Conception des systèmes

V. Felea / vafelea at femto-st dot fr / 411C

- système d'exploitation – SE (définition, fonctionnalités)
- gestion mémoire
- gestion processus
- allocation des ressources

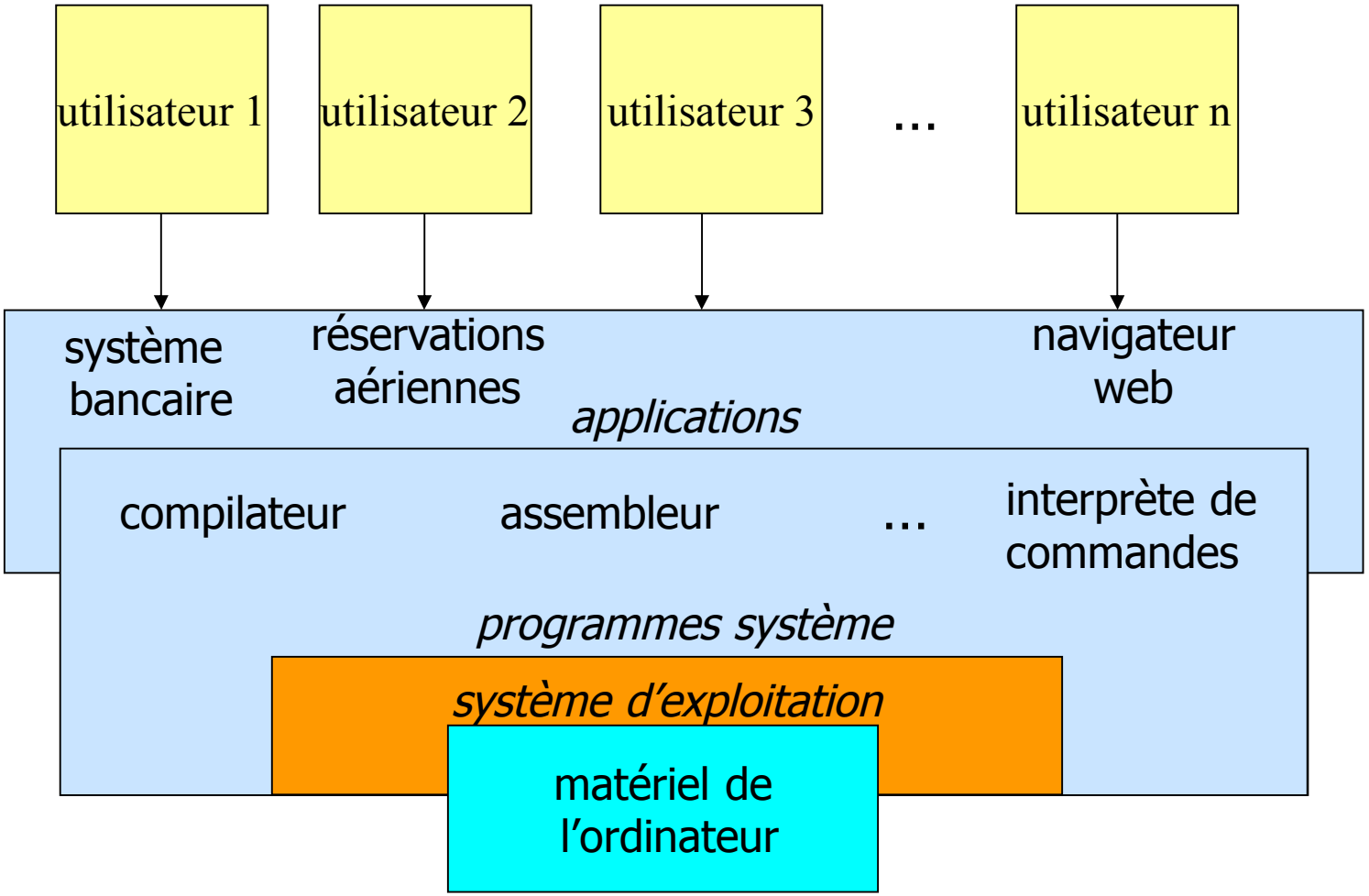
Bibliographie

- "Principes des systèmes d'exploitation des ordinateurs", S. Krakowiak, Dunod, 1987
- "Systèmes d'exploitation", A. Tanenbaum, Pearson Education, 2003 (et multiples éditions ultérieures)
- "Principes des systèmes d'exploitation", 4ème édition, A. Silberschatz et P. Galvin, Addison-Wesley, 1994

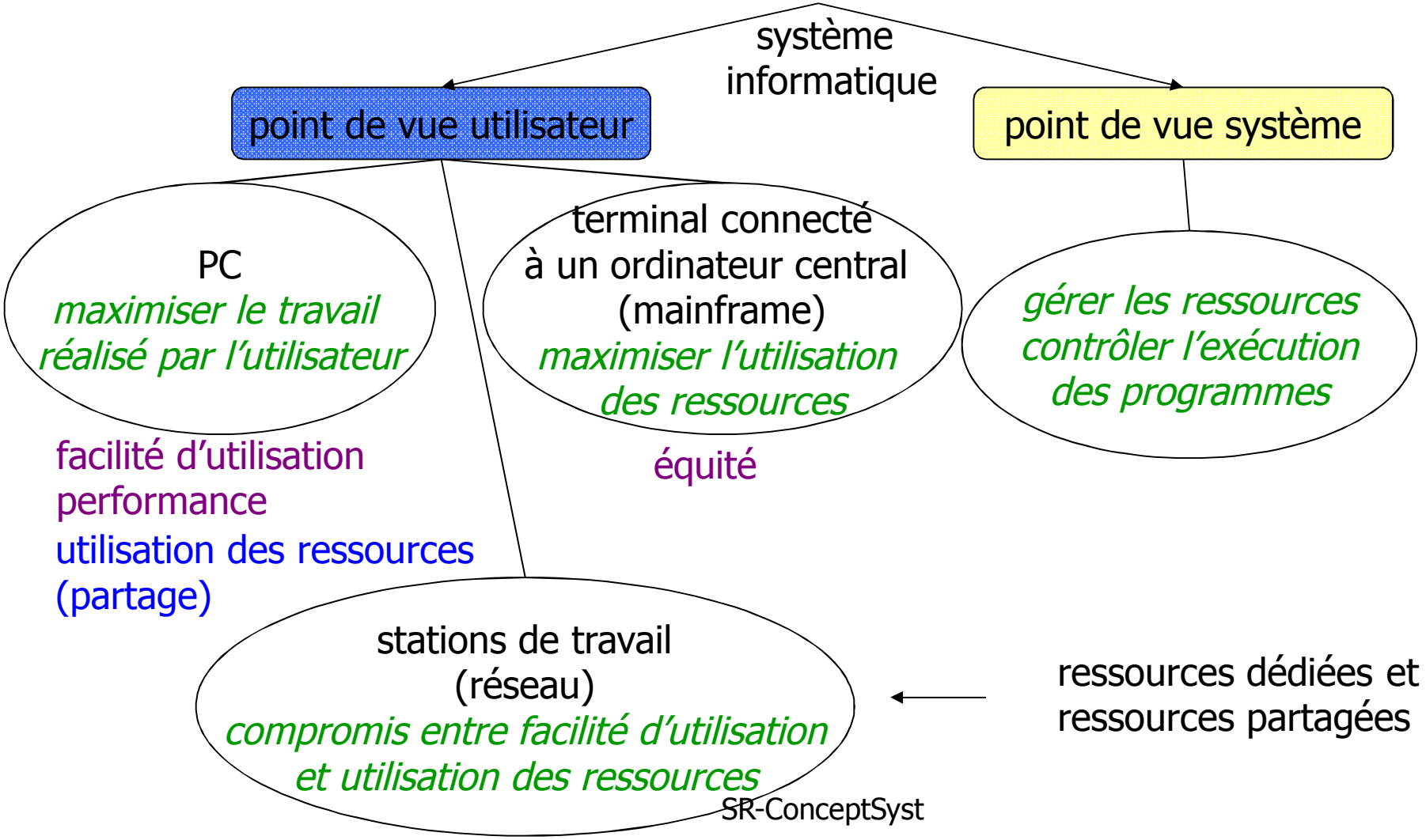
Conception des systèmes

Généralités d’un système d’exploitation

Vue générale d'un SE



Objectifs d'un SE



Système d'exploitation

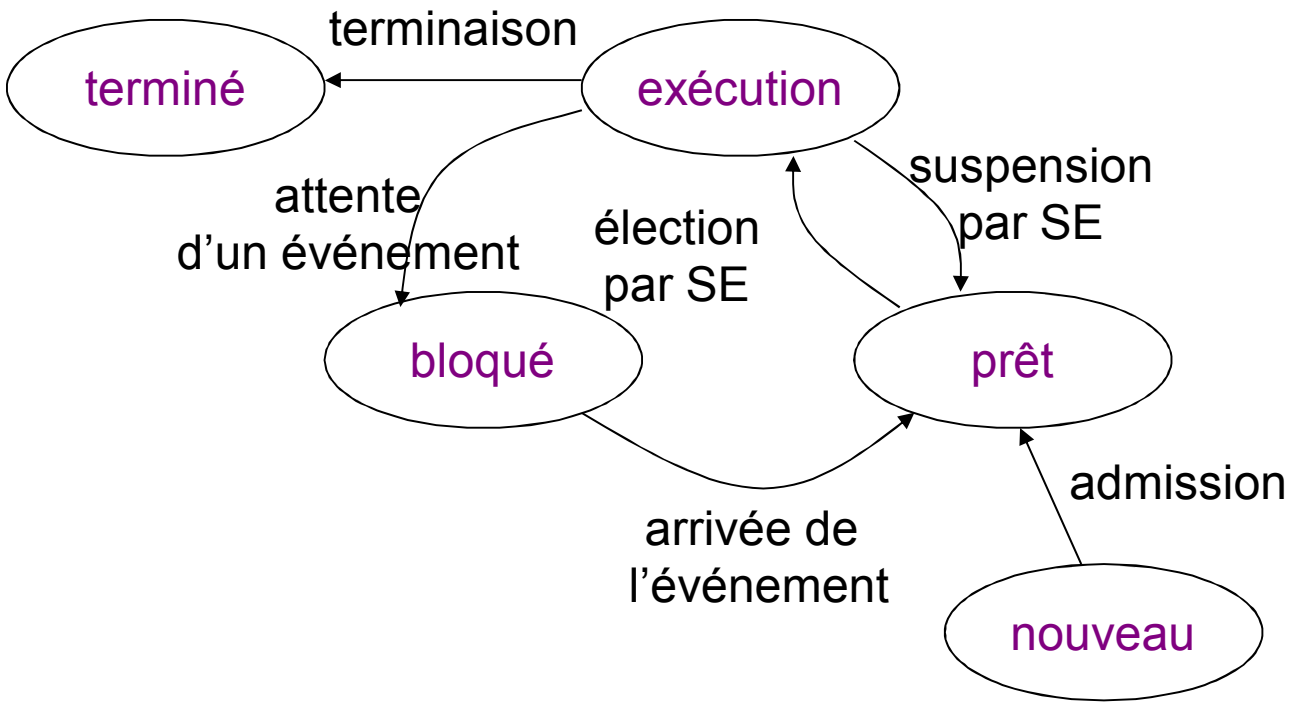
- place
  - intermédiaire entre un ordinateur et les applications qui l'utilisent
- rôles
  - d'***interface*** : fournit une interface plus commode à utiliser que celle du matériel
  - ***contrôleur de l'exécution*** des applications (***ordonnanceur***)
  - ***gestionnaire de ressources*** : gère les ressources matérielles et logicielles – mémoire, processeurs, programmes, communications (et l'allocation, le partage et la protection)

Ressources et leur gestion

- types de ressources
  - *physiques* : mémoire, unités E/S, processeur, ...
  - *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

  $\Rightarrow$ aux *processus*

Processus – états et transitions



Etats Nouveau, Terminé

- *Nouveau*
  - le SE a créé le processus
    - construit un identificateur pour le processus (PID)
    - construit une nouvelle entrée (PCB) dans la table de processus
  - des ressources ne lui sont pas encore allouées $\Rightarrow$ pas encore *admis pour être ordonnancé*

- *Terminé*
  - le processus n'est plus exécutable, mais ses ressources sont encore requises par le SE

PCB

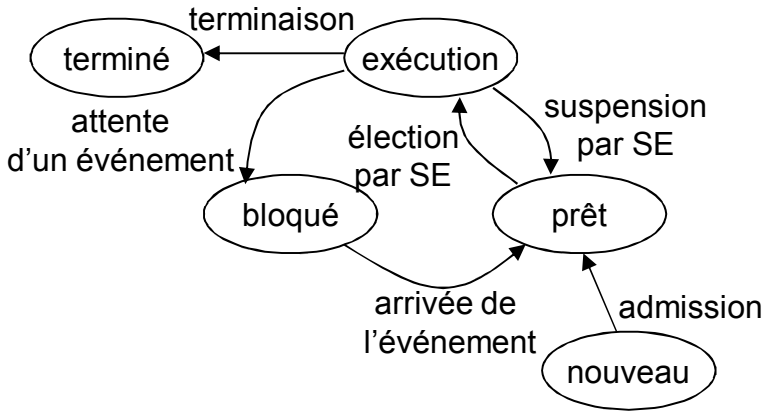
- **P**rocess **C**ontrol **B**lock (bloc de contrôle de processus)
  - état du processus
  - compteur d'instructions
  - registres CPU
  - information d'ordonnancement
  - information sur la gestion mémoire (table des pages/segments)
  - information de comptabilisation
  - état des E/S
- sert à sauvegarder et restaurer le contexte mémoire et processus lors d'une commutation de contexte

Transitions entre états (1)

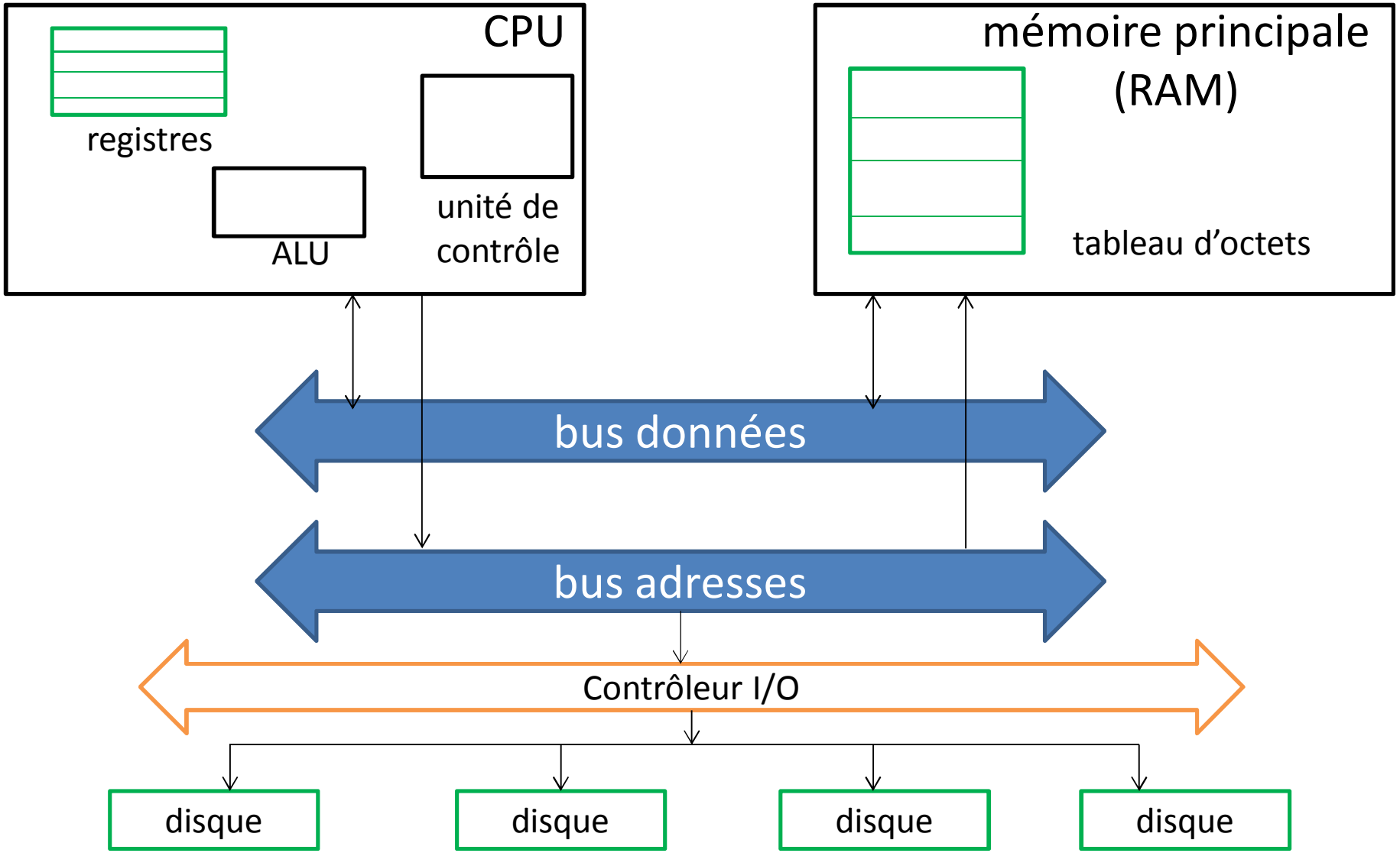
- *Prêt* → *Exécution*
  - lorsque le SE choisit un processus pour exécution
- *Exécution* → *Prêt*
  - résultat d'une interruption causée par un événement indépendant du processus
    - il faut traiter cette interruption, donc le processus courant perd l'UC
  - cas important : le processus a épuisé son intervalle de temps (minuterie)

Transitions entre états (2)

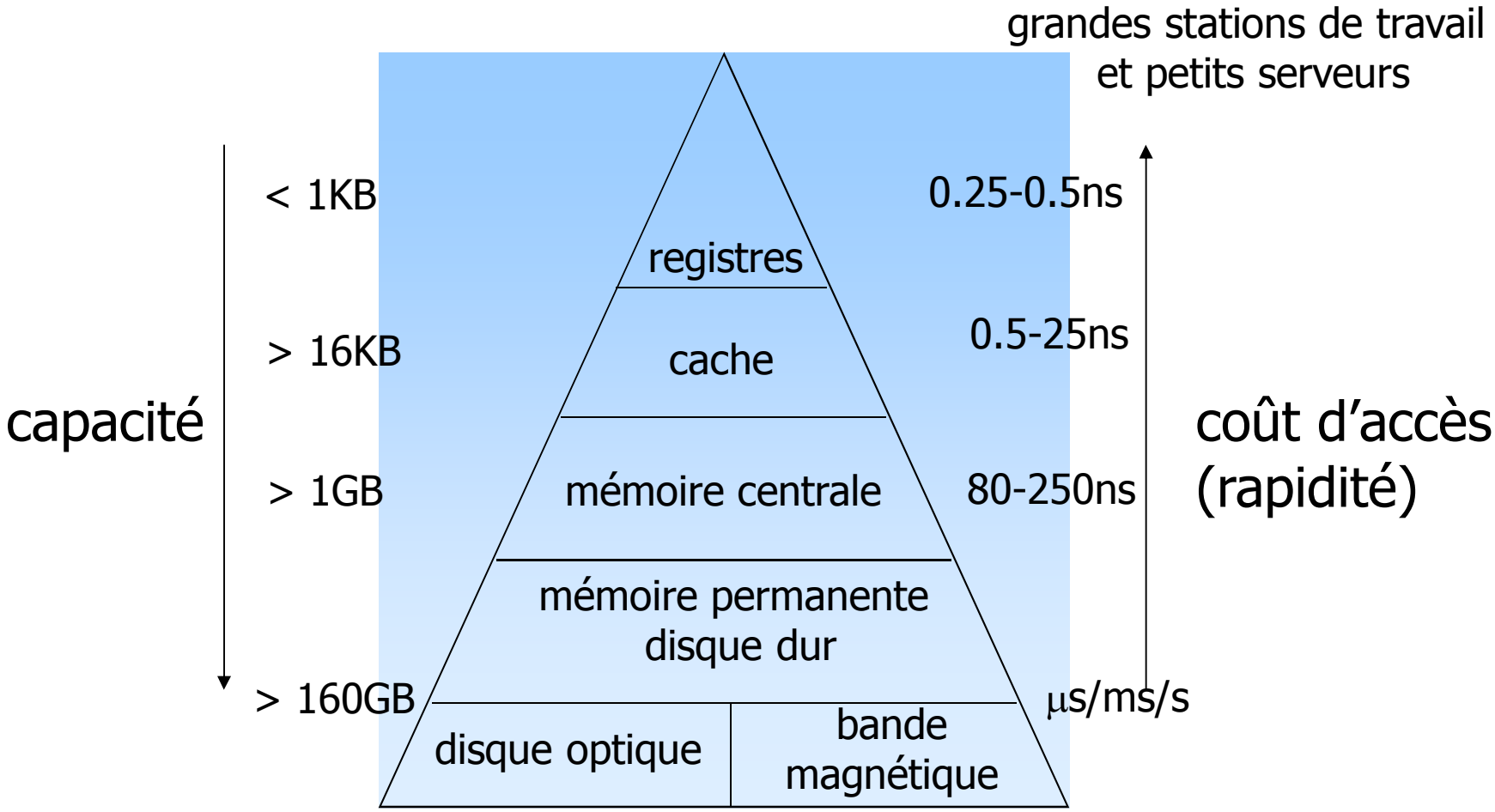
- *Exécution* → *Bloqué*
  - lorsqu'un processus demande un service du SE que le SE ne peut pas offrir immédiatement
    (interruption causée par le processus lui-même)
    - accès à une ressource pas encore disponible
    - opération d'E/S
    - attente d'une réponse d'un autre processus
- *Bloqué* → *Prêt*
  - lorsque l'événement attendu se produit



Mémoire

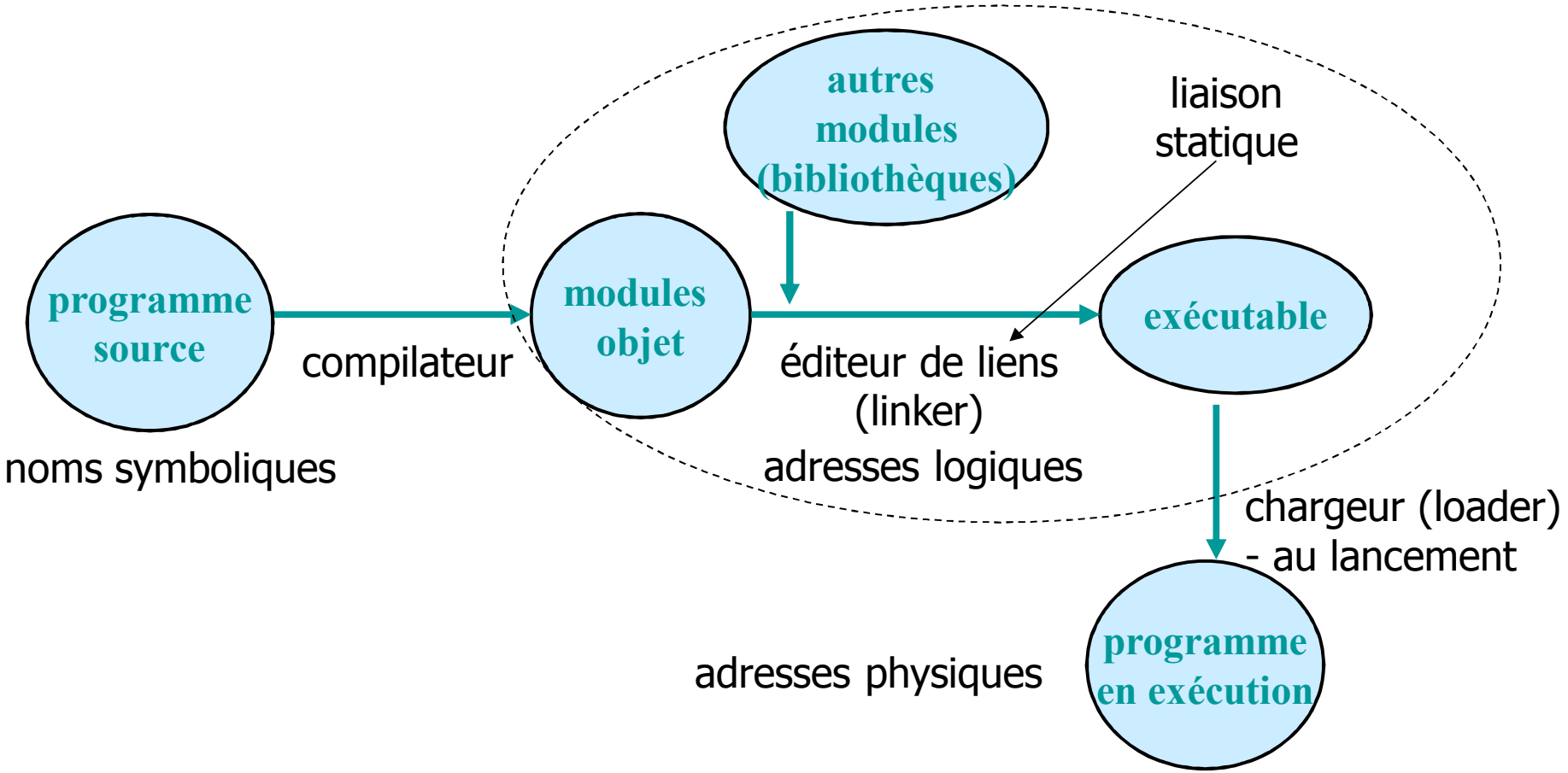

CPU
registres
ALU
unité de
contrôle
mémoire principale
(RAM)
tableau d'octets
bus données
bus adresses
Contrôleur I/O
disque
disque
disque
disque

Hiérarchie de mémoire



Mémoire d'un programme/processus

- mémoires logique / physique



Linker/Loader in C

building process (including linker) → loader

gcc p1.c p2.c → a.out *./a.out*

cc1
internal cmd

cpp preprocessor
cpp p1.c -o /tmp/p1.i

cpp compiler - generate assembler code
csl /tmp/p1.i –o /tmp/p1.s

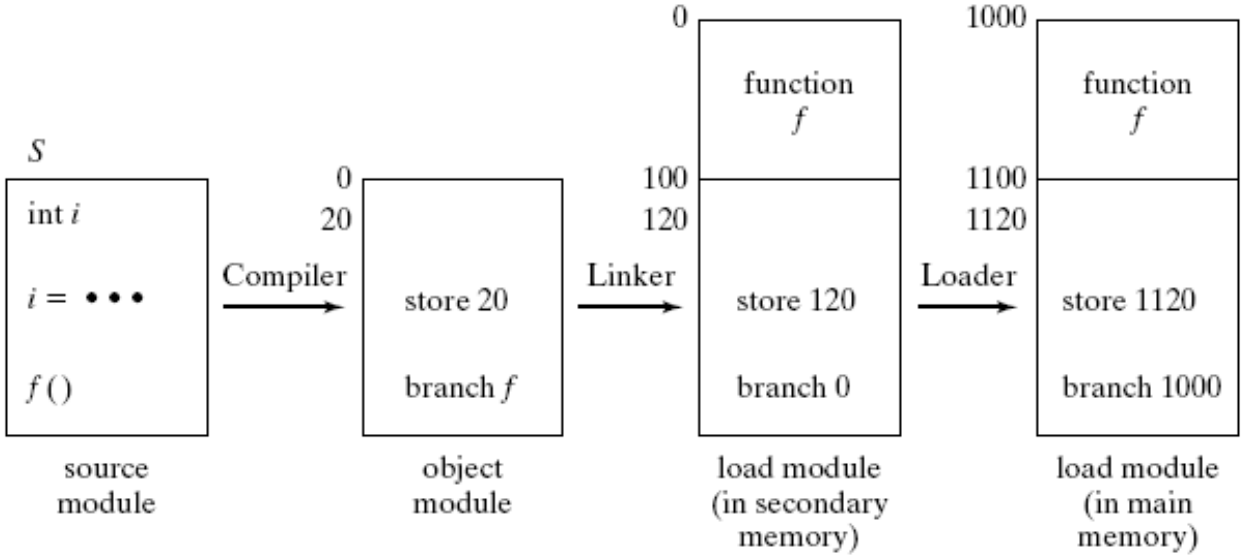
assembler - generate object
as /tmp/p1.s –o /tmp/p1.o

linker
ld /tmp/p1.o /tmp/p2.o –o a.out

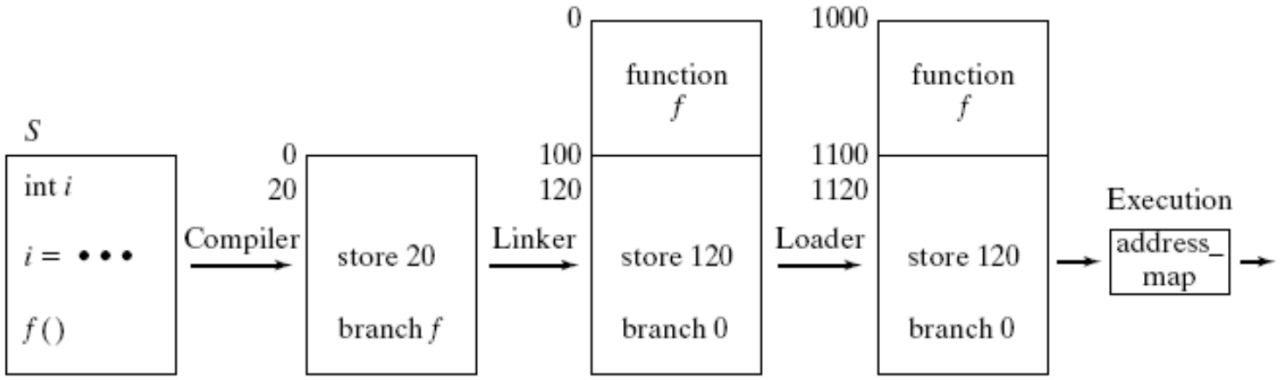
building process option :
--save-temps
→ keeps intermediate files

(Linux)
more assembly_file
readelf –a object_file

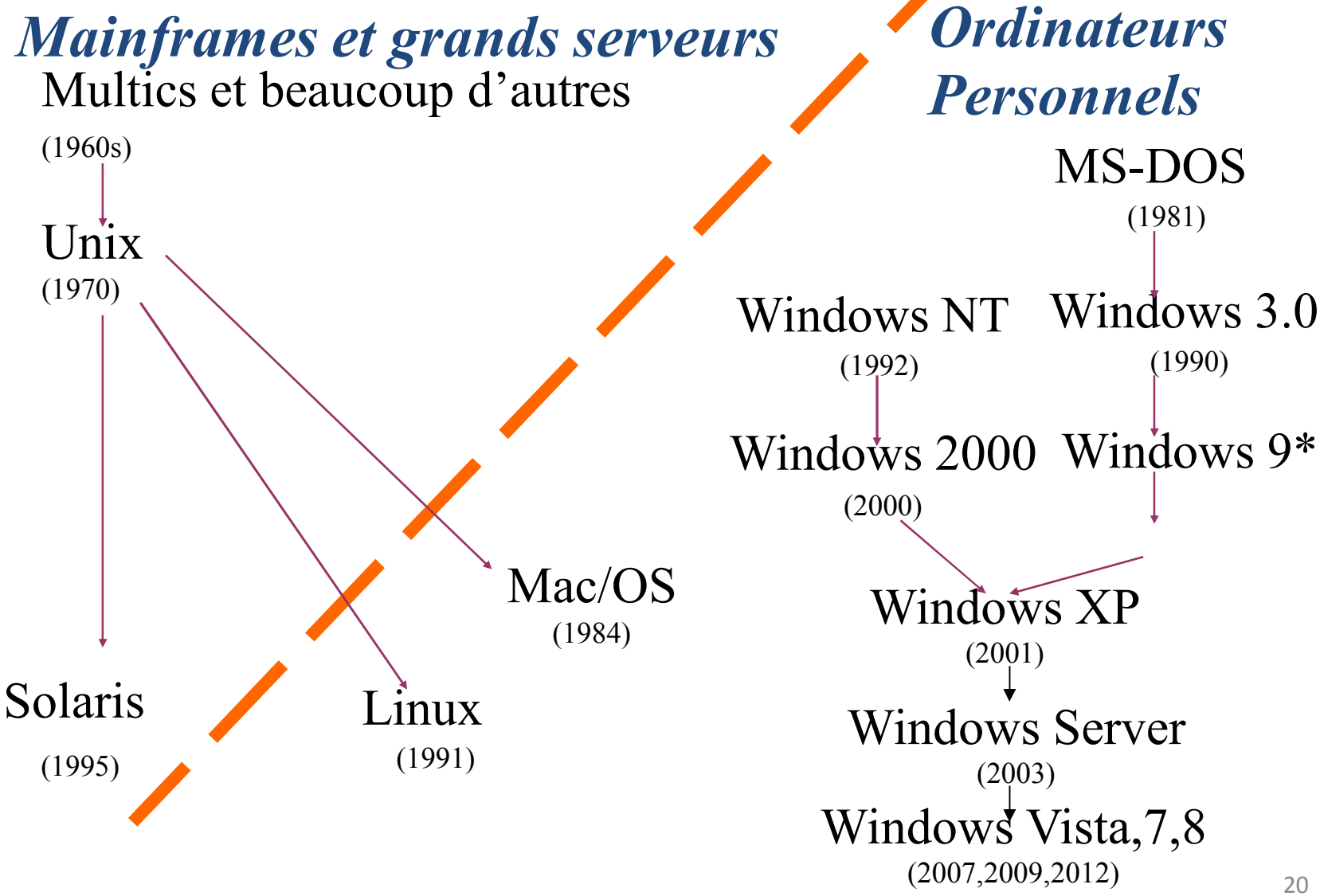
Adresses Linker/Loader (liaison statique)



Adresses Linker/Loader (liaison dynamique)



Synthèse historique des SE



Windows - évolution

The Guardian 2.10.2014 – Samuel Gibbs

- https://www.theguardian.com/technology/2014/oct/02/from-windows-1-to-windows-10-29-years-of-windows-evolution

Windows – évolution sur 29 ans (1)

- 1985 – Windows 1.0 (1er essai de IHM pour architecture processeur 16b) / jeu Reversi
- 1987 – Windows 2.0 (fenêtres recouvrantes, minimiser/maximiser les fenêtres) + Microsoft Word et Excel
- 1990 – Windows 3.0 (plus de disque dur pour l'installation mais version préinstallée) + jeu Solitaire

Windows – évolution sur 29 ans (2)

- 1992 - Windows 3.1 (caractères TrueType – représentation à base de courbes Bézier quadratiques, 1M RAM, contrôle des programmes MS-DOS par la souris, distribution sur CD-ROM) + jeu Minesweeper
- 1995 – Windows 95 (premier Menu/bouton start, plug&play sur périphériques, compatible archi 32b, multitâches + InternetExplorer)
- 1998 – Windows 98 (connecteur USB + IE4, Outlook Express, Microsoft Chat, Windows Media Player)

Windows – évolution sur 29 ans (3)

- 2000 – Windows ME (outils de restauration, auto-complétion dans WE + IE 5.5, Windows Media Player 7, Windows Movie Player) + buggy !
- 2000 – Windows 2000 basé sur Windows NT et base de Windows XP (mise à jour automatique, 1er système de mise en veille)
- 2001 – Windows XP (effort sur l'IHM) de longévité remarquable (maintenu jusqu'en 2014), 430m PC + problème : sécurité (Bill Gates « Trustworthy Computing »)

Windows – évolution sur 29 ans (4)

- 2007 - Windows Vista (options de recherche, sécurité, reconnaissance vocale, distribution sur DVD + Windows DVD player, Photo Gallery) + buggy !
- 2009 – Windows 7 (convivialité – moins de boîtes de dialogue, plus rapide, plus stable, plus facile d'utilisation, plus de mécanismes automatiques de redimensionnement des fenêtres) = upgrade de Windows XP

Windows – évolution sur 29 ans (5)

- 2012 - Windows 8 (écran tactile, interface ‘tuilée’, périphériques rapides USB 3.0, réintégration de l’usage de la souris et du clavier malgré l’apparition des IPad et smarphones)
- 2014 – Windows 10 (réapparition du menu Start, redésigné pour clients ‘traditionnels’, compatible toutes plateformes pour multiples dispositifs – iPhones, tablettes)

Linux – évolution sur 25 ans (1)

- 1991 – Linus Benedict Torvalds (étudiant finois de 21 ans) propose un S.E. gratuit : Linux 0.01
- 1993 – distributions Linux (Slackware, Debian)
- 1994 – Linux 1.0 pour architecture uni-processeur i386 (176 250 lignes de code)
- 1996 – Linux 2.0 pour architecture multiprocesseur type SMP et divers types de processeur + projet KDE
- 1998 – Google Search Engine (basé sur Linux)

Linux – évolution sur 25 ans (2)

- 1999 – Gnome desktop (l'environnement orienté bureau par défaut dans la plupart des distributions Linux : Debian, Fedora, RedHat, SUSE)
- 2000 – 2ème rang parmi les plus populaires S.E. pour les serveurs (après Windows NT)
- 2004 – Ubuntu 4.10 (!! 1st release) en octobre + adoption de Linux pour les portables
- 2007 – Google annonce Android

Linux – évolution sur 25 ans (3)

- 2009 – revue NYT :”More than 10 million people are estimated to run Ubuntu today”
- 2011 – Google : cloud Chrome OS for ChromeBooks + Linux 3.0 “Big change : nothing” (L. Torvalds)
- 2016 – Linux 4.0 (stable : 4.7.2 en 2016)

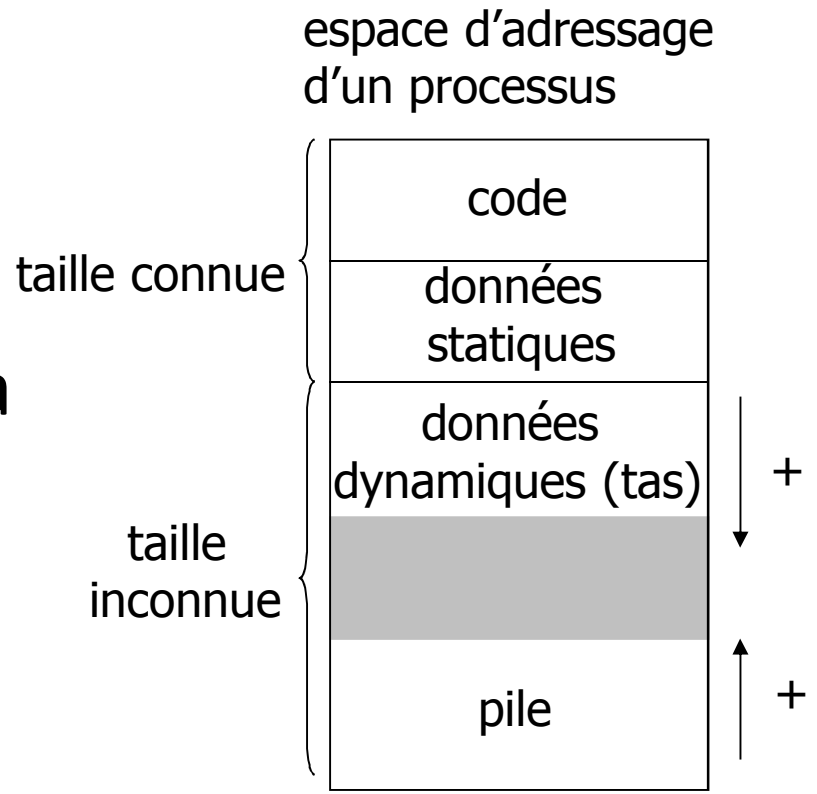
Conception des systèmes

Gestion mémoire

Mémoire d'un processus

= espace d'adressage d'un processus

- le code et les données statiques du programme à exécuter
- la pile
- les données dynamiques



Rôles d'un gestionnaire mémoire

- comment référencer les informations (instructions /données) ? directement dans la mémoire RAM ?
- comment organiser la mémoire ?
  - une ou plusieurs partitions, le nombre et la taille des partitions fixes ou variables au cours du temps
  - comment mémoriser l'état de la mémoire ?
- comment allouer de la mémoire à un processus ?

  ⇒***politique d'allocation / politique de placement***
- comment allouer de la mémoire à plusieurs processus ?
  - libérer de l'espace mémoire

  ⇒ ***politique de remplacement***
  - assurer la protection d'accès

Exigences

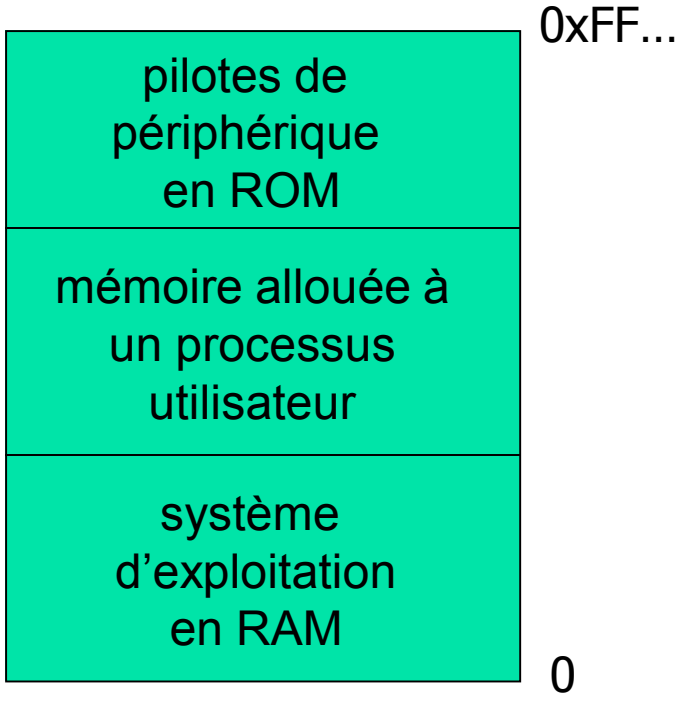
- ***efficacité***
  - allouer la mémoire de manière équitable et à moindre coût, en assurant une bonne gestion des ressources (mémoires, processeurs, disque)
- ***protection***
  - les processus ne peuvent pas se corrompre
- ***transparence***
  - chaque processus ignore l'existence des autres processus (concernant la mémoire)
- ***relocation***
  - la possibilité de déplacer un exécutable en mémoire (lui changer de place) – besoin d'adresses relatives

Solutions

- monoprogrammation
- multiprogrammation
  - partition
  - pagination
  - mémoire virtuelle
  - segmentation

Monoprogrammation

- la mémoire est allouée à un seul processus utilisateur à un instant donné
- adresse physique = adresse logique
- exemple : DOS

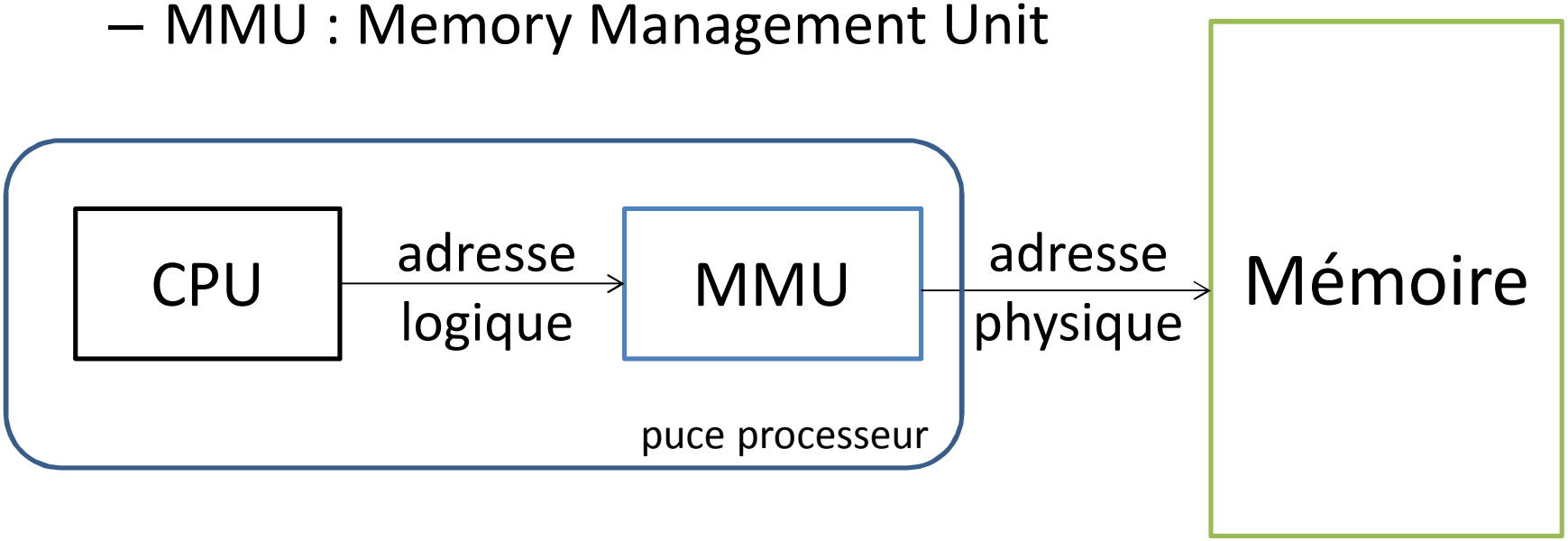


Multiprogrammation

- plusieurs allocations mémoire à divers processus à la fois
- objectif : maximiser la ressource processeur (processus interactifs - processus de calcul)
- protéger les adresses (physiques) entre processus
  - mêmes adresses pour l'espace d'adressage mais adresses distinctes en mémoire RAM pour éviter corruption (sauf en cas de partage mémoire)

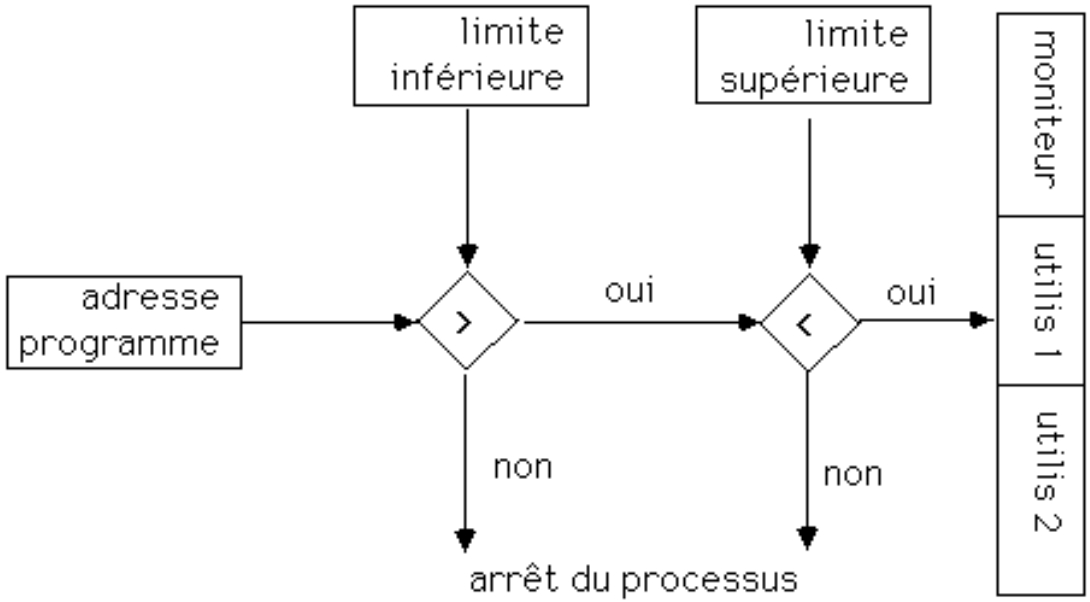
Mécanisme de conversion d'adresses

- traduction d'une adresse logique (appartenant à l'espace d'adressage du processus) en adresse physique (en mémoire RAM) = translation d'adresses
  - MMU : Memory Management Unit



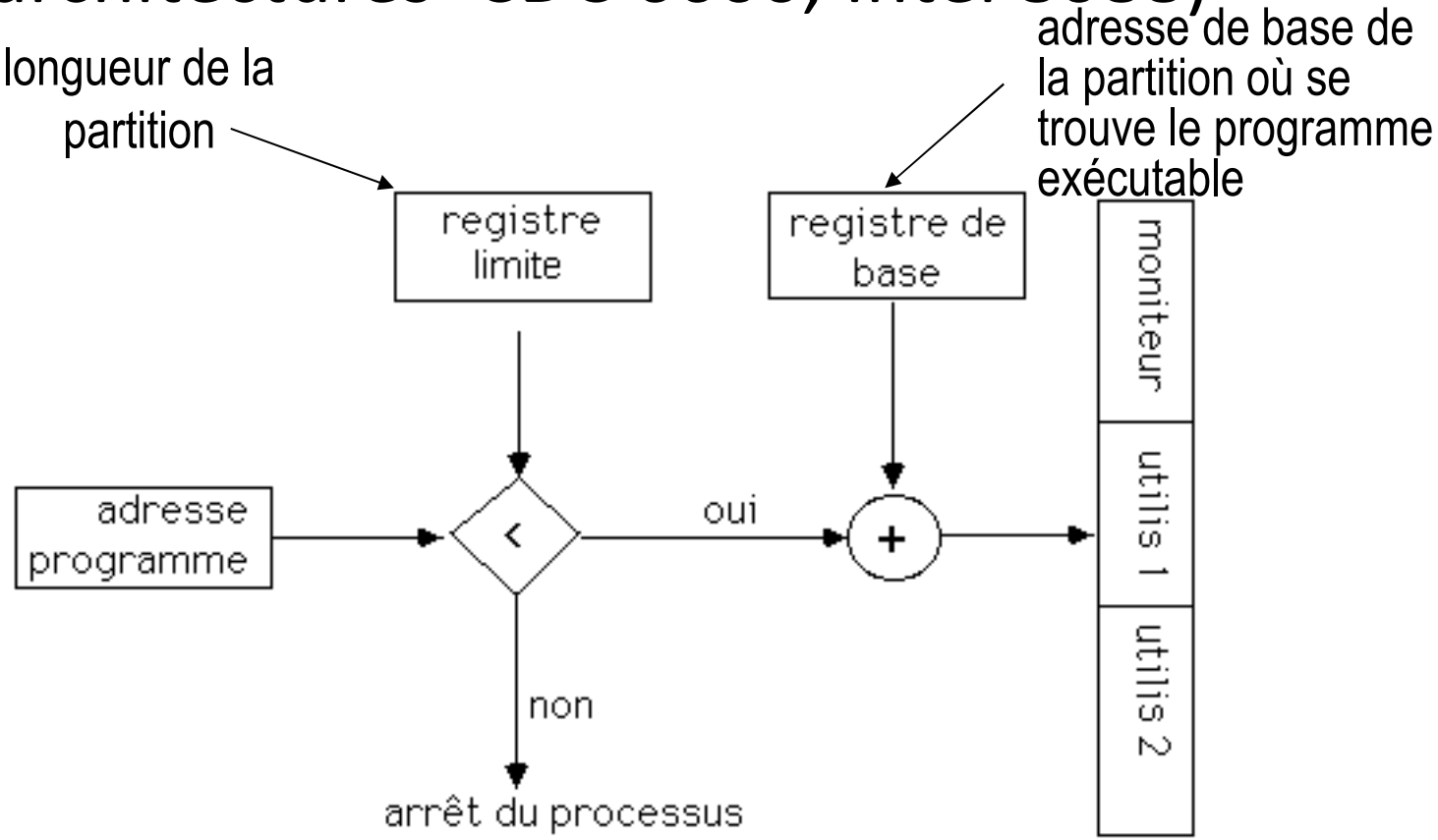
Mécanismes de protection d'adresses (1)

- paire de registres limites
  - registre limite inférieure (LBR)
  - registre limite supérieure (UBR)



Mécanismes de protection d'adresses (2)

- registres de base et limite (architectures CDC 6600, Intel 8088)



Partitionnement

- division de la mémoire en *partitions*
- cas
  - partitionnement fixe : partitions de tailles fixes (égales ou inégales)
  - partitionnement variable : partitions de tailles variables

- état de la mémoire
- allocation/libération mémoire

Partitionnement fixe

- partitions fixes

  = multiples partitions de mémoire à tailles fixes (égales ou inégales)

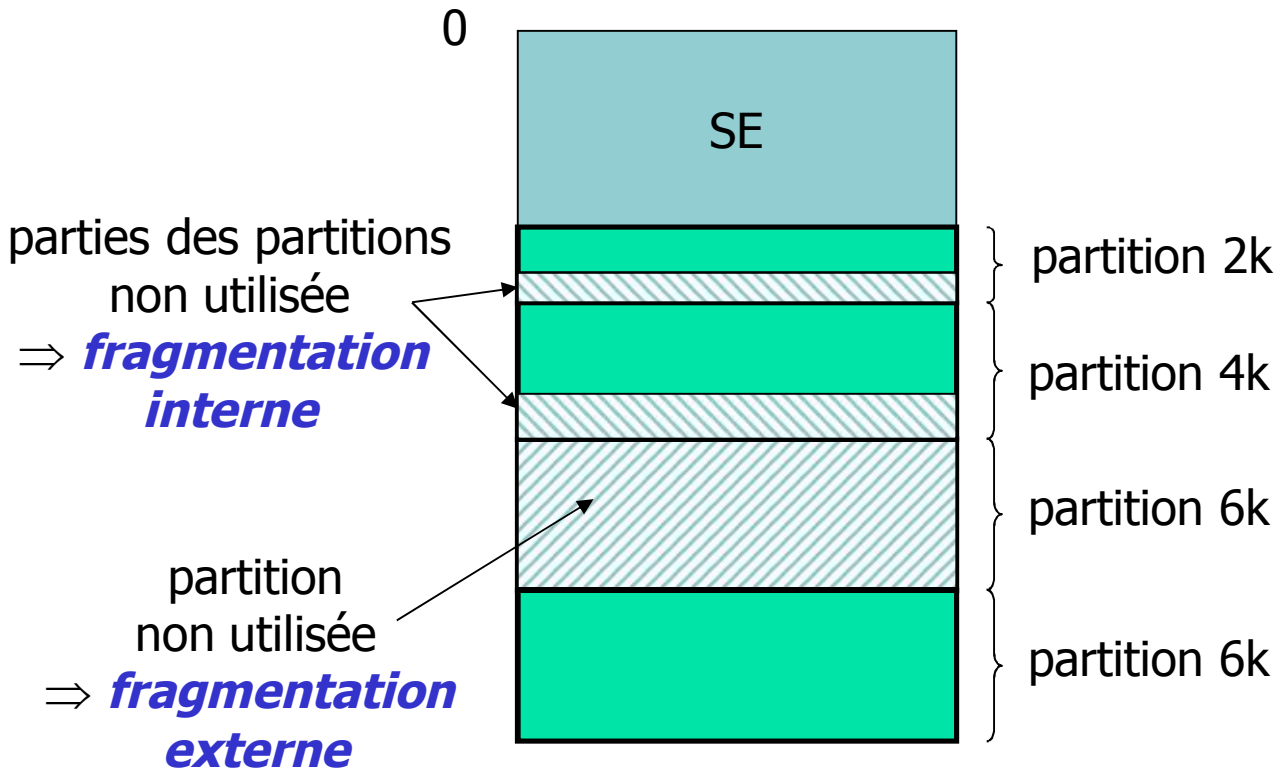
Problème : ***fragmentation***

- 2 types d'espace mémoire non utilisé
  - partie d'une partition non utilisée

  → ***fragmentation interne***
  - partition non utilisée

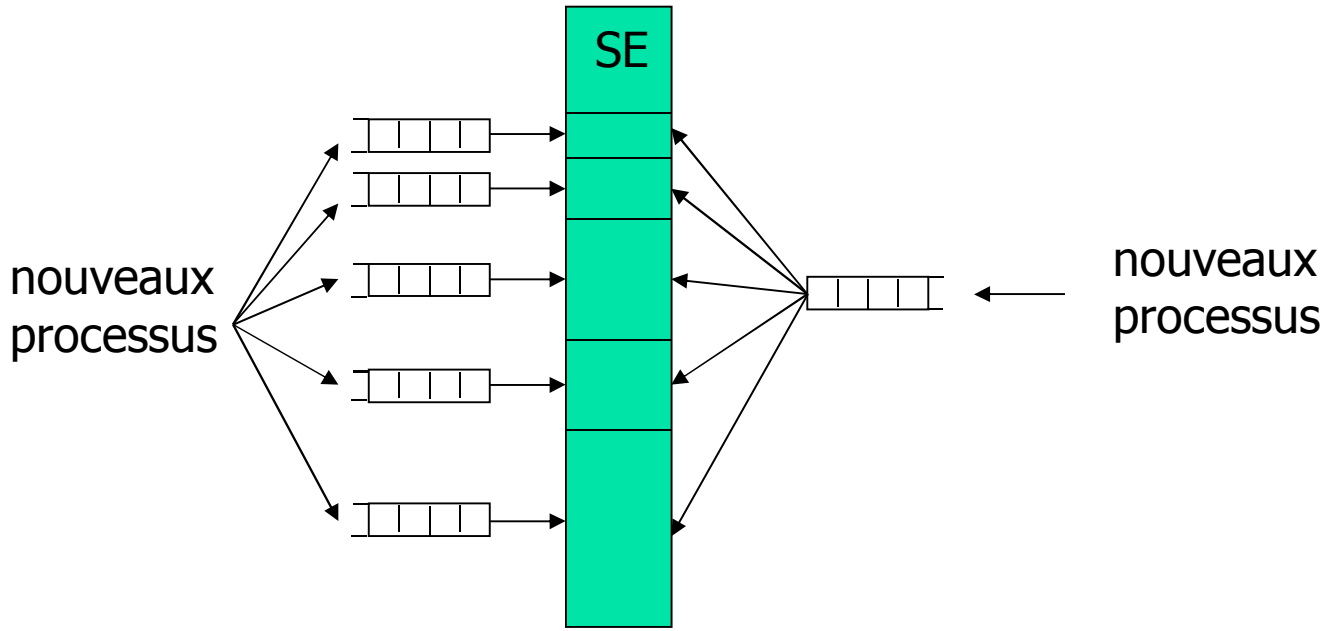
  → ***fragmentation externe***

Fragmentations interne/externe



Stratégies d'allocation mémoire

- file unique pour toutes les partitions
- file d'entrée multiple



Limites du partitionnement fixe

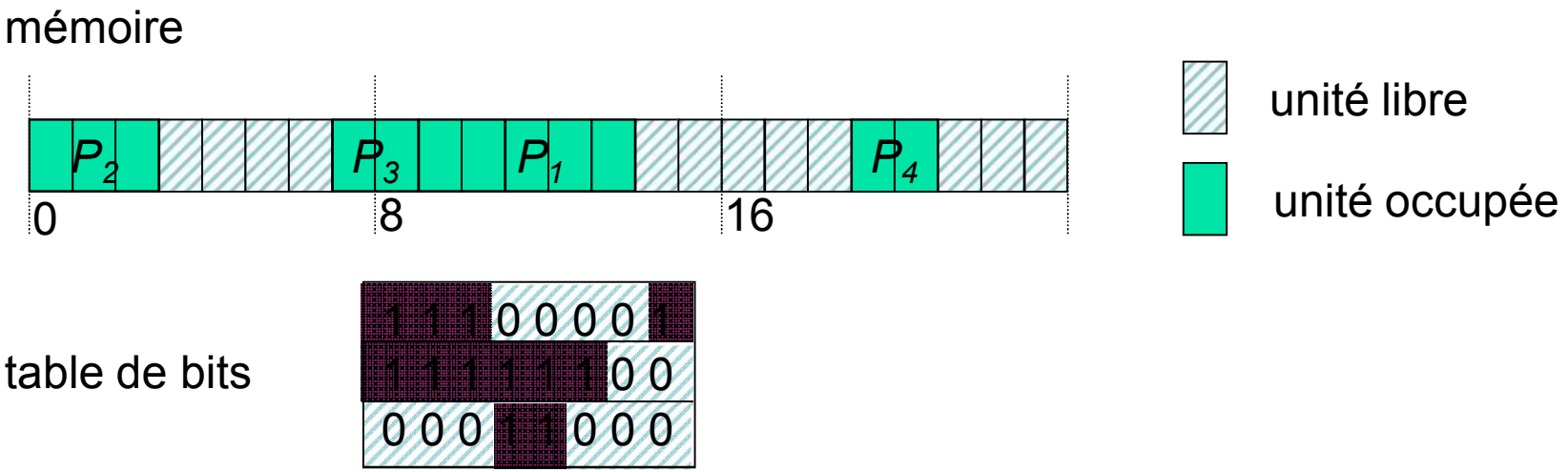
- objectifs du partitionnement
  - chargement simultané de plusieurs images processus en mémoire
  - protection d'adresses
- inconvénient
  - fragmentation
- solution : technique de partitionnement variable
  - multiples partitions de mémoire à taille variable
  - mémorisation des zones libres et occupées
  - réduire au maximum la fragmentation

Partitionnement variable

- la mémoire présente des zones (libres/occupées) de tailles variables
- connaître l'***état de la mémoire***
  - méthodes (principales) : table de bits et listes chaînées
- avoir une ***politique de placement*** et de ***récupération d'espace mémoire***
  - la première zone libre
  - la zone libre suivante
  - le meilleur ajustement
  - le plus grand résidu

Gestion par table de bits (1)

- mémoire divisée en unités d'allocation
  - unité de taille variant d'un mot à plusieurs Ko
  - unité ↔ bit à 1 si occupée
    ↔ bit à 0 si libre

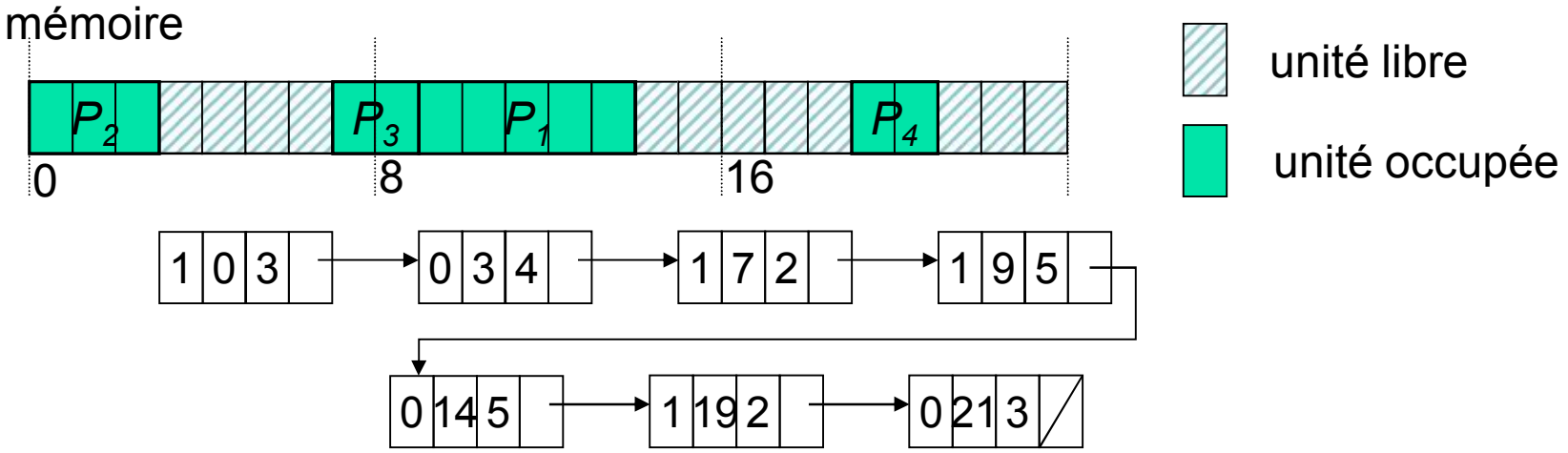


Gestion par table de bits (2)

- inconvénients
  - nécessite un espace mémoire important
    - directement dépendant de la taille de l'unité d'allocation et de la taille de la mémoire principale
  - recherche de *k* zéros consécutifs pour un processus demandant *k* unités mémoire (lente)
- avantage
  - simple à mettre en œuvre

Gestion par listes chaînées (1)

- segments libres et occupés
- chaque segment
  - bit d'état (1 occupé, 0 libre)
  - adresse de début
  - taille
- liste triée par adresse de début (d'autres critères : taille du segment libre)



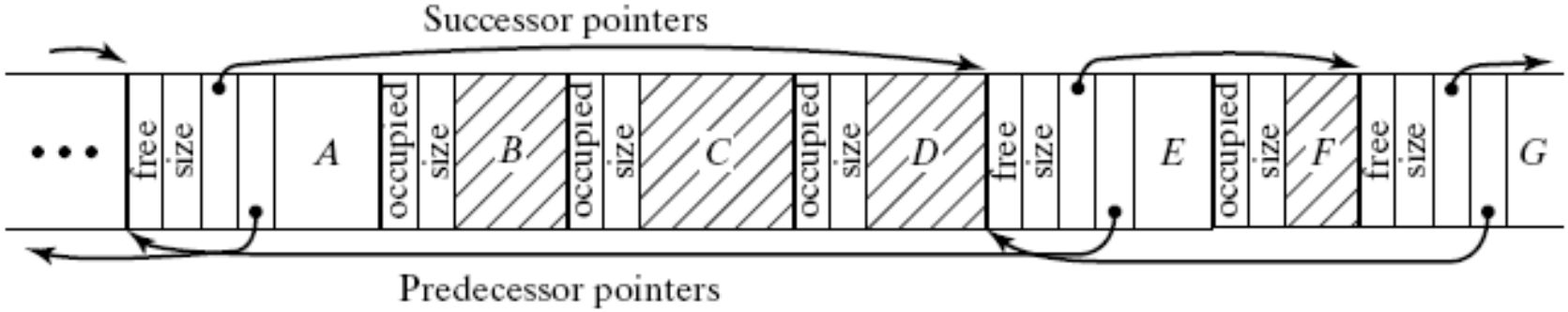
Gestion par listes chaînées (2)

- inconvénients
  - maintien plus complexe de la liste lors des libérations de mémoire

  $\Rightarrow$ double chaînage
- avantages
  - moins d'espace mémoire pour stocker l'information

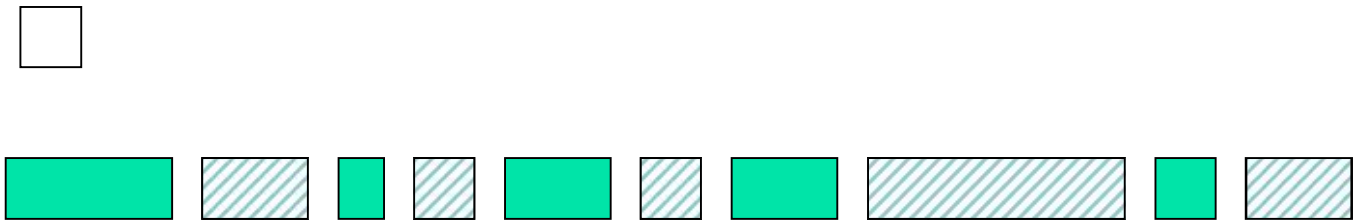
Gestion par listes chaînées (3)

- optimisation

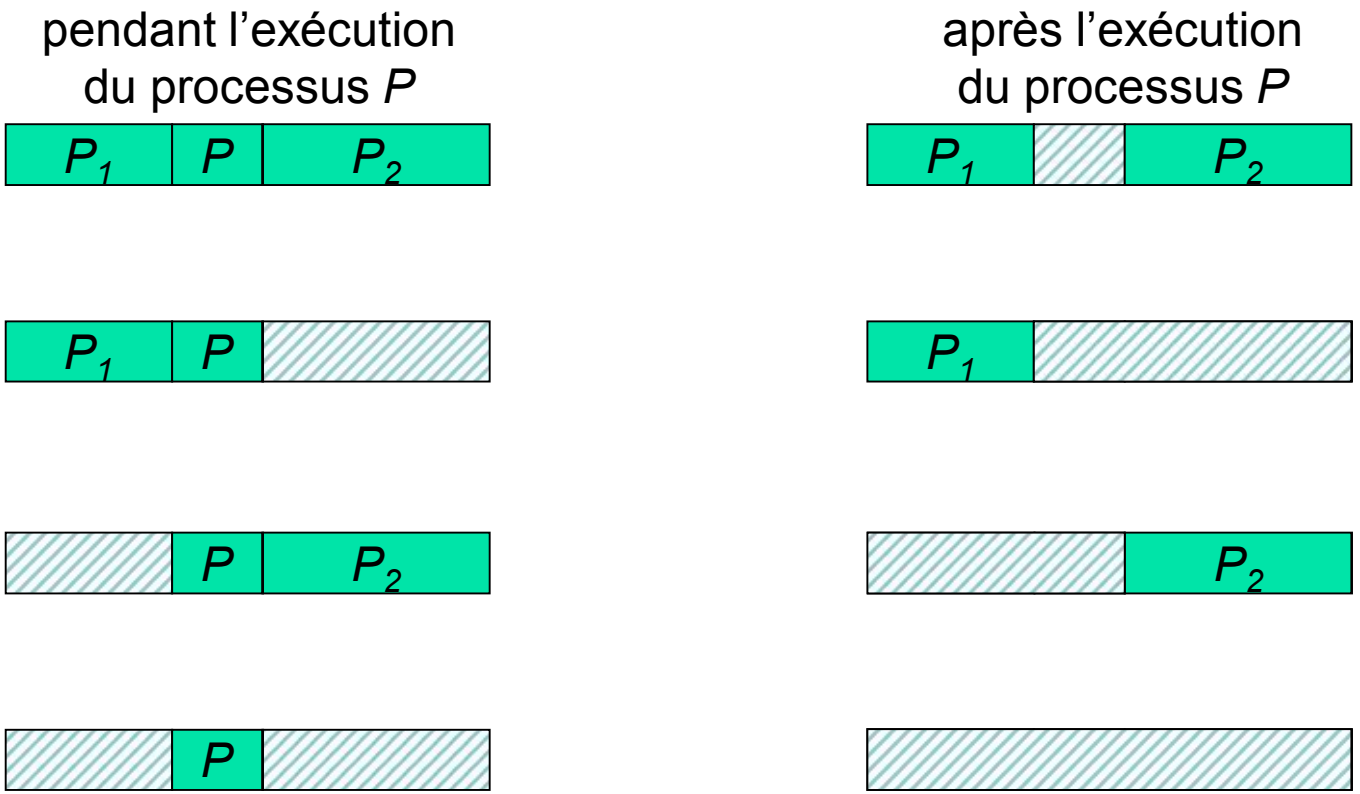


Allocation de la mémoire

- stratégies
  - la première zone libre (first fit)
  - la zone libre suivante (next fit)
  - le meilleur ajustement (best fit)
  - le plus grand résidu (worst fit)



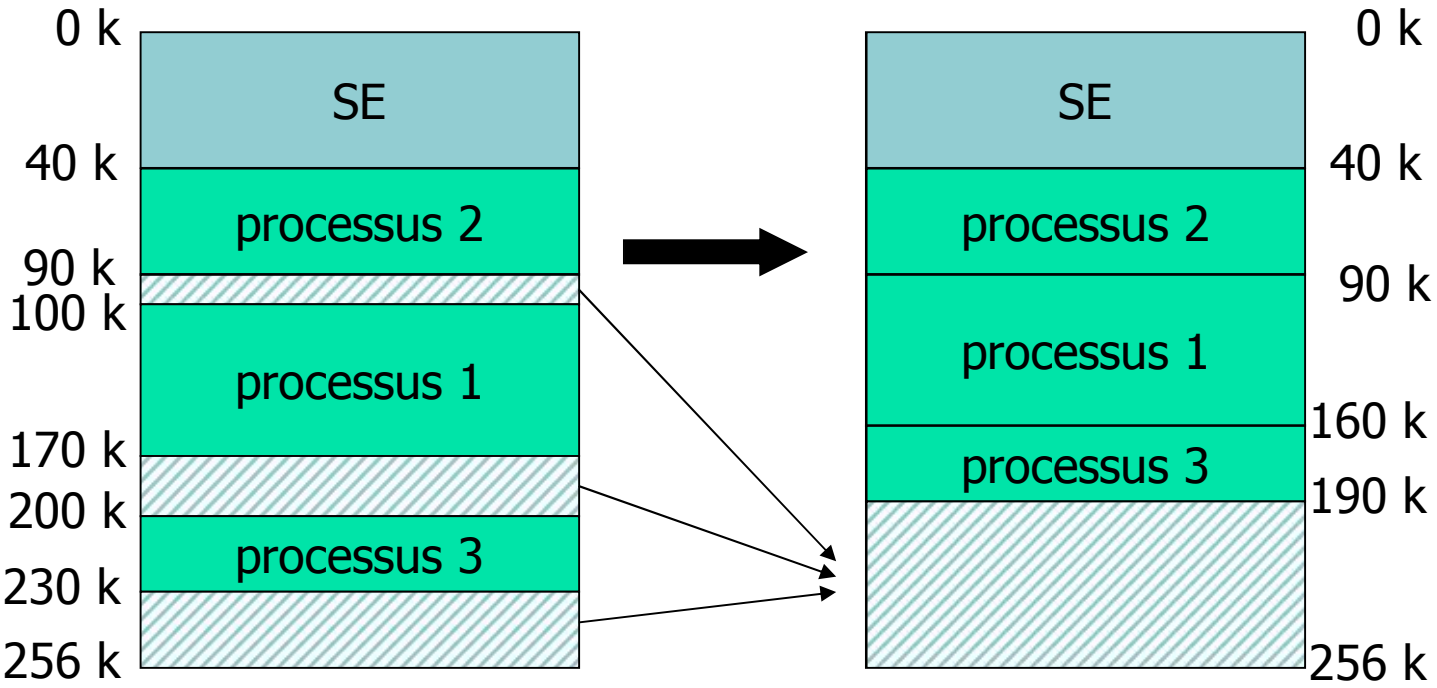
Libération de mémoire



Récupération de la mémoire

- problèmes lors de l'allocation / libération
  - fragmentation
  - fuite
- solutions
  - le ***compactage***
  - le ***ramasse-miettes*** (garbage collector)

Compactage

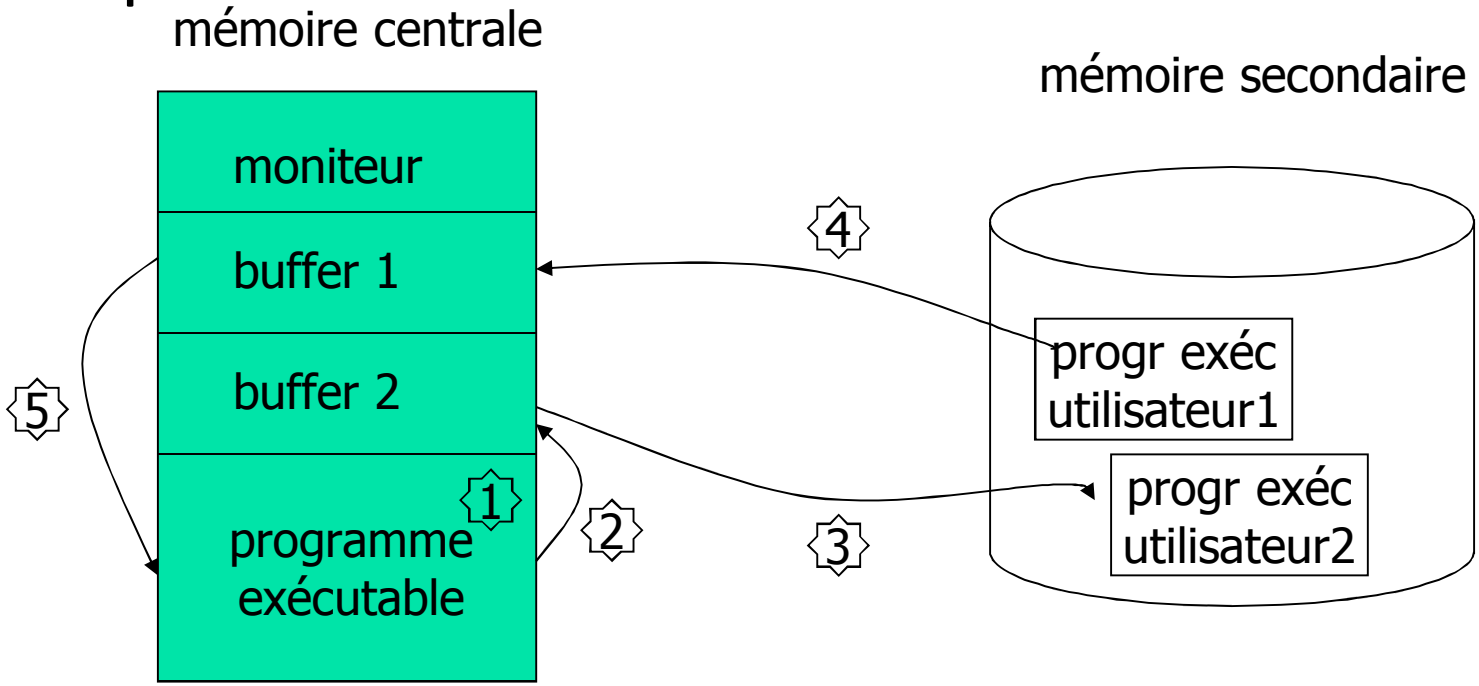

0 k
SE
40 k
processus 2
90 k
100 k
processus 1
170 k
200 k
processus 3
230 k
256 k
0 k
SE
40 k
processus 2
90 k
processus 1
160 k
processus 3
190 k
256 k

Ramasse-miettes

- libération de la mémoire facilitée explicitement par le programmeur
  - fonctions de bibliothèque (par exemple *free* ())
- récupération automatique
  - langage Java

Swapping

- lors de l'allocation, si aucune partition ne peut être allouée
- mouvement des exécutables (image mémoire du processus) entre la mémoire principale et le disque



Limites du partitionnement

- cas : tous les processus ne peuvent pas tenir simultanément en mémoire
- déplacement temporaire de certains processus sur une mémoire provisoire (swap)
- permet de pallier le manque de mémoire nécessaire
- n'autorise pas l'exécution de programmes de taille supérieure à celle de la mémoire centrale

Taille de l'espace mémoire d'un processus (1)

- ***image mémoire*** d'un processus
  - informations statiques : programmes, données statiques
  - structures dynamiques : tas, pile d'exécution
- fixer arbitrairement une taille pour la pile et le tas
- lorsqu'un processus a utilisé toutes ses ressources
  - soit le système met fin au processus
  - soit le système réalloue de la mémoire au processus dans une nouvelle zone en lui réservant plus de ressources

Question

- le processus doit résider dans sa totalité en mémoire (et les gros processus ?)
- besoin de recopier toute l'imagine mémoire d'un processus pour seulement agrandir quelques unes de ses structures de données

  $\Rightarrow$ ***mémoire virtuelle***

  $\Rightarrow$ ***segmentation***

Pagination (1)

- simplification et accélération des mécanismes d'allocation de mémoire (pourquoi ???)
- la mémoire physique est découpée en pages de taille fixe (quelques kilo-octets) : ***cadre*** (***page frame***)
- l'espace d'adressage d'un processus est également découpé en ***pages***
  - pages du processus chargées à des cadres libres de mémoire
  - correspondance → ***table des pages***

Correspondance page-cadre

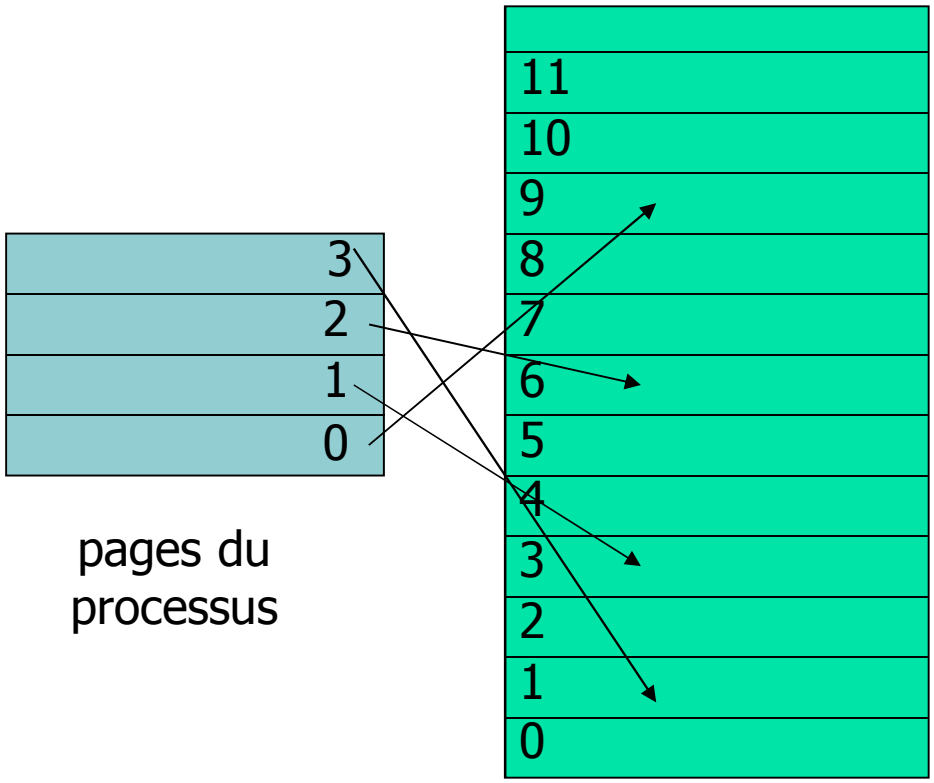


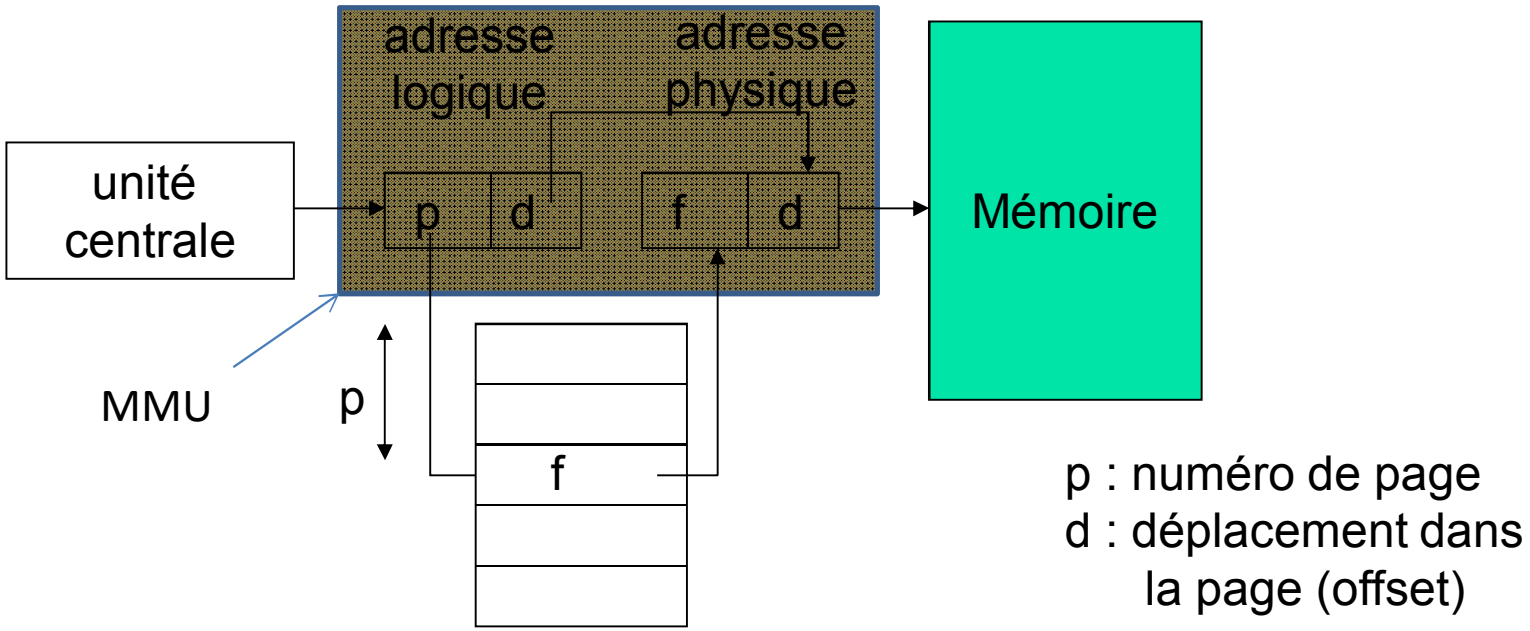
table des pages

0	9
1	3
2	6
3	1

9
3
6
1

Pagination (2)

- adresse dans un système paginé
  - numéro de page
  - position / déplacement dans la page



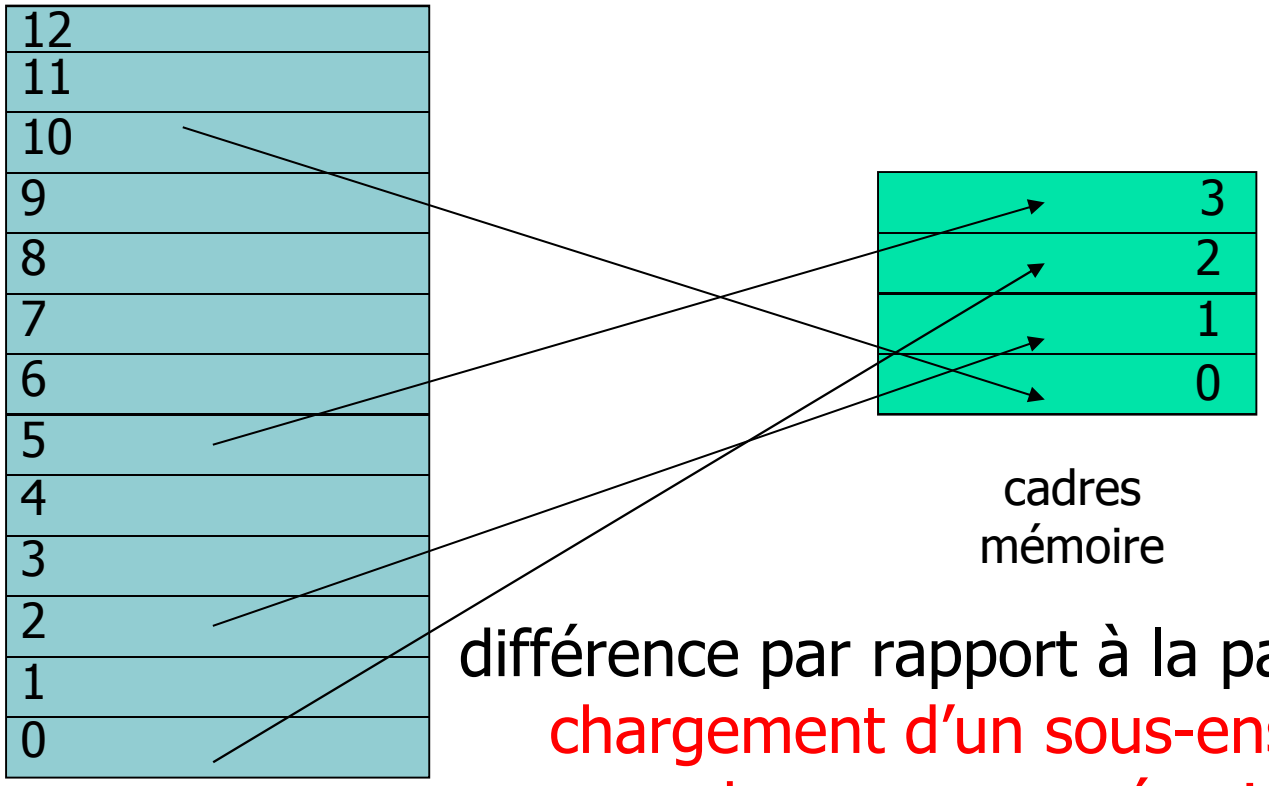
Avantages et inconvénients de la pagination

- avantages
  - pas besoin de zone contiguë de mémoire
  - allouer de la mémoire à un processus sans forcément devoir réaliser le compactage
- inconvénient
  - fragmentation interne

Mémoire virtuelle (1)

- méthodes swap, pagination
  - espace *adressable* par un processus qui s'exécute présent en mémoire
- remarque : pas toutes les parties d'un programme sont utilisées en même temps
- ***mémoire virtuelle***
  - permet d'exécuter des programmes qui ne tiennent pas entièrement en mémoire centrale lors de leur exécution
  - découpage du processus et de la mémoire en *pages /cadres*
  - chargement partiel des pages du processus

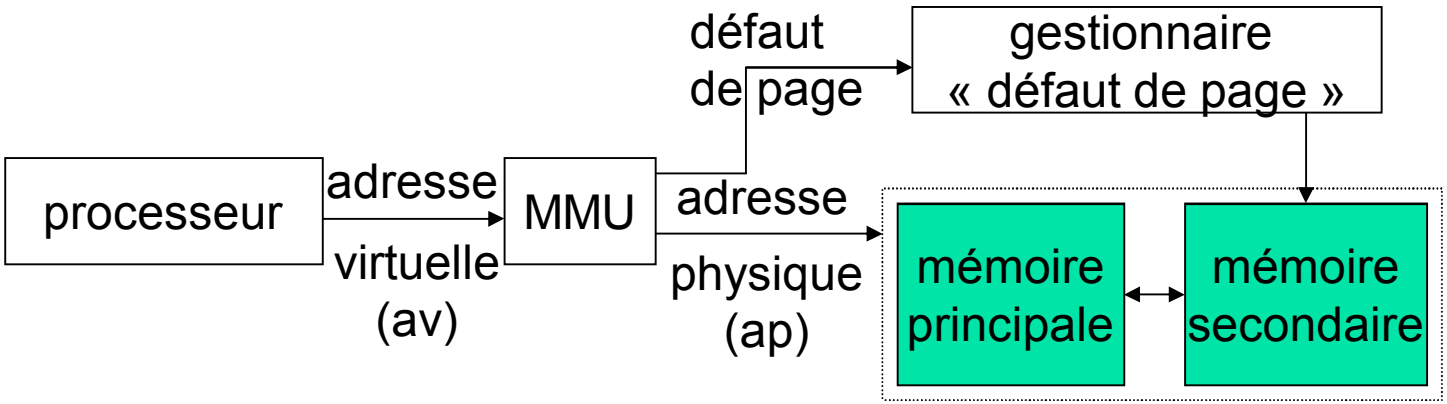
Mémoire virtuelle (2)



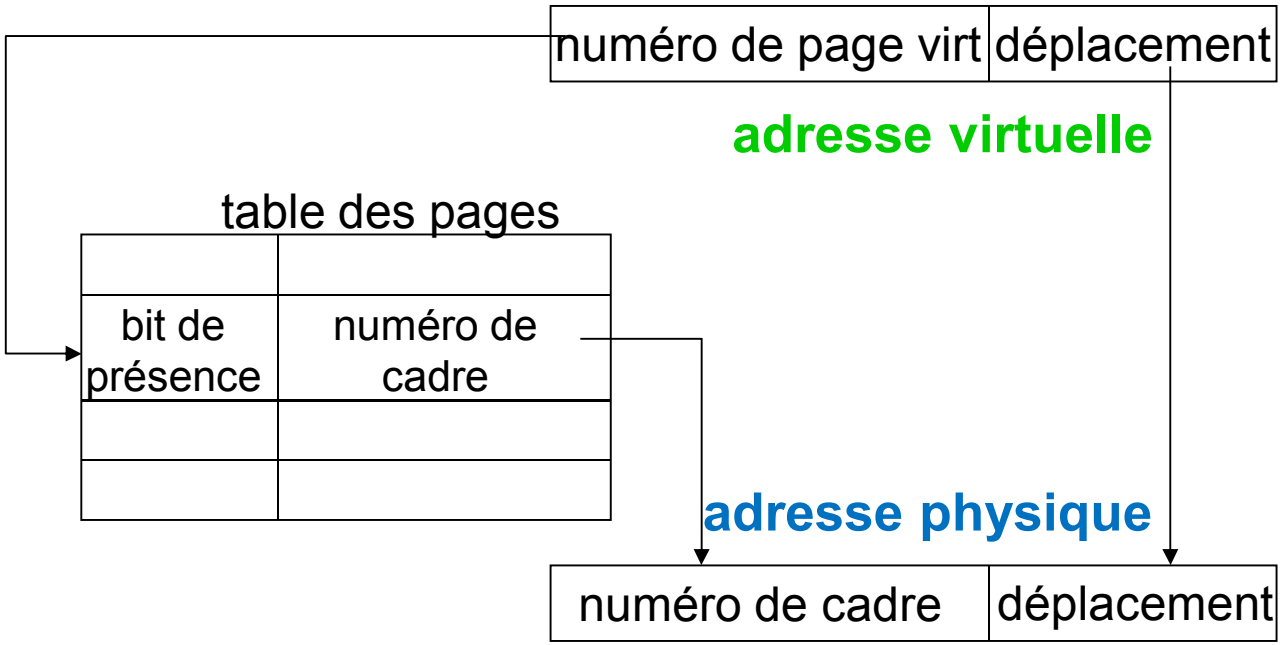
différence par rapport à la pagination :
chargement d'un sous-ensemble de pages en mémoire

Mémoire virtuelle (3)

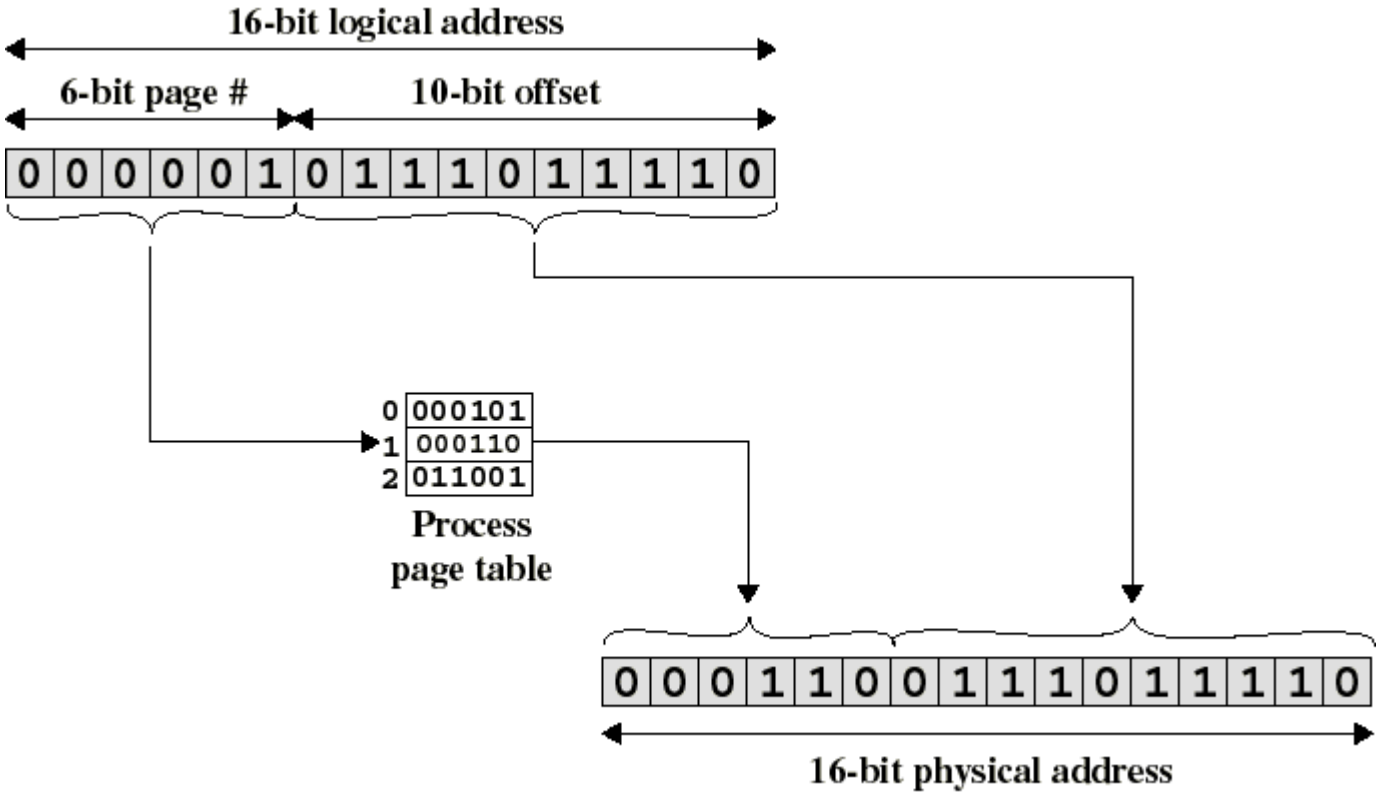
- *adresse virtuelle*
  - manipulée par le programme
  - *espace d'adressage virtuel*
    - divisé en pages
- Memory Management Unit (MMU)
  - unité de gestion de la mémoire (dispositif matériel)
  - traducteur des adresses virtuelles en adresses physiques



Transcodage des adresses virtuelle (1)

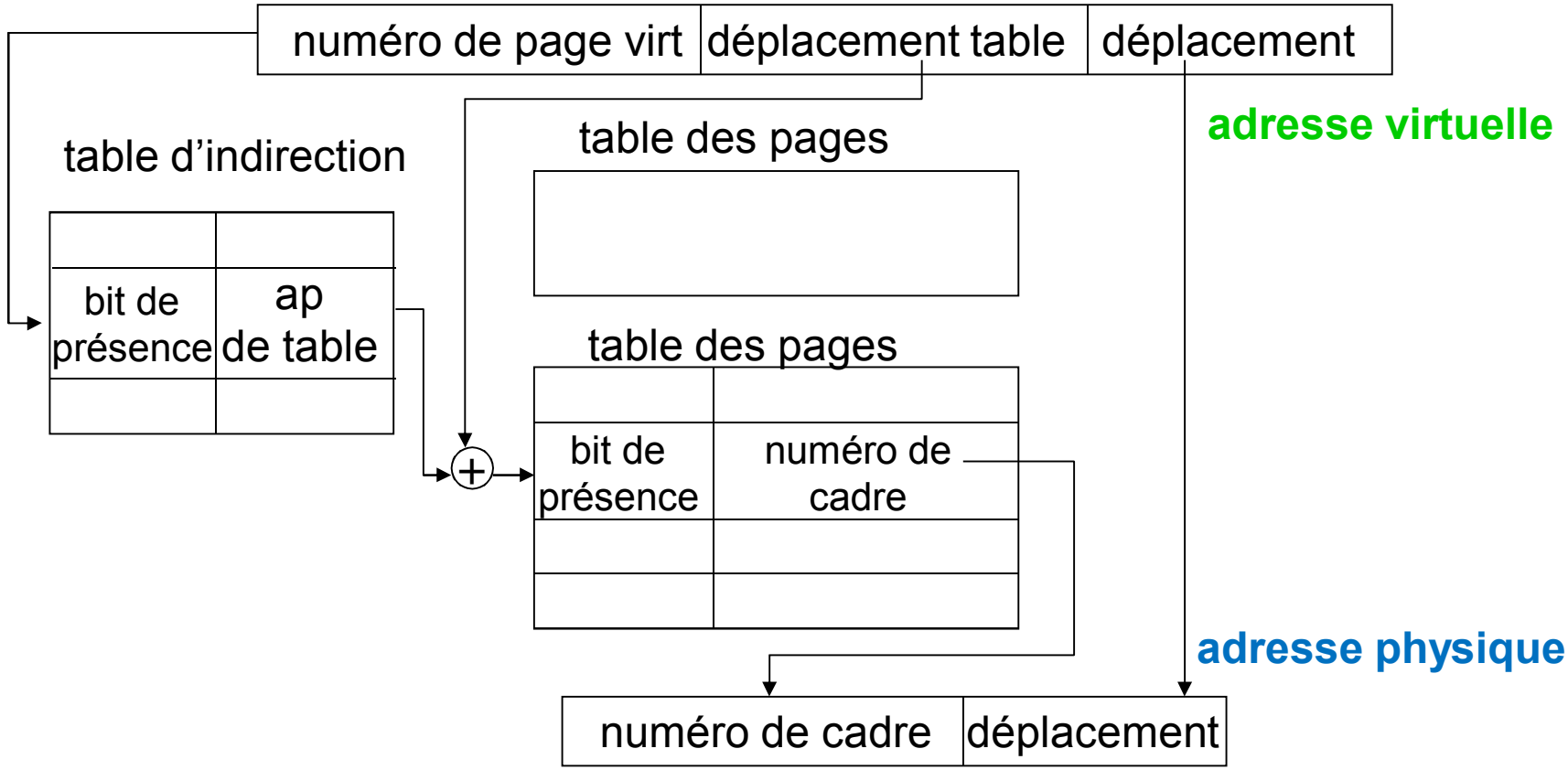


Transcodage des adresses virtuelles – exemple



Transcodage des adresses virtuelles (2)

- table des pages multi-niveaux



Défaut de page

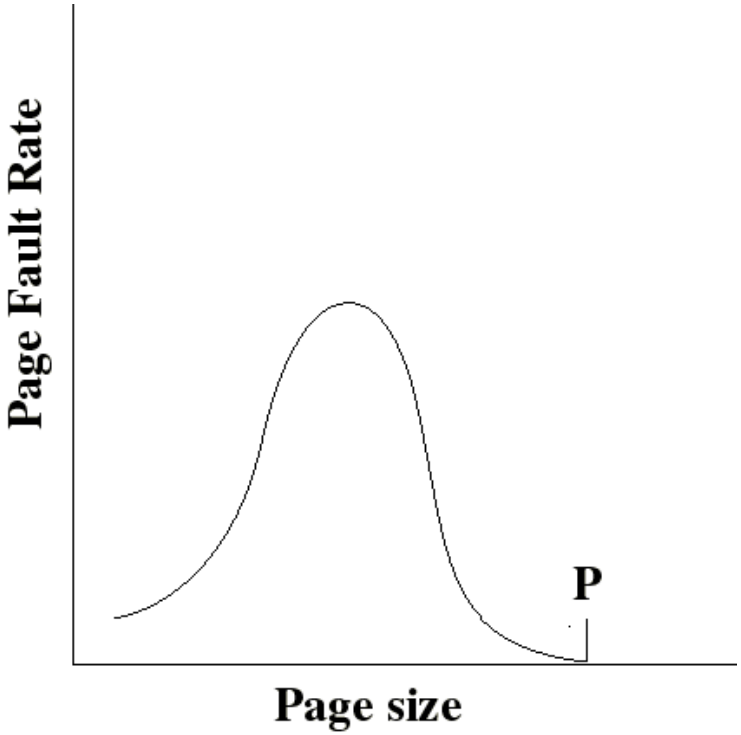
- contexte : un processus veut accéder à une page logique de la mémoire virtuelle
- définition : page non présente en mémoire physique
- traitement du défaut de page
  - cas 1 : cadre disponible en mémoire physique
    - solution : y charger la page logique
  - cas 2 : pas de cadre libre en mémoire physique
    - solution : remplacement de page

Algorithmes de remplacement de pages

- choix des pages à supprimer de la mémoire centrale pour faire place aux nouvelles pages
- algorithmes
  - remplacement optimal (OPT)
  - remplacement de page première entrée, première sortie (First In First Out - FIFO)
  - remplacement d'une page non récemment référencée (Not Recently Used - NRU)
  - remplacement de la page la moins récemment utilisée (Least Recently Used - LRU)
  - remplacement d'une page peu utilisée (Not Frequently Used - NFU)
- critère d'évaluation : *minimiser le taux de défaut de pages à long terme*

Taille d’une page

- avec une petite taille
  - mettre un grand nombre de pages en mémoire centrale
  - chaque page contient uniquement le code utilisé
  - peu de défauts de page
- en augmentant la taille
  - moins de pages peuvent être gardées en mémoire centrale
  - chaque page contient plus de code qui n’est pas utilisé
  - plus de défauts de page
  - mais le taux de défaut de pages diminue lorsque nous approchons le point P où la taille d’une page est celle d’un programme entier



Situation considérée normale

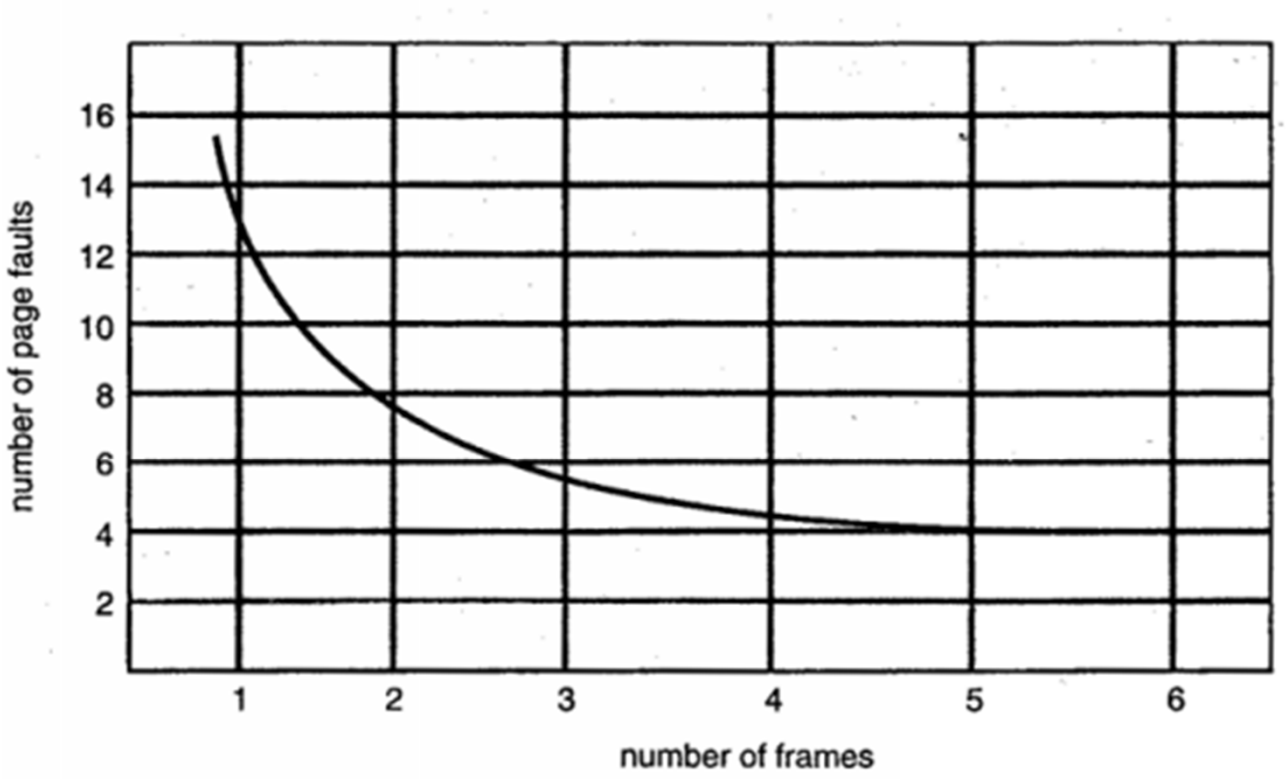
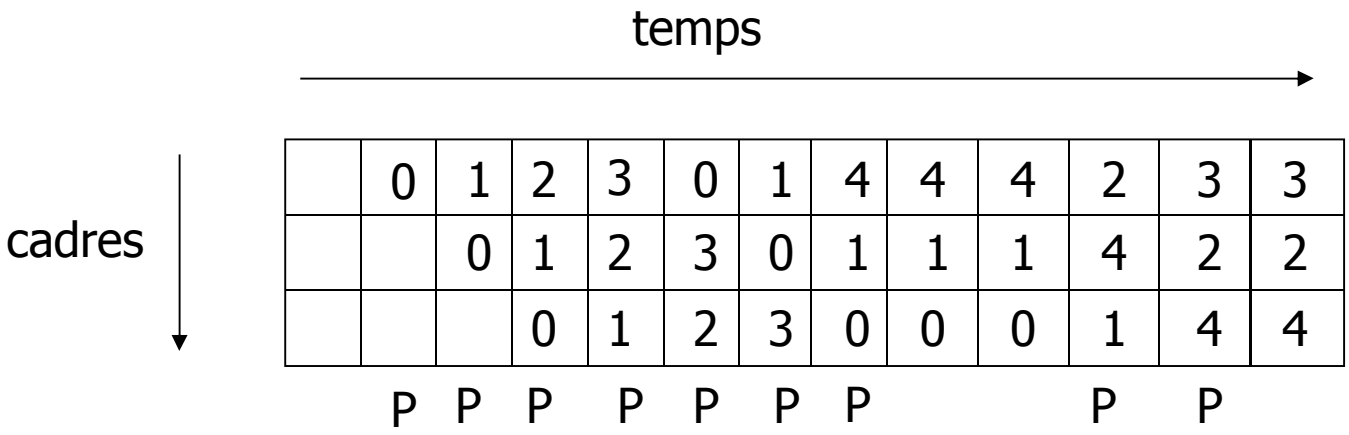


Figure 10.7 Graph of page faults versus the number of frames.

Anomalie de Belady

- plus de défauts avec plus de mémoire !
  - exemple d'algorithmes : FIFO, mais pas LRU, OPT
- exemple : FIFO
  - pages référencées : 0 1 2 3 0 1 4 0 1 2 3 4
  - nombre de cadres disponibles : 3 / 4

Anomalie de Belady – exemple



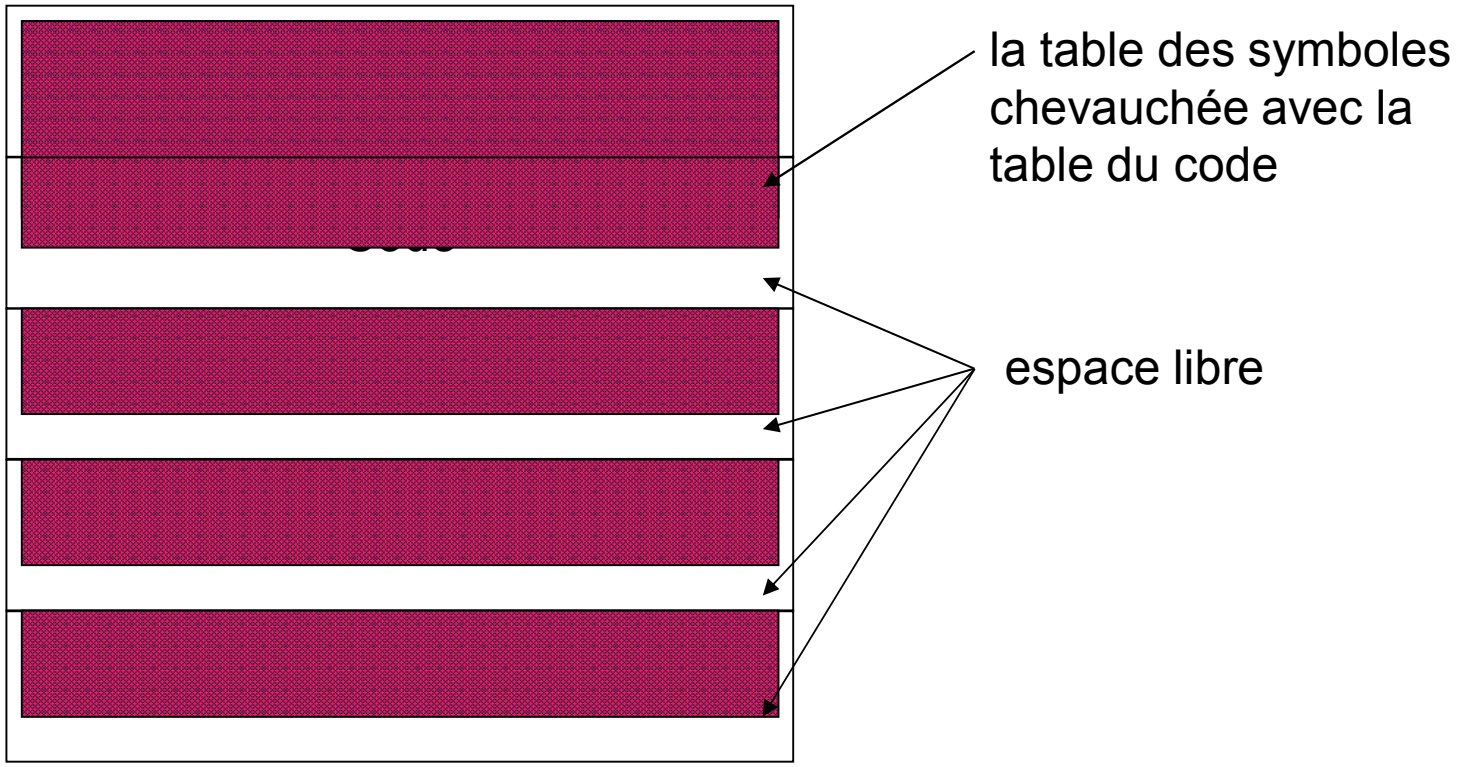
pages référencées : 0 1 2 3 0 1 4 0 1 2 3 4

	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
				0	0	0	1	2	3	4	0	1
	P	P	P	P			P	P	P	P	P	P

Segmentation (1)

- Intérêt : exemple des besoins en mémoire d'un compilateur
- informations construites au cours de la compilation
  - le code source sauvegardé
  - la table de symboles contentant le nom et les propriétés des variables
  - la table des constantes utilisées
  - l'arbre d'analyse syntaxique du programme
  - la pile utilisée pour les appels de sous-programmes

Segmentation (2)



Segmentation (3)

- espaces d'adresses indépendants = ***segments***
- segment
  - suite d'adresses contiguës
  - correspond à un découpage logique du programme (segment de pile, de données, de code…)
  - sa taille varie en cours d'exécution et indépendamment d'un autre segment

Segmentation (4)

- adresse dans une mémoire à segmentation
  - numéro de segment
  - adresse au sein du segment

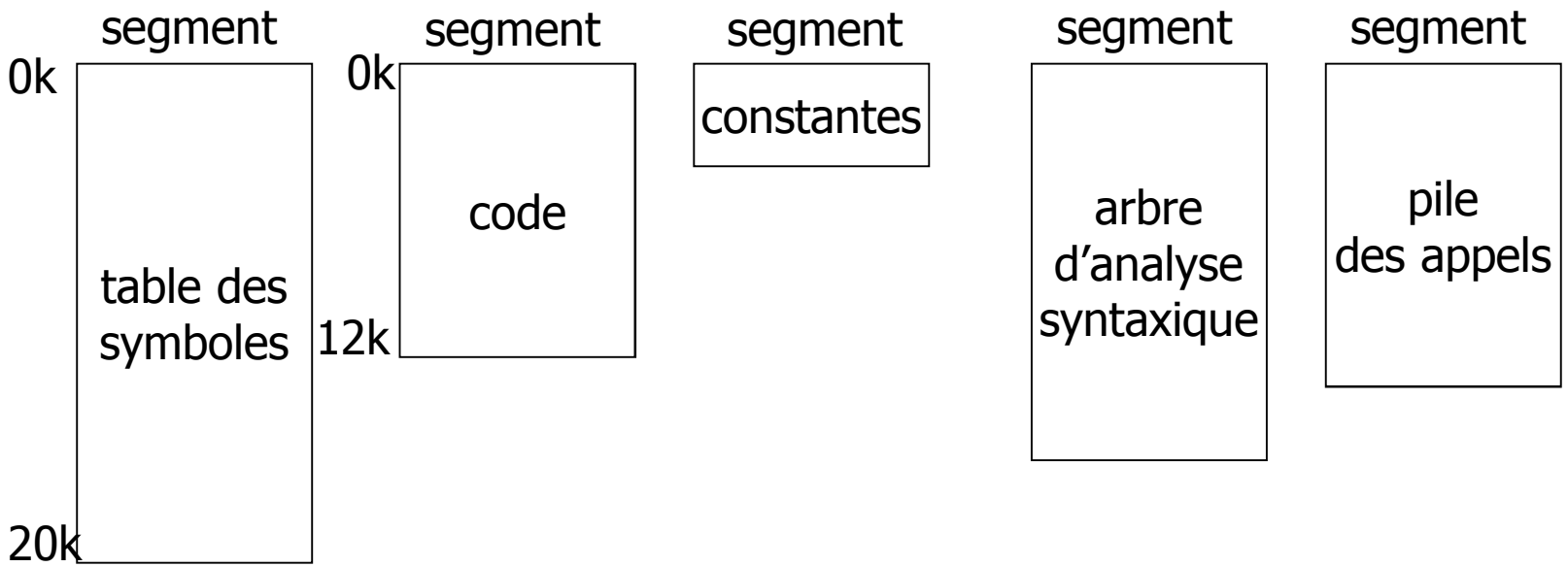
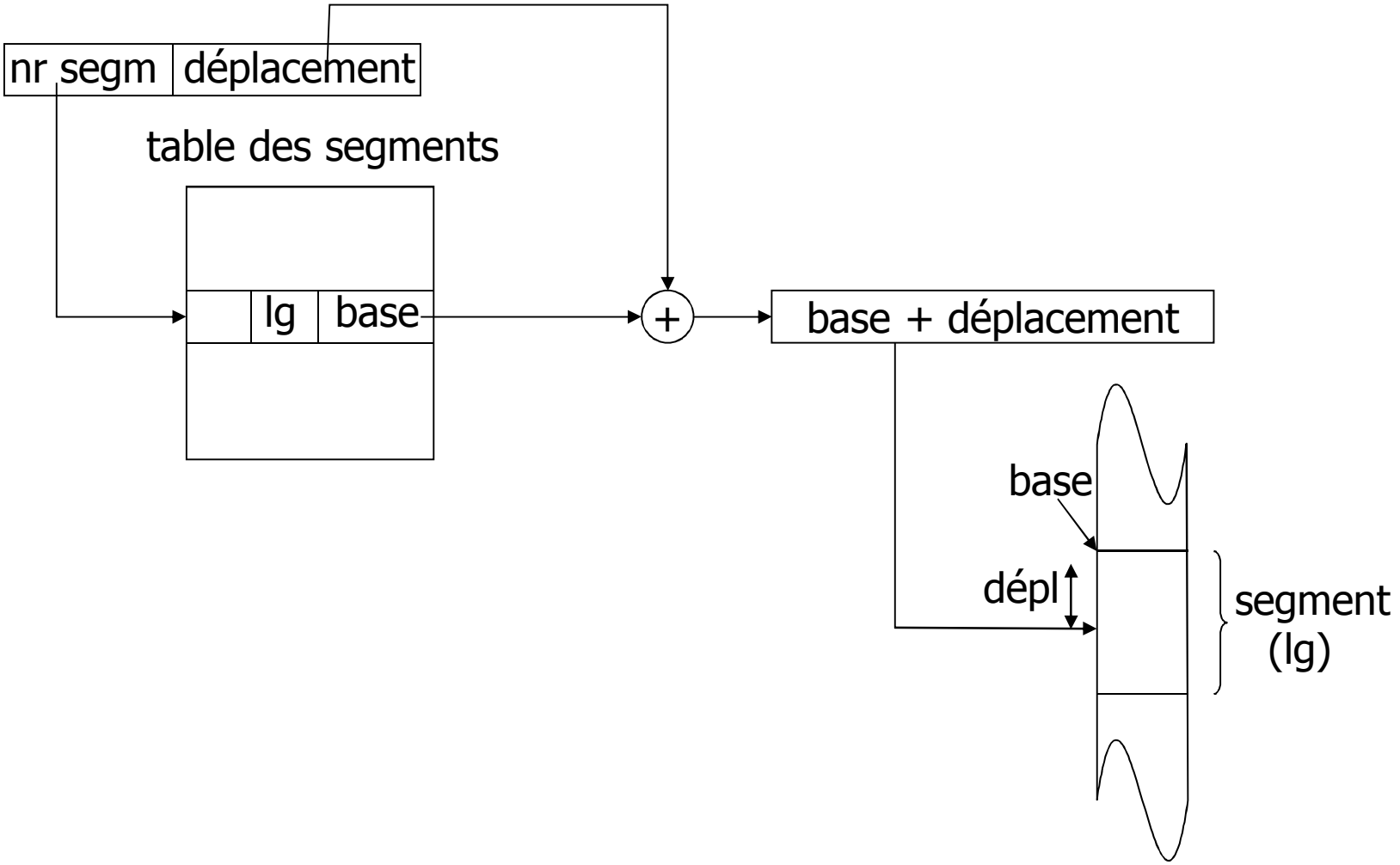


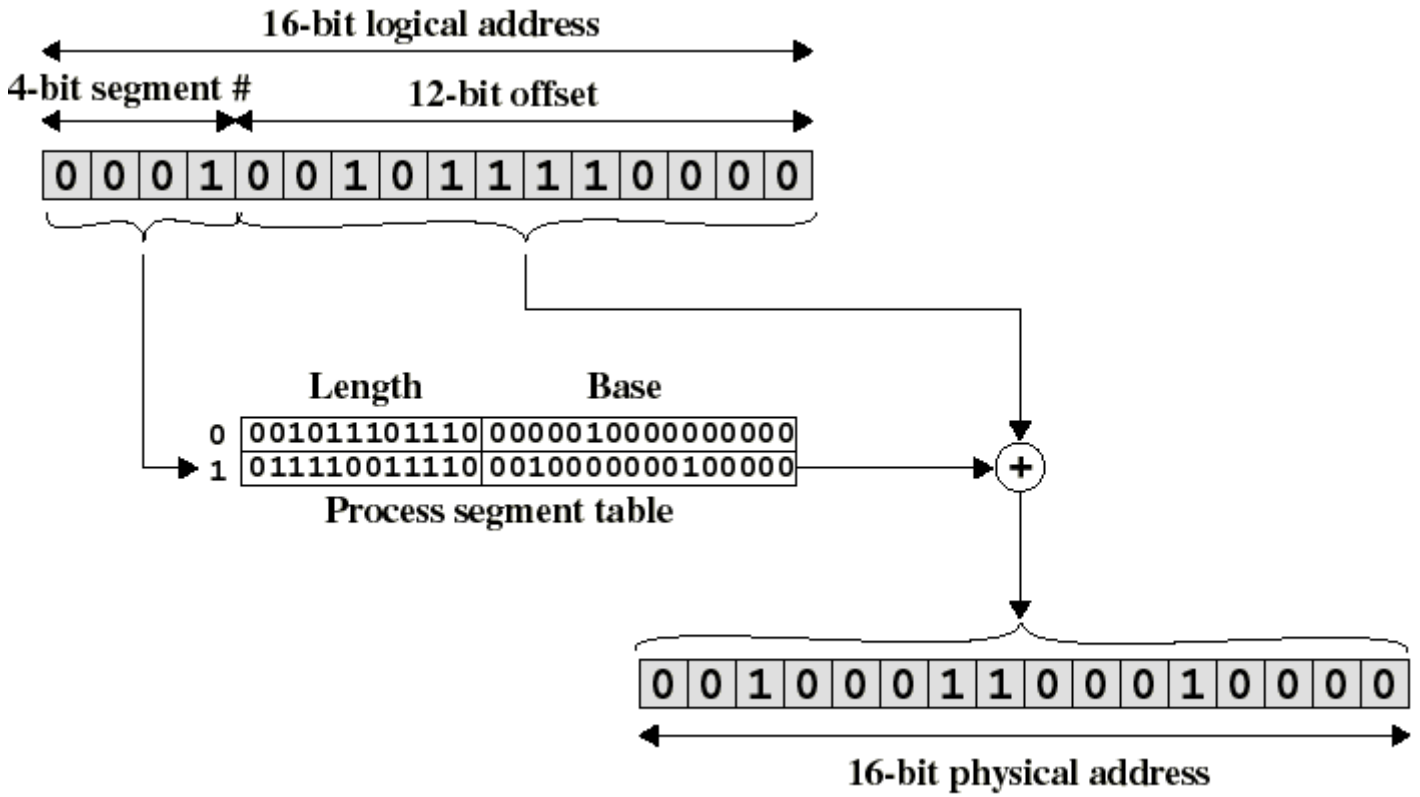
Table de segments

- le PCB du processus
  - contient un pointeur à l'adresse en mémoire de la table de segments
- la ***table de segments*** contient descripteurs de segments
  - adresse de base
  - longueur du segment
  - infos de protection

Transcodage des adresses en mémoire segmentée



Transcodage des adresses en mémoire segmentée - exemple



Segmentation – motivations (1)

- Gestion dynamique de la mémoire
  - on ne peut prévoir à l'avance la taille de certaines structures de processus
  - prévoir un trop grand espace mémoire peut être pénalisant
  - prévoir un espace trop petit peut limiter l'utilisation du programme

  - permettre l'évolution de la taille des structures de données dynamiques d'un processus

Segmentation – motivations (2)

- partage des bibliothèques chargées en mémoire
  - modularité
  - éviter la duplication de code dans plusieurs processus
  - plus simple d'avoir des bibliothèques partagées avec un système segmenté (qu'avec un système paginé)

Segmentation – motivations (3)

- gestion de la sécurité
  - chaque segment représente une entité logique : procédure, tableau, pile, etc.
  - des segments différents peuvent avoir des protections différentes
  - segment de procédure peut être déclaré en exécution seule : pas de lecture ni d'écriture
  - tableau de réels peut être déclaré en lecture écriture (pas d'exécution) : pas de branchement vers ce segment
  - type de protection utile pour détecter les erreurs de programmation

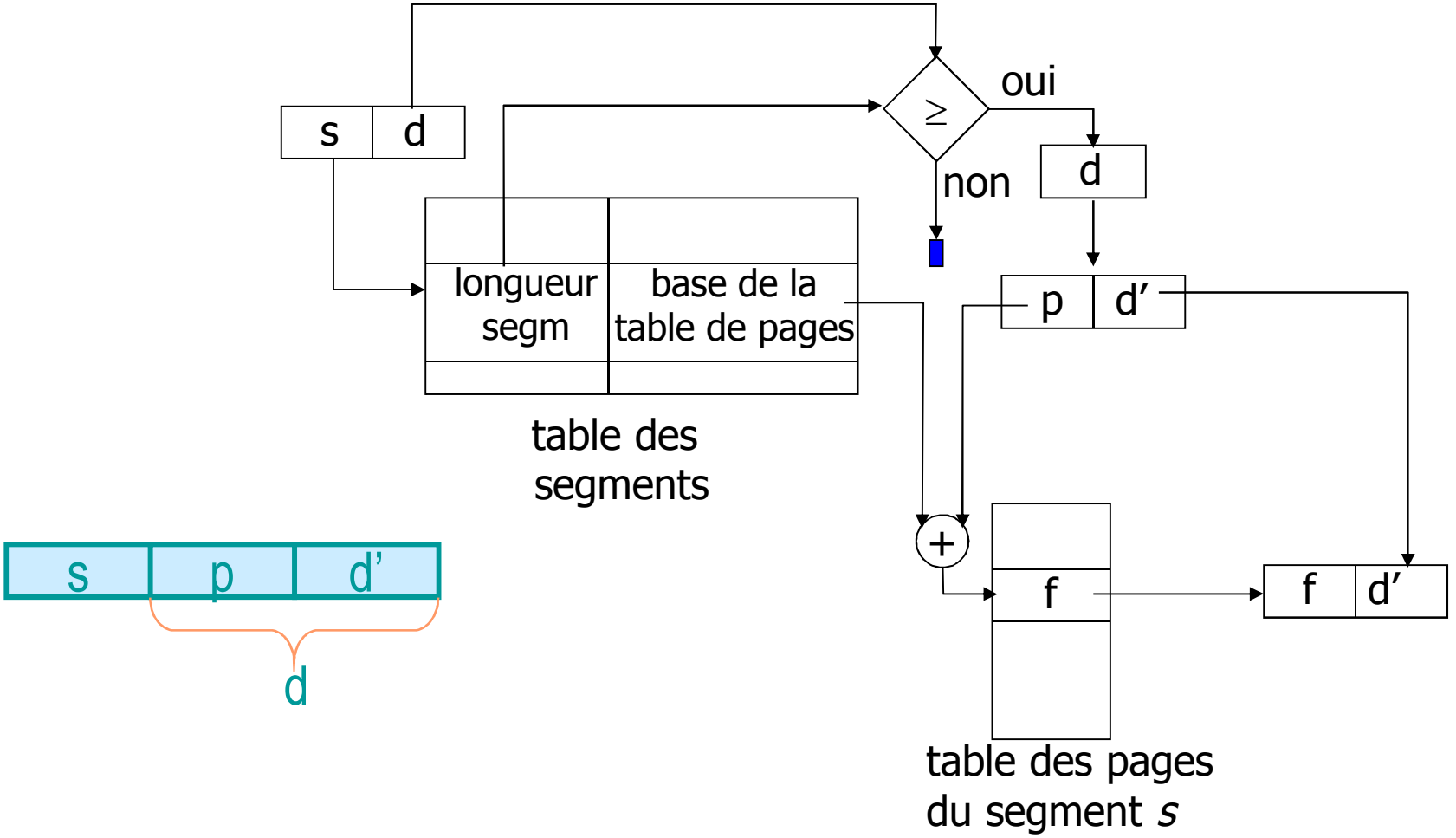
Pagination, segmentation – conclusions

- pagination
  - ??
- segmentation
  - ??

Segmentation et pagination

- les espaces d'adressage des processus sont divisés en segments et les segments sont paginés
- donc chaque adresse de segment n'est pas une adresse de mémoire, mais une adresse à la table de pages du segment

Transcodage des adresses en mémoire segmentée et paginée



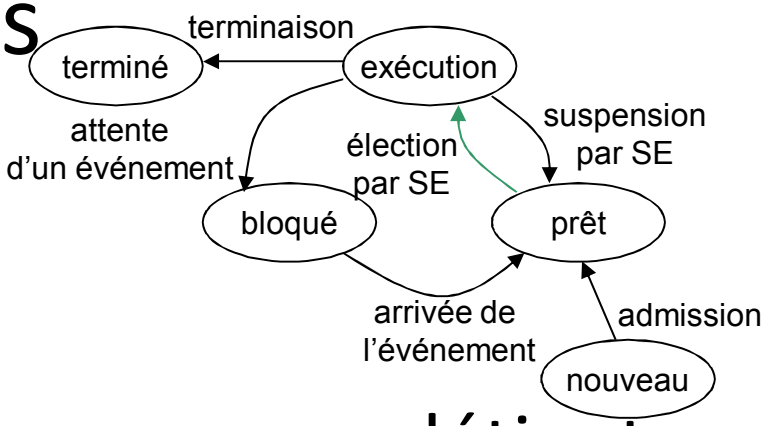
Conception des systèmes

Gestion processus

Ordonnancement de l'exécution

- définition
- contexte : multiprogrammation
- cycle d'exécution
- ordonnancement
  - préemptif
  - non préemptif

Sauvegarde des informations du processus



- en multiprogrammation, un processus détient l'UC de façon intermittente
- reprise cohérente

  $\Rightarrow$ sauvegarde des informations essentielles

  – PCB

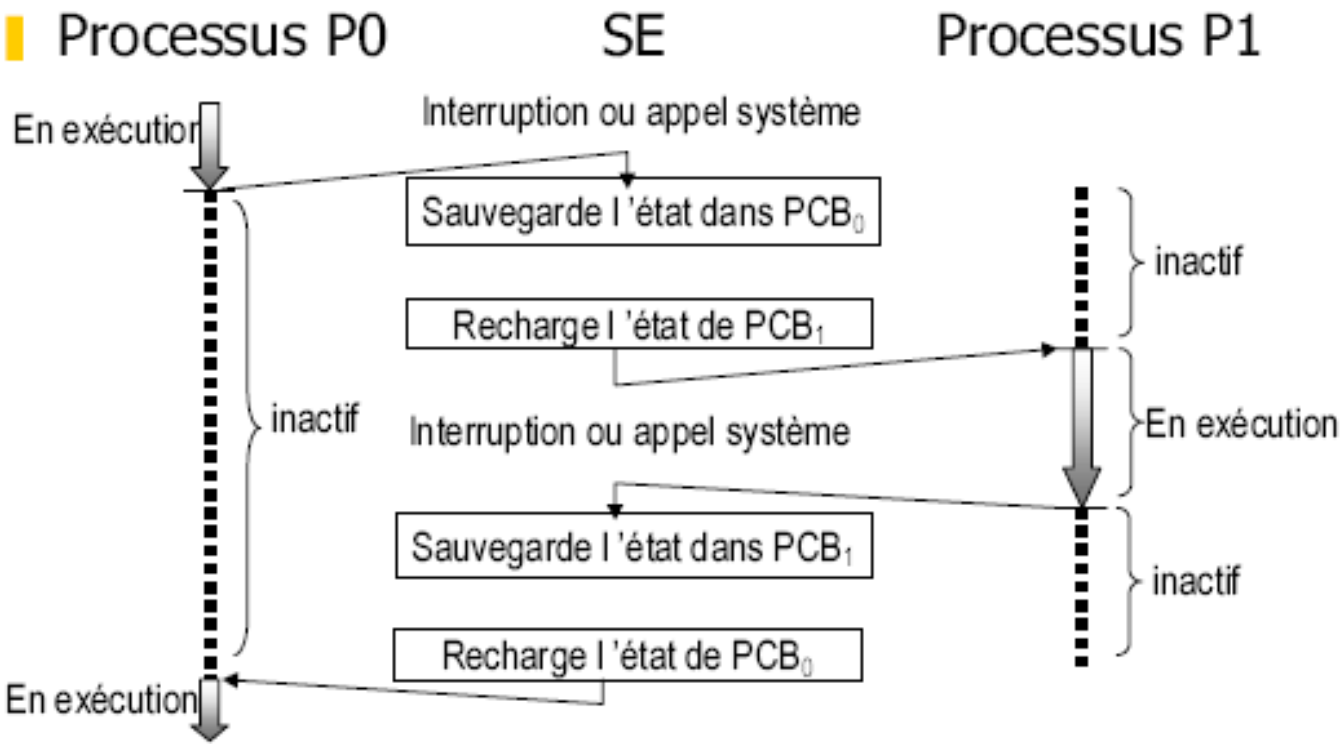
  – l'état des données du programme : *image du programme* en mémoire principale ou secondaire (le PCB pointera à cette image)

Création de processus

- allocation PID
- allocation image mémoire (clone de l'image mémoire du père)
- allocation + initialisation PCB (à partir du PCB du père)
- ajout PCB dans la table des processus
- ajout du processus créé dans la file des processus prêts
- blocage du père :
  - arrête le père et sauvegarde son contexte dans le PCB
  - état du père : bloqué
  - donne le processeur à un autre processus (commutation)
- renvoie le PID du processus créé

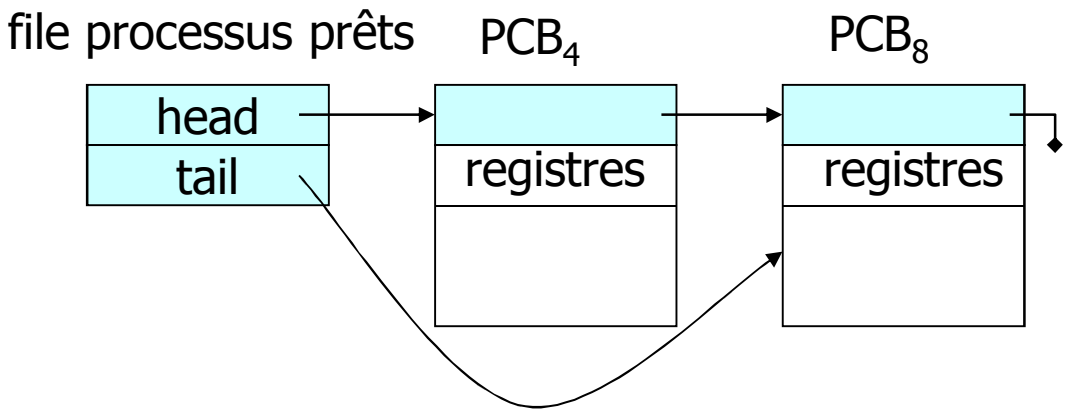
Commutation de l'UC entre processus

changement de contexte (context switching)



Ordonnancement des processus

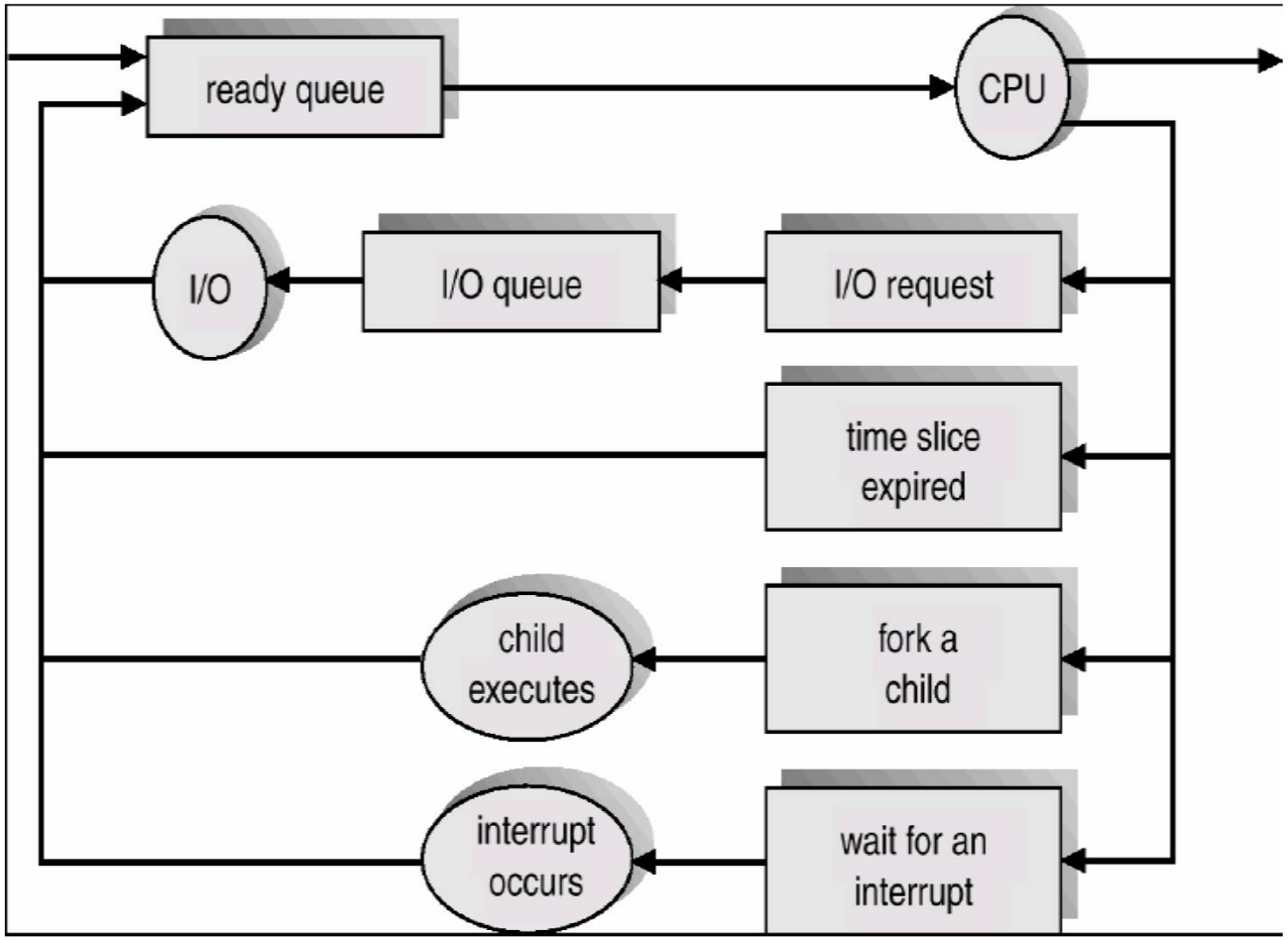
- objectif : avoir à chaque moment un processus en exécution afin de maximiser l'utilisation de l'UC
- files d'attente
  - pointeurs vers les PCBs



Files d'attente (1)

- les ressources d'ordinateur sont souvent limitées par rapport aux processus qui en demandent
- file de processus en attente associée à chaque ressource
  - file prêt : les processus en état prêt
  - files associées à chaque unité E/S
  - etc.
- en changeant d'état, les processus se déplacent d'une file à l'autre
  - les PCBs ne sont pas déplacés en mémoire pour être mis dans les différentes files : les pointeurs changent

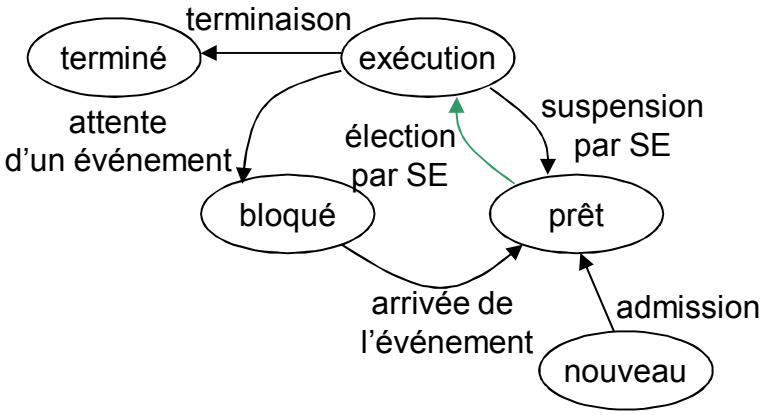
Files d'attente (2)



Ordonnancement des processus

- objectif : avoir à chaque moment un processus en exécution afin de maximiser l'utilisation de l'UC
- files d'attente
  - pointeurs vers les PCBs
- solution : quand l'UC devient disponible, le SE choisit un processus parmi ceux de la file d'attente des processus prêts pour être exécuté $\Rightarrow$ ***ordonnanceur de l'UC***

Quand invoquer l'ordonnanceur ?



- l'ordonnanceur intervient quand :
  - un processus se présente en tant que nouveau ou se termine ou
  - un processus exécutant devient bloqué
  - un processus change d'exécutant à prêt
  - un processus change de bloqué à prêt

⇒ tout événement dans un système cause une interruption de l'UC et l'intervention de l' ordonnanceur, qui devra prendre une décision concernant quel processus aura l'UC après

- préemption : si on enlève l'UC à un processus qui l'avait et peut continuer à s'en servir
  - dans les premiers deux cas, il n'y a pas de préemption

Critères d'ordonnancement

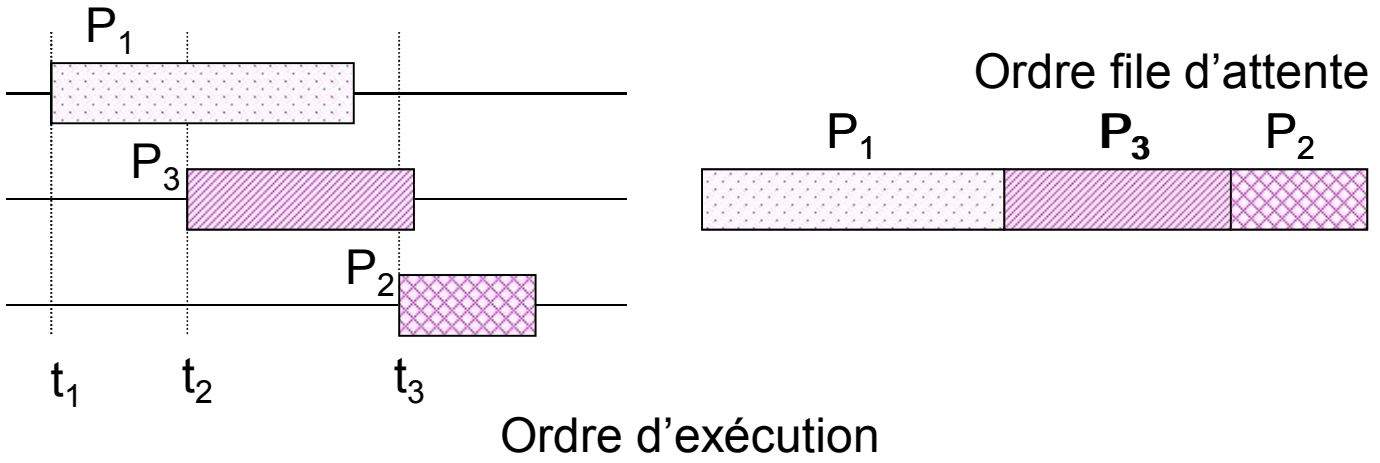
- situation :
  - plusieurs processus prêts (file *prêt*)
  - quand l'UC devient disponible lequel des processus choisir pour exécution ?

Critères

- l'*utilisation de l'unité centrale* : à maximiser
- le *débit* (throughput) : à maximiser
- le *délai de rotation* (*turnaround*) : à minimiser
- le *temps de réponse* : à minimiser
- le *temps d'attente* : à minimiser

Algorithmes d'ordonnancement (1)

- premier arrivé, premier servi (FCFS)

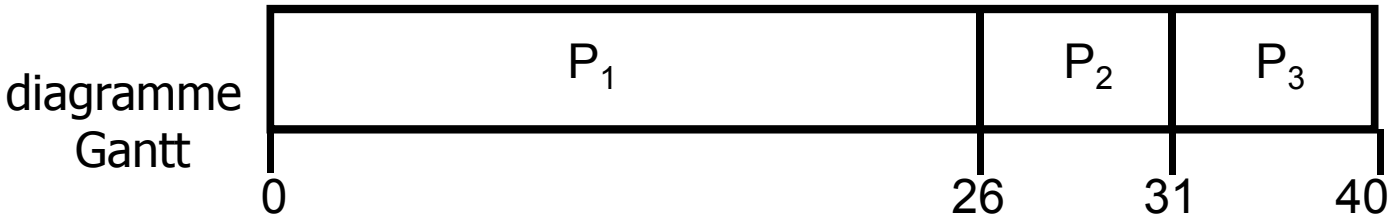


FCFS – exemple (1)

Exemple:

Processus	Temps de cycle
P_1	26
P_2	5
P_3	9

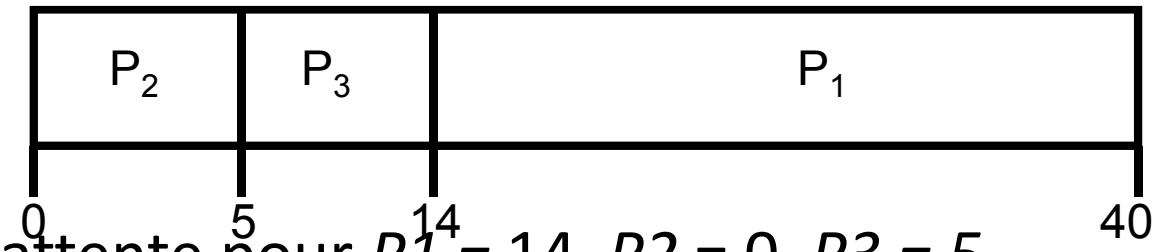
- les processus arrivent au temps 0 dans l'ordre P_1 , P_2 , P_3



- temps d'attente pour *P*1= 0 *P*2= 26 *P*3= 31
- temps moyen d'attente : (0 + 26 + 31)/3 = 19
- utilisation UC : 100%
- débit : 3/40 = 0,075
  - 3 processus complétés en 40 unités de temps
- temps moyen de rotation : (26+31+40)/3 ≅ 32,3

FCFS – exemple (2)

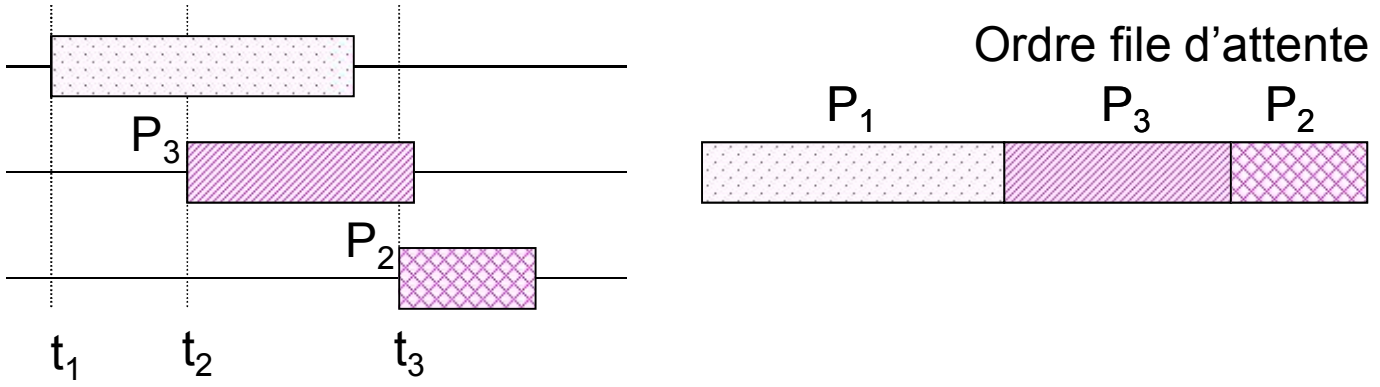
- les processus arrivent au temps 0 dans l'ordre P_2, P_3, P_1



- temps d'attente pour *P1* = 14 *P2* = 0 *P3 = 5*
- temps moyen d'attente : (14 + 0 + 5)/3 ≅ 6,3
  - beaucoup mieux !
  - donc pour cette technique, le temps d'attente moyen peut varier grandement
- utilisation UC : 100%, débit : 3/40=0,075
  - mêmes
- temps moyen de rotation : (5+14+40)/3 ≅ 19,6

Algorithmes d'ordonnancement (2)

- travail le plus court d'abord (SJF)



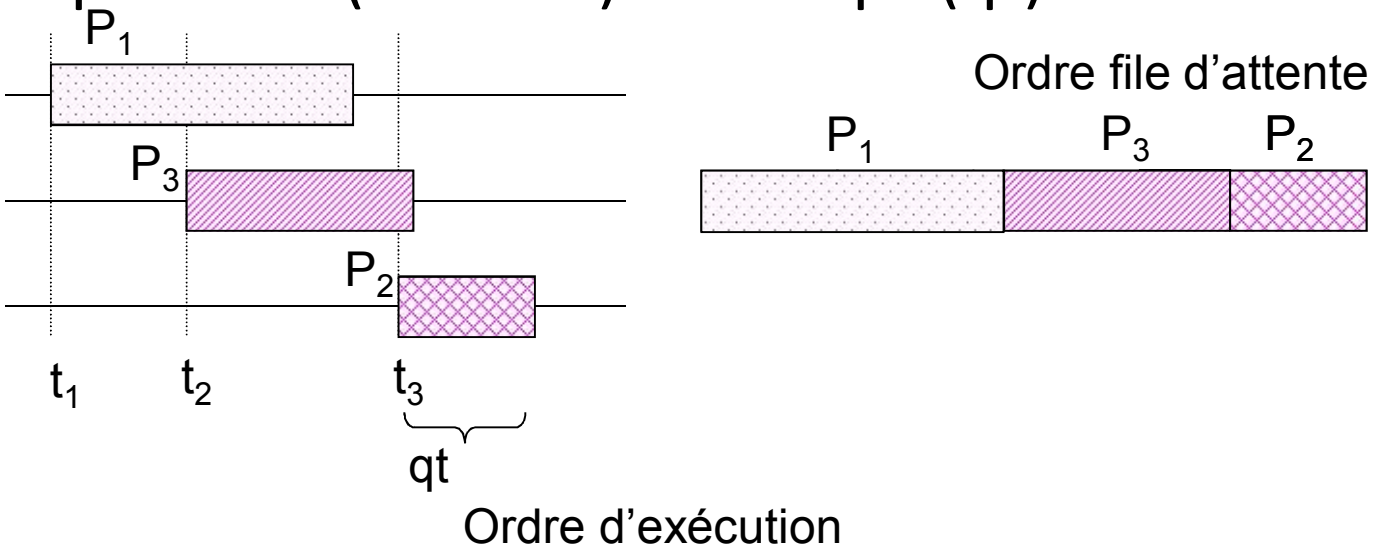
Ordre d'exécution

Algorithme SJF - critique

- difficulté d'estimer la longueur à l'avance
- les processus longs souffriront de ***famine*** lorsqu'il y a un apport constant de processus courts
- la préemption est nécessaire pour des environnements à temps partagé
  - un processus long peut monopoliser l'UC s'il est le 1er à entrer dans le système et il ne fait pas d'E/S

Algorithmes d'ordonnancement (3)

- ordonnancement à tourniquet (circulaire ou Round Robin)
  - quantum (tranche) de temps (qt)

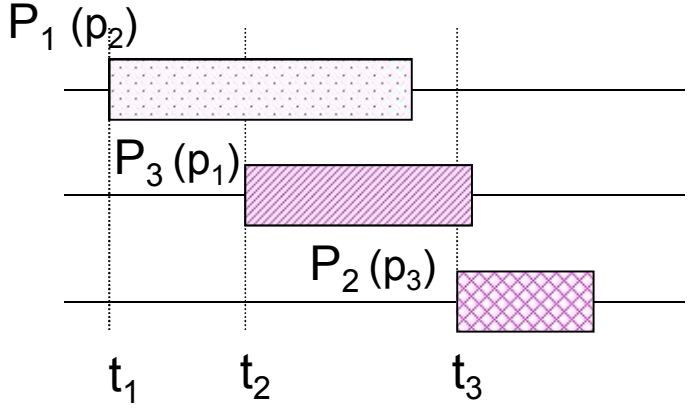


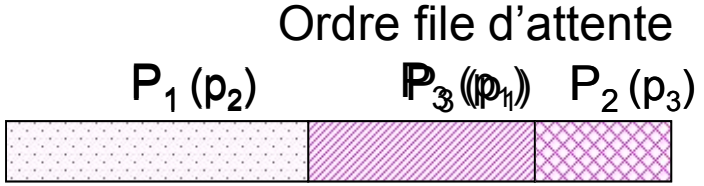
Priorité

- 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'UC est donnée au processus prêt avec la plus haute priorité
  - avec ou sans préemption
  - il y a une **file *prêt*** pour chaque priorité

Algorithmes d'ordonnancement (4)

- ordonnancement avec priorité





$p_1 < p_2 < p_3$

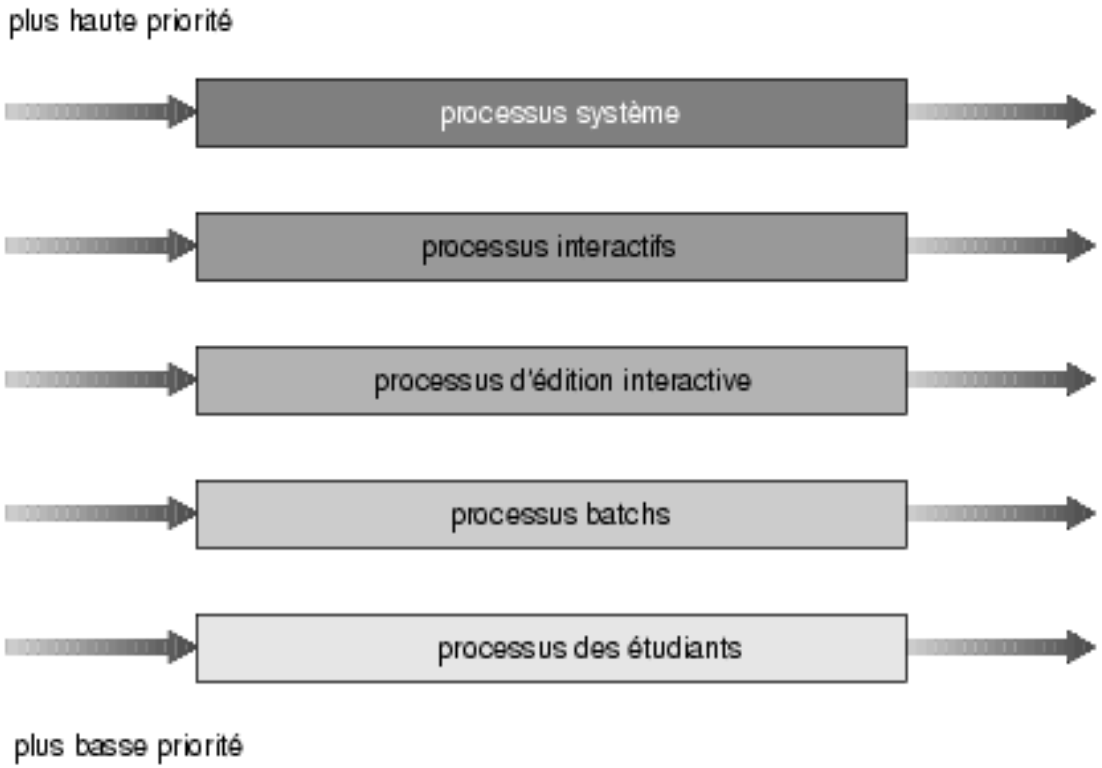
Ordre d'exécution

Problème possible avec priorités

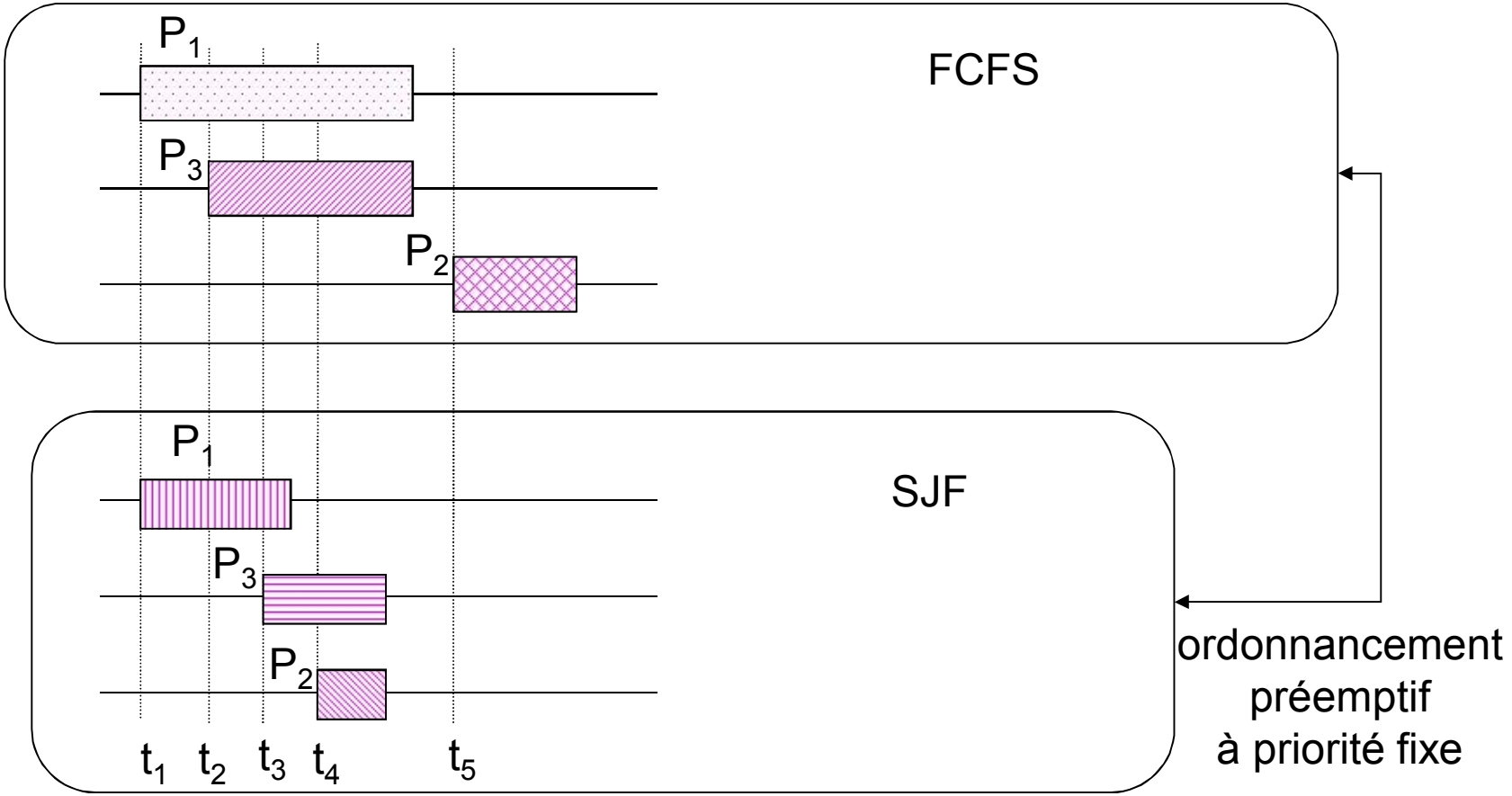
- ***famine*** : les processus moins prioritaires n'arrivent jamais à s'exécuter
- Solutions :
  - modifier la priorité d'un processus en fonction de son âge et de son historique d'exécution ⇒ ***vieillissement***
  - assignation dynamique des priorités
  - grouper les processus en catégories de priorités

Algorithmes d'ordonnancement (5)

- ordonnancement à files multiniveaux (Multilevel Queue)



Ordonnancement à files multiniveaux



Algorithmes d'ordonnancement (6)

- ordonnancement avec files d'attente multiniveaux et à retour (Multilevel Feedback Queue)
  - ordonnancement avec files multiniveaux
  - les processus peuvent passer d'une file à une autre
    - déplacer un processus vers une file d'attente de priorité plus/moins élevée
  - empêche la famine
  - paramètres : le nombre de files, l'algorithme d'ordonnancement pour chaque file, méthode de déplacement d'un processus

En pratique

- les méthodes d'ordonnancement sont toutes utilisées en pratique (sauf plus court servi *pur* qui est impossible)
- les SE sophistiqués fournissent au gérant du système une bibliothèque de méthodes, qu'il peut choisir et combiner au besoin après avoir observé le comportement du système
- pour chaque méthode, plusieurs paramètres sont à définir : p.ex. durée du quantum, coefficients, etc.

Conception des systèmes

Allocation ressources

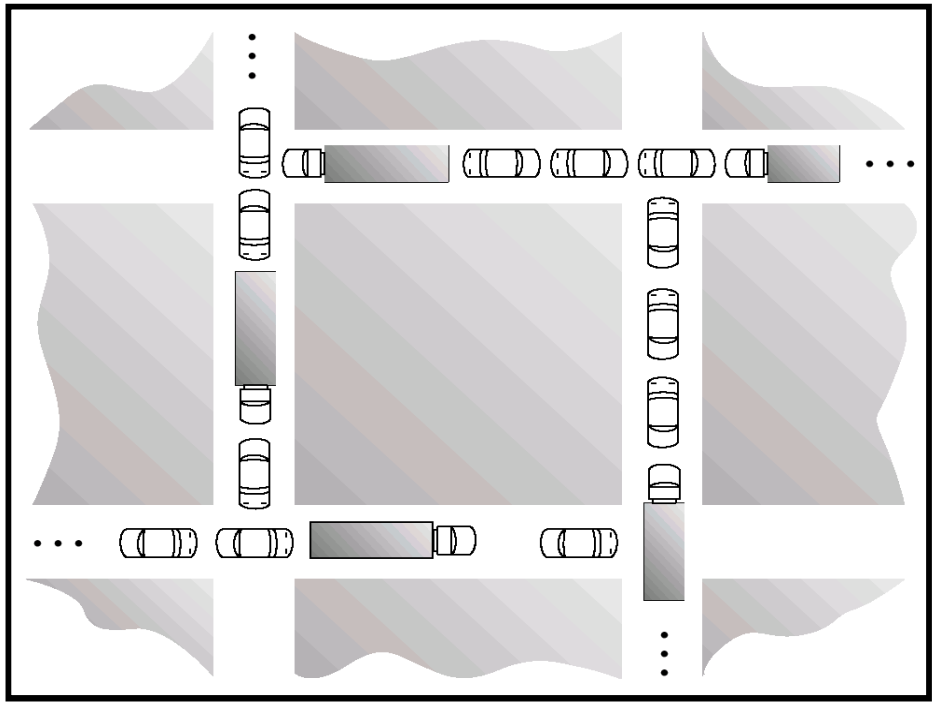
Types de ressources

- ***ressource préemptible***
  - peut être retirée sans risque au processus qui la détient
  - exemple : la mémoire
- ***ressource non préemptible***
  - exemple : l'imprimante

Interblocage

- un ensemble de processus est en ***interblocage*** si chaque processus attend un événement que seul un autre processus de l'ensemble peut l'engendrer
  - la plupart du temps, événement = libération d'une ressource détenue par un autre processus

Exemples



Sémaphores

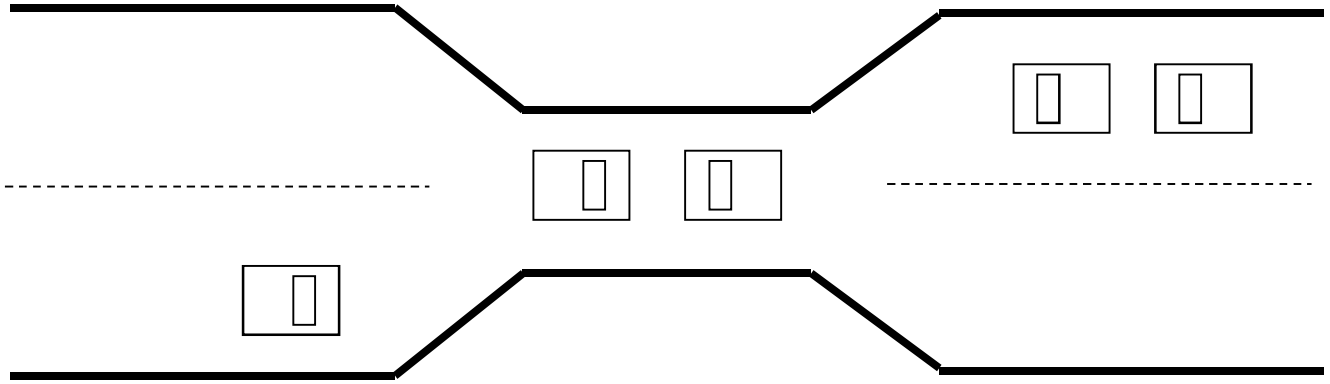
P_0	P_1
P(A)	*P(B)*
P(B)	*P(A)*

A et *B* initialisés à 1

Interblocages – caractéristiques

- conditions nécessaires
  - exclusion mutuelle
  - détention et attente
  - pas de préemption
  - attente circulaire

Exercice : passage sur un pont



- des voitures sont dans une situation d'interblocage sur un pont
- Comment les différentes conditions sont-elles satisfaites ?

Interblocages – solutions

Stratégies pour traiter les interblocages

- les ignorer (exemple : Linux)
- détection et correction
- prévention
- empêchement

Détection des interblocages

- cas d'un seul type de ressources
  - modélisation sous la forme d'un graphe
    - des demandes d'allocation de ressources
    - des détentions de ressources
  - détection de cycle
- cas de plusieurs types de ressources
  - construire
    - matrice d'allocations courantes
    - matrice de demandes
    - matrice de ressources disponibles

Correction d'interblocages

- au moyen de la préemption des ressources
- au moyen de rollback (basé sur des points de reprise - checkpoints)
- au moyen de suppression (terminaison) de processus

Prévention des interblocages

= éliminer au moins une des conditions

- exclusion mutuelle
- détention et attente
- pas de préemption
- attente circulaire

Elimination de la condition « occupation et attente »

- un processus demande toutes ses ressources en même temps
- le système lui alloue sur une base tout ou rien
- inconvénient : beaucoup de processus ne connaissent pas à l'avance les ressources dont ils ont besoin

Elimination de la condition « pas de préemption »

- un processus qui attend pour une ressource doit libérer toutes les ressources qu'il détient
- inconvénients
  - coûteux
  - famine
  - l'état de certaines ressources ne peut pas être sauvegardé

Elimination de la condition « attente circulaire »

- allocation des ressources dans un ordre bien défini
- $R = \{r_1, \ldots, r_n\}$ l'ensemble des types de ressources
- fonction $f : R \rightarrow \mathrm{N}$
- règle : un processus qui détient une ressource r_i peut demander une ressource r_j ssi $f(r_i) < f(r_j)$ (ordonner les ressources)
  = empêche l'apparition de cycles dans le graphe des ressources
- inconvénients
  – nombres élevés
  – ajout de nouvelles ressources

Empêchement des interblocages

- ***état sûr***
  - séquence <P_0, P_1, …, P_n> sûre
    - si ∀i, les ressources que P_i pourrait encore demander peuvent être satisfaites par celles disponibles et celles des processus P_j, j < i
- Exemple : 12 ressources disponibles

cas 1 : <P_1, P_0, P_2> séquence sûre ?

Processus	Besoin max	allouées
P_0	10	5
P_1	4	2
P_2	9	3

cas 2 : <P_1, P_0, P_2> séquence sûre ?

Processus	Besoin max	allouées
P_0	10	5
P_1	4	2
P_2	9	2

Empêchement

- algorithme permettant une allocation ssi celle-ci laisse le système dans un état sûr
  - un interblocage est un état non sûr
  - un état non sûr n'implique pas un interblocage
  - un processus demandant une ressource peut attendre même si elle est disponible
- algorithme du banquier (Dijkstra, 1965)

Algorithme du banquier (1)

- supposition : un seul type de ressources
- soit
  - dem_i le vecteur des demandes de P_i
  - *disponible* le nombre de ressources disponibles
  - $allouées_i$ le nombre de ressources allouées à P_i
  - max_i le maximum de ressources requises par P_i
  - $besoin_i$ les besoins restants de P_i (max_i – $allouées_i$)

Algorithme du banquier (2)

- lors d'une demande d'allocation du processus P_i
  - si $dem_i > besoin_i$ alors erreur
  - si $dem_i > disponible$ alors attente
  - le système simule l'allocation
    - $disponible = disponible - dem_i$
    - $allouées_i = allouées_i + dem_i$
    - $besoin_i = besoin_i - dem_i$
  - si l'état résultant est sûr alors l'allocation est effectuée

Algorithme du banquier (3)

Algorithme pour vérifier qu'un état est sûr

- *travail* = *disponible* et $\forall P_i$: *fini*[i] = *faux*
- pour $\forall$i, tel que *fini* [i]= faux et

  $$besoin_i \leq travail$$

  - *travail* = *travail*+*allouées*$_i$
  - *fini* [i] = vrai
- si $\forall$i, *fini* [i] = vrai alors l'état est sûr

Algorithme du banquier (4)

- inconvénients
  - nombre fixe de ressources
  - nombre fixe de demandeurs
  - besoin d'informations supplémentaires (par ex. maximum de ressources nécessaires)

Bilan du cours (conception des systèmes d'exploitation)

- Gestion mémoire
  - partionnement
  - pagination
  - segmentation
- Gestion processus : ordonnancement de l'exécution
  - algorithmes (FCFS, SJF, RR, priorité)
- Allocation des ressources
  - interblocage
  - stratégies de traitement