

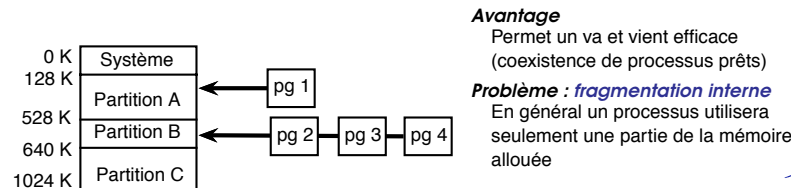


### Solution directe

allouer à chaque processus un bloc (distinct) de mémoire, de taille suffisante pour contenir son image mémoire

#### Partitions fixes (OS 360 MFT)

- La mémoire est divisée en zones de taille fixe (*partitions*).
- Une file d'attente est associée à chaque partition
- Quand un processus est lancé, le SX lui alloue la plus petite partition pouvant contenir son image mémoire (*best fit*)

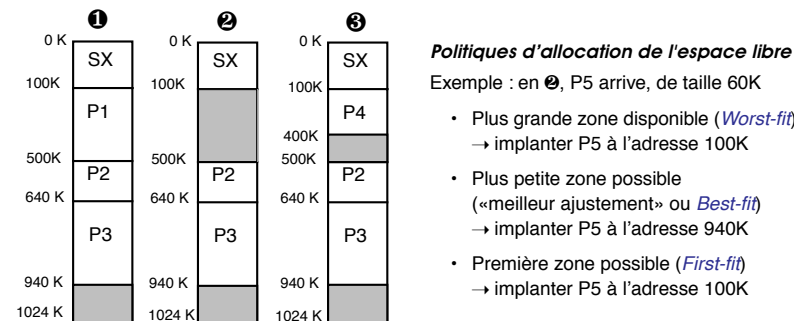


5 / 69

### Eviter la fragmentation interne

#### → Partitions variables

- Allouer à chaque processus une partition de la taille exacte de son image mémoire
- En fin d'exécution, l'espace occupé est libéré

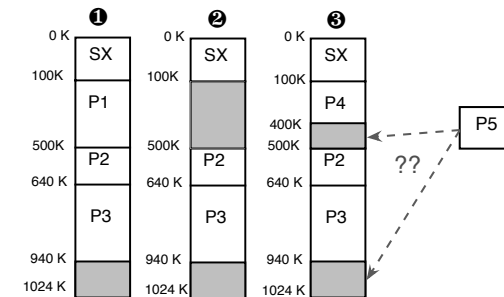


6 / 69

### Difficulté : fragmentation externe (émiettement)

L'espace libre est suffisant, mais divisé en nombreux fragments de petite taille, inutilisables séparément (*miettes* (garbage))

Exemple : en ③, que faire si P5 arrive, de taille 150K ?



→ (*Re*)compactage : regrouper les zones libres en un seul bloc  
⇒ recopie, réimplantation...

7 / 69

### Bilan (provisoire)

#### Difficultés

- zones de taille fixe : manque de souplesse (tailles prédéfinies → *fragmentation interne*)
- zones de taille variable : *fragmentation externe*
  - recompactage : simple, mais coûteux en temps
  - traitement algorithmique : gérer les zones libres (recherche ; fusion de zones libres voisines)
    - lourd/complexé en général
      - table d'occupation de la mémoire (recherche lente, fusion ok)
      - listes de zones libres, classées par tailles (fusion lente)
  - possible lorsque l'ensemble des tailles demandées est régulier (puissances de 2) ou fixe (évite l'émiettement)  
Exemples : VM 370 (IBM) ; structures de données noyau Solaris

#### Intérêt

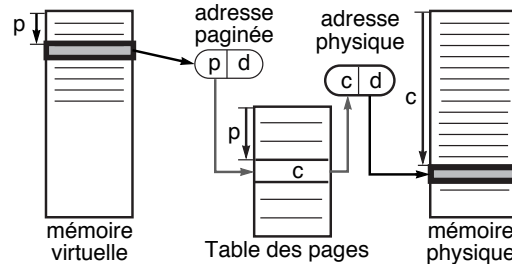
- d'éviter la fragmentation externe (zones fixes)
- de disposer d'un mécanisme d'allocation non contiguë...

8 / 69

- 1 Gestion de la mémoire physique
  - Allocation contiguë
  - Allocation fragmentée
- 2 Mémoire virtuelle
- 3 Synthèses
- 4 Exemples

**Idée** : fractionner l'image mémoire d'un processus  
 ⇒ gérer une carte d'implantation de l'image mémoire (*table des pages*)

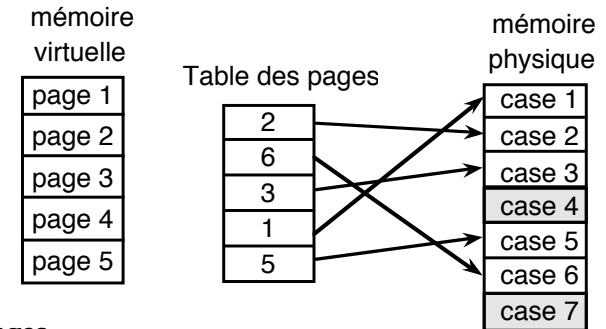
- L'image mémoire (virtuelle) est découpée en *pages*
- La mémoire physique est découpée en *cases*
- Cases et pages ont une **taille P** fixée et identique



Evaluation de l'adresse virtuelle ( $A_v$ )

$A_v = (n^\circ \text{ de page, déplacement dans la page}) = (p, d)$ ,  
 avec  $p = A_v \text{ div } P$  et  $d = A_v \bmod P$  (efficace avec  $P = 2^k$ )

## Exemple



## Avantages

- pas de fragmentation externe
- fragmentation interne : 1/2 page par processus, en moyenne  
 → ok si la taille des pages est assez petite (souvent 2K-8K)
- (vu plus loin) **interprétation des adresses** (adressage indirect)  
 → possibilité de
  - déplacer une page (réimplantation dynamique)
  - partager une page
  - contrôler l'accès à une page

En général, une table des pages par (image mémoire de) processus,  
 référencée dans le descripteur du processus

## Difficulté

Table des pages en RAM ⇒ temps d'accès mémoire doublé  
 → mécanismes accélérateurs nécessaires

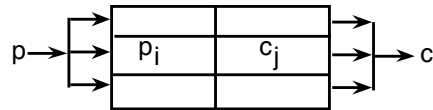
## Idée naïve

conserver la table des pages en mémoire rapide (tableau de registres)

La taille d'une table des pages est importante  
 (adresses de 32 bits → espace de 4Gio, pages de 4Kio →  $10^6$  pages)  
 → coût prohibitif

- matériel
- commutation de contexte

→ mécanisme matériel spécifique :  
*registre associatif* (TLB : translation lookaside buffer)



- contient les derniers accès (page → case) (LRU) : *principe de localité*
- recherche en parallèle
- en cas d'échec dans la recherche associative
  - On consulte la table complète (en mémoire)
  - On met le registre associatif à jour, avec la nouvelle référence
- LRU est une très bonne heuristique (mesures sur des applications types)  
 16 entrées → 80% de succès, 32 (80486) → 98%, 512 → 99%, 4096 → 99,99%

13 / 69

Mémoire vue comme un *ensemble* de zones (*segments*)

- de longueurs variables
- sans ordre relatif

### Idée

Un segment est destiné à contenir des objets de *même* « type »

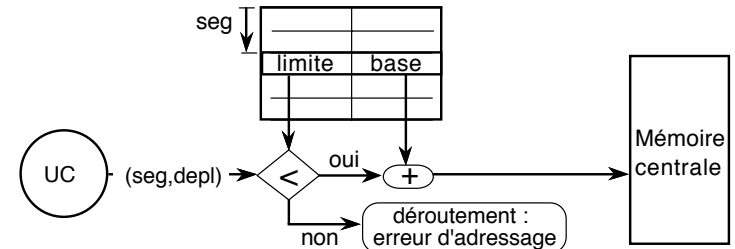
→ possibilité de contrôles et protection adaptés au type :  
 Pour chaque segment

- contrôle des *accès*  
 opérations possibles limitées aux opérations spécifiques au type des objets du segment
- contrôle de l'*adressage*  
 Ex : si le segment contient un tableau, on conserve sa dimension

14 / 69

Analogie à la pagination :

- Adresse segmentée : (id. segment, dépl. dans le segment)
- Mécanisme



- Mise en œuvre de la table des segments  
 cf table des pages : table en mémoire, registres associatifs

Différence : *fragmentation externe*

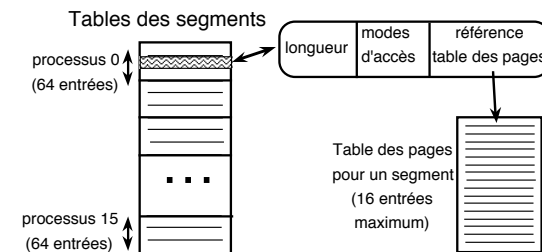
le recompactage est facilité (indirections), mais reste coûteux.

15 / 69

- conserver l'abstraction offerte par la segmentation,
- ... sans la fragmentation externe

Idée : *paginer les segments*

- une table des segments pour chaque processus
- une table des pages pour chaque segment



- 680x0 → 16 tables, pour 16 processus (maximum)
- Chaque table de segments contient 64 entrées
- Chaque table de pages contient 16 entrées

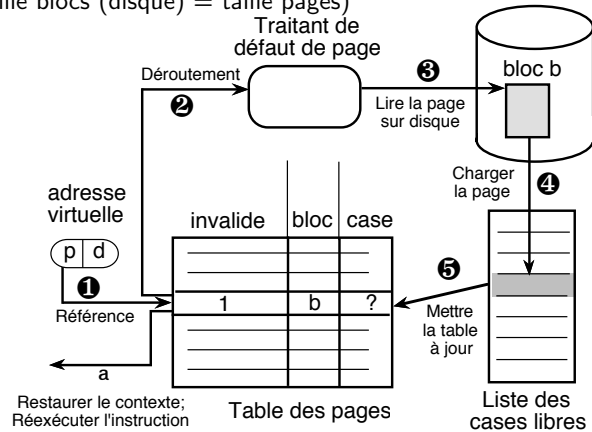
16 / 69



## Mécanisme de défaut de page (1/3)

### Chargement

- ajout d'une colonne (*invalide*) à la table des pages, indiquant pour chaque page si elle est chargée en mémoire
- ajout d'une *table de couplage* <n° page, n° bloc> (colonne *bloc*), qui localise la copie de chaque page en mémoire secondaire (taille blocs (disque) = taille pages)

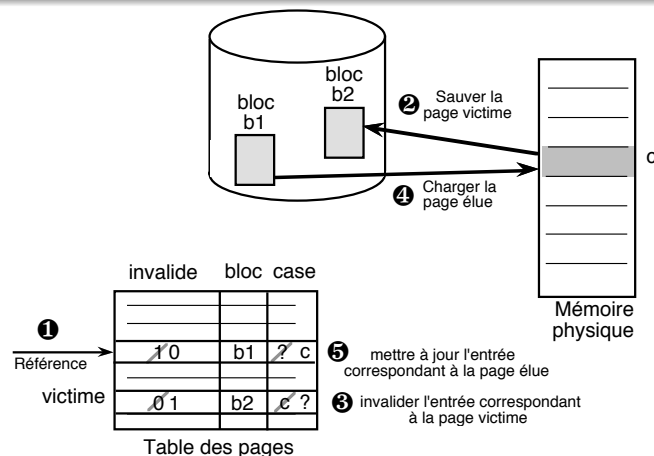


21 / 69

## Mécanisme de défaut de page (2/3)

### Remplacement

Va-et-vient « page à page » s'il n'y a plus de cases libres

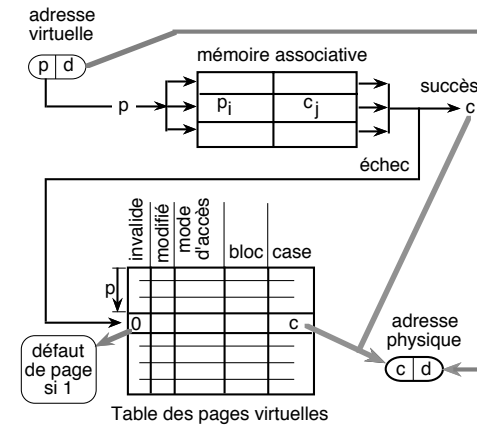


22 / 69

Remarque : le nombre d'E/S causées par un défaut de page double

## Mécanisme de défaut de page (3/3)

### Optimisation : mémoire associative (Translation Lookaside Buffer-TLB)



- invalide* : bit de présence (faux si page en mémoire)
- modifié* ("dirty bit") : indique si une écriture a été effectuée sur la page (utilisé pour le remplacement)
- mode d'accès* : opérations autorisées / la page (utilisé pour la protection)

23 / 69

## Plan

- Gestion de la mémoire physique
- Mémoire virtuelle
  - Pagination et va-et vient
  - Mise en œuvre d'un espace virtuel de grande taille
  - Gestion de la mémoire virtuelle
  - Utiliser la mémoire virtuelle pour échanger des données
- Synthèses
- Exemples

24 / 69



### Temps d'accès effectif à la mémoire

$$\text{TAE} = (1-p) \times \text{TAM} + p \times \text{TDP}$$

avec :

- $p$  : probabilité d'un défaut de page ;
- TAM : temps d'accès mémoire
- TDP : temps d'accès, en cas de défaut de page (gestion de l'IT + va et vient)

### Ordres de grandeur

- TAM  $\approx 100$  ns
- TDP  $\approx 1$  ms +  $2 \times 8$  ms  $\approx 17$  ms

$$\rightarrow \text{TAE} \approx 100 + 17 \times 10^6 p$$

Pour avoir une dégradation des performances  $\leq 10\%$  de TAM, on doit avoir :  $100 + 17 \times 10^6 p \leq 110$   
soit :  $p \leq 1 / 1,7 \times 10^6$



29 / 69

### Peaufiner le mécanisme

- éviter les lectures/écritures disque superflues
  - *bit d'utilisation* (dirty bit) : indique si la page victime a été modifiée (auquel cas elle doit effectivement être recopiée)
  - pool de *pages libres et sales* (utilisées auparavant)
    - $\rightarrow$  mémoriser le numéro des pages libérées
    - $\rightarrow$  (localité) peut éviter un va-et-vient lors d'un défaut de page
- anticiper
  - pool de pages libres : évite d'attendre la sauvegarde de la victime
  - *préchargement* de pages : souvent possible et efficace (localité/ensemble de travail)
  - *restitution* volontaire de page : ok si automatique (compilateur)

### Choisir la bonne victime

$\rightarrow$  algorithmes de remplacement

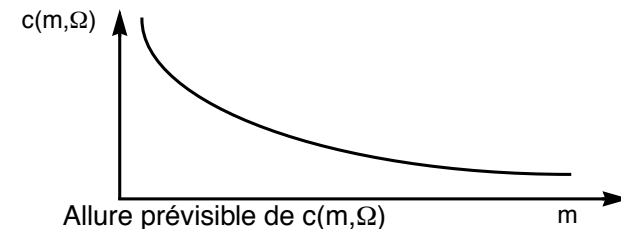
### Ecarter les processus causant trop de défauts de pages

$\rightarrow$  *régulation* basée sur un *modèle de comportement* des processus



30 / 69

- $c(m, \Omega)$  : *nombre de chargements de pages* causés par  $\Omega$ 
  - $\Omega$  : suite de références (numéros de pages virtuelles demandées par le processus),
  - $m$  : nombre de pages mémoire disponibles pour le processus



- *taux de défaut de pages*  $F = c(m, \Omega) / |\Omega|$   
avec  $|\Omega|$  : longueur de la suite de références



31 / 69

### Algorithme aléatoire

Victime = page choisie aléatoirement

### FIFO

Victime = première page chargée (parmi les pages présentes)

*Remarque* : l'allure de  $c(m, \Omega)$  peut être contre-intuitive pour FIFO

*Exemple* (anomalie de Belady) :

- gestion FIFO des requêtes de remplacement
- Pour  $\Omega = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$  on a :  $c(3, \Omega) < c(4, \Omega)$

### Algorithme optimal

Victime = page dont la prochaine utilisation est la plus tardive

$\Rightarrow$  évaluer *a priori* la chaîne des références

$\rightarrow$  irréalisable en général



32 / 69



### Heuristique : localité temporelle

Le passé récent est une bonne image du futur proche

→ base pour évaluer le futur

→ LRU : victime = page inutilisée depuis le plus longtemps

### Remarques

- LRU ne présente pas l'anomalie de Belady
- Mise en œuvre directe coûteuse (activée à chaque référence)

- gérer une *liste* doublement chaînée.

*Fonctionnement*

- à chaque nouvelle référence, placer la page référencée en tête
- victime = queue

- numéroté les références successives

→ associer un compteur (*date logique*) à chaque page

*Difficultés* : débordement du compteur, arithmétique, tri

nf

33 / 69

utilisent un indicateur (bit) de référence associé à chaque page  
(0 initialement, mis à 1 lors de l'accès)

### Représentation « économique » des dates logiques

Date logique = historique des bits de référence sur une période de temps

- le bit de gauche est le bit de référence courant
- le mot est décalé vers la droite, à intervalles réguliers
- la plus petite valeur correspond à l'accès le moins récent

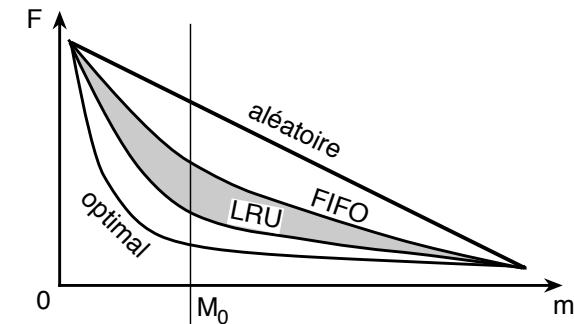
0	0	1	1	
0	1	0	0	
0	1	1	0	
0	0	0	1	
0	0	0	0	

### Algorithme de la deuxième chance

- Index parcourant la table des pages de manière circulaire
- L'entrée pointée de la table des page a un bit de référence valant
  - 1 → bit remis à 0 ; l'index progresse
  - 0 → c'est la victime

nf

34 / 69



**Point saillant** : existence d'un seuil  $M_0$  à partir duquel  $F$  croît très vite

- avant ce seuil, les performances s'améliorent nettement avec  $m$
- après ce seuil, augmenter  $m$  est peu rentable

### Moralité

L'algorithme compte moins que la taille de la mémoire allouée

nf

35 / 69

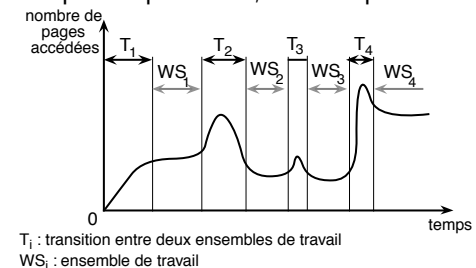
Essentiel : fournir un espace suffisant à chaque processus

Espace suffisant déterminé à partir de 2 heuristiques de localité

- *temporelle* : sur une (courte) période d'observation, l'ensemble des pages utilisées est **stable**
- *spaciale* : les références se concentrent sur peu de pages

→ notion d'*ensemble de travail* ("working set") :

pages référencées par un processus, sur une période de temps

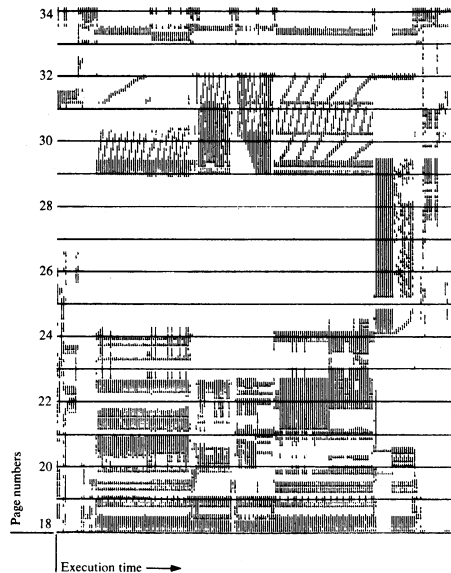


$T_i$  : transition entre deux ensembles de travail  
 $WS_i$  : ensemble de travail

nf

36 / 69

Illustration expérimentale [Hatfield 1972 – IBM Syst. Journal]



## Algorithmes basés sur le modèle de comportement

Idée

### Prévenir l'écroulement en limitant le nombre de processus prêts (actifs)

### Algorithme de l'ensemble de travail (WS)

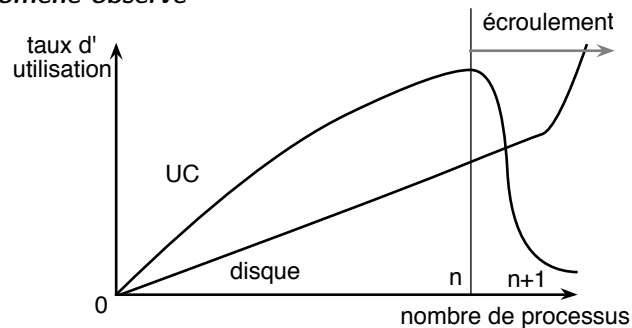
- l'ensemble de travail de chaque processus est mémorisé
  - en cas de défaut de page
    - on remplace une page qui n'est dans aucun ensemble de travail
    - s'il n'y en a pas, on réquisitionne les pages détenues par le processus le moins prioritaire (qui est suspendu)
- tout processus actif dispose des pages de son ensemble de travail

Exemple : VMS (VAX)

relevé périodique des bits de référence pour estimer l'ensemble de travail

## Illustration empirique : écroulement du système (“trashing”)

### Phénomène observé



### Explication

Dès qu'un processus a moins de mémoire que son ensemble de travail

→ il engendre des défauts de pages nombreux et fréquents

→ s'il n'y a plus de cases libres, retrait de pages à d'autres processus qui, à leur tour, ne disposent plus de leur ensemble de travail. . .

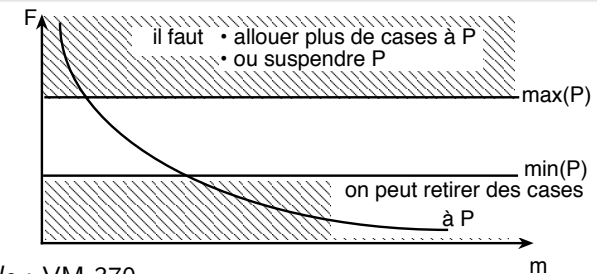
→ boule de neige

## Algorithmes basés sur le modèle de comportement (suite)

Algorithme du taux de défaut de page par processus (PFF, page fault frequency)

Idée

Utiliser le taux de défauts de page pour détecter si un processus dispose de son espace de travail



*Exemple : VM 370*

## Remarques

- PFF est plus aisé à réaliser que WS
- Les capacités mémoire actuelles permettent (et ont pour objectif) de réduire très fortement le va-et-vient, et donc l'écroulement.

- 1 Gestion de la mémoire physique
- 2 Mémoire virtuelle
  - Pagination et va-et vient
  - Mise en œuvre d'un espace virtuel de grande taille
  - Gestion de la mémoire virtuelle
  - Utiliser la mémoire virtuelle pour échanger des données

### 3 Synthèses

### 4 Exemples



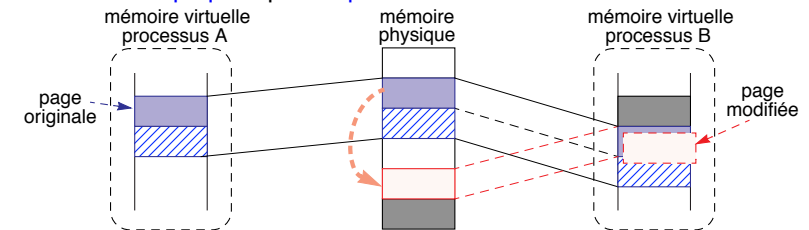
41 / 69

### But

Fournir à un processus (B) une copie des données d'un autre processus (A)  
*Contrainte* : les **copies** doivent être **indépendantes** (locales à (A) et à (B))

### Observation

inutile de dupliquer les données si elles ne sont pas modifiées  
 → **partager la page** des données d'origine **en lecture**,  
 et ne **dupliquer** qu'à la **première écriture**



Exemple : mécanisme UNIX de création de processus

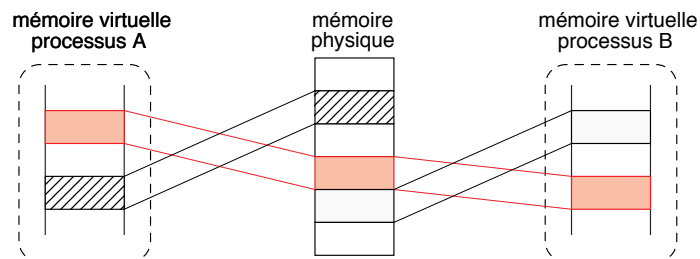
duplication sur écriture de l'image mémoire du processus père



43 / 69

### Principe

Référencer une **même case** mémoire à partir de **pages distinctes** de processus distincts



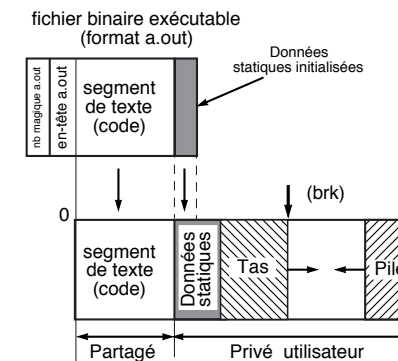
### Protection

- contrôle de l'**espace d'adressage**
  - indicateur spécifique (bit d'« invalidité »)
  - valeur limite associée à la table des pages
- **indicateurs** pour chaque page, précisant le **mode d'accès** (lecture, écriture, ajout, exécution...)



42 / 69

**Idée** : au chargement, les fichiers binaires de code exécutable sont liés (via la table de couplage) à l'image mémoire des processus.  
*[i.e. les blocs du fichier sont associés à des pages de l'image mémoire]*



- utiliser le même mécanisme pour coupler les fichiers de données
- plus d'E/S : contenu du fichier = **variables** en mémoire
  - plus de tampons → moins de recopies → transferts plus **efficaces**



44 / 69

## Intégration (*couplage*) des fichiers dans l'espace virtuel (suite)

### *Couplage* (*mapping*) du fichier en mémoire virtuelle

- fixer 1 adresse de base  $B$  en mémoire virtuelle pour le contenu du fichier
- associer le contenu du fichier aux adresses successives à partir de  $B$  (coupler les blocs du fichier aux pages successives à partir de  $B$ )

### Remarques

- couplage  $\equiv$  l'ouverture
- accès au fichier  $\equiv$  accès à l'image mémoire
- transferts réalisés via le mécanisme de défaut de page
- possibilité de partager un fichier (cf TD)

### Systèmes proposant ce mécanisme

Multics (1965) (réalisation systématique des E/S par couplage des fichiers)  
Appolo Domain, OS/2, Windows NT, Mach, Linux.



45 / 69

## Plan

- 1 Gestion de la mémoire physique
  - Allocation contiguë
  - Allocation fragmentée
- 2 Mémoire virtuelle
  - Pagination et va-et vient
  - Mise en œuvre d'un espace virtuel de grande taille
  - Gestion de la mémoire virtuelle
  - Utiliser la mémoire virtuelle pour échanger des données
- 3 Synthèses
  - Hiérarchies de mémoires
  - Étapes de production d'un programme exécutable
- 4 Exemples
  - Unix 4.3 BSD
  - Solaris
  - Windows NT



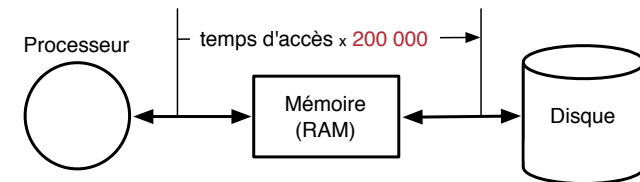
46 / 69

## Hiérarchie de mémoires

### Caractéristiques de la pagination avec défaut de page

Deux niveaux de mémoire

- un niveau rapide, mais coûteux et restreint (haut)
- un niveau lent, mais économique et de grande capacité (bas)



Les données sont

- conservées sur le disque (capacité = 1000 fois celle de la RAM)
- traitées en mémoire centrale (RAM)



47 / 69

**Idéal** : amener dans le niveau haut les informations dont la fréquence / la probabilité d'utilisation sont les plus grandes

→ mémoire

- comparable au niveau haut pour les temps d'accès
- comparable au niveau bas pour les capacités

**Comment ? (rappel)**

heuristiques de localité

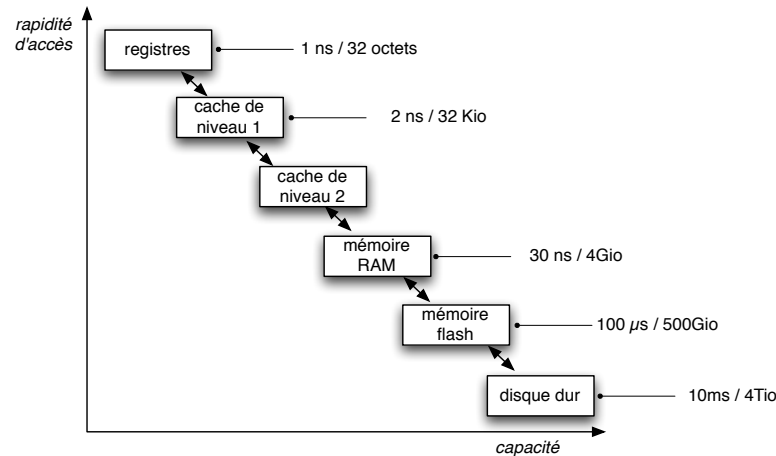
- spatiale (ensemble de travail) et
- temporelle (LRU)

On dit que le niveau haut (mémoire centrale) joue un rôle d'*antémémoire* (*cache*) pour le niveau bas (mémoire secondaire).



48 / 69

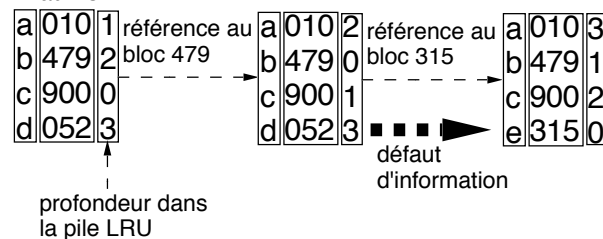
Les idées développées dans le cadre de la pagination sont utilisées pour la hiérarchie des différentes mémoires existantes



## Fonctionnement d'une mémoire cache(niveau matériel)

- une ligne du cache contient, outre les données, un indicateur de validité des données, et une étiquette (*tag*) identifiant l'adresse mémoire centrale à laquelle correspondent les données du cache.
- toutes les informations à lire passent d'abord dans le cache
- écriture dans le cache avec recopie vers la mémoire centrale
- la mémoire cache est découpée en *blocs* (p. ex. 64 octets)
- le cache gère les N dernières références dans une « pile » LRU

info adMC



## Les colonnes du cache (cache à correspondance prédéfinie)

Pour réduire le nombre de comparateurs nécessaires,

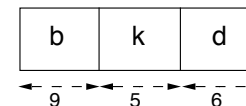
- La mémoire centrale est divisée en *colonnes* correspondant à une partition de la mémoire, espérée équiprobable.
- Chaque colonne est divisée en *rangées*. Une *rangée* d'une colonne a la *taille d'un bloc*.
- De même, le cache est divisé en colonnes, puis en rangées.

Chaque colonne du cache joue le rôle de cache pour une colonne de la mémoire.

## Les colonnes du cache (suite)

Chaque colonne du cache joue le rôle de cache pour une colonne de la mémoire.

### Analyse d'une adresse



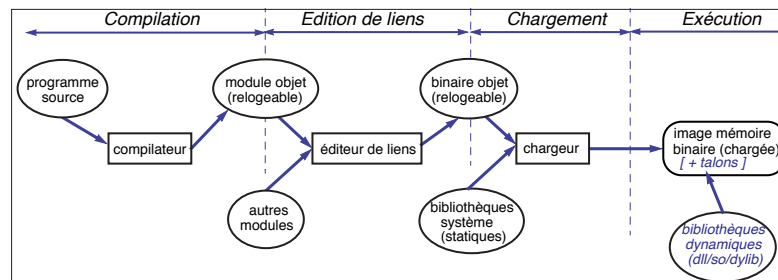
b : numéro de rangée dans la colonne k (ici 512 blocs)  
k : numéro de colonne (32 colonnes)  
d : déplacement dans le bloc (64 octets)

Exemple : cache de 8 Ki

- $8 \text{ Ki} / (64 \times 32) = 4$  blocs par colonne dans le cache,
- donc 4 comparateurs (au lieu de 128) ( $128 = 8 \text{ Ki} / 64$ )

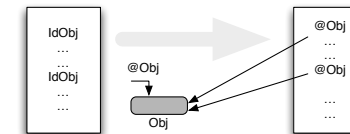
Si ce cache correspond à une mémoire de 1 Mio, celle-ci comportera 512 rangées par colonne.

- 1 Gestion de la mémoire physique
- 2 Mémoire virtuelle
- 3 Synthèses
  - Hiérarchies de mémoires
  - Étapes de production d'un programme exécutable
- 4 Exemples

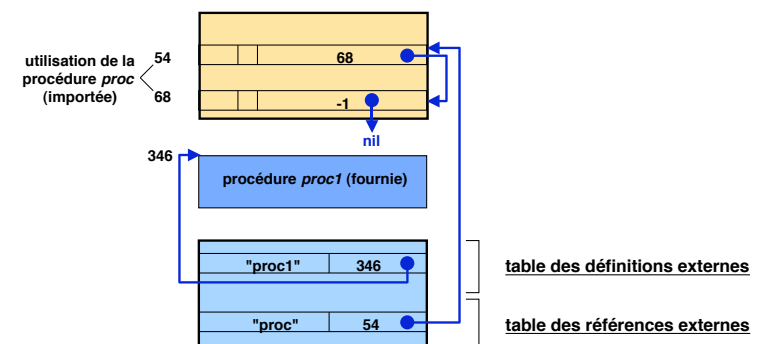
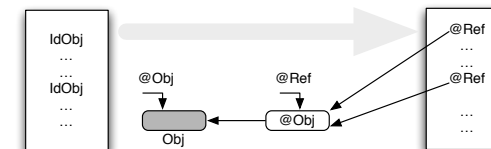


- chargement souvent dynamique → gain de place mémoire : les procédures sont stockées sur disque au format binaire relogeable, et chargées en mémoire à leur premier appel
- L'édition de liens peut être dynamique (bibliothèques)
  - partage et mise à jour automatique du code
  - les procédures à lier dynamiquement sont représentées par un talon qui gère la liaison et se remplace par la procédure effective au premier appel.

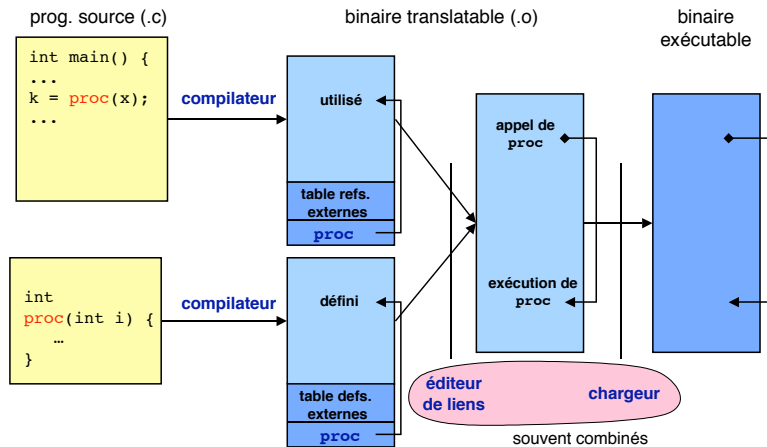
- **substitution**/inclusion : remplacer chaque occurrence de l'objet abstrait (symbole) par sa correspondance concrète (adresse)



- simple, statique, efficace
- une **table de traduction**, indiquant la position des occurrences de chacun des objets substitués dans le code d'origine doit être conservée, si l'on veut pouvoir refaire l'étape
- **chaînage**/indirection : remplacer chaque occurrence de l'objet abstrait par l'adresse d'une référence vers l'objet concret.
  - plus souple, mais moins efficace

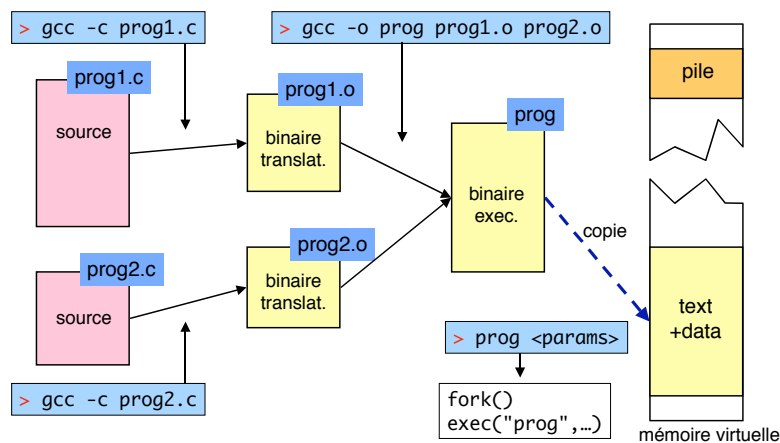


1. Source : S. Krakowiak



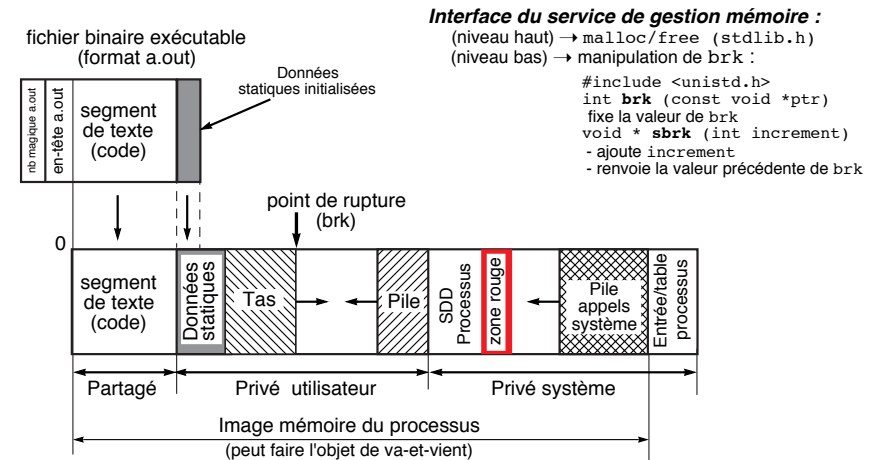
2. Source : S. Krakowiak

57 / 69



3. Source : S. Krakowiak

58 / 69



59 / 69

- 1 Gestion de la mémoire physique
  - Allocation contiguë
  - Allocation fragmentée
- 2 Mémoire virtuelle
  - Pagination et va-et vient
  - Mise en œuvre d'un espace virtuel de grande taille
  - Gestion de la mémoire virtuelle
  - Utiliser la mémoire virtuelle pour échanger des données
- 3 Synthèses
  - Hiérarchies de mémoires
  - Étapes de production d'un programme exécutable
- 4 Exemples
  - Unix 4.3 BSD
  - Solaris
  - Windows NT

60 / 69

## Gestion mémoire sous UNIX 4.x BSD

Va et vient

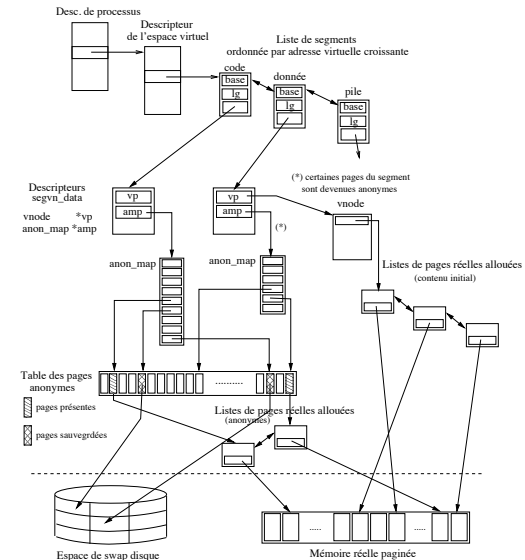
- le va et vient est appliqué, mais dans des cas peu fréquents (prévention de l'écroulement, processus inactifs) : les processus sont suspendus par le processus scheduler
  - lorsque la charge est élevée
  - et que la mémoire libre est en dessous d'un seuil minimum ( $\text{lotsfree} \approx 1/64 \text{ mémoire totale}$ ), durant trop longtemps
- le scheduler suspend alors (et répète jusqu'à avoir plus de  $\text{lotsfree}$  mémoire libre)
  - le processus inactif depuis le plus longtemps, parmi ceux inactifs depuis plus de 20s
  - le plus gros processus inactif depuis le plus longtemps
- la gestion « courante » de la mémoire est basée sur la pagination



61 / 69

## Solaris

## Gestion de la mémoire virtuelle : segmentation paginée



63 / 69

## Gestion mémoire sous UNIX 4.x BSD (suite)

## Pagination

- Pagination à 2 niveaux
- taille des pages indépendante du matériel ( $\geq 512$  octets)
- pages verrouillables (non éjectables) (attente E/S, va-et-vient)
- les pages d'un processus sont généralement préchargées (et mises dans la liste des pages libres, étant marquées récupérables)
- une *table des cases* est gérée (liste inverse)
- et parcourue circulairement, s'il faut libérer de la place (proc. *pagedaemon*)
  - les cases déjà libres ou verrouillées ne sont pas touchées
  - sinon, si le processus auquel la case est allouée a suffisamment d'espace mémoire
    - la page est éventuellement sauvée sur disque (bit « modifié »)
    - la case est marquée comme libre mais récupérable
    - la page correspondante est marquée comme invalide
- *pagedaemon* est réveillé
  - lorsque le processus scheduler en a besoin
  - quand le nombre de cases libres est au dessous d'un seuil (*lotsfree*) (1/64 mémoire totale)
- *pagedaemon* utilise moins de 10% du temps processeur.



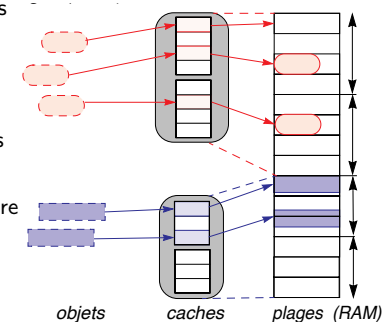
62 / 69

## Solaris (suite)

## Gestion de la mémoire noyau : allocation de plages (slabs)

### Principe

- une **page** (slab) est une suite de cases contiguës
- un **cache** est composé d'une ou plusieurs pages
- chaque structure de données (SdD) du noyau (descripteurs de processus, de fichiers...) est associée à un cache, qui contiendra les instances (objets) de cette SdD
- lorsqu'un cache est créé, il est peuplé d'un nombre déterminé d'objets marqués comme **libres**
- les objets d'un cache sont alloués et marqués **utilisés** au fur et à mesure des besoins
- une demande d'allocation est servie en cherchant d'abord une page partiellement remplie, puis une page vide
- si toutes les pages sont pleines, une nouvelle page est créée et associée au cache



### ***Advantages***

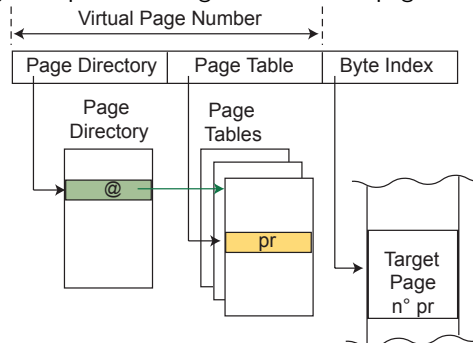
- **avantages** :
  - **pas de fragmentation** : ensemble de tailles prédéterminé (structures du noyau), extension dynamique des plages, allocation aux plages par pages de taille unique
  - **efficacité** : allocation (objets libres pré-créés) et libération (indicateur libre)



64 / 69



- Mémoire à deux niveaux : segmentation et pagination
- Adresses virtuelles sur 32 bits  $\Rightarrow$  4G octets adressables
- Découpage logique :
  - Intelx86 : 1024 segments de 1024 pages de 4K
  - Digital Alpha : 256 segments de 2048 pages de 8K



Page directory = table des segments, implantée à une adresse fixe

65 / 69

### Règles de couplage

- Une zone interdite : 0x0 à 0xFFFF (64Ko de début, référencée par les pointeurs "nuls" )
- Zone programme applicatif : 0x10000 à 0x7FFEFF (2G - 128K)
- Une zone interdite : 0x7FFF0000 à 0x7FFFFFFF (64Ko)
- Zone noyau : 0x80000000 à 0xFFFFFFFF (2G)

### Programmation du couplage en 2 étapes

- réservation/libération d'un espace d'adressage (région);
- couplage/découplage proprement dit d'un contenu.

### Primitives

- *VirtualAlloc/VirtualFree* : réservation/libération (et (dé)couplage de pages)
- *VirtualLock/VirtualUnlock* : (dé)verrouillage en mémoire
- *CreateFileMapping/MapViewOfFile/UnmapViewOfFile* : définition/couplage/découplage de fichiers

66 / 69

```
LPVOID VirtualAlloc (
    LPVOID lpAddress, // adresse de base
    DWORD dwSize, // taille en octets
    DWORD flAllocationType, // type d'opération
    DWORD flProtect, // attributs de protection
);
```

```
LPVOID VirtualFree (
    LPVOID lpAddress, // adresse de base
    DWORD dwSize, // taille en octets
    DWORD dwFreeType, // type d'opération
);
```

avec :

```
flAllocationType = {MEM_RESERVE, MEM_COMMIT, MEM_RESET}
dwFreeType={MEM_RELEASE, MEM_DECOMMIT}
```

67 / 69

## Couplage de fichiers sous Windows NT : primitives

### Couplage de contenu

#### Etapas

- 1 Créer ou ouvrir le fichier : *CreateFile, OpenFile* ;
- 2 Définir le couplage du fichier : *CreateFileMapping* ;
- 3 Associer une région au fichier : *MapViewOfFile*.

```
HANDLE CreateFileMapping (
    HANDLE hFile, // le fichier
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    DWORD flProtect, // attributs de protection
    DWORD dwMaximumSizeHigh, // forts poids
    DWORD dwMaximumSizeLow, // faibles poids
    LPCTSTR lpName
);
```

avec :

```
flProtect = {PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY}
```

☞ PAGE\_WRITECOPY : voir FILE\_MAP\_COPY

68 / 69

## Couplage de fichiers sous Windows NT : primitives

Réserver/Libérer une région en mémoire virtuelle

```
LPVOID MapViewOfFile (
    HANDLE hFileMappingObject, // l'objet couplage
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh, // forts poids
    DWORD dwFileOffsetLow, // faibles poids
    DWORD dwNumberOfBytesToMap
);
BOOL UnmapViewOfFile(
    LPCVOID lpBaseAddress
);
```

avec :

```
dwDesiredAccess =
    {FILE_MAP_WRITE, FILE_MAP_READ, FILE_MAP_COPY}
```

👁 FILE\_MAP\_COPY : fichier original non modifié

