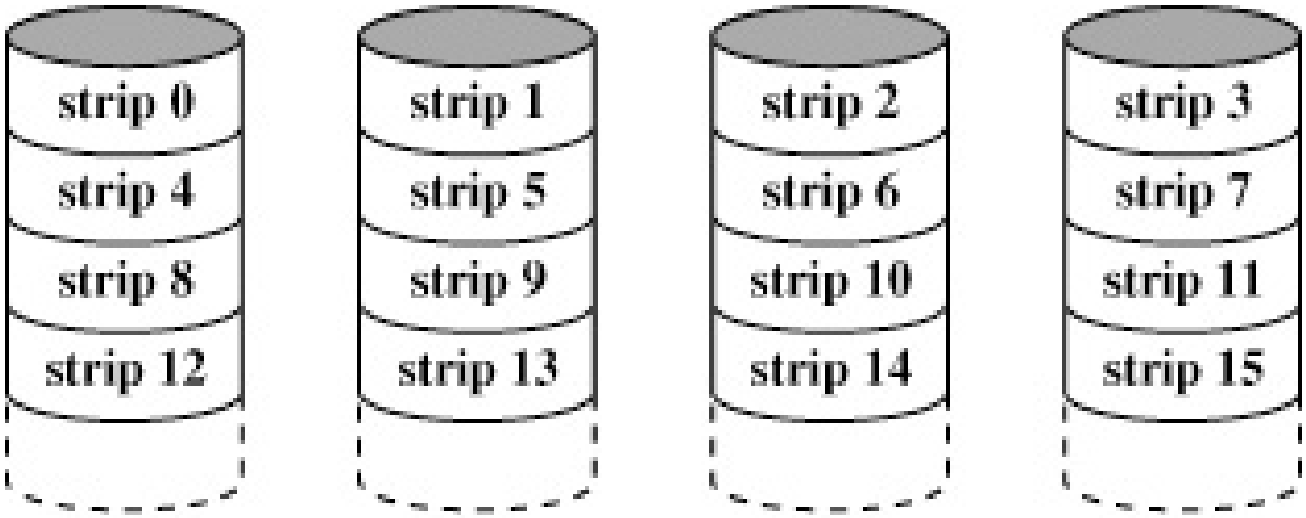
- **Les données changent de taille plus souvent que le programme**
- **Chaque fois qu'un proc demande plus de mémoire, on lui donne le double**



Autres mécanismes en Unix

- **Les mécanismes sont différents dans différentes versions de Unix**
- **Les différentes versions fonctionnent avec autres mécanismes, comme pagination, systèmes compagnons (buddy) etc.**

RAID: Redundant Array of Independent Disks

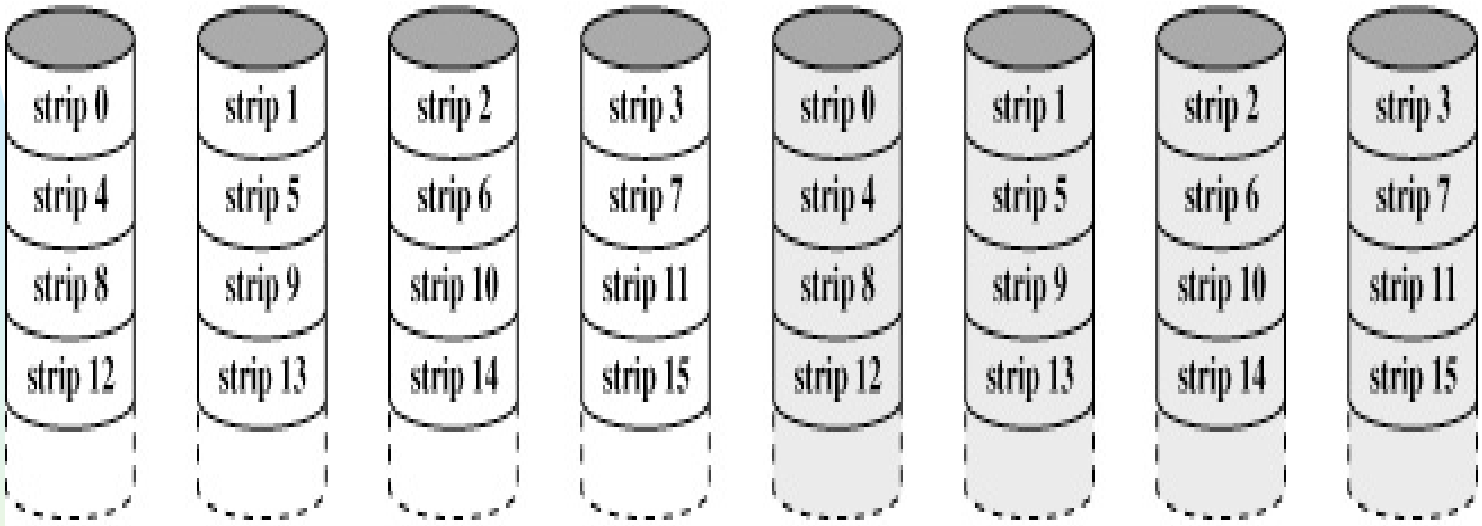


(a) RAID 0 (non-redundant)

Stallings

En distribuant les données sur différents disques, il est probable qu'une grosse lecture puisse être faite en parallèle (au lieu de lire strip0 et strip1 en séquence, cette organisation permet de les lire en même temps)

Redondance dans RAID

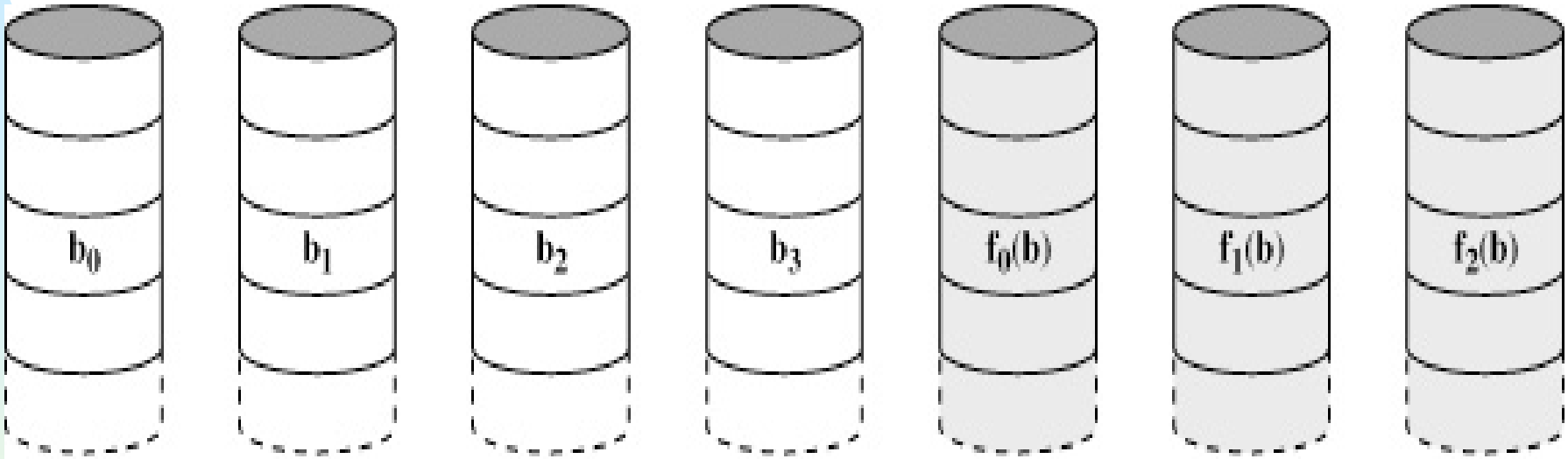


(b) RAID 1 (mirrored)

Stallings

Dupliquer les données pour incrémenter le parallélisme et remédier aux pertes de données (coûteux mais utilisé en pratique)

RAID: Correction d'erreurs par codes de correction



(c) RAID 2 (redundancy through Hamming code)

Stallings

Les codes de correction d'erreur pour des données enregistrées sur un disque sont sauvegardés sur un autre disque (plus de résistance aux erreurs)

Concepts importants du chapitre 13

- **Fonctionnement et structure des unités disque**
- **Calcul du temps d'exécution d'une séquence d'opérations**
- **Différents algorithmes d'ordonnancement**
  - Fonctionnement, rendement
- **Gestion de l'espace de permutation**
  - Unix
- **RAID: réorganisation des fichiers pour performance et résistance aux erreurs**

Par rapport au livre

- **Seulement 13.1, 13.2, 13.4**