Master d'Informatique UPMC **M1**
Spécialité SAR

Architecture Avancée des Noyaux des Systèmes d'Exploitation

UE 4I401 : : **Noyau**

2017/2018

Polycopié 2 :
Gestion de Fichiers
Mémoire

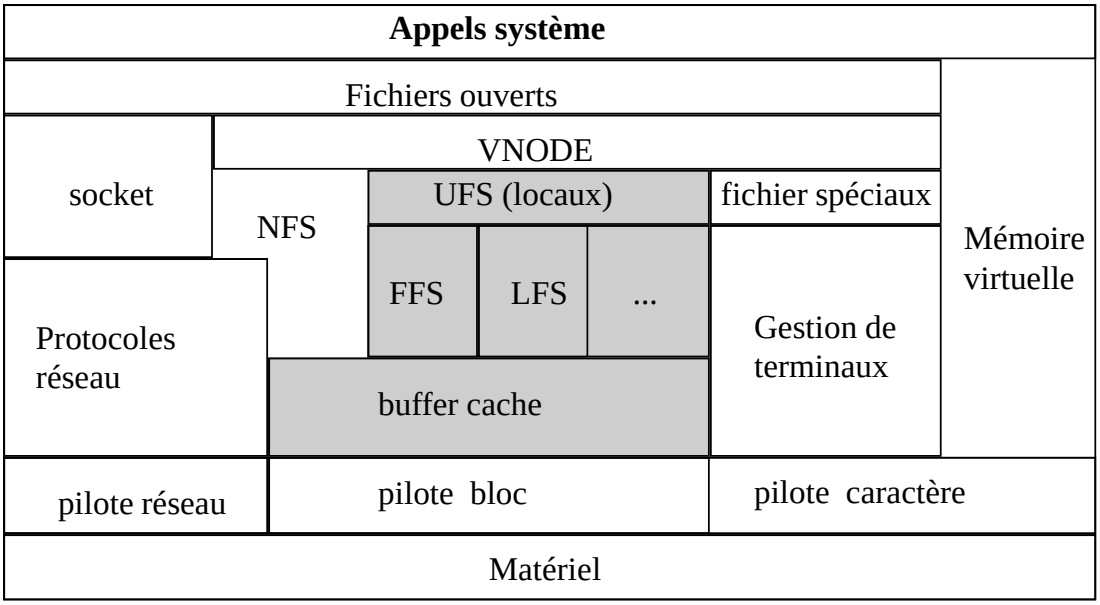
Pierre Sens

Système de gestion des Entrées/sorties

1- Le sous système d'entrées/sorties

2- Les systèmes de fichiers locaux

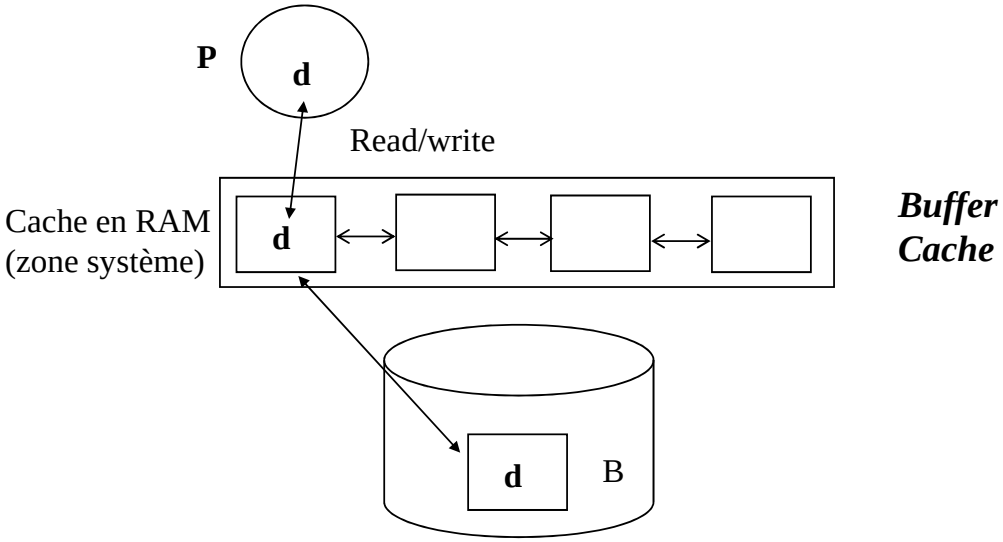
I - Système de fichiers locaux



PARTIE 1 : Cache

Principe du cache

Accès donnée **d** dans bloc B

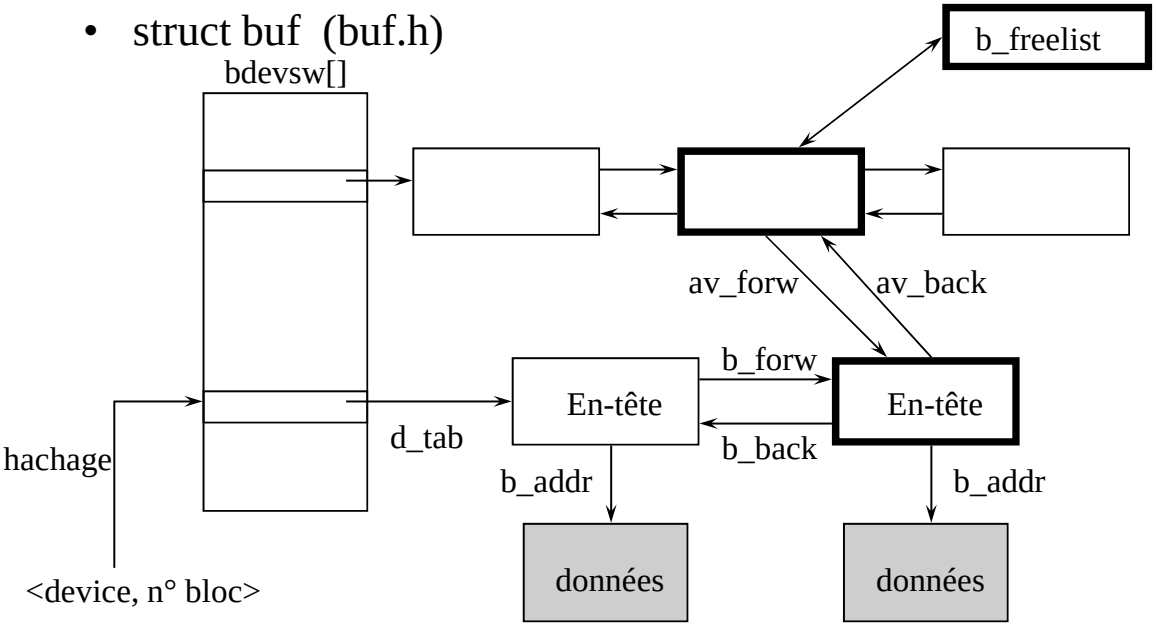


1 - Gestion du cache - le buffer cache

- Principe:
  - Les lectures/écritures par blocs
  - Les blocs sont conservés en mémoire dans une zone du système = buffer cache
- Avantages:
  - Limiter le nombres d'E/S (localité)
  - Dissocier E/S logique et E/S physique (asynchronisme)
  - Anticipation en cas d'accès séquentiel
- Inconvénient:
  - Risque d'incohérence (perte de données) en cas de défaillance

Structure générale

- struct buf (buf.h)



En-tête du buffer cache

- Extraits de struct buf:
  - b_flags : états du bloc
  - *b_forw : pointeur buffer suivant dans le même pilote
  - *b_back : pointeur buffer précédent dans le même pilote
  - *av_forw : pointeur buffer libre suivant (dans la b_freelist)
  - *av_back : pointeur buffer libre précédent
  - b_addr : pointeur vers les données
  - b_blkno : numéro logique du bloc
  - b_error : code de retour après une E/S

Etats d'un buffer

- Valeurs du champs b_flags
- Disponible : pas d'E/S en cours => dans la b_freelist
- Indisponible (B_BUSY positionné dans b_flags)

  B_DONE : E/S terminée

  B_ERROR : E/S incorrecte

  B_WANTED : désiré par un processus (réveiller en fin E/S)

  B_ASYNC : ne pas attendre fin E/S (E/S asynchrone)

  B_DELWRI : retarder l'écriture sur disque (tampon «sale»)

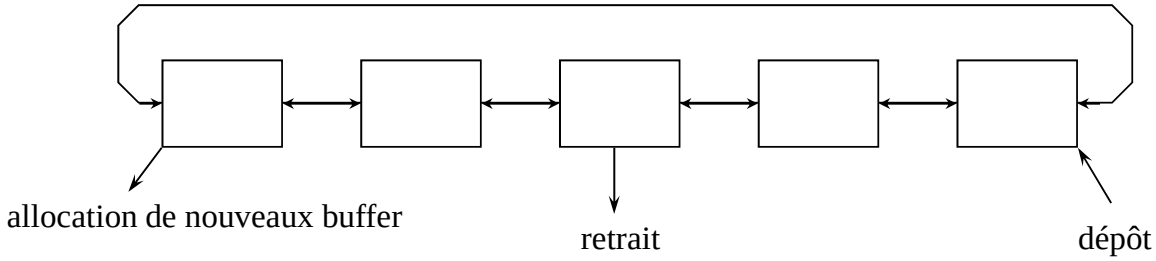
Les primitives

- Lecture d'un bloc : bread
- Lecture par anticipation d'un bloc : breada
- Ecriture différée d'un bloc : bdwrite (buffer *delayed* write)
- Ecriture asynchrone: bawrite
- Ecriture synchrone : bwrite
- Libération d'un buffer : brelse

- Recherche ou allocation d'un buffer : getblk

Gestion des tampons

- Liste des buffer libres : listes circulaire avec gestion LRU



- Accès à un buffer par hash-coding
  - fh(b_dev, b_blkno, nombre de files)
  - Distribution uniforme des tampon dans les files

Recherche/Allocation de buffer (getblk)

```
Entrées : numéro de bloc, device
Tant que (tampon non trouvé)
    Si (bloc dans file indique par fh(bloc, device, nfiles) ) {
        Si (état buffer = B_BUSY) {
            marquer le buffer B_WANTED
            sleep(tampon libre);
            continue;
        }
        marquer le buffer B_BUSY, le retirer de la b_freelist
        Retourner le buffer;
    Sinon { // Le bloc n'est pas dans le buffer cache
        Si (b_freelist vide) { // Plus de tampon libre !
            marquer la b_freelist B_WANTED;
            sleep(un tampon se libère);
            continue;
        }
        Etat buffer tête = B_BUSY; Retirer le buffer de la b_freelist;
        Si (B_DELWRI positionné) { // le tampon est «sale»
            écriture asynchrone sur disque;
            continuer;
        }
        Placer le buffer dans la file correspond au couple <bloc, device>
        Retourner le buffer;
    }
```


Libération d'un buffer (brelse)

- Réveiller tous les processus en attente qu'un buffer devienne libre
- Réveiller tous les processus en attente que ce buffer devienne libre
- Masquer les interruptions
- Si (contenu du buffer valide)

  mettre le tampon en queue de la b_freelist
- Démasquer les interruptions
- retirer bit B_BUSY

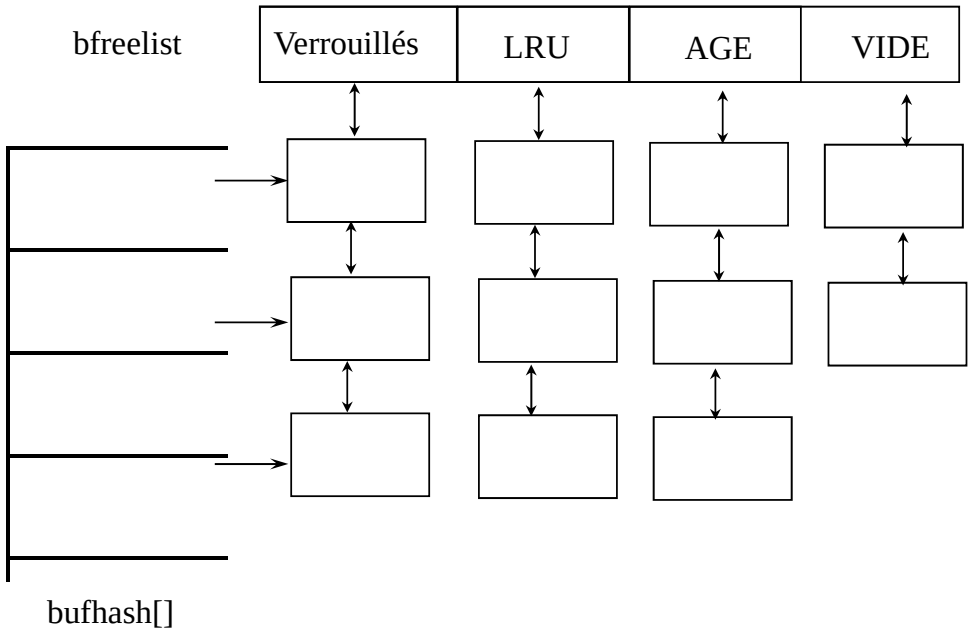
Lecture d’un buffer (bread)

- **Entrées** : device, bloc
- Rechercher ou allouer le bloc (getblk)
- Si (buffer valide et B_DONE)
  retourner le tampon
- Lancer une lecture sur disque (appel du pilote - strategy)
- sleep(attente fin E/S)
- retourner le buffer

Ecriture d’un buffer sur disque

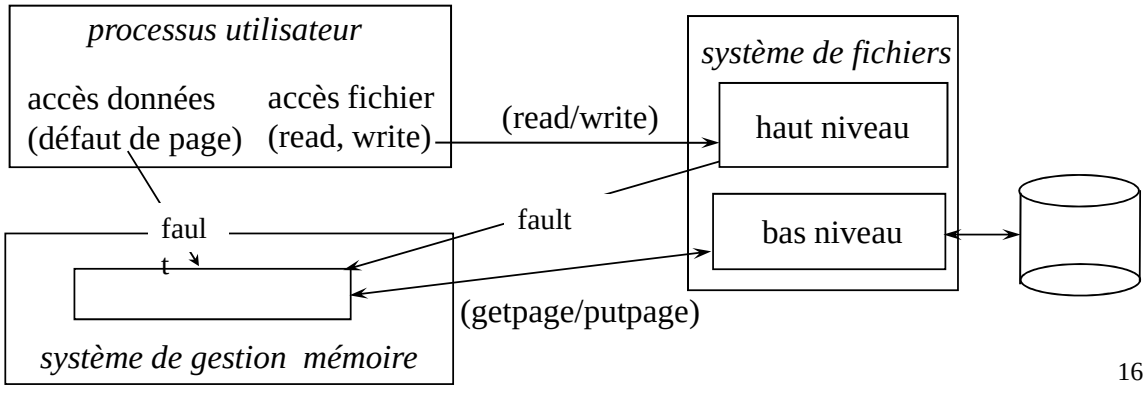
- Bdwrite
  Positionnement de B_DELWRI pour E/S asynchrone
- Besoin de place
  Dans getblk: Si le bloc n’est pas dans le cache
  => allouer un nouveau buffer
  Si B_DELWRI => Ecriture
- Régulièrement **sync** parcourt la liste des buffers
  Si B_DELWRI => Ecriture

Organisation du buffer cache (BSD)



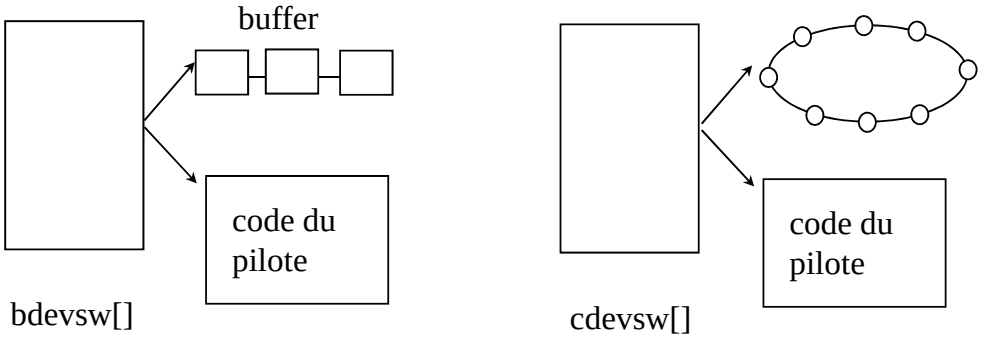
Mémoire virtuelle et buffer cache

- Mécanismes très voisins
  (cases => tampons, swap => fichier)
- Buffer cache integré dans la pagination (SunOs, SVR4)
  - Cases pour les pages **et** les tampons
  - Fichier correspond à une zone de mémoire virtuelle (seg_map)
  - lecture d'un bloc non présent => **défaut de page**



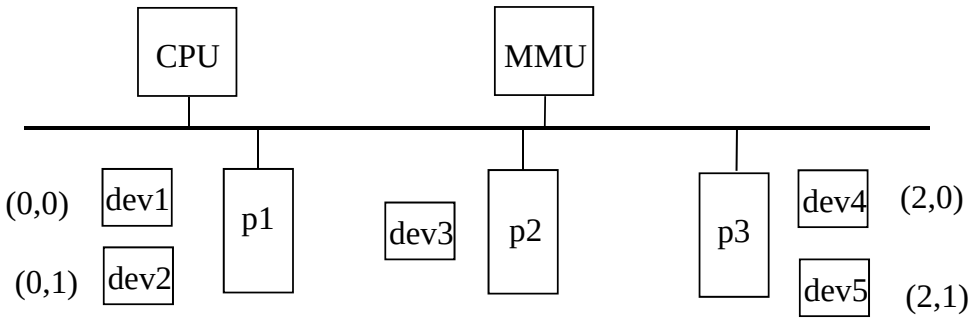
2 - Les Entrées/sorties

- Les types de périphérique
- Mode bloc : accès direct + structuration en bloc
- Mode caractère : accès séquentiel, pas de structuration des données (flux)



Les pilotes de périphérique

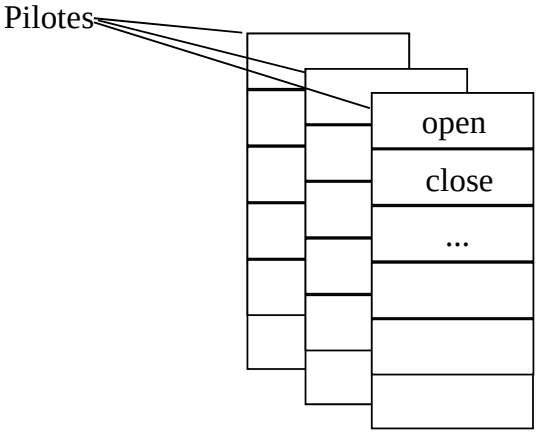
- Configuration typique



- Adresse logique :
  - majeur : numéro de pilote
  - nimeur : numéro d'ordre de l'unité logique

Les tables internes

- Pour chaque pilote un ensembles de fonction (points d'entrées)
- Une table pour chaque pilote (device switch)



Interface structurée

- 2 types de tables : bdevsw (bloc), cdevsw (car.)

Pilotes en mode bloc

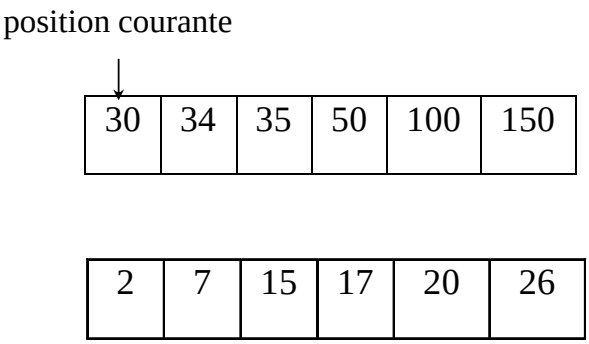
- Table bdevsw :

```
struct bdevsw {
    int (*d_open)();          /* ouverture */
    int (*d_close)();         /* fermeture */
    int (*d_strategy)();      /* Tranfert : Lecture/Ecriture */
    int (*d_size)();          /* Taille de la partition */
    int (*d_dump)();          /* Ecrire toute la mémoire physique
                                 sur périphéque */
    ( int *d_tab;             /* Pointeur vers tampon */ )
    ...
} bdevsw[];
```

(*bdevsw[major(dev)].d_open)(dev, ...);

Requêtes d'E/S

- Algorithme ascenseur (C_LOOK) - BSD
- => limiter les déplacements de têtes



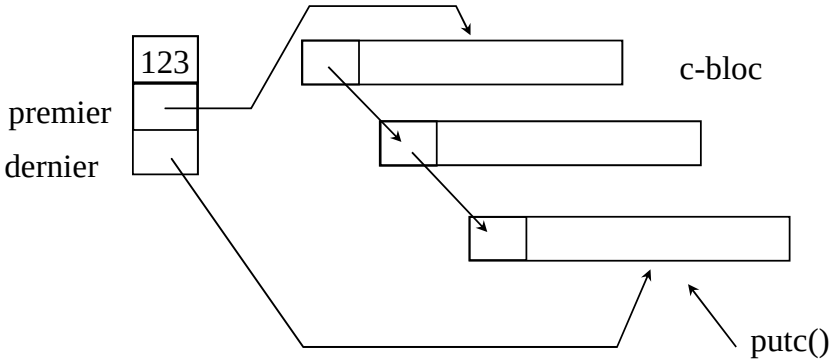
Pilotes en mode caractère

- table cdevsw

```
struct cdevsw {
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();          /* lecture */
    int (*d_write)();         /* ecriture */
    int (*d_ioctl)();         /* contrôle */
    ...
} cdevsw[];
```

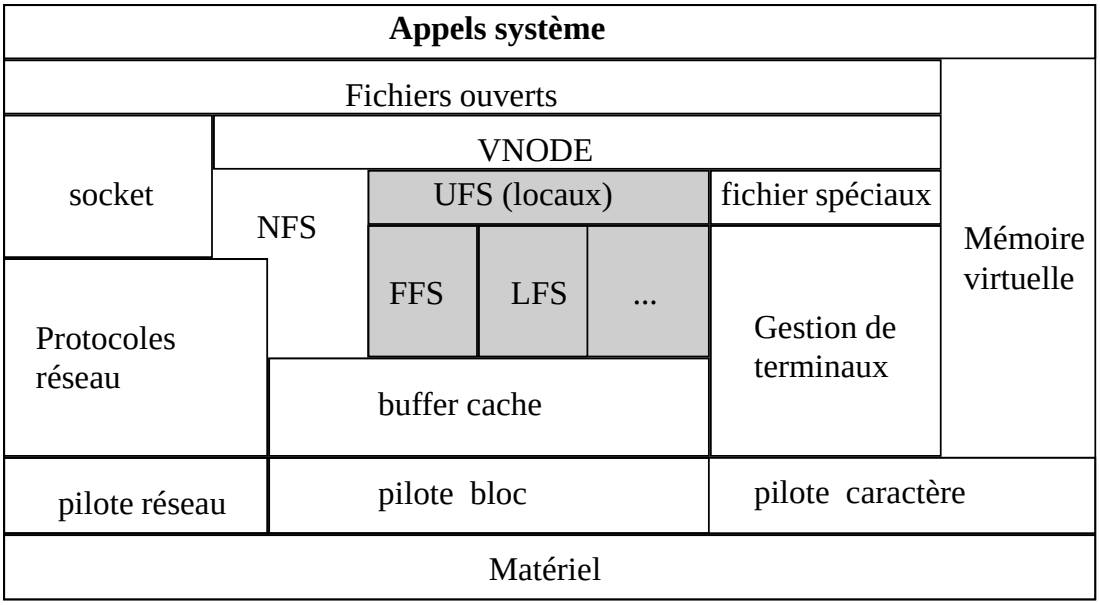
Tampon

- Terminaux : une c-list par terminal



  - Les caractères sont copiés vers le contrôleur soit par le processeur soit par le contrôleur (DMA)
- Chaque type de périphérique gère ses propres tampons (possibilité de transférer directement depuis espace util.)

PARTIE 2 : Systèmes de Fichiers



Les différents systèmes de fichiers

- 2 principaux systèmes de fichiers locaux :

  System V File System (s5fs)
  - Système de fichier de base (78)

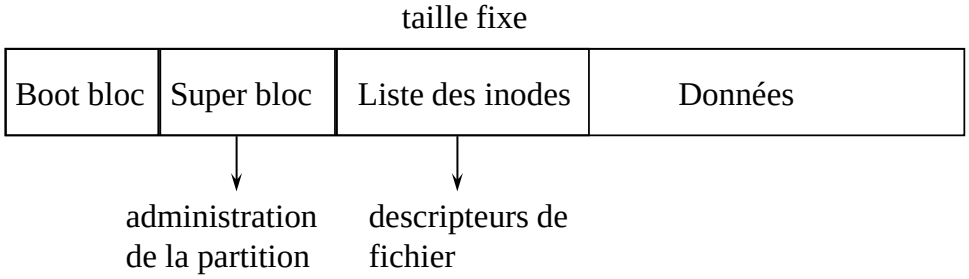
  Fast File System (FFS)
  - Introduit dans 4.2BSD
- Système de fichiers générique
  - Virtual File System (Sun 86)

- De nombreux autres systèmes de fichiers :
  - Ext2fs (linux, FreeBSD) ...

Organisation générale du disque (s5fs)

- Disque divisé en partitions
  - Chaque partition possède ses propres structures

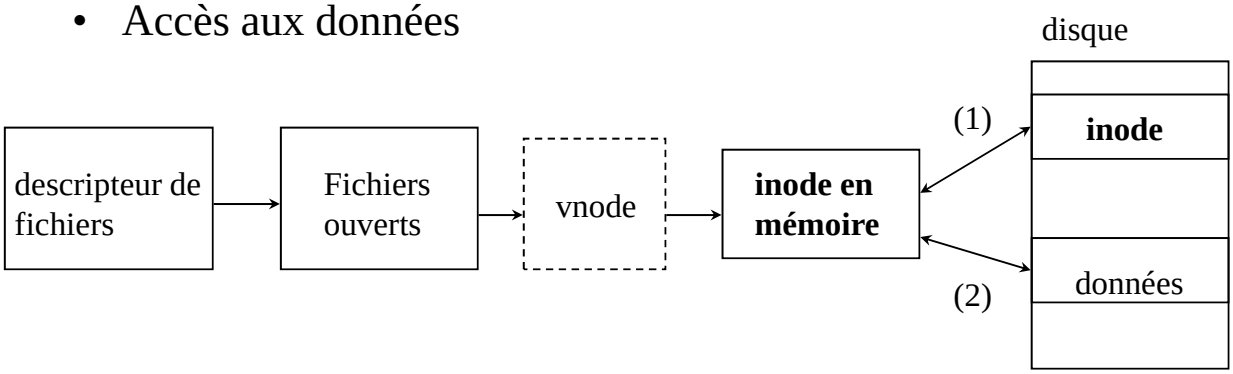
- Organisation d'un partition :



- Numéro d'inode => accès aux blocs du fichier

Les structures

- Accès aux données



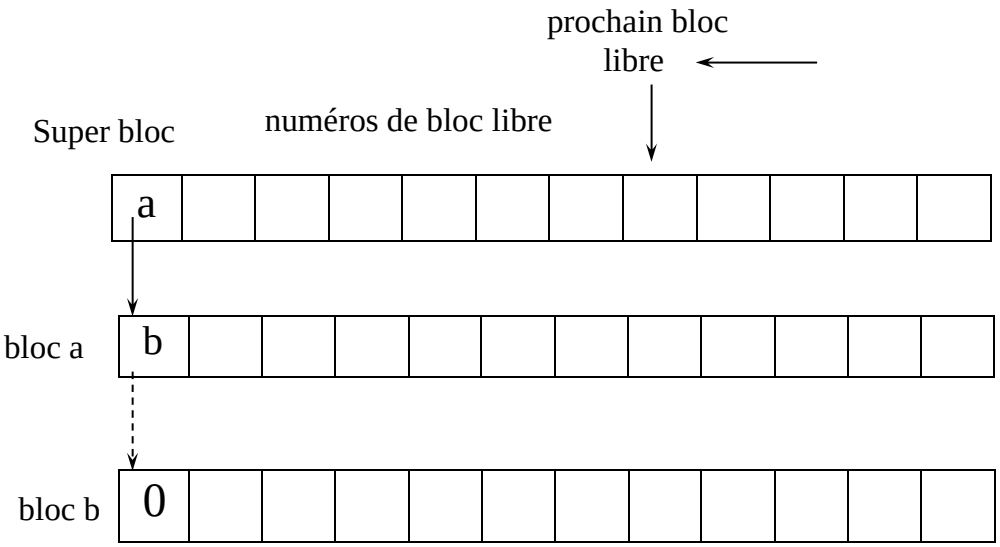
- Allocation de bloc
  - Le superbloc contient la liste des blocs libres, des inodes libres

Gestion des blocs libres

- Le superbloc : struct fs (fs.h)

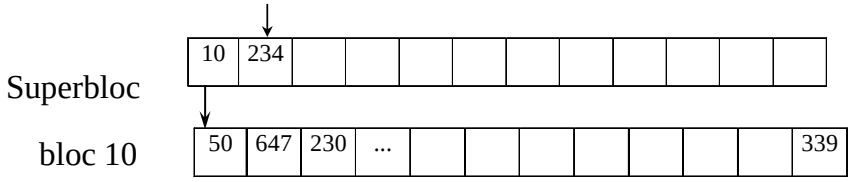
  Bloc d'administration de la partition qui contient :
  - Taille en blocs du système de fichier
  - Taille en blocs de la table des inodes
  - Nombre de blocs libres et d'inodes libres
  - Liste des blocs libres
  - Liste des inodes libres sur disque

Allocation des blocs libres

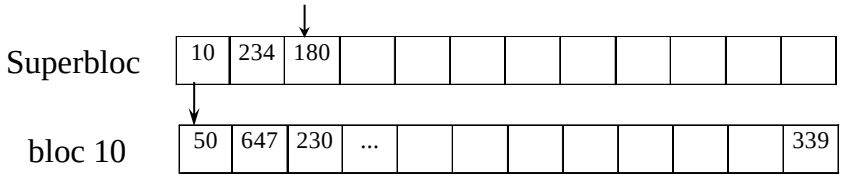


Allocation/libération de bloc : exemple

Configuration initiale



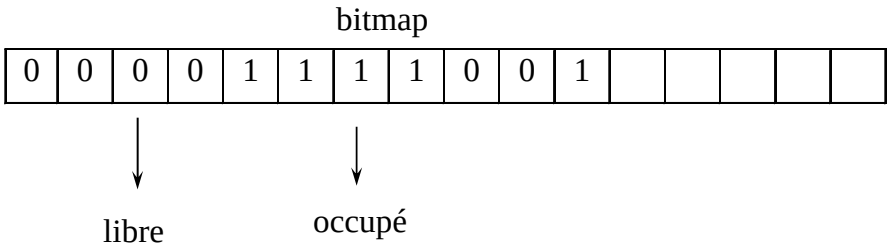
Libération du bloc 180



Allocation de 3 blocs
(180, 234, **10**)

Allocation dans les nouveaux systèmes de fichiers

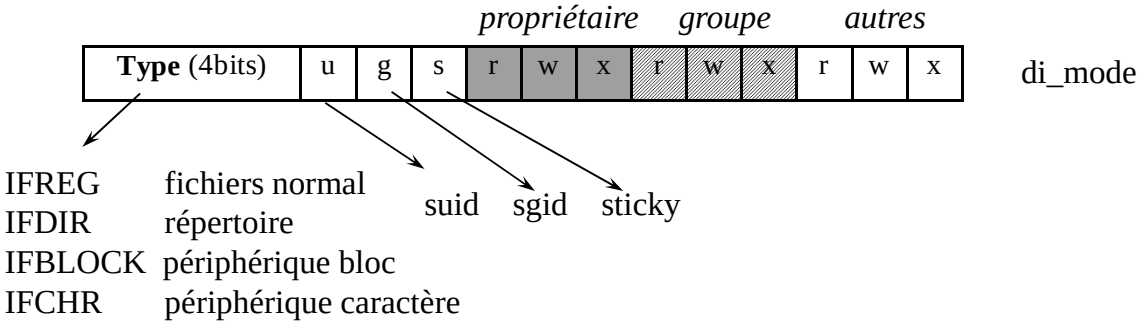
- Pb de la stratégie "classique" : pas de prise en compte de la contiguïté des blocs libres
- => vecteur binaire



- Exemple Ext2fs

Accès au données

- Structure des **inodes** = caractéristiques du fichier
- Sur disque : struct dinode
  - di_mode : type + droits
  - di_nlink nombre de liens physique
  - di_uid, di_gid
  - di_addr : table de blocs de données
  - di_atime, di_mtime, di_ctime : dates consultation, modification, modification inodes



Structure inode

- En mémoire : struct **inode**
  - dinode avec en plus :
  - i_dev : device (partition)
  - i_number : numéro d'inode

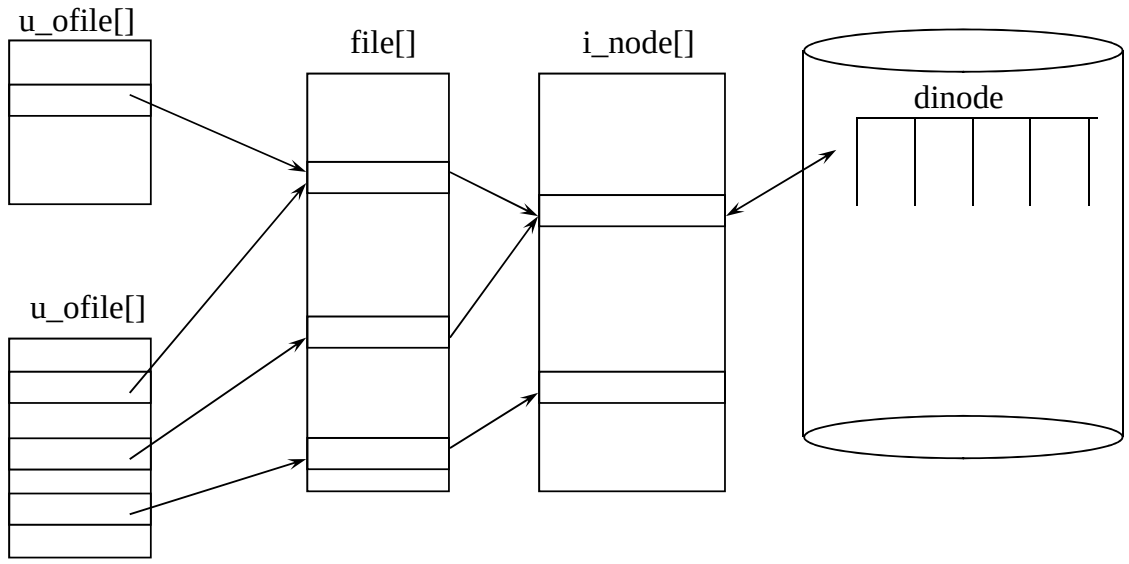
  } accès à l'inode sur disque (mises à jour)
  - i_flags : Flags (synchronisation, cache)
  - pointeurs sur la freelist (liste des inodes libres)

Les autres structures

- file[] : table globale des fichiers ouverts
  - f_flag : mode d'ouverture (Lecture, Ecriture, Lecture/Ecriture)
  - f_offset : déplacement dans le fichier
  - f_inode : numéro d'inode
  - f_count : nombre de références

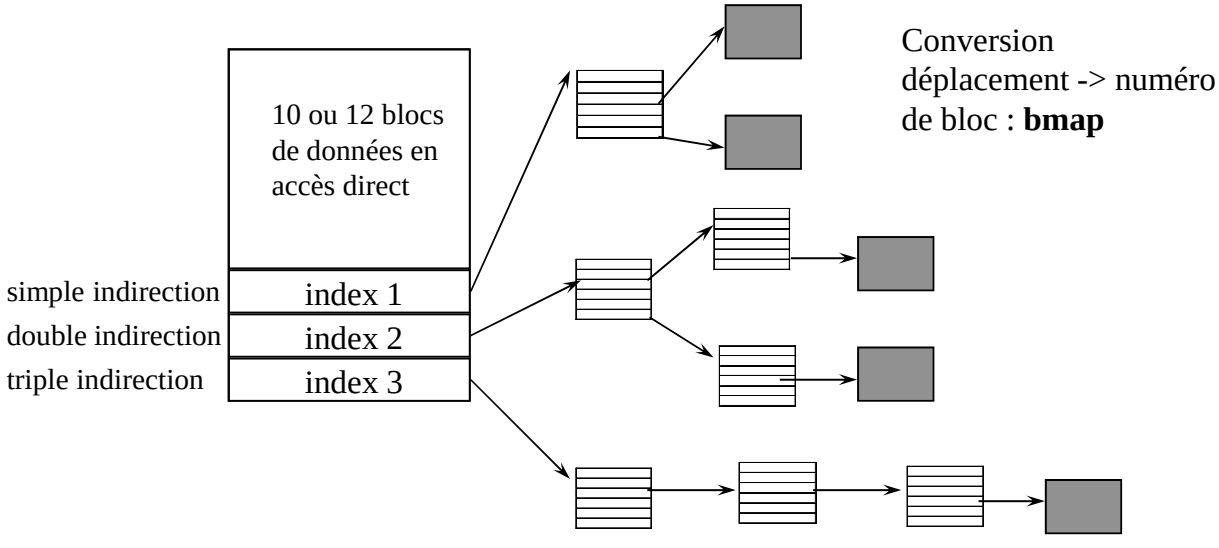
- u_ofile[] : Table locale des ouverts ouverts par un processus
  - entrée dans file

Résumé des structures



Où trouver les blocs ?

- Liste de blocs dans l'inode

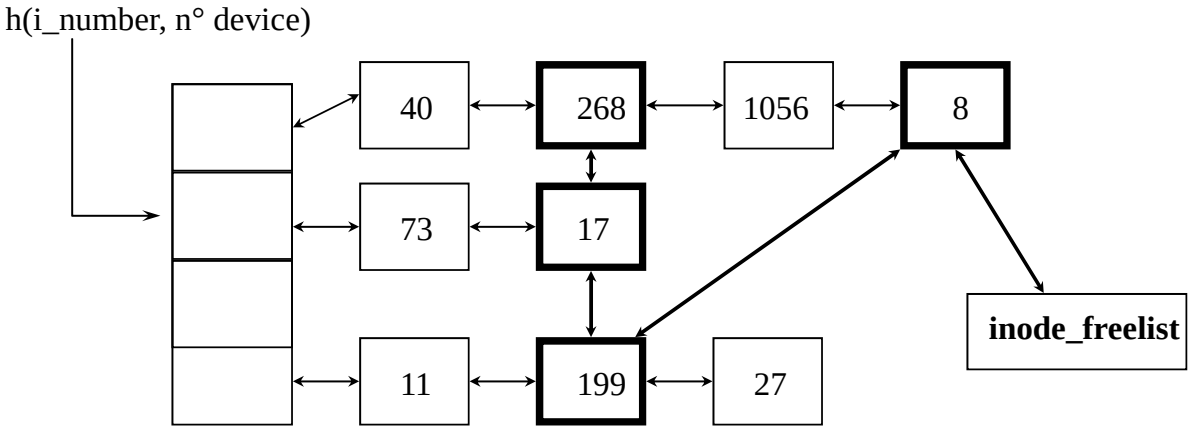


Conversion déplacement -> numéro de bloc : **bmap**

- Vision de l'utilisateur :

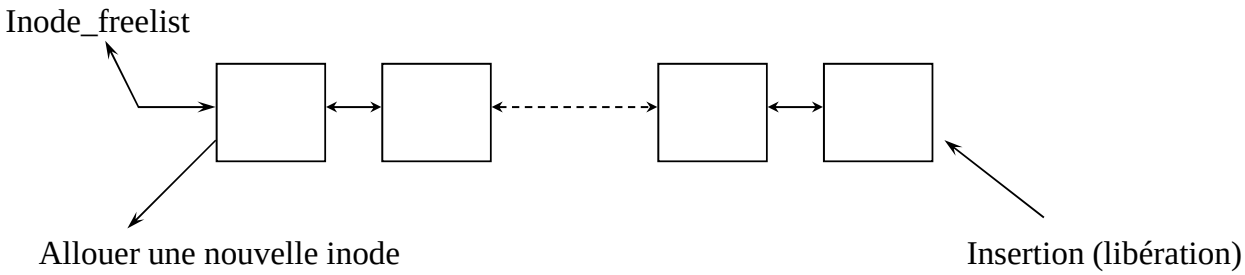
Les inodes en mémoire

- Les entrées de la table des inodes sont chaînées
- Pour trouver rapidement une inode à partir de son numéro utilisation d'une fonction de hachage



Gestion des inodes libres en mémoire

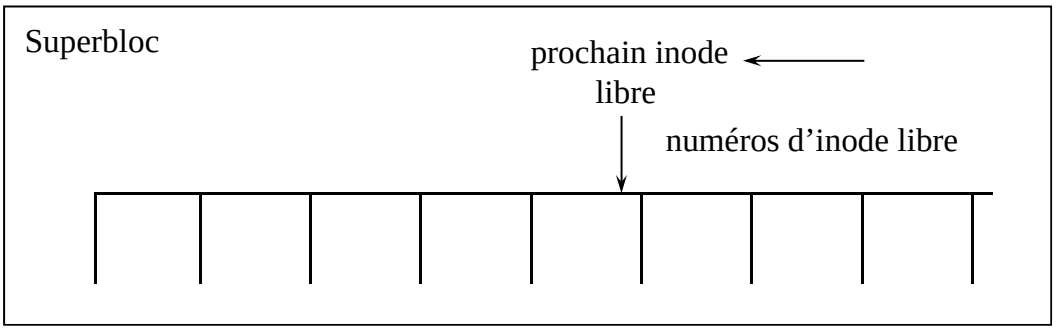
- Si une inode n'est plus utilisée par aucun processus => insertion dans inode_freelist.
- Inode_freelist = cache des anciennes inodes



- Gestion LRU (Least Recently Used) SVR3 (autre critère dans SVR4)

Gestion des inodes libres sur disque

- Le superbloc contient une liste partielle des inodes libres



- Si liste vide, réinitialiser la liste en «scannant» la table des inodes sur disque

Fonction de manipulation des inodes

- namei : retrouve une inode à partir d'un nom de fichier (open)
- ialloc : allouer une nouvelle inode disque à un fichier (creat)
- ifree : détruire une inode sur disque (unlink)
- iget : allouer/initialiser une nouvelle inode en mémoire
- iput : libérer l'accès à une inode en mémoire

Principe de ialloc

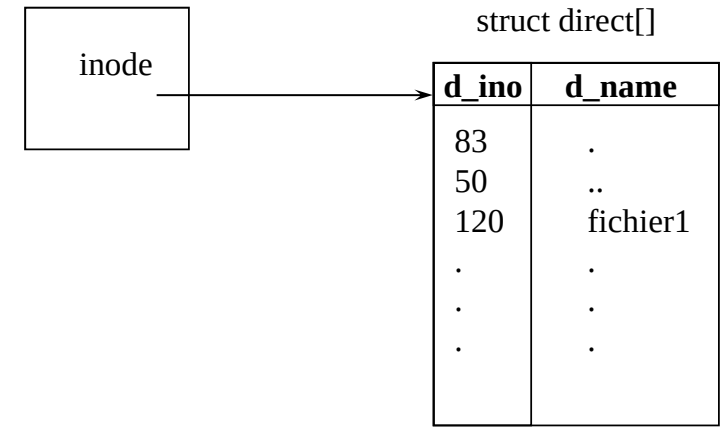
- Vérifier si aucun autre processus n'a verrouillé le superbloc (sinon sleep)
- Verrouiller le superbloc
- Si liste des inodes libres sur disque non vide
  - Prendre l'inode libre suivante dans superbloc
  - attribuer une inode en mémoire (iget)
  - mise à jour sur disque (inode marquée prise)
- Si liste vide
  - Verrouiller le superbloc
  - parcourir la liste des inodes sur disque pour remplir le superbloc
- Tester à nouveau si l'inode est vraiment libre sinon la libérer et recommencer (conflit d'accès à un même inode !)

Principe de iget

- Trouver l'inode en mémoire à partir du couple <i_number, device>
- Si inode non présente allouer un inode libre en mémoire (à partir de la inode_freelist)
- Remplir l'inode à partir de l'inode sur disque

Les répertoires

- Répertoire = un **fichier** de type répertoire
  => référencé par une inode



d_name (14 caractères) SVR4
(255 caractères) BSD

Algorithme de namei

Entrées: nom du chemin

Sortie: inode

```
Si (premier caractère du chemin == '/')
    dp = inode de la racine (rootdir)   (iget)
sinon
    dp = inode du répertoire courant (u.u_cdir) (iget)
Tant qu'il reste des constituants dans le chemin {
    lire le nom suivant dans le chemin
    vérifier les droits et que dp désigne un répertoire
    si dp désigne la racine et nom = ".."
            continuer
    lire le contenu du répertoire (bmap pour trouver le bloc puis bread)
    si nom suivant  appartient au répertoire
            dp = inode correspond au nom
    sinon
            // Pas d'inode
}
retourner dp
```

Exemple

Les liens

- Fichiers spéciaux
  - Liens symboliques : contienne le nom d'un fichier
  - Liens physiques : désigne la même inode

ln -s /users/paul/f1 /users/pierre/lsf1

rm /users/pierre/lsf1

Droits : 1) droits sur le lien
2) droits sur le fichiers

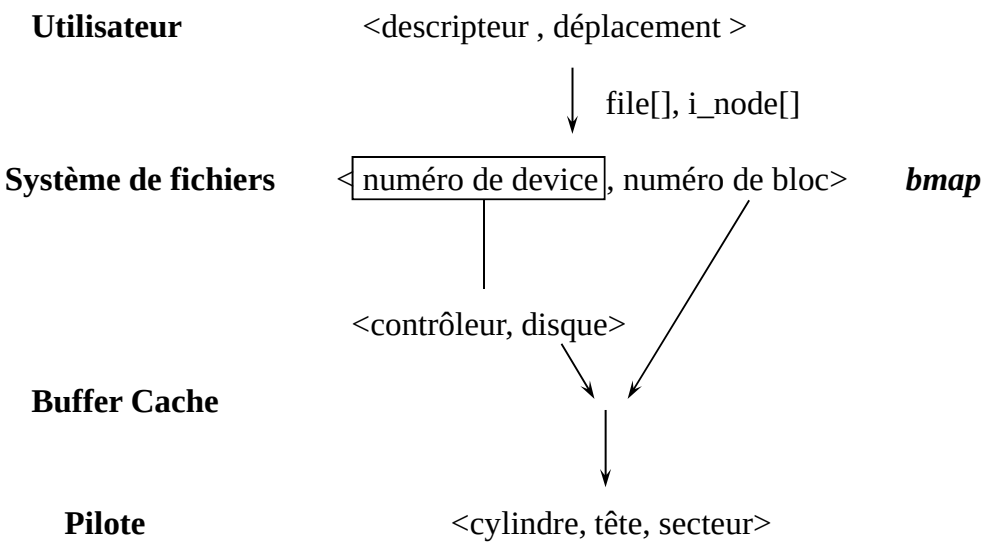
ln /users/paul/f2 /users/pierre/lhf2

rm /users/pierre/lhf2

Droits : droits sur le fichier

Implémentation des appels système

- Systèmes d'adressage :

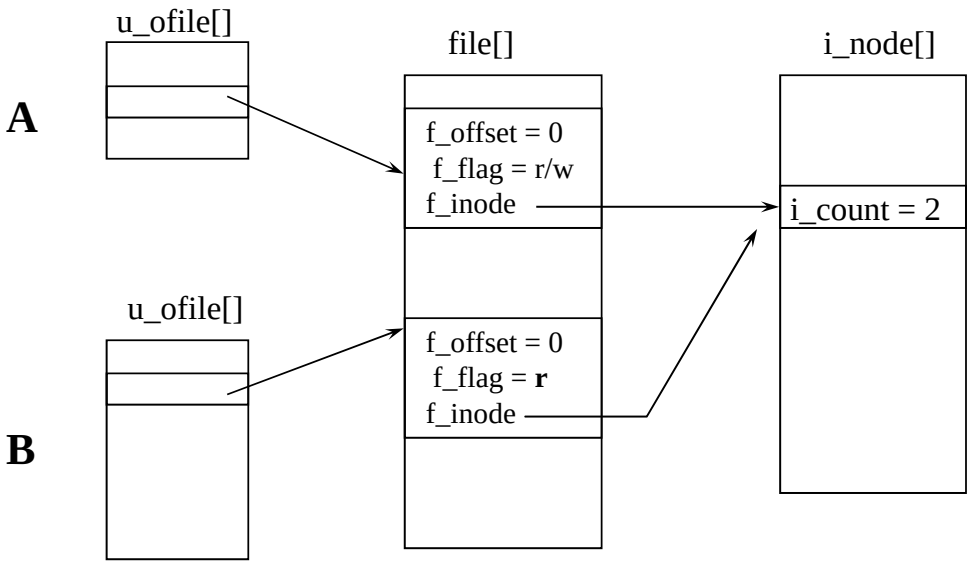


Algorithme de open

- Retrouver l'inode à partir du nom de fichier (namei)
- Si (fichier inexistant ou accès non permis) retourner erreur
- allouer et initialiser un élément dans la table file[]
- allouer et initialiser une entrée dans u_ofile du processus
- Si (mode indice une troncature) libérer les blocs (free)
- déverrouiller inode
- retourner le descripteur

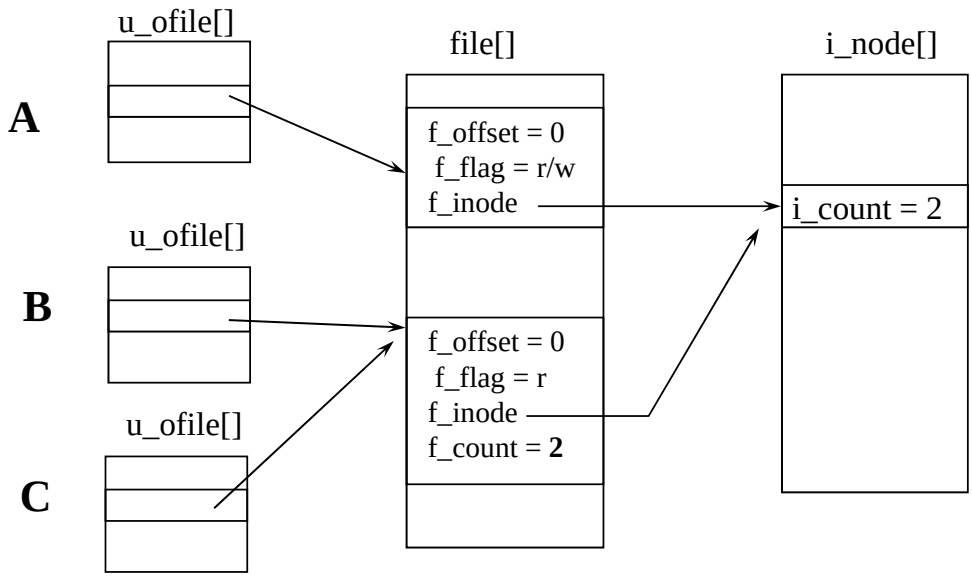
Exemple

Processus A : fd = open ("/home/sens/monfichier", O_RDWR|OCREAT, 0666);
Processus B : fd = open("/home/sens/monfichier", O_RDONLY);

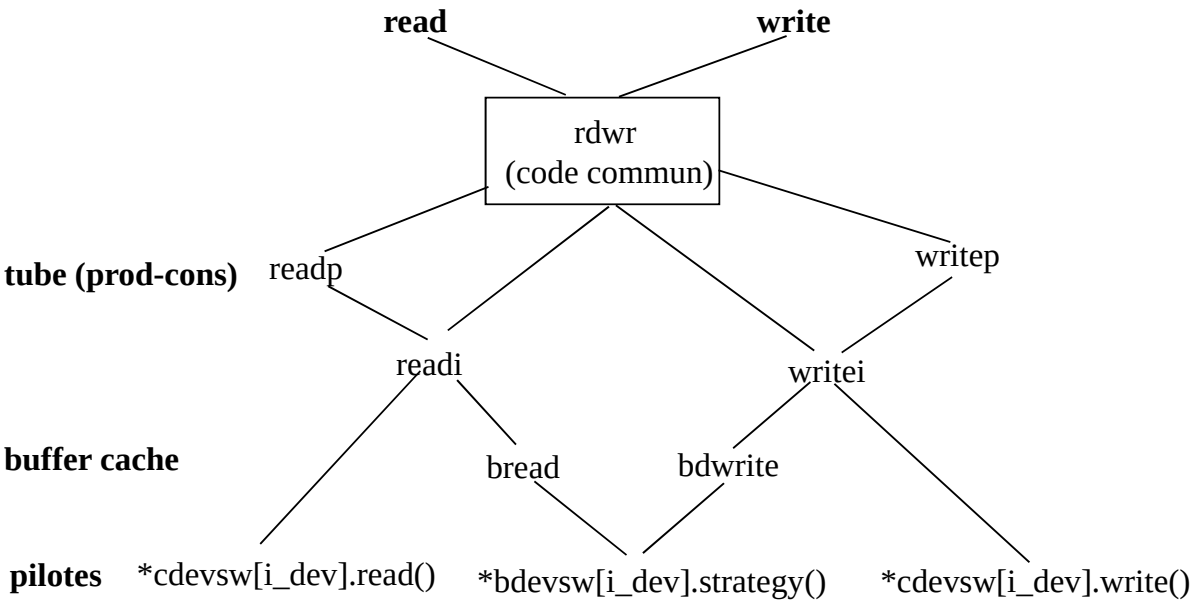


Exemple (2)

Processus B : fork()



Appels read et write

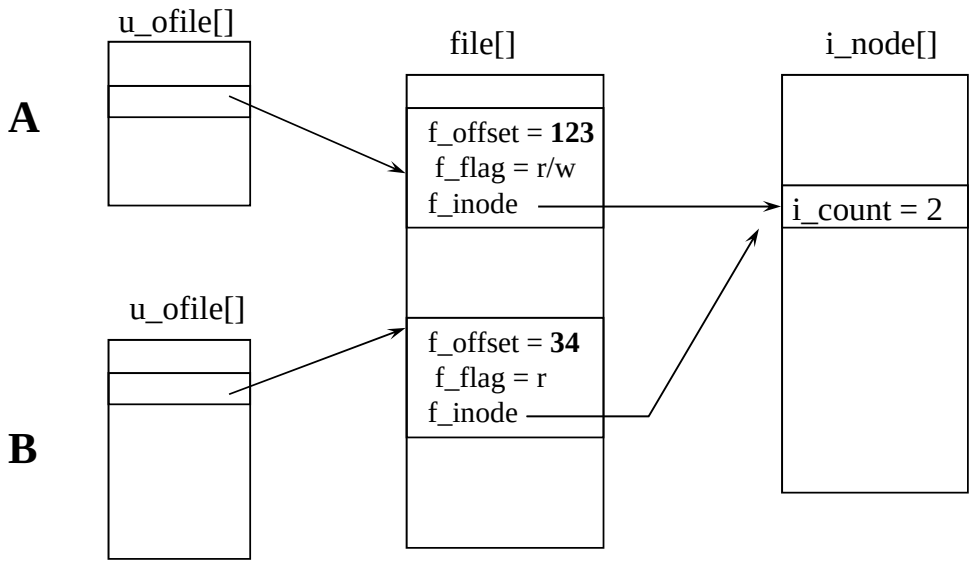


Algorithme de read

- Accéder à l'entrée de file[] à partir de u_ofile[fd]
- Vérifier le mode d'ouverture (champs f_flag)
- Copier dans la zone u les informations pour le transfert
- Verrouiller l'inode (f_inode)
- Tant que (nombre octets lus < nombre à lire)
  - Conversion déplacement numéro bloc (bmap)
  - Calculer le déplacement dans le bloc
  - Si (nb octets restants == 0) break; // Fin de fichier
  - Lecture du bloc dans le cache (bread)
  - Transférer tampon dans zone u
  - libérer le tampon (verrouiller par bread)
- Déverrouiller l'inode; Mettre à jour file[]
- Retourner nombre octets lus

Exemple

Processus A : nb = write(fd, buf, 123);
Processus B : nb = read(fd, buf, 34);



Les optimisations : fast file system (ffs)

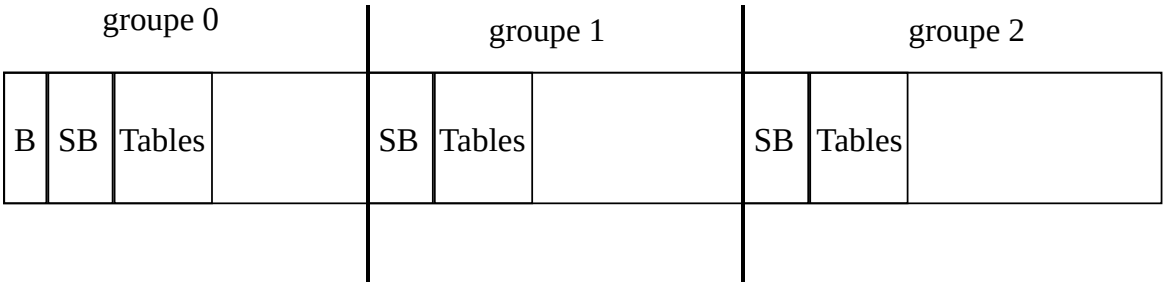
- Intégrer dans tous les unix (connu comme ufs)
- De nombreuses améliorations

=> Augmenter la fiabilité

=> Augmenter les performances

Organisation en groupe

- Disque divisé en groupe de cylindre



- Réplication du superbloc => augmenter la fiabilité
- Dissémination des tables => réduction des temps d'accès

Blocs et fragments

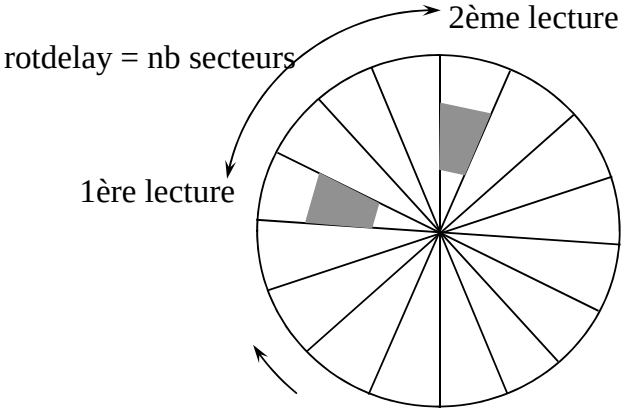
- Problème sur la taille des blocs
  - Taille de blocs importante => plus de données transférées en une E/S
    plus d'espace perdu (1/2 bloc en moyenne)
- Idée : partager les blocs entre plusieurs fichiers
- => Blocs divisés en fragment
  - 2,4,8 fragments par bloc
  - Taille "classique" : blocs 8Ko, fragment 512 octets
- Unité d'allocation = fragment
  - => perte réduite
  - => plus de structures de données

Optimisations

- Optimisations :
  - 1) Regrouper toutes les inodes d'un même répertoire dans un même groupe
  - 2) Inode d'un nouveau répertoire sur un autre groupe
    => distribution des inodes
  - 3) Essayer de placer les blocs de données d'un fichier dans un même groupe que l'inode
- => limiter les déplacements de tête

Politique d'allocation de bloc

- Constatations : la plupart des lectures sont séquentielles
- => placement des blocs d'un même fichier
  - En fonction de la vitesse de rotation pour optimise lecture séquentielle
  - Objectif : faire en sorte que lors de la lecture suivant le bloc soit sous la tête



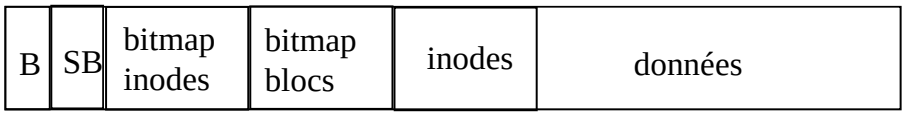
Performances

- Stratégie d'allocation efficace si disque pas trop plein (< 90%)

- Sur VAX/750
- Accès lecture débit = 29 Ko/s s5fs
  débit = 221 Ko/s ffs
- Accès écriture débit = 48Ko/s s5fs
  débit = 142 Ko/s ffs

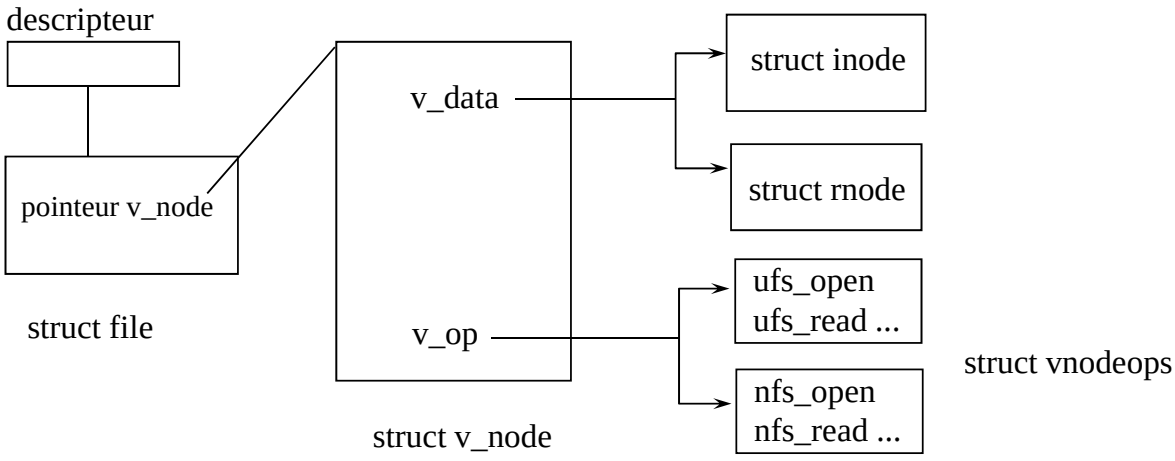
D'autres organisations

Exemple Ext2fs

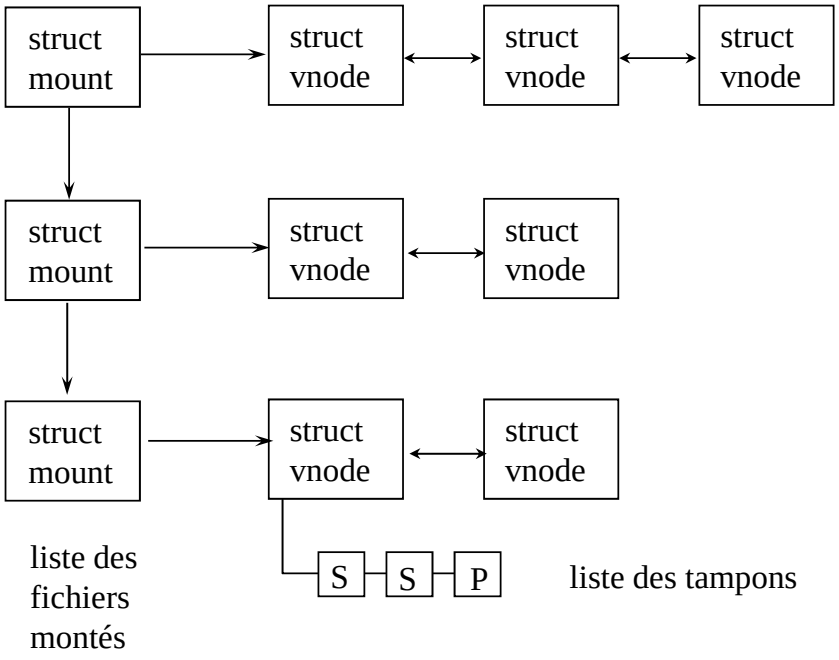


Systèmes génériques : VFS

- Objectifs : gérer différents systèmes de fichiers locaux et distants => Virtual File System
- Ajout d'une couche supplémentaire responsable de l'aiguillage : couche vnode (virtual node)



Architecture VFS

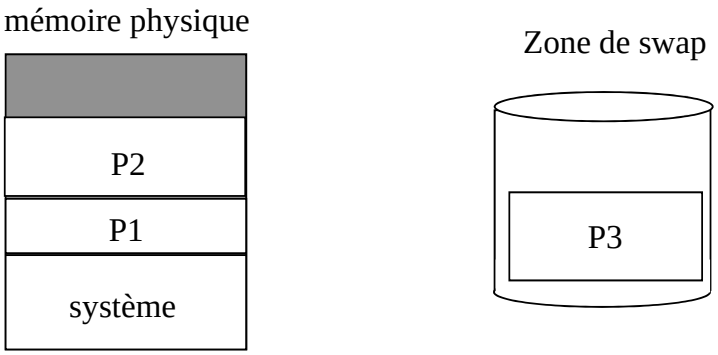


Mémoire virtuelle

1. Notions de base
2. Historique
3. Support Matériel
4. Etude de cas : 4.3BSD
   Pagination, Gestion du swap
5. Les nouveaux système de pagination : 4.4BSD - SVR4

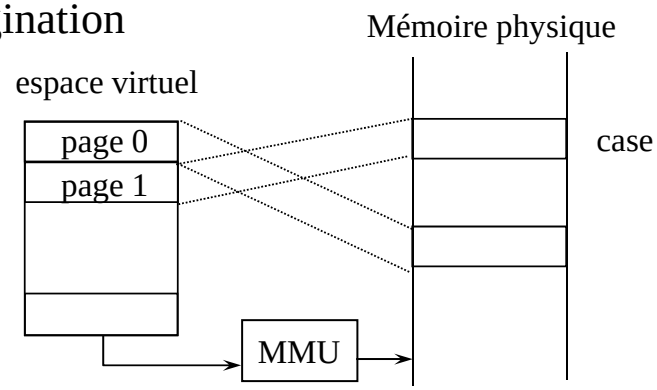
Notions de base

- Le swapping
  - Processus alloués de manière contiguë en mémoire physique
  - chargés /déchargés en entier
  - séparation du code pour optimiser la mémoire (segmentation)



Notions de base (2)

- La pagination



Le processus est partiellement en mémoire

  - les pages sont chargées **à la demande**
- Le remplacement de page
  - Evincer une page lorsqu'il n'y a plus assez de cases libres
- Notion d'espace de travail
  - ensemble des pages les plus utilisées par un processus (localité)

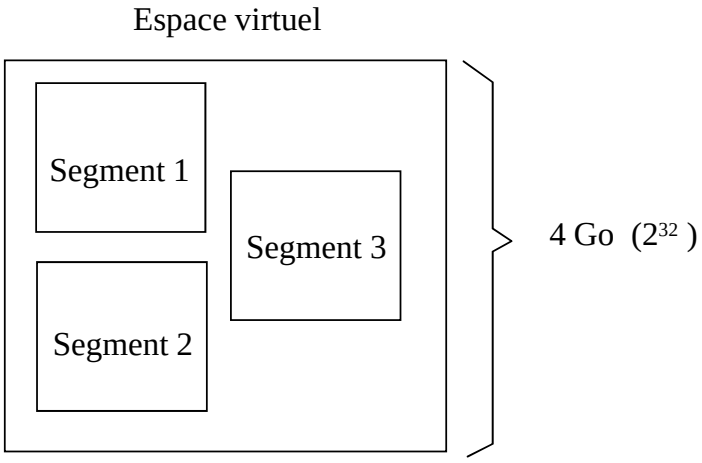
Historique

- Apparition tardive de la mémoire paginée dans Unix
- Jusqu'en 1978 utilisation exclusive du swap de processus
  PDP-11 16 bits
- 1979 introduction de la pagination
  3BSD sur vax-11/780 - 32 bits
  => 4 Go d'espace d'adressage
- Milieu de années 80 toutes les versions d'Unix incluent la mémoire virtuelle

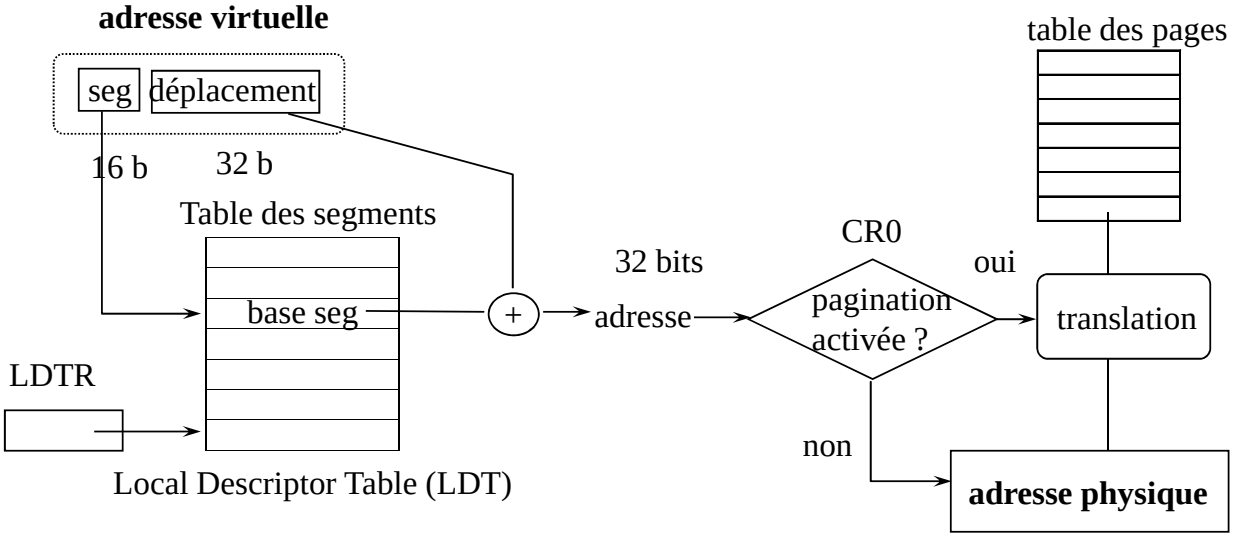
- Dans Unix, segmentation cachée à l'utilisateur, utilisée uniquement pour le partage et la protection

Support matériel :
Exemple Pentium

- A partir de Intel 80386 adresses sur 32 bits
  => 4 Go d'espace d'adressage
- Mémoire segmentée paginée



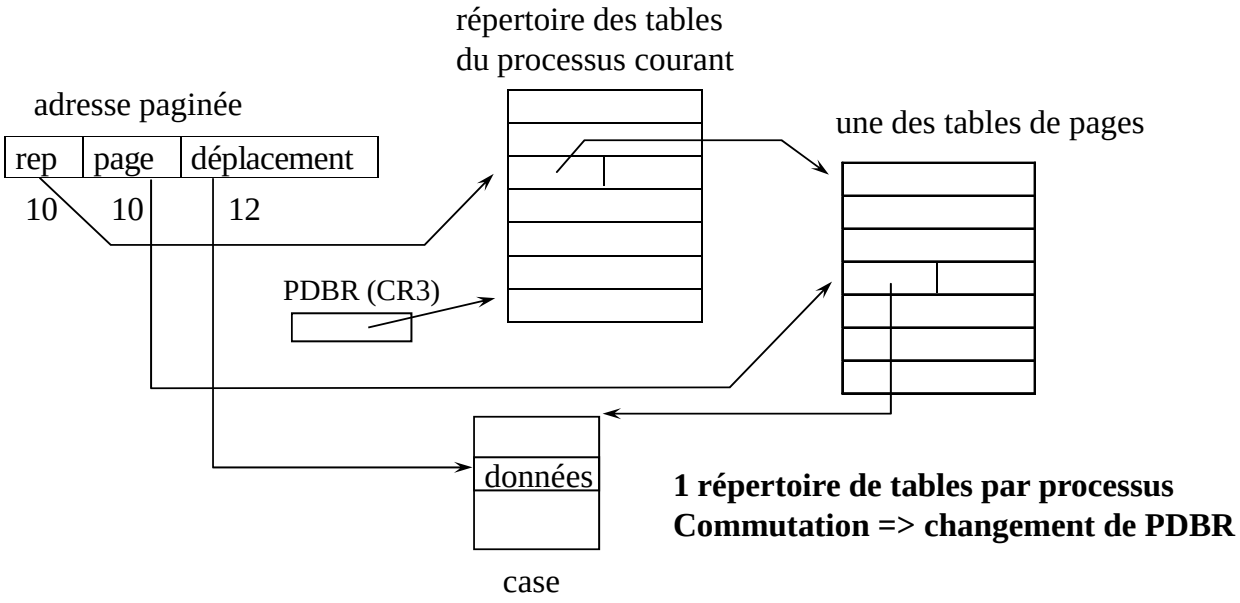
Architecture



- **1 table des segments (LDT) par processus**
- **1 table globale (Global Descripteur Table) = table des segments du système**
- 1 segment particulier : task state segment (TSS) pour sauvegarder les registres lors des commutations

Pagination multi-niveau

- Adressage 32 bits => impossible de maintenir table des pages du processus courant entièrement en mémoire (4 Mo par table !)



Format table des pages

D	Dirty bit (Modification)
A	Accessed bit (Référence)
U	User bit (0: mode utilisateur, 1: mode système)
W	Write bit (0: lecture, 1: écriture)
P	Present bit

Intel prévoit 4 niveaux de protection : Unix en utilise que 2 (util. / syst.)
En mode u les adresses hautes ne sont pas accessibles

Cache d'adresse : la TLB

- Problème de pagination multi-niveaux : accès aux tables
  - => temps d'accès fois 3 (2 niveaux - Intel x86),
    fois 4 (3 niveaux - Sparc),
    fois 5 (4 niveaux - Motorola 680x0)
- Effectuer la traduction uniquement au premier accès

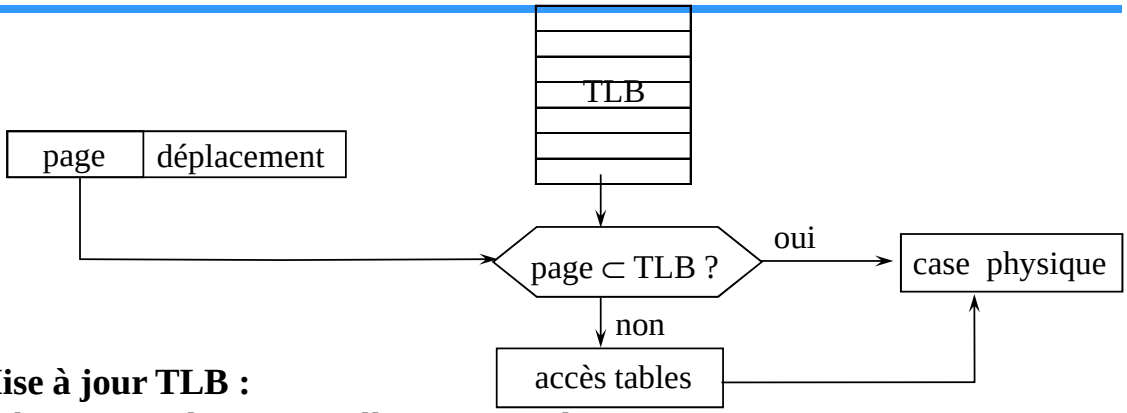
Mémoire associative : Translation Lookaside Buffer

Page	case

= cache des adresses

- TLB nombre d'entrées limité

Gestion de la TLB



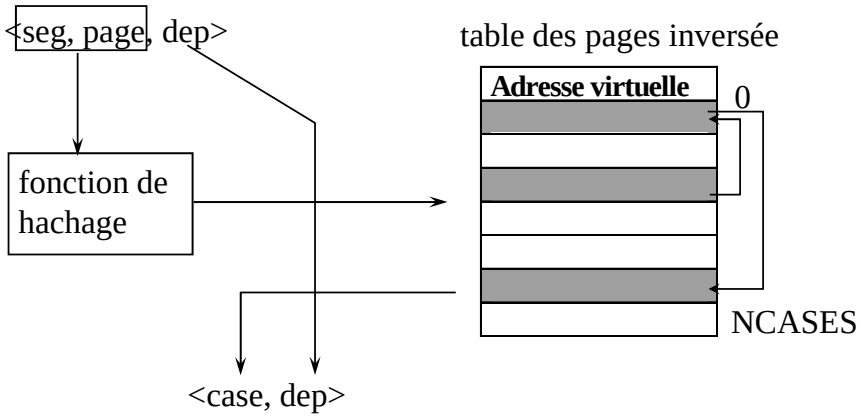
Mise à jour TLB :

- Chargement d'une nouvelle page pour le processus courant
- Commutation => invalidation de **toute** la TLB
  (automatique Intel x86 avec mise à jour PDBR)
- Déchargement page => invalidation entrée TLB
- Recouvrement (exec)

Autre approche : RS/6000

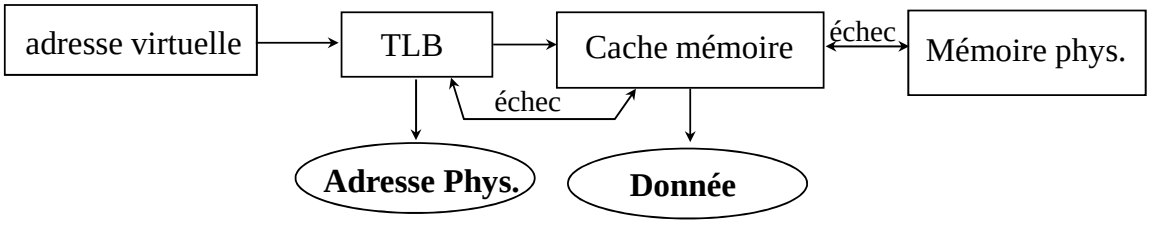
- Architecture RISC base pour AIX
- Utilisation d'une table des pages inversée = 1 entrée par **case** => taille réduite (page 4Ko, 32Mo de RAM => 128 Ko)
  1 seule table globale

adresse virtuelle du processus courant

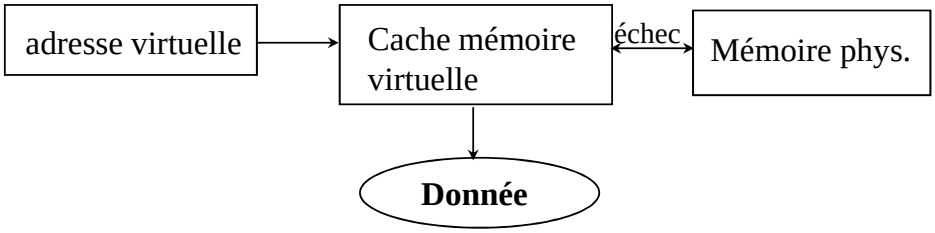


Architecture récente : Cache virtuel

- Architecture «classique» = 2 niveaux de cache mémoire

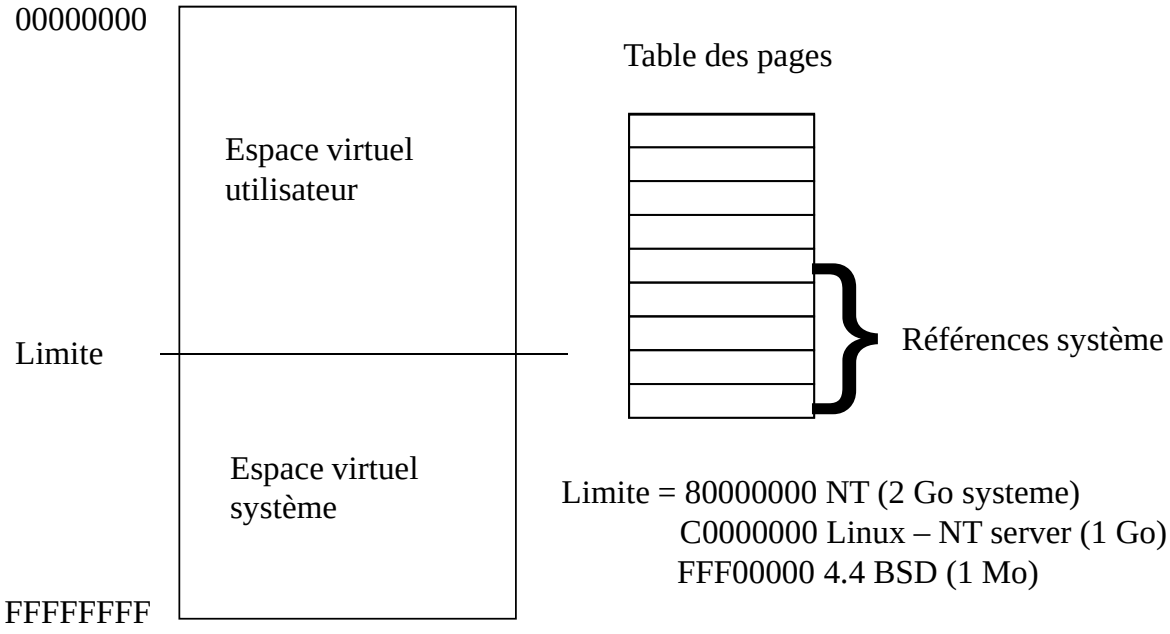


- Architecture à cache virtuel



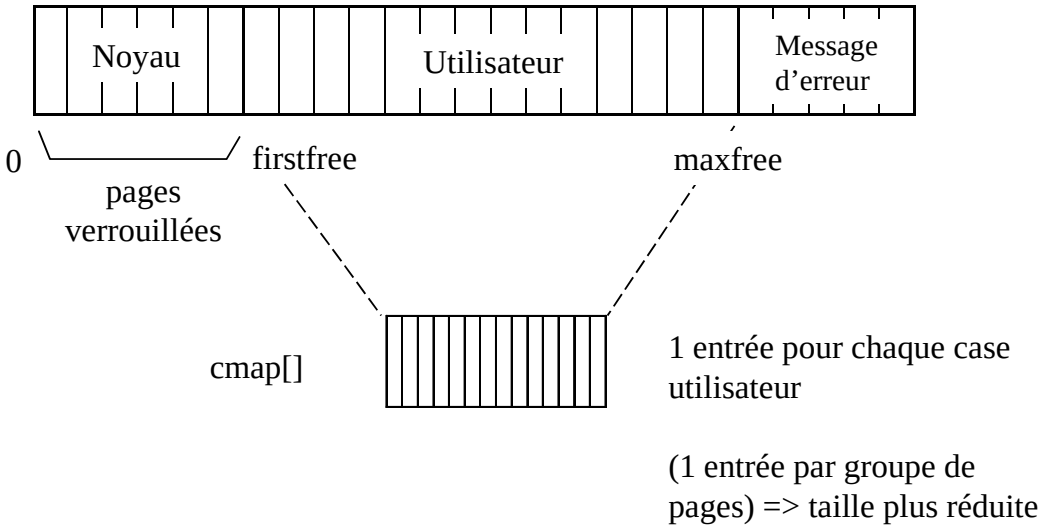
- Avantage : 1 seul niveau + pas de «flush» à la commutation

Les processus en mémoire



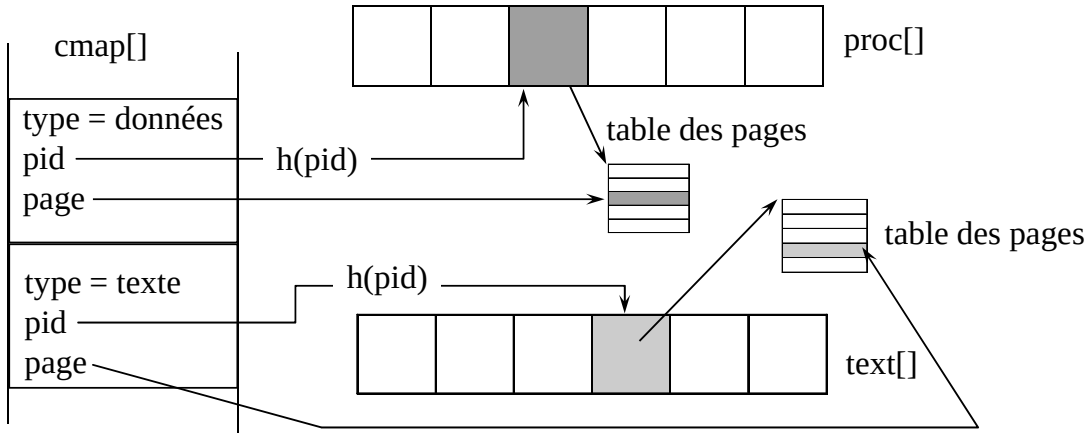
Etude de cas : 4.3BSD

Représentation de la mémoire physique



Structure de contrôle

- La structure cmap:
  - Noms : ID processus, Numéro de page, type (pile, données, texte)
  - Liens sur la freelist (listes des cases libres)
  - Synchronisation : verrous (pendant les chargements/déchargements)
  - Informations utiles pour le cache des pages de code



Etat d'une page

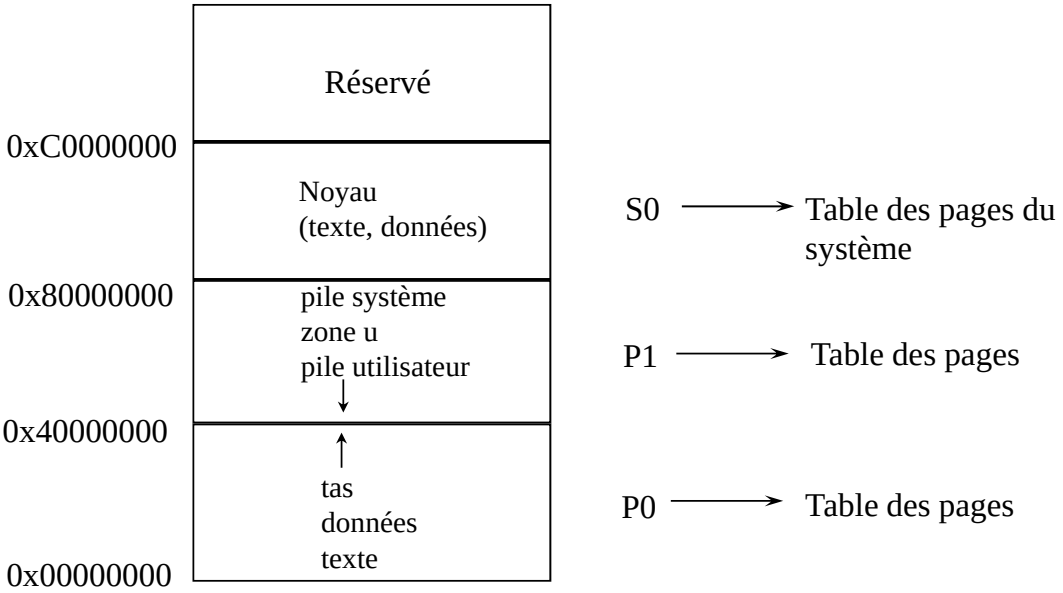
- Résidente : présente en mémoire physique
- Chargée-à-la-demande (Fill-on-demand):
  - Page non encore référencée qui doit être chargée au premier accès
  - 2 types :

    Fill-from-text : lue depuis un exécutable

    Zero-fill: page de pile ou de donnée créée avec des 0
- Déchargée (Outswapped)

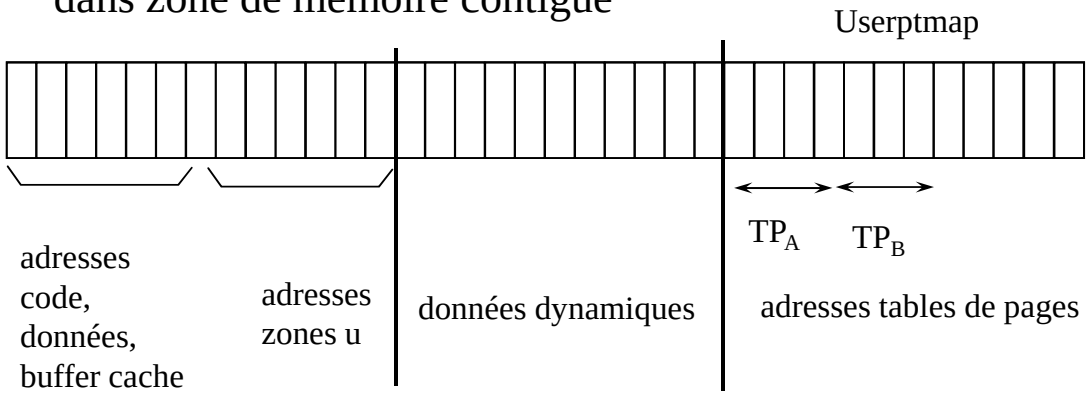
Structure de l'espace d'adressage

4.3BSD sur VAX-11 - adresses sur 32 bits => 4Go



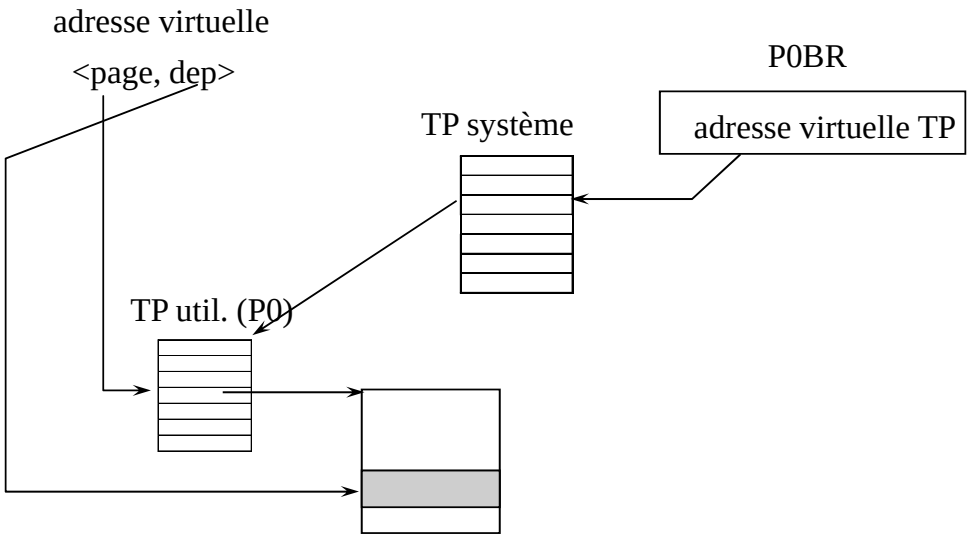
Organisation de l'espace virtuel noyau

- Table des pages du système (TPS) allouée statiquement dans zone de mémoire contiguë



- Tables des pages des processus contiguë dans l'espace virtuel du noyau

Accès aux données utilisateur



Double indirection (passage par la table du système)
=> fait une seule fois ensuite l'adresse est dans la TLB !

Défaut de page - pagein

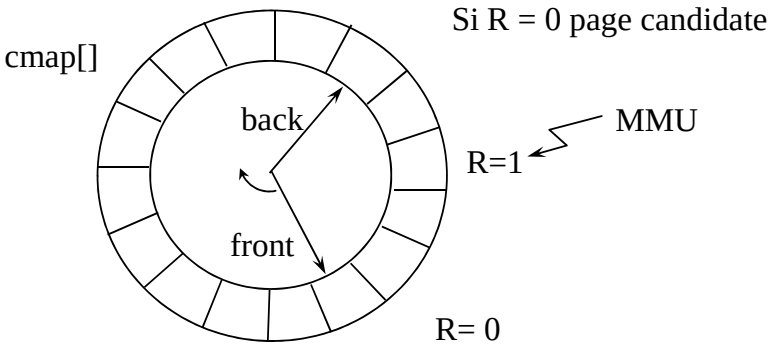
```
Pagein(adresse virtuelle) {
    Verrouiller la table des pages
    Si (adresse non valide) {
         envoyer SIGSEGV au processus;
         aller fin;
    Si  (page dans le cache des pages) { // page de code
         extraire page du cache
         mise à jour de table des pages;
         tant que (contenu page non valide)
                 sleep(contenu_valide);
    }
    sinon {
         attribuer une nouvelle case;
         Si (page non précédemment chargée et «Zero-fill») initialisée à 0
         sinon {
                 lire la page depuis le périphérique de swap ou fichier exécutable
                 sleep(E/S);
         }
    }
    wakeup(contenu-valide);
    Positionner bit valide; Effacer bit modifié;
    Déverrouiller;
}
```

Remplacement de pages

- Objectif: minimiser le nombre de défauts de page
- Idée : exploiter la localité des programmes
  - «Une page anciennement utilisée a une faible probabilité d'être référencée dans un futur proche»
- Algorithme LRU (**Least** Recently Used) trop coûteux
  => approximation de LRU : NRU «**Not** Recently Used»
- Choix d'un remplacement **global**
  => meilleure répartition des pages
  moins bon contrôle de nombre de défauts de page

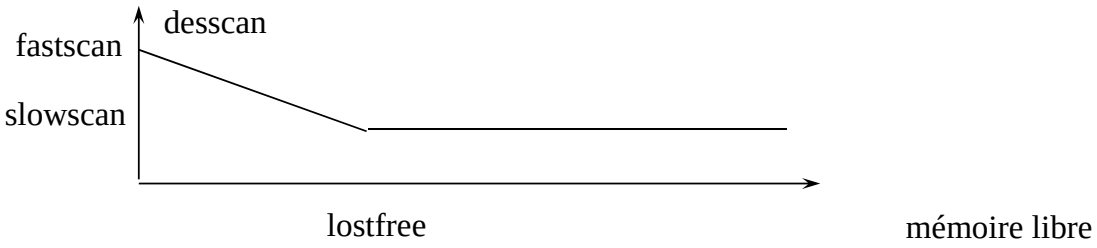
Implémentation du NRU

- Objectif : maintenir une liste de cases libres avec une taille minimum = freelist (taille = freemem).
- Utilisation du bit de référence positionné par la MMU
- 2 passes : 1) Mettre à 0 le bit de référence
  2) Tester (plus tard) ce bit, si toujours à 0 la page peut être récupérée si nécessaire



Le démon de pagination

- Maintient le nombre de cases libres au-dessus d'un seuil
- Réveillé 4 fois par seconde pour tester les cases
- Choix des pages victimes (NRU) à insérer dans freelist
  - si les victimes ont été modifiées, lancer une écriture asynchrone sur le swap
  - écriture terminée => insertion freelist
- Paramètres de bases :
  - Nombre de pages à tester (descan) en moyenne 20 à 30 % des pages testées par seconde
  - Arrêt du démon lorsque freemem > lostfree (= 25% mémoire utilisateur)



Le démon de pagination (2)

- Autres paramètres :
  - desfree : nombre de cases libres à maintenir par le démon (1/8 4.3BSD, 7% 4.4BSD (free_target), 6.25% System V R4)
  - minfree : nombre de cases minimum pour le système (1/16 4.3BSD, 5% 4.4BSD, 3% System VR4)
- Si freemem < minfree activer stratégie de swap
  - => déchargement de processus en entier
  - le démon n'arrive plus à maintenir assez de cases libres

Gestion du swap

- Gérer par le **swapper** (processus 0)
- rôle : charger (swapin) / décharger (swapout) des processus
- Dans les Unix récents intervient uniquement dans les cas de pénurie de mémoire importante

Quand décharger un processus ?

- 3 cas :
  - 1) Userptmap fragmentée ou pleine : impossible d'allouer des pages contiguës pour les tables des pages (propre à 4.3BSD)
  - 2) Plus assez de mémoire libre
    freemem < minfree (BSD)
    < GPGSLO (SVR4)
  - 3) Processus inactifs plus de 20 secondes
    (exemple : un utilisateur ne s'est pas déconnecté)
- => le processus victime est entièrement déchargé
  - Toutes les pages + zone u + tables des pages

Quel processus évincé

- 2 critères :
  - Temps processus endormi en mémoire
  - Taille du processus
- Choisi d'abord les processus endormis depuis plus de 20 sec. (maxslp)
- Si non suffisant : les 4 plus gros processus
- Si non suffisant : ???

Le swapper

- Algorithme de sched

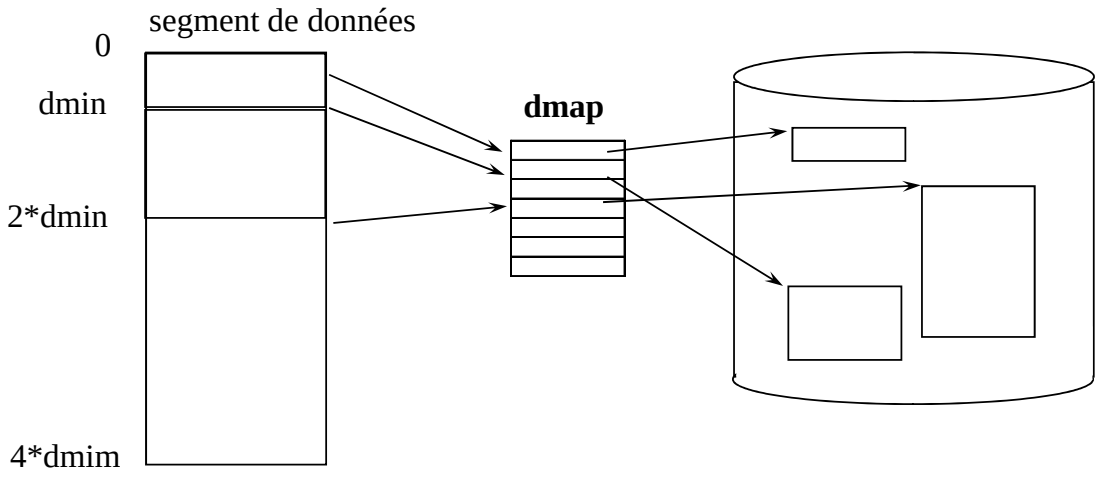
```
boucle
    recherche processus SRUN et non SLOAD le plus ancien
    si non trouvé
        alors sleep (&runout, PSWP); continuer
    sinon
        si swapin(p); continuer
        /* place insuffisante en mémoire */
        Si existe processus endormis ou en mémoire depuis longtemps
          alors swapout(p); continuer
          sinon sleep(&runin, PWSP);
        fin si
    fin si
fin boucle
```

Gestion de l’espace de swap

- Une ou plusieurs partitions (sans système de fichiers)
- Le swap est préalloué à la création du processus (pour les données et la pile)
  - => pouvoir toujours décharger un processus
- Swap du code :
  - Code déjà présent sur disque dans système de fichier
  - Swappé pour des raisons de performance !
  - Code swappé uniquement si plus utilisé par un processus en mémoire (champs x_ccount indique le nombre de processus en mémoire utilisant le code)

Espace de swap (2)

- Pour chaque segment une structure dmap stocké dans zone U
  - Premier bloc de taille 16K (= dmim)
  - Chaque bloc suivant est le double du précédent



Attention: 1 seule copie pour le code => dmap du code dans struct text

Algorithme swapout

- Swapout : décharger un processus sur disque

1- Allouer espace de swap pour zone U et table des pages
2- Décrémenter x_ccount, si x_ccount = 0 décharger les pages de code
3- Décharger les pages résidentes et modifiées sur le swap
4- Insérer toutes les pages déchargées dans freelist
5- Décharger table des pages, zone U, pile système
6- Libérer zone U
8- Mémoriser dans struct proc l'emplacement zone U sur disque
7- Libérer tables des pages dans Userptmap

Algorithme swapin

- swapin : chargement d'un processus

1- Allouer table de pages dans Userptmap
2- Allouer une zone U
3- Lire table des pages, zone U
4- Libérer espace table des pages, zone U sur disque
5- charger éventuellement le code et l'attacher au processus
6- Si le processus à l'état prêt (SRUN), l'insérer dans file des processus prêts

Création d'un processus

- BSD : données et pile dupliquées, code partagé
- **Swap** :
  - Allouer espace sur le swap pour le fils (données pile)
  - Espace pour le texte déjà alloué par le père (exec)
- **Table des pages** :
  - Allouer des pages pour les tables de pages du fils
    (trouver des entrées contiguës dans Userptmap, prendre des cases dans la freelist)
- **Zone U** :
  - créer une nouvelle zone U avec le contenu de la zone U du père
- **Code** :
  - Ajouter le fils dans la liste des processus partageant le code
  - x_count++, x_ccount++

Création de processus (2)

- **Données et pile** :
  - Pages référencées par les segments de données et de pile copiées
  - Pages marquées modifiées
  - Pages swappées copiées

=> très coûteux => Création d'un nouvel appel le **vfork**

- Constatation : le fork et très souvent suivi d'un exec
  => recopie inutile !
- vfork : pas de recopie en attendant le exec
  - Père et fils partagent le même espace d'adressage
  - Création uniquement de proc, zone U, table des pages
  - Père reste bloquer jusqu'à ce que le fils fasse exec ou exit (pb de cohérence)

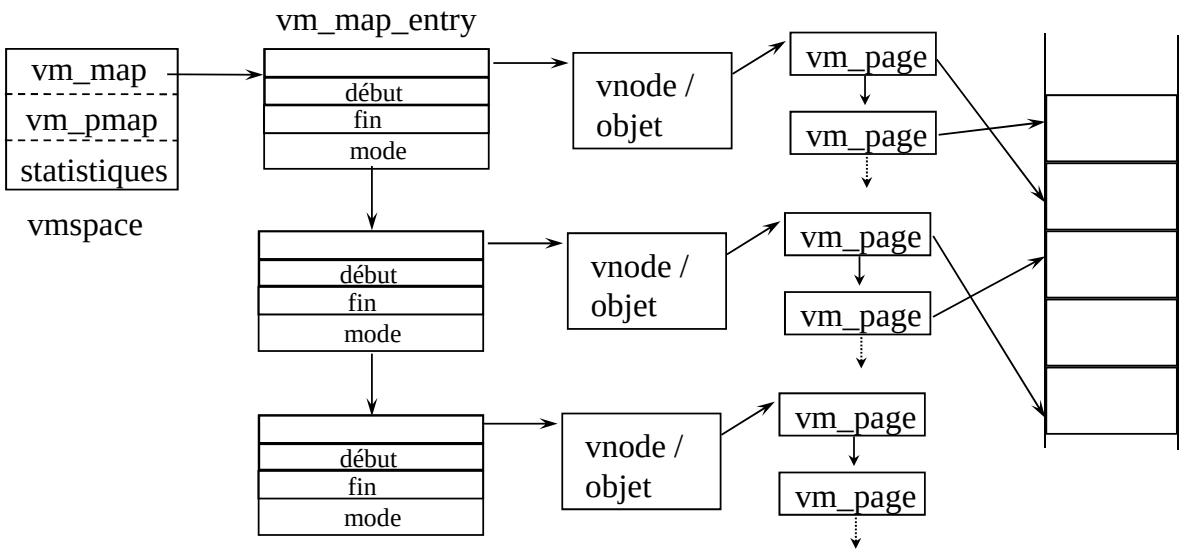
Les nouveaux systèmes

- Système V Release 5
- Solaris
- 4.4 BSD
- Linux

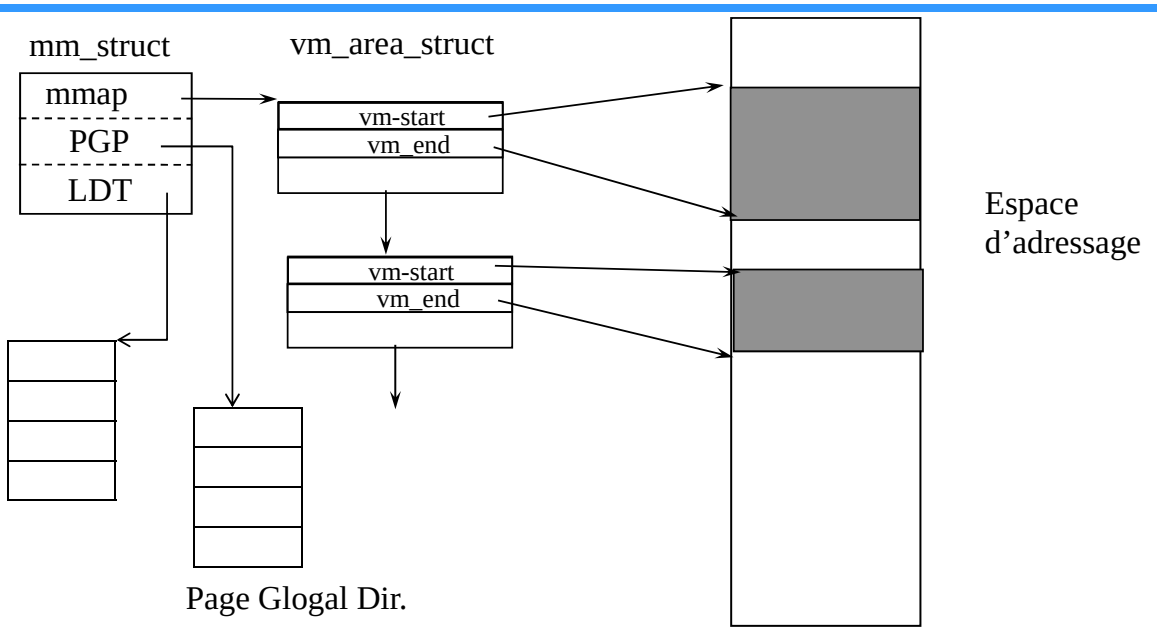
} Mémoires virtuelles très proches

- Nouveautés :
  - Structures générales
  - Fichiers «mappés»
  - Copie-sur-écriture

Structure d'un espace 4.4BSD

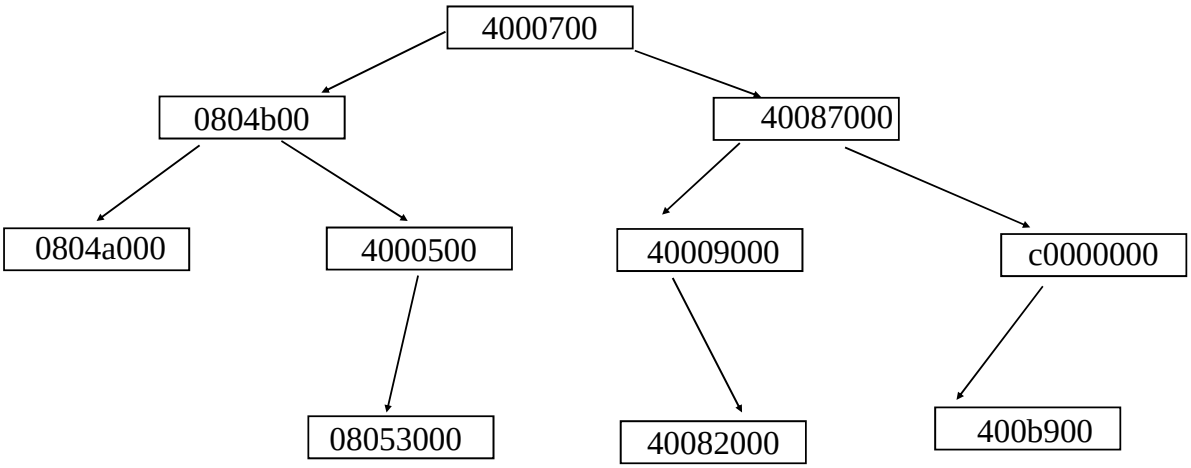


Structure dans Linux



Organisation de l'espace mémoire d'un processus

- Lorsque beaucoup de régions
  - Liste de région => arbre des régions (arbre AVL)
- Ex : linux : /proc/pid-processus/maps



Visualisation mémoire sous linux

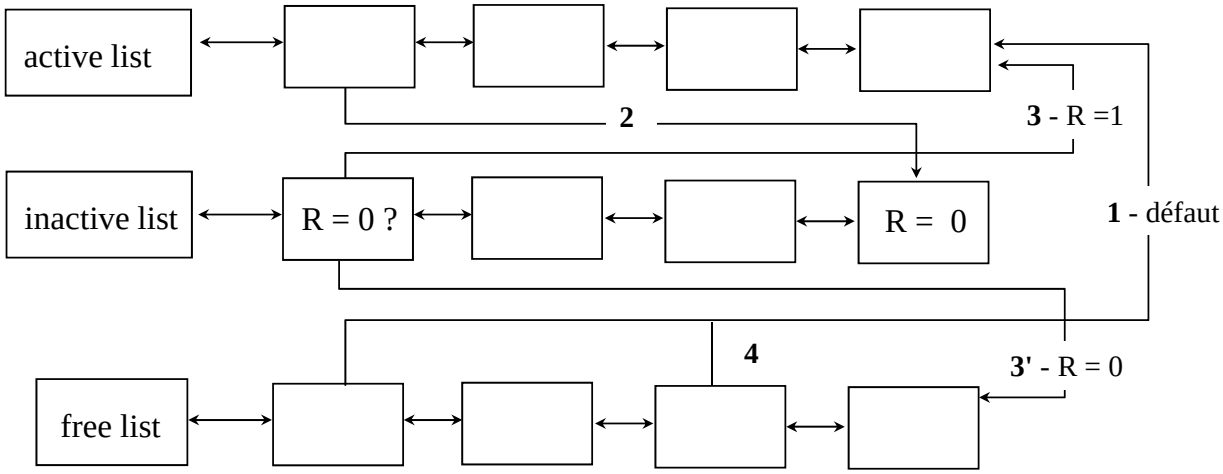
```
# more /proc/1/maps
08048000-0804f000 r-xp 00000000 03:06 80252      /sbin/init
0804f000-08051000 rw-p 00006000 03:06 80252      /sbin/init
08051000-08055000 rwxp 00000000 00:00 0
40000000-40012000 r-xp 00000000 03:06 69906      /lib/ld-2.1.3.so
40012000-40013000 rw-p 00011000 03:06 69906      /lib/ld-2.1.3.so
40013000-40014000 rw-p 00000000 00:00 0
4001d000-400fc000 r-xp 00000000 03:06 69912      /lib/libc-2.1.3.so
400fc000-40101000 rw-p 000de000 03:06 69912      /lib/libc-2.1.3.so
40101000-40104000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Les objets et paginateurs

- Un **paginateur** par type d'objet
  => chargement/déchargement des pages de l'objet

- Structure vm_pmap : dépendante de la machine
  Conversion adresse physique <--> adresse logique
  Fonction de manipulation de la table de page
  - Gérer les protections (copie-sur-écriture)
  - Mise à jour
  - Création ..

Remplacement de pages



Algorithme: Fifo avec seconde chance

Optimisation : copie sur écriture

- Objectif : éviter les recopies du fork
- Autoriser le partage en écriture
  - segment de pile de données partagées, les pages sont recopiées uniquement si elles sont modifiées