Cours n° 1

Introduction au Noyau

Supports conçus par Pierre Sens

Cours donné par G. Thomas

Objectifs

Mécanismes internes du noyau (UNIX)

Processus,
Ordonnancement
Fichiers,
Mémoire virtuelle

Organisation

Début TD/TME : Aujourd'hui

Equipe enseignante

Gaël Thomas : gael.thomas@telecom-sudparis.eu

Le cours AISE est basé sur le cours NOYAU donné à l'UPMC Sorbonne Universités

Bibliographie

Programmation système

J.M. Rifflet, «La Programmation sous UNIX»,
J.M. Rifflet, «Les communications sous UNIX»,
C. Blaess, «Programmation système en C sous Linux»

Mécanismes internes du noyau UNIX

M.J. Bach, «Conception du système UNIX»,
S.J Leffler & al. , «Conception et implémentation du système 4.4 BSD»,
U. Vahalia, «Unix internals --the new frontiers»
Noyau Linux :
D. P. Bovet, M. Cesari, « Le noyau Linux »
R. Love, « Linux Kernel Development »
R. Card & al., «Programmation Linux 2.0»
Linux Magazine Sept./Octobre 2003 - « Voyage au centre du noyau

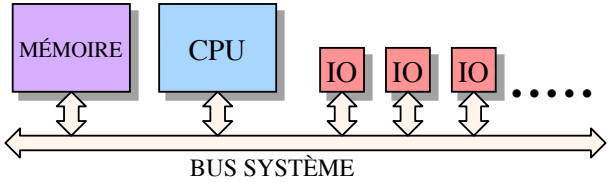
Mécanismes internes Windows

Inside Windows 2000, 3rd Edition

Vue d'une machine

Ensembles de composants organisées autours de bus

Composants de base: mémoire, CPU, I/O, bus système



I/Os "standard": cartes SCSI et/ou IDE, clavier, souris, haut-parleurs, etc.

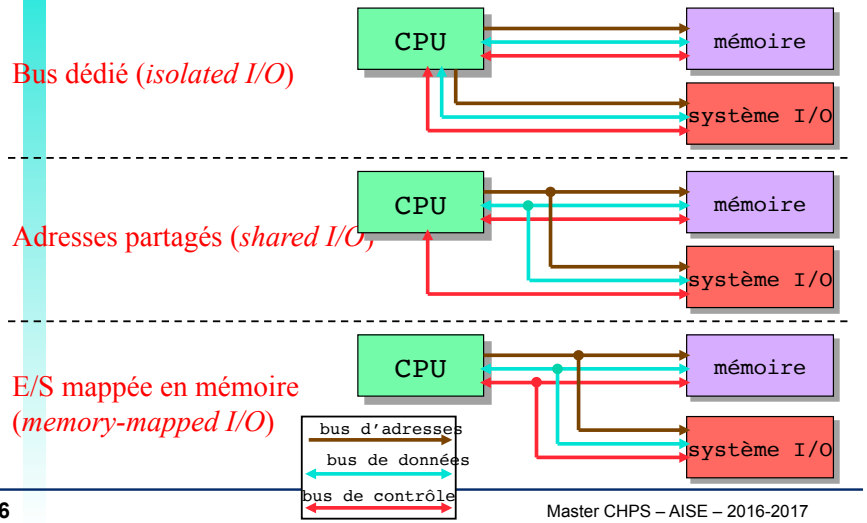
La vitesse du bus système devient le facteur prédominant pour la performance d'un ordinateur.

Vue d'une machine : Accès aux périphériques

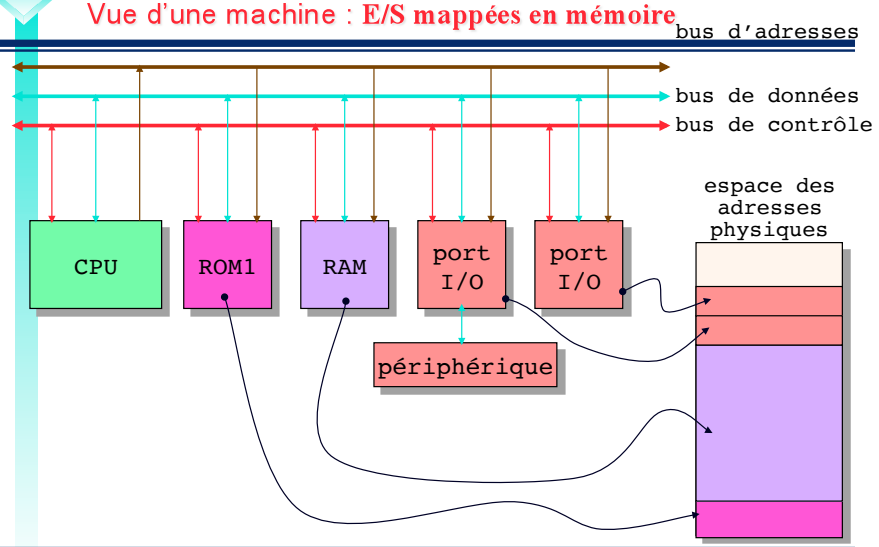
Bus dédié (*isolated I/O*)

Adresses partagés (*shared I/O*)

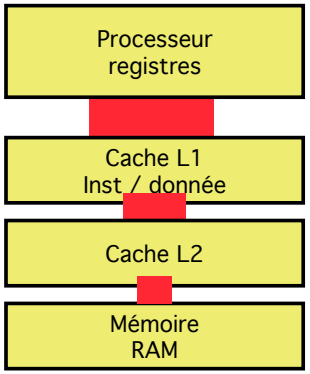
E/S mappée en mémoire (*memory-mapped I/O*)



Vue d'une machine : E/S mappées en mémoire



Vue d'une machine : Les niveaux de caches



Temps accès < 1ns

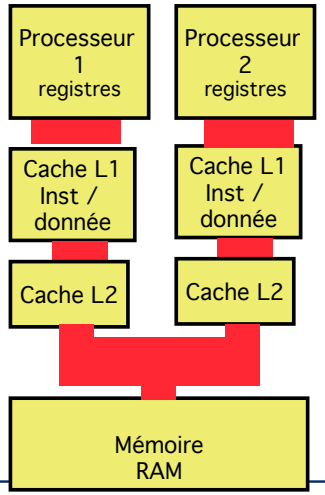
Temps accès ~1ns

Temps accès 2-3 ns

Temps accès 10-50 ns

Disque : 10 ms (x10^6 RAM !)

Vue d'une machine : Multi-processeur SMP



Architecture *Symmetric Multiprocessor (SMP)*

• Processeurs distincts, pas de partage de caches

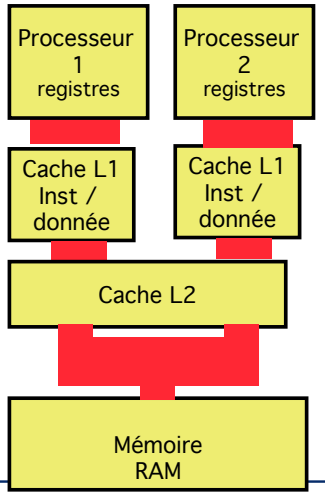
• Partage de la RAM

+ Gestion de flux indépendants

- Maintien de la cohérence des caches

- Conflit d'accès au bus mémoire (nb de processeurs limités)

Vue d'une machine : Multi-core



Architecture *Multi-core*

(ex. Intel Dual/Quad core, Cell)

• Deux cœurs de processeurs distincts sur un même support, partage de cache possible (L2 ou L3)

• Partage de la RAM

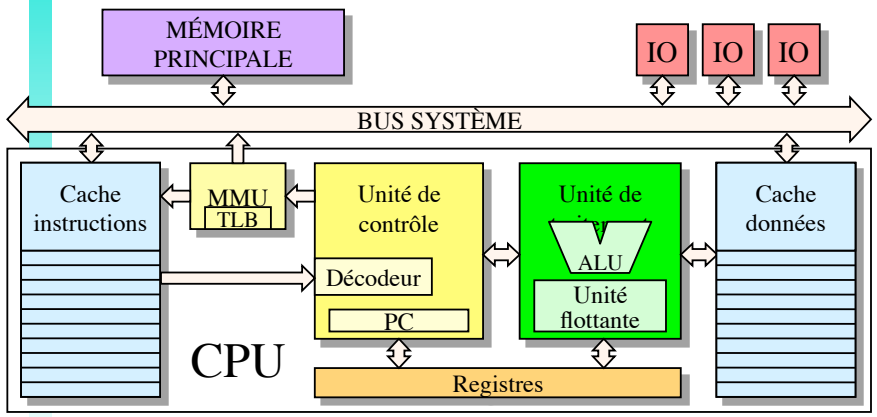
+ Gestion de flux indépendants

+ Moins coûteux que SMP

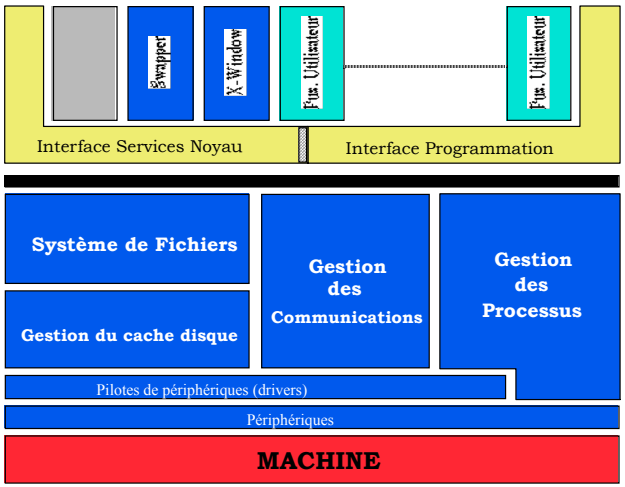
+ Moins de pb de cohérence

- Moins de cache disponible

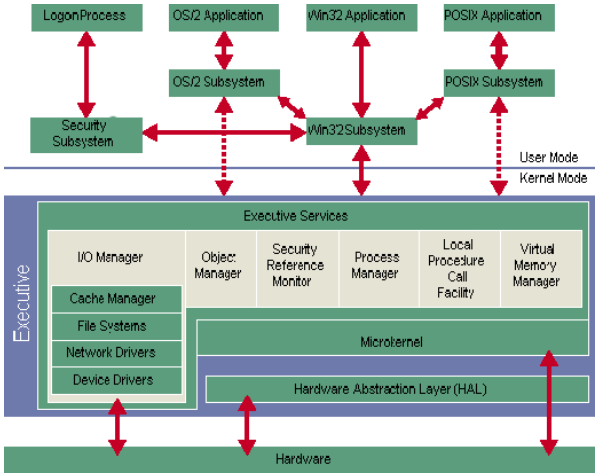
Vue d'une machine



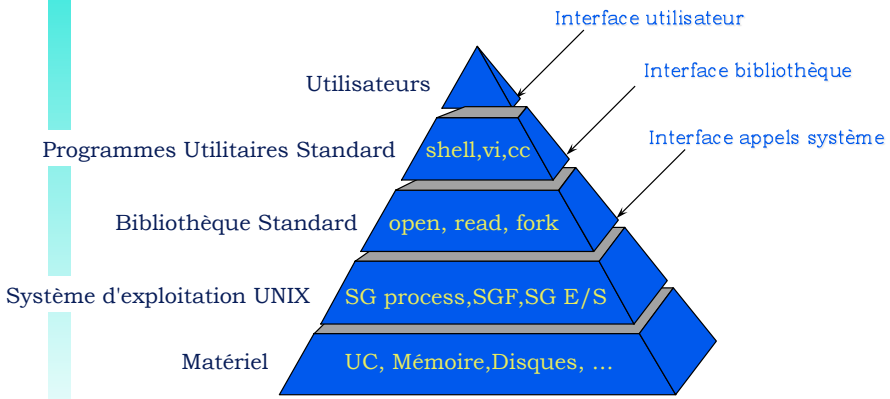
Vue du système : architectures classiques



Vue du système : architectures modulaire (Windows NT)



Généralités (1)



Généralités (2)

Système interactif en temps partagé, Multi-Utilisateurs et Multi-Tâches

Principes

- Système de gestion de fichiers hiérarchisé
- Entrées/Sorties
- Création dynamique de processus (Père / Fils)
- Communication inter-processus (Pipes, Sockets)

Langage de commande extensible (Shell)

Noyau monolithique portable

- Le noyau est écrit en C à 95%.
- UNIX existe sur de nombreuses machines (PC, Stations RISC, CRAY-YMP, Hypercubes, ...)

Les objectifs

Simplicité et efficacité (par opposition aux gros systèmes MULTICS ...):

- **Efficacité dans la gestion des ressources**

Fournir des services d'exécution de programmes

- **Charger, Exécuter, Gérer les erreurs, Terminaison**
- **Entrées / Sorties à partir de périphériques (Créer, Lire. Ecrire, ...).**
- **Détecter les erreurs (CPU, mémoire, E/S, ...).**

Fournir des services d'administration

- **Allocation des ressources système.**
- **Gestion des utilisateurs.**
- **Comptabilité et statistiques (accounting)**
- **Configuration.**
- **Protection des ressources.**
- **Ajout et retrait de gestionnaires de périphériques (drivers).**

Tentatives de Normalisation

POSIX : Compromis entre BSD et Systeme V

Proche de V7 de Bell Labs + signaux + gestion des terminaux

O.S.F (Open Software Fundation) : IBM, DEC, HP, …

Conforme aux normes IEEE + outils

X11 : système de fenêtrage,
MOTIF : interface utilisateur
DCE : calcul réparti
DME : gestion répartie
…

U.I (UNIX International) : AT&T, Sun, …

Système V release 4.0

mais aussi ...

AIX (IBM), Spix (Bull), Ultrix (Digital), HP-UX (HP), SCO-UNIX (SCO), SunOS & Solaris(Sun Microsystems), … LINUX, FreeBSD, ...

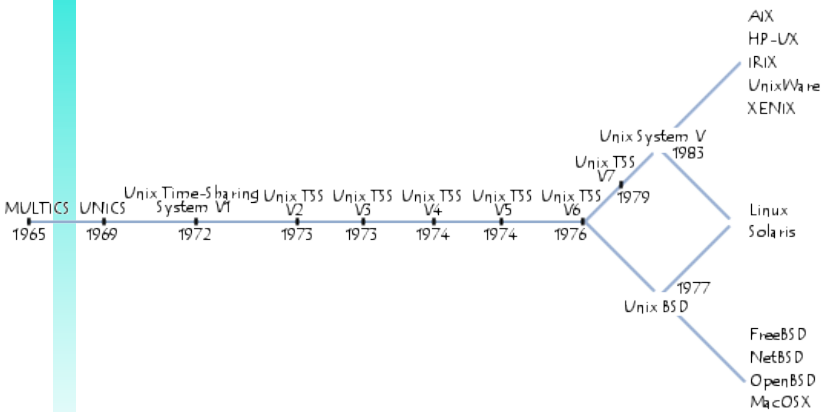
Historique (1)

A l 'origine UNICS (UNIplexed information and Computing System)

1969 UNIX/PDP-7 Ecrit en assembleur sur un PDP-7 pour développer un traitement de texte aux Bell Labs
1973 UNIX V5 Langage C (90%)
1976 UNIX V 6 (coopération de Bell Labs/universités américaines)

	Berkeley	Bell Labs	AT&T
1977	1.0 BSD	-	-
1978	2.0 BSD	V7	-
1979	3.0 BSD	-	-
1980	4.0 BSD	-	-
1981	4.1 BSD	-	-
1982	-	-	System III
1984	4.2 BSD	V8	System VR2
1986	4.3 BSD	-	-
1989	-	V 10	-
1993	4.4 BSD	-	System VR4
1991		**Linux**	

Historique (2)



Historique (3)

POSIX (Portable Open System Interface eXchange)

1993 - Compromis entre BSD et Systeme V

1003.0 Guide et présentation
1003.1 Appels Systèmes
1003.2 Shell et Utilitaires
1003.3 Méthodes de tests et conformité
1003.4 Extensions temps réel
1003.6 Sécurité
1003.7 Administration système
1003.8 Accès commun aux fichiers
1003.9 Appels Fortran-77
1003.10 Interface pour les supercalculateurs
1003.11 Extensions pour le transactionnel
1003.12 Extensions communications inter processus
...

Types d'API

Contrôler l'exécution d'un processus,

Faire des accès et de la gestion sur le système de fichiers,

Faire et contrôler des accès réseau,

Créer et gérer de l'espace mémoire,

Envoyer et recevoir des messages entre les processus,

Gérer ou s'informer sur l'état du système,

Contrôler des permissions d'accès,

Allouer et gérer des ressources diverses

Limites d'Unix

Complexité de certaines versions => problèmes de robustesse

Interface utilisateur

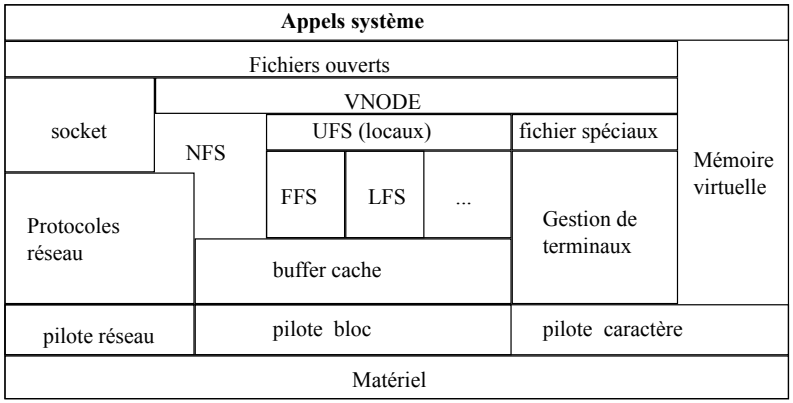
Prolifération des versions => situation chaotique

Approche monolythique => difficilement extensible

Mécanismes Internes

- Architecture générale
- Gestion de processus
  - Ordonnancement/ Signaux / Multi-thread
- Gestion de fichiers
  - SGF / Entrées-sorties disque
- Gestion mémoire
  - Mémoire virtuelle

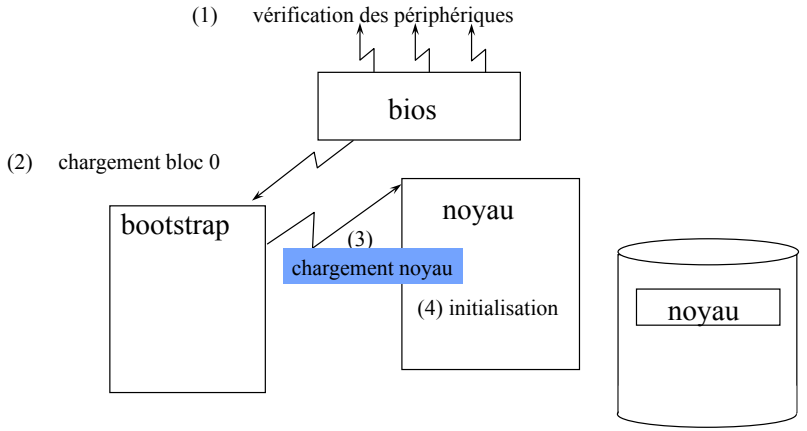
Architecture générale



Accès au service du noyau

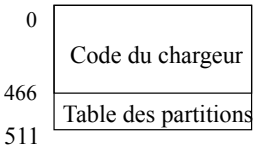
- Appels système
  - A l'initiative du processus courant (synchrone)
  - Utilisation de la pile système du processus courant
- Interruptions matérielles
  - Evénements externes asynchrones indépendants du processus courant (exemple : périphériques d'E/S)
  - Utilisation de la pile d'interruption
- Trappes matérielles
  - Evénements externes liés au processus courant (ex : division par zéro)
- Interruptions logicielles
  - Utiliser par le noyau pour demander une action à un moment précis (ex : délivrer un message à un processus).

Démarrage



boot bloc et multi-boot

Format :



Multi-boot (grub, lilo, boot manager) :

Changer le code du chargeur du boot bloc de la première partition (MBR)

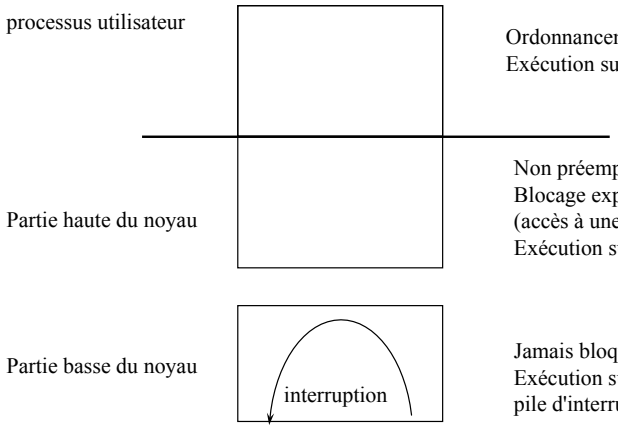
Ex: lilo

1) Exécution chargeur lilo (inclus dans bloc 0 du premier disque)
   `lilo:`
2) Choix de la partition à booter
3) Lire la table des partition pour trouver le boot bloc de la partition
4) Exécuter le chargeur du boot bloc trouvé

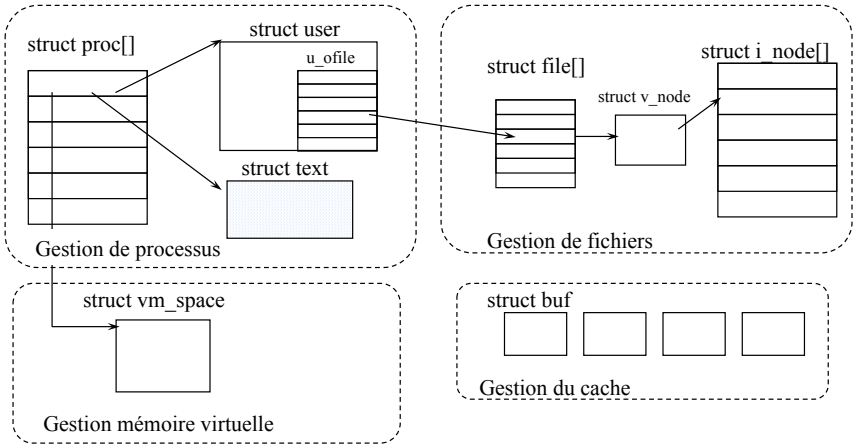
Bios vs EFI

- Limitation du BIOS
  - Mode 16 bits
  - Utilisation du MBR => taille de partition limitée :
    - Entrée de 32 bits :
    - Taille max = 2^32 blocs = 2 To
- => EFI (Extensible Firmware Interface)
  - 1 partition pour stocker les information de boot **GUID Partition Table (GPT)**
  - GPT contient adresse programme d'armossage et le table des partitions
  - Entrée sur 64 bits => Taille max = 2^64 blocs = 9.44Zeta octets !

Organisation et mode d'exécution



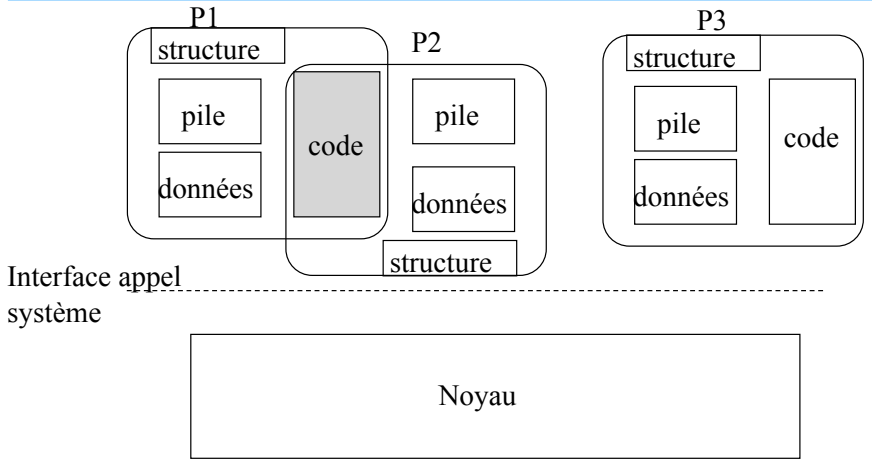
Les principales structures



Gestion de processus

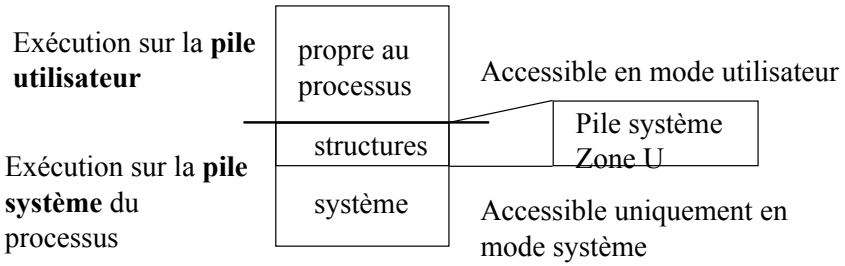
1. Architecture - Mode d'exécution- Etats
2. Création/terminaison
3. Signaux
4. Les processus du système/ Initialisation du système
5. Ordonnancement
6. Processus légers – threads
7. Linux
8. Windows

Architecture



Mode d'exécution

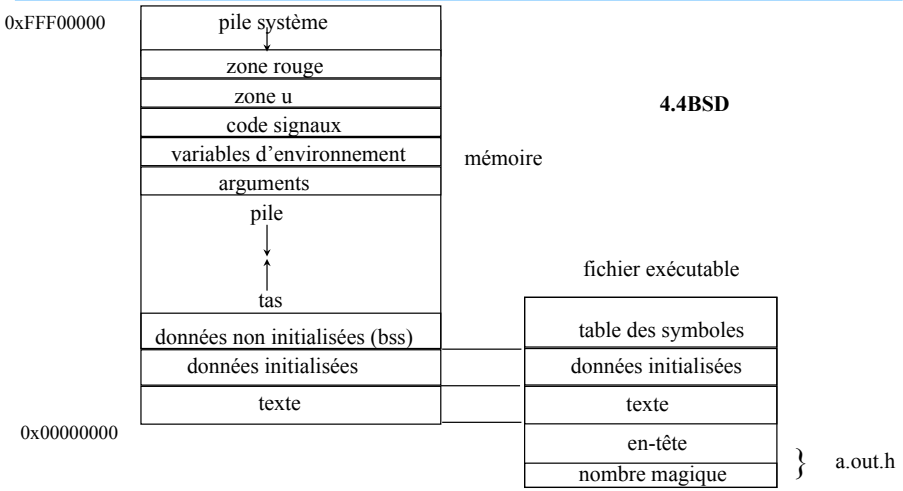
- Deux modes : utilisateurs / systèmes
  - OS/2 3 niveaux, Multics 7
- => 2 zones de mémoire virtuelle :
  - le noyau fait partie de l'espace virtuel du processus courant !



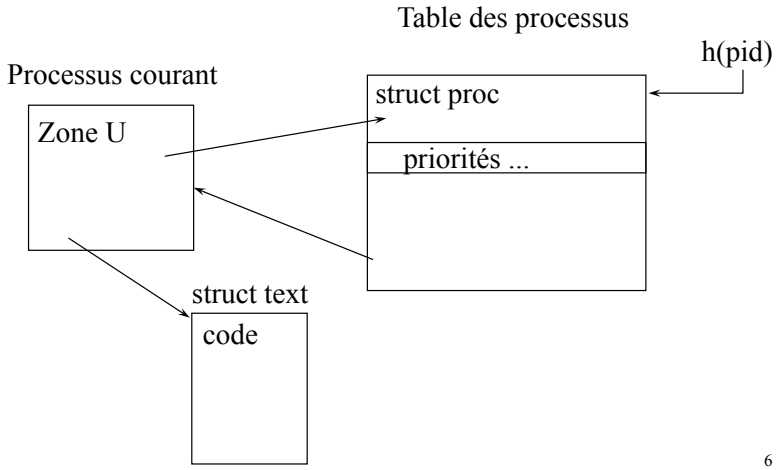
Structure interne

- Contexte :
  - Espace utilisateur (données, pile)
  - Information de contrôle (zone u, struct proc)
  - Variables d'environnement
- Contexte matériel :
  - Compteur ordinal
  - Pointeur de pile
  - Mot d'état (Process Status Word) : état du système, mode d'exécution, niveau de priorité d'interruption
  - Registre de gestion mémoire
  - registres FPU (Floating point unit)
- Commutation => sauvegarde du contexte mat. dans zone u (pcb : process control bloc)

Processus en mémoire et sur disque



Les structures en mémoire



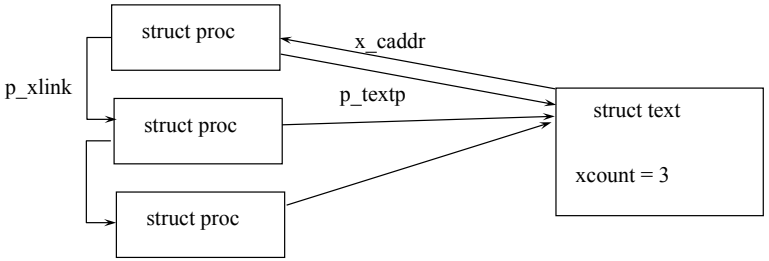
Structure - Zone U

- Zone u (struct u - user.h) :
- Fait partie de l'espace du processus => swappable
  - pcb
  - pointeur vers struct proc
  - uid et gid effectif et réel
  - arguments, valeurs de retour , erreurs de l'appel système courant
  - information sur les signaux
  - entête du programme
  - table des fichiers ouverts
  - pointeurs vers vnodes du répertoire courant, terminal
  - statistiques d'utilisation CPU, quotas, limites
  - [pile système]

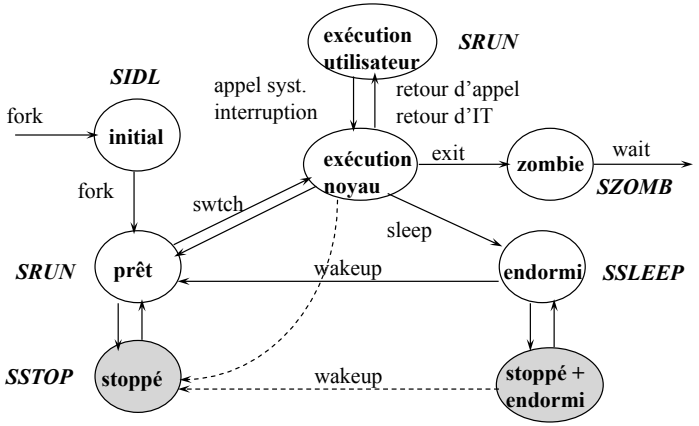
Structure résidente

- Struct proc - proc.h
  - pid, gid
  - pointeur zone U
  - état du processus
  - pointeurs vers liste de processus prêts, bloqués ...
  - événement bloquant
  - priorité + information d'ordonnancement
  - masque des signaux
  - information mémoire
  - pointeurs vers listes des processus actifs, libres, zombies

Partage de code



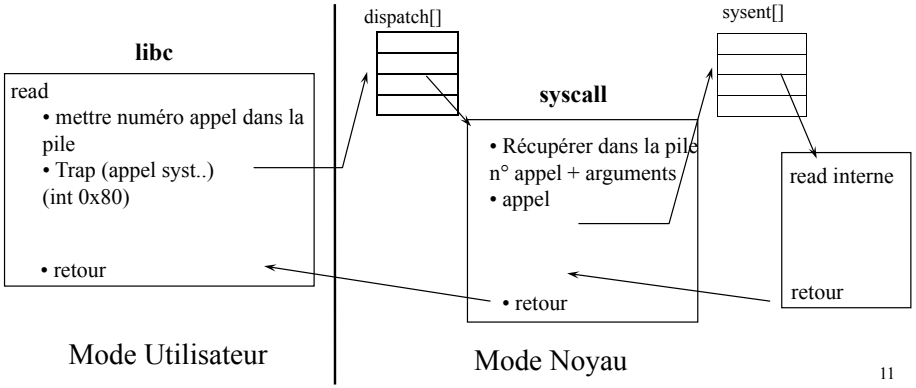
Etat d'un processus



BSD

Interface des appels systèmes

- Appels système encapsulés par des fonctions de librairie
- Chaque appel est identifié par un numéro



Algorithme de syscall

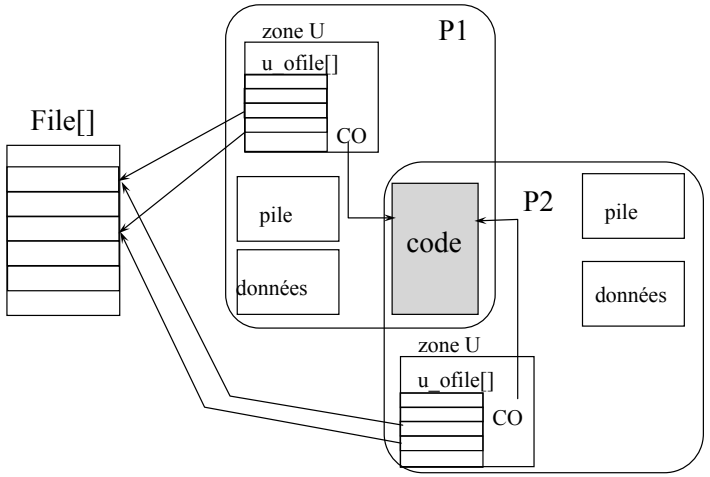
- Trouver les paramètres dans la pile du processus
- Copier les paramètres dans zone U (champs u_arg)
- Sauvegarder le contexte en cas de retour prématuré (interruption par des signaux)
- Exécuter l'appel
- Si erreur : positionner le bit report du mot d'état
  mettre le numéro d'erreur dans un registre

- Au retour de l'appel tester le bit report

Création/Terminaison

- Fork : Créer un fils à l'image du père
  1. Réserver espace de swap
  2. Allouer un nouveau PID
  3. Initialiser struct proc
  4. Allouer tables zone de mémoire virtuelles
  5. Allouer zone U (copier du père)
  6. Mise à jour zone U
  7. Augmenter le nombre de processus partageant le code
  8. Dupliquer données + piles du père
  9. copier le contexte matériel du père
  10. Mettre le fils à l'état prêt + insertion dans la file
  11. retourner 0 au fils
  12. retourner nouveau pid au père

Création (2)



Exec : invocation d'un nouveau programme

1. Vérifier le nom de l'exécutable et si l'appelant a les droits d'accès
2. Lire l'entête et vérifier si l'exécutable est valide
3. Si le fichier a les bits SUID ou SGID positionnés, affecter les UID ou GID effectifs au propriétaire du fichier
4. Copier les arguments et variables d'environnement dans le noyau
5. [Allouer espace de swap pour les données et pile]
6. Libérer l'ancien espace d'adressage et les zones de swap associées
7. Allouer tables pour code, données et piles
8. Initialiser le nouvel espace d'adressage. Si le code est déjà utilisé le partager
9. Copier les arguments et l'environnement dans espace utilisateur
10. Effacer les routines de traitement de signaux définies. Masques de signaux restent valides
11. Initialiser le contexte matériel (registres)

Terminaison

1. Annuler tous les temporisateurs en cours
2. Fermer les descripteurs ouverts
3. Sauver la valeur de terminaison dans le champs p_xstat de la structure proc
4. Sauver les statistiques d'utilisation dans champs p_ru
5. Changer le processus à l'état SZOMB et mettre le processus dans la liste des processus zombies.
6. Libérer l'espace d'adressage, zone u, tables de pages, espace de swap
7. Envoyer le signal SIGCHLD au père (ignorer par défaut)
10. Réveiller le père si il était endormi (wakeup)
11. Appeler swtch() pour élire un nouveau processus

Gestion des signaux

- Structures
  - Dans la zone U :
    - u_signal[] routines de traitements
    - u_sigmask[] masque associé à chaque routine
    - ...
  - Dans struct proc :
    - p_cursig masque des signaux “pendants”
    - p_sig signal en cours de traitement
    - p_hold masque des signaux bloqués
    - p_ignore masque des signaux ignorés

Signaux : Génération

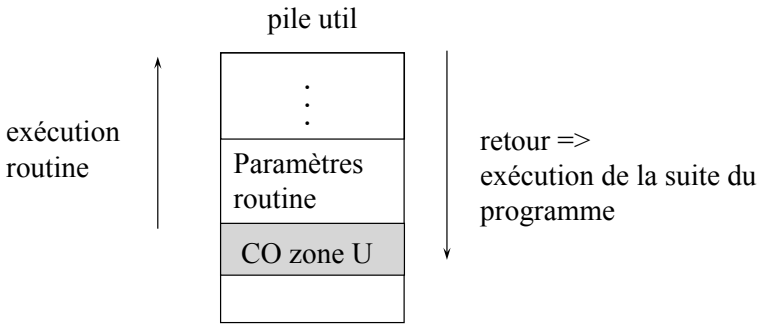
- Lors d’un “kill”
  - Chercher la structure proc du processus cible
  - Tester p_ignore, si signal ignoré retourner directement
  - Ajouter le signal dans p_cursig
  - Si le processus est bloqué dans le noyau, le réveiller (rendre prêt)
- => 1 seul traitement pour plusieurs instances du même signal
- Le signal ne sera traité que lorsque le processus cible passera sur le processeur

Signaux : traitement (1)

- Vérifier la présence de signaux : appel à issig
  - issig est appelé lors : retour au mode utilisateur (après appel système ou interruption)
  - issig :
    - Vérifier les signaux positionnés dans p_cursig
    - Vérifier si le signal est bloqué (test de p_hold)
    - Si non bloqué mettre le numéro de signal dans p_sig
    - retourner TRUE
- Si issig retourne TRUE traiter le signal : appel de psig
  - psig:
    - Trouver la routine de traitement dans u_signal du processus courant
    - Si aucune routine exécuter le traitement par défaut
    - ...p_hold |= ...u_sigmask
    - Appel de sendsig qui exécute la routine lors du retour en mode util.

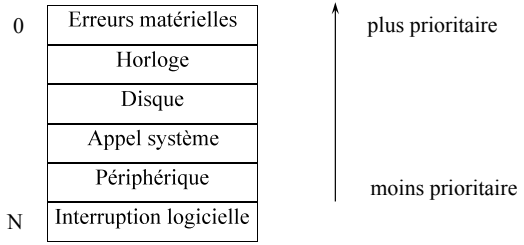
Signaux : traitement (2)

- sendsig : appel dépendant de la machine



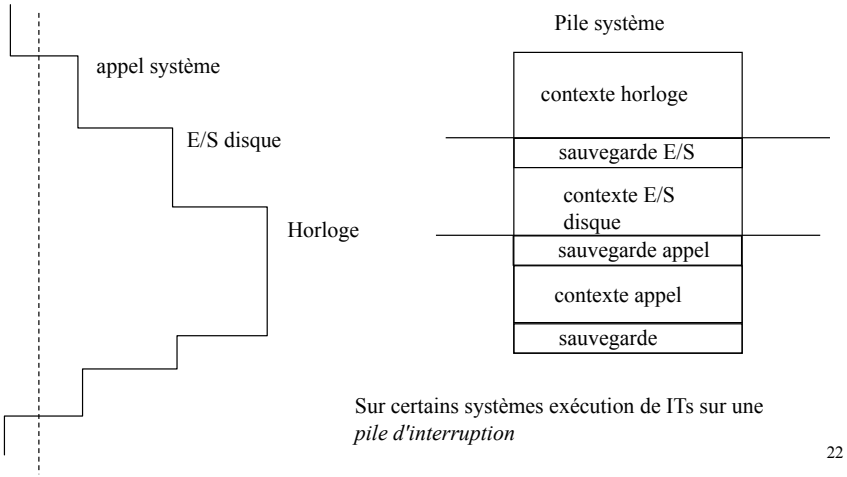
Les interruptions

- Pour chaque interruption, un niveau de priorité (ipl: interrupt priority level)
- 7 niveaux Unix de base, 32 niveaux Unix BSD



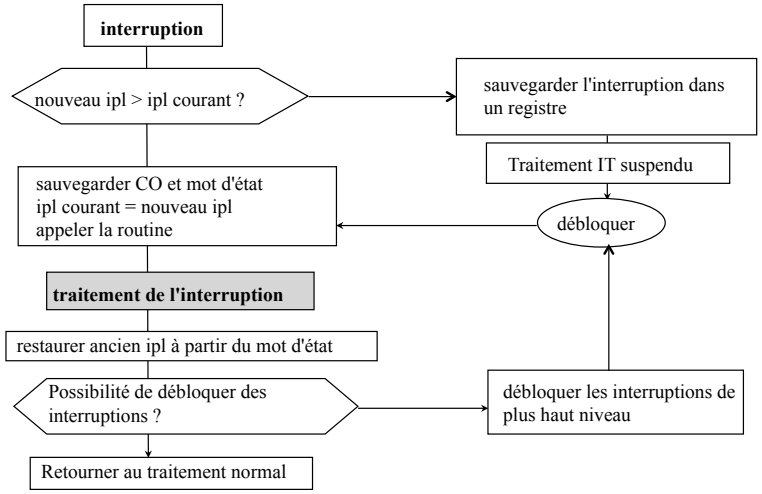
- ipl stocké dans le mot d'état

Les niveaux d'exécution



Sur certains systèmes exécution de ITs sur une *pile d'interruption*

Traitement des interruptions



Synchronisation

- Unix est **ré-entrant** => A un instant donné, plusieurs processus dans le noyau :
  - un seul est cours d'exécution
  - plusieurs bloqués
- Problème si manipulation des mêmes données
  - nécessité de protéger l'accès aux ressources

    => noyaux (la plupart) **non préemptifs** : Un processus s'exécutant en mode noyau ne peut être interrompu par un autre processus *sauf blocage explicite*

  => 1) synchronisation uniquement pour les opérations bloquantes

  ex: lecture d'un tampon => verrouillage du tampon pendant le transfert

  2) possibilité d'interruption par les périphériques => définition de section critique

Section critique

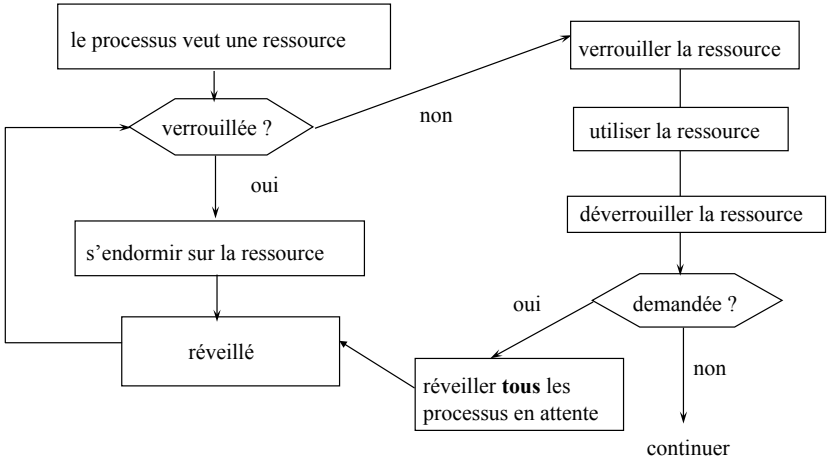
- Appel à set-priority-level pour bloquer les interruptions

Primitive	Activité bloquée
slp0()	aucune
splsoftclock()	horloge faible priorité
splnet()	protocoles réseaux
spltty()	terminaux
splbio()	disques
splimp()	périphériques réseaux
splclock()	horloge
splhigh()	toutes les interruptions

- Exemple :

```
s=splbio(); /*augmenter la priorité pour bloquer les IT disques */
...
splx(s);    /* restaurer l'ancienne priorité */
```


Verrouillage de ressources



Exemple de code

- Verrouillage

```
/* Attente d'une E/S */                 /* Fin d'E/S */
iowait(bp) {                            iodone(bp) {
    ps = slpbio();                          ...
    while ( !bp->b_flags & B_DONE )         bp->b_flags |= B_DONE ;
        sleep(bp, PRIBIO);                  if (bp->b_flags & B_ASYNC)
    slpx(ps);                                   brelse(bp) ;
    ...                                     else wakeup(bp) ;
}                                       }
```

iodone exécuter ici => sleep inutile !

Primitive sleep

- 2 paramètres :
  - adresse de l'obstacle
  - Priorité en mode noyau (priorité du processus endormi)
- Priorité (4.3BSD):
  - PSWP Swapper
  - PMEM Démon de pagination
  - PINOD Attente d'une inode
  - PRIBIO Attente E/S disque
  - PZERO Seuil
  - PPIPE Attente sur tube (plein ou vide)
  - TTIPRI Attente entrée sur un terminal
  - TTOPRI Attente écriture sur un terminal
  - PWAIT Attente d'un fils
  - PSLEP Attente d'un signal

Non interruptibles par des signaux

Interruptibles par des signaux

Algorithme de sleep

- Masquer les interruptions
- Mettre le processus à l'état SSLEEP
- Mise à jour du champs p_wchan (obstacle)
- Changer le niveau de priorité du processus
- Si (priorité non interruptible) {
  commutation (swtch) /* le processus dort */
  /* réveil */
  démasquer les interruptions
  retourner 0
  }
- /* priorité interruptible */
- Si (pas de signaux en suspens) {
  commutation (swtch) /* le processus dort */
  Si (pas de signaux en suspens) {
  démasquer les interruptions /* Pas réveillé par un signal */
  retourner 0;
  }
- /* Signal reçu ! */
- démasquer interruption
- restaurer le contexte sauvegardé dans appel système
- saut (longjmp)

Algorithme de wakeup

- Réveiller tous les processus en attente sur l'obstacle
  - Masquer interruption
  - pour (tous les processus endormis sur l'obstacle) {
    mettre à l'état prêt
    si (le processus n'est pas en mémoire)
    réveiller le swapper
    sinon si(processus plus prioritaire que processus courant)
    marquer un flag
  - }
  - démasquer interruptions
- Retour en mode utilisateur => test du flag :
  - Si (flag positionné) réordonner

Initialisation du système

- Initialisation des structures :
  - liste des inodes libres, table des pages
- montage de la racine
- construire le contexte du processus 0
  (struct U, initialisation de proc[0])
- Fork pour créer le processus 1 (init)
- Exécuter le code du swapper (fonction sched)

Processus du système

- Processus 0 : swapper
  gère le chargement/déchargement des processus sur le swap
- Processus 1 : init lance les démons d'accueil (gettyd)
- Processus 2 : paginateur (pagedaemon) - gère le remplacement de pages
- Autres "démons" :
  inetd, nfsd, nfsiod, portmapper, ypserv....

Visualisation : commande ps

```
>nice ps aux
USER       PID %CPU %MEM   SZ  RSS TT STAT START  TIME COMMAND
sens     17820 18.0  2.9  300  640 p0 S    17:07   0:03 -tcsh (tcsh)
root         1  0.0  0.0   52    0 ?  IW   Dec 11  0:02 /sbin/init -
root         2  0.0  0.0    0    0 ?  D    Dec 11  0:02 pagedaemon
root     16023  0.0  0.0   40    0 co IW   Jan 15  0:00 - cons8 console (getty)
root     17818  0.0  1.3   44  300 ?  S    17:07   0:00 in.rlogind
root         0  0.0  0.0    0    0 ?  D    Dec 11  0:11 swapper
root       100  0.0  0.4   72   88 ?  S    Dec 11  0:10 syslogd
root       117  0.0  0.2  108   52 ?  I    Dec 11  2:54 /usr/local/sbin/sshd
root       110  0.0  0.0   52    0 ?  IW   Dec 11  0:00 rpc.statd
root       128  0.0  0.0   56    0 ?  IW   Dec 11  0:35 cron
root       141  0.0  0.4   48   92 ?  S    Dec 11  0:05 inetd
root       144  0.0  0.0   52    0 ?  IW   Dec 11  0:00 /usr/lib/lpd
daemon   16012  0.0  0.0   96    0 ?  IW   Jan 15  0:00 rpc.cmsd
root        87  0.0  0.0   16    0 ?  I    Dec 11  0:01  (biod)


sens     17847  0.0  2.1  216  464 p0 R N  17:07   0:00 ps -aux
```


Ordonnancement

1. Interruption horloge
2. Les structures
3. Ordonnanceurs classiques (BSD, SVR3)
4. Classes d'ordonnancement (SVR4)
5. Ordonnancement temps réel (SVR4,Solarix 2.x)

Les horloges matérielles

- RTC : Real-Time Clock
  - Horloge temps-réel
  - Maintenue par batterie lorsque l'ordinateur est éteint
  - Précision limitée, accès lent
  - Utilisée au démarrage pour mettre à jour l'horloge système
- TSC : Time Stamp Counter
  - Compteur 64 bits (Intel)
  - Incrementé à chaque cycle horloge
  - ex: 1G HZ => incrémentation toutes les ns (1/1E9) => sur 64 bits débordement au bout de 584 ans !
  - Mesure précise du temps
  - Mesure directement dépendante de la fréquence du processeurs => pb avec portable
- PIT : Programmable Interval Timer
  - Registre horloge => agit comme un minuteur
  - Décrémentation régulière, Passage à 0 => interruption horloge ITH (IRQ0)
  - Outils de base de l'ordonnanceur
  - Précision de 100 HZ (10 ms) sur la plupart des UNIX – 1000 HZ (1 ms) dans linux 2.6

Interruption horloge : hardclock()

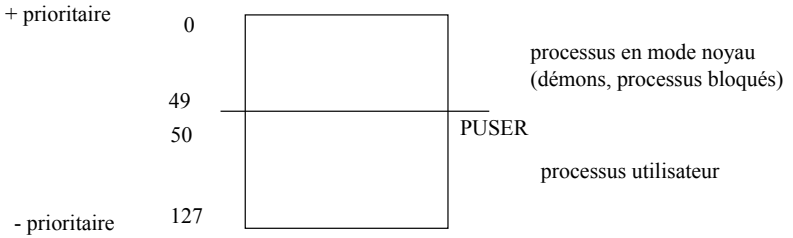
- Horloge matérielle interrompt le processus à des intervalles de temps fixes = tics horloge
- tic - 10 ms
  - HZ dans param.h indique le nombre de tics par seconde (par ex. 100)
- Routine de traitement dépendant du matériel
- Doit être courte !
- Très prioritaire
- 1 quantum = 6 ou 10 tics

Routine de traitement de IT horloge

1. Réarmer l'interruption horloge
2. Mise à jour des statistiques d'utilisation CPU du processus courant (p_cpu)
3. Recalculer la priorité des processus
4. Traiter fin de quantum
5. Envoyer SIGXCPU au processus courant si quota CPU dépassé
6. Mise à jour de l'horloge
7. Réveiller processus système si nécessaire
8. Traiter les alarmes

Structures

- Ordonnancement basé sur les priorités
- Les informations sont stockées dans struct proc (résident)
  - p_pri Priorité courante
  - p_usrpri Priorité du mode utilisateur (égale à p_pri en mode U)
  - p_cpu mesure de l'activité CPU récente
  - p_nice incrément de priorité contrôlable par l'utilisateur



Ordonnancement classique

- Répartir équitablement le processeur =>
  - baisser la priorité des processus lors de l'accès au processeur
- A chaque tic p_cpu++ pour le processus courant
- Régulièrement appel de schedcpu() (1 fois par seconde)
  - Pour tous les processus prêts :

    p_usrpri = PUSER + p_cpu/4 + 2*p_nice

    p_cpu = p_cpu * decay

    decay = 1/2 System V Release 3

    decay = (2 * load) / (2*load +1) BSD
- Un processus qui a eu un accès récent => p_cpu élevé => p_usrpri élevé.

Les primitives internes

- Après 4 tics appel de **setpriority**() pour mettre à jour la priorité du processus courant
- 1 fois par seconde appel de **schedcpu**() pour la mise à jour des priorités de tous les processus
- **roundrobin**() appelée en fin de quantum (10 fois par seconde) pour élire une nouveau processus

Exemple - System V Release 3

- Quantum = 60 tics

Priorité des processus bloqués

- Les processus sont bloqués avec une haute priorité ≠ priorité utilisateur (p_pri ≠ p_usrpri)

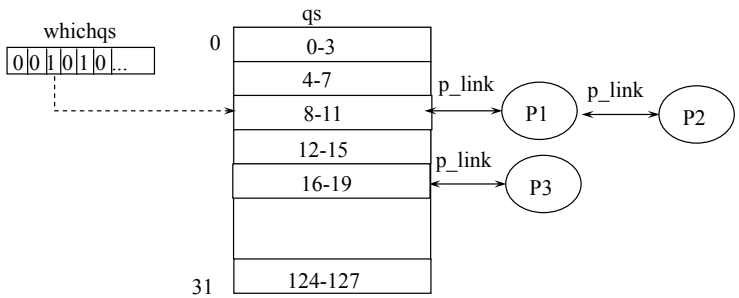
=> Au réveil le processus a une plus grande probabilité d'être élu

=> privilégier l'exécution dans le système

- Au passage au mode U l'ancienne priorité est restaurée (p_pri = p_usrpri)

Implémentation

- Problème : trouver **rapidement** le processus le plus prioritaire
- BSD : 32 files de processus prêts

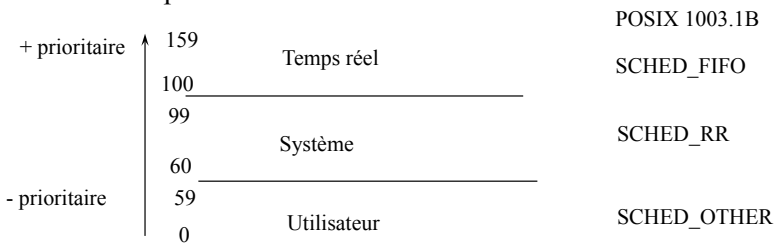


Algorithme de swtch

- Trouver le premier bit positionné dans whichqs
- Retirer le processus le plus prioritaire de la tête
- Effectuer la commutation :
  - Sauvegarder le PCB (Process Control Bloc) du processus courant (inclus dans zone U)
  - Changer la valeur du registre de la table des page (champs p_addr de struct proc)
  - Charger les registres du processus élu à partir de la zone u

SVR4 : Classes d'ordonnancement

- 3 classes de priorités



- Les processus temps réel prêt s'exécutent tant qu'ils restent prêt
- Définition des processus temps réel réservée au superviseur (appel système priocntl SVR5 - sched_setparam POSIX)

SVR4 : Structures

- Ajout dans struct proc :
  - p_cid : identificateur de la classe ...
- Une liste des processus temps réel (rt_plist)
- Une liste de processus temps partagé (ts_plist) ...

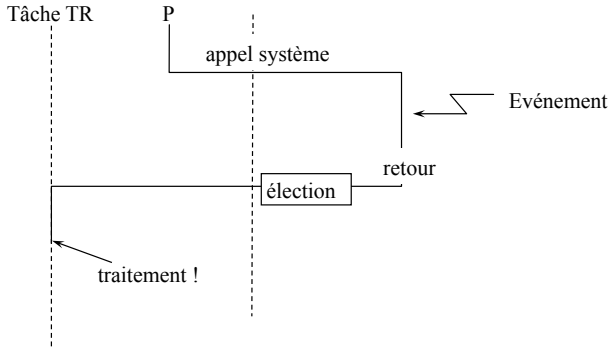
Classe "temps partagé"

- Quantum variable d'un processus à l'autre
- inversement proportionnel à la priorité !
- Définis statiquement pour chaque niveau de priorités

pri	quantum	pri suiv.	maxwait	pri wait
0	100	0	5	10
1	100	0	5	11
...	...	...	...	...
15	80	7	5	25
...	...	...	...	...
40	20	30	5	50
...	...	...	...	...
59	10	49	5	59

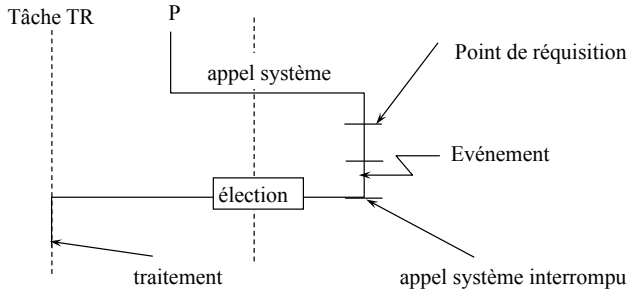
Classe "temps réel"

- Objectif : satisfaire des contraintes de temps
  - Processus temps réel très prioritaire en attente d'événement
- Impossible dans la plupart des Unix car noyau non-préemptif !



Points de réquisition

- Solution 1 : vérifier *régulièrement* si un processus plus prioritaire doit être exécuté (SVR4)



- Pratiquement il est difficile de placer de nombreux points
  => latence de traitement importante

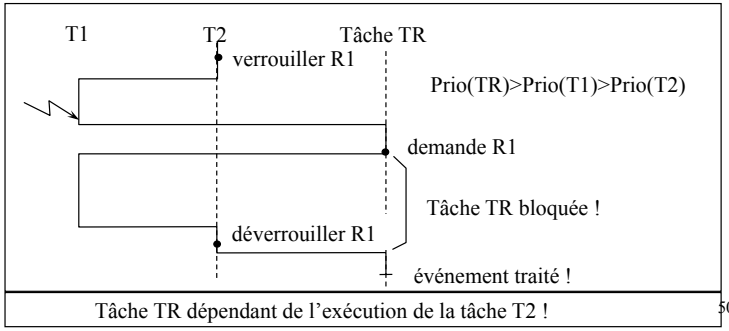
Noyaux Préemptifs (Solaris 2.x)

- Solution 2 : rendre le noyau préemptif

=> en mode noyau l'exécution peut être interrompue par des processus plus prioritaires

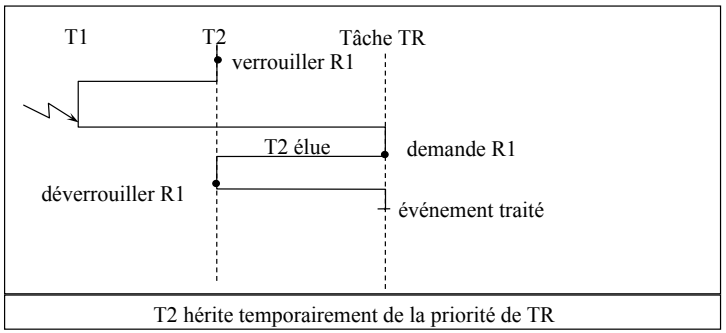
- Protéger toutes les structures de donnée du noyau par des verrous (~ sémaphores)

Inversion de priorité

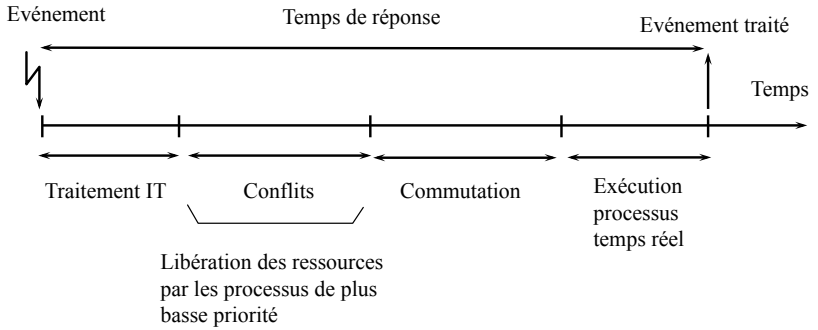


Héritage de priorité

- Solution : Donner à la tâche qui possède la ressource la priorité de la tâche temps réel

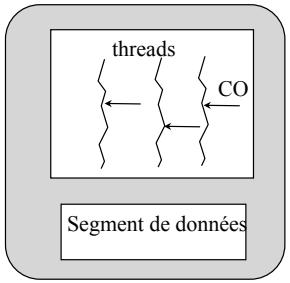


Temps de réponse



Processus légers

- Motivations :
  - 1) avoir une structure plus légère pour le parallélisme
  - 2) partage de données efficace



thread = code + pile + registres

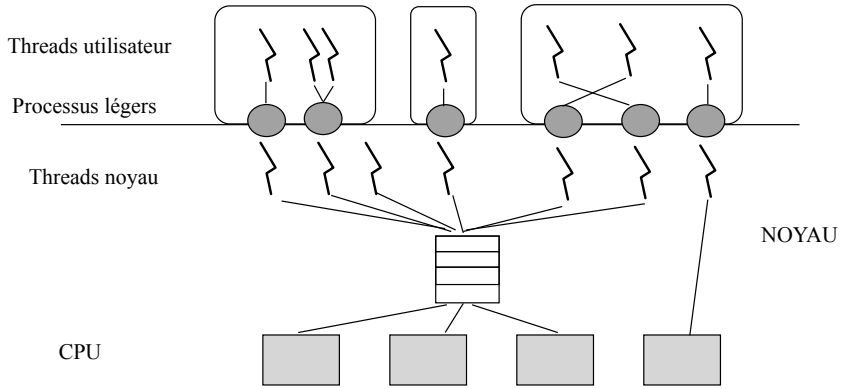
- thread (processus léger) : unité d'exécution

Propriétés des threads

- Partage le même espace => commutation plus rapide
- Echange de données par simple échange de référence
- Création/synchronisation plus rapide

- 3 types de threads (Solaris 2.x)
  - thread noyau : unité d'ordonnancement dans le noyau
  - processus léger (lightweight process LWP) : associé à un thread noyau
  - thread utilisateur : multiplexé dans les LWP

Exemple Solaris 2.x



Comparaison

	Temps de création (microsecondes)	Temps de synchronisatic en utilisant des sémapho (microsecondes)
Thread utilisateur	52	66
Processus léger	350	390
Processus	1700	200

Solaris sur Sparc2

Gestion des processus dans LINUX

- Structures
- Ordonnancement
- Nouveautés depuis 2.6

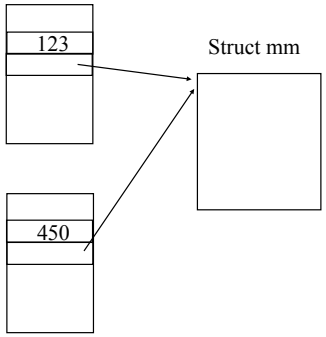
Structure de données : struct task

- 1 table des processus de type struct task (equivalent à proc + user)
- 1 entrée par processus
- Première entrée réservée au processus **init**

- **Task_struct** :
  - **policy** (SCHED_OTHER, SCHED_FIFO, SCHED_RR) : stratégies d'ordonnancement
  - **state** : running, waiting, stopped, zombie
  - **priority: quantum de base**
  - **counter** : compte le nombre de tics restants avant la prochaine commutation
  - **next_task**, **prev_task : liste**
  - **mm_struct : contexte mémoire**
  - **pid**, **pid : identifiant**
  - **fs_struct : fichiers ouverts**
  - …

Threads

- Implémentation des threads dans la noyau :
  - Simple partage de la structure struct mm



Processus Linux (<=2.4)

- **Trois classes de processus**
  - Processus interactifs : attente événement clavier/souris, temps de réponse court
  - Processus « batch » : lancement en arrière plan, plus pénalisé par ordonnanceur
  - Processus temps-réel : forte contraintes de synchronisation (multi-média, commandes robots ..)
- **Etats :**
  - Running
  - Waiting
  - Stopped
  - Zombie

Stratégies d'ordonnancement (<=2.4)

- Noyau **non-preemptif** mais ordonnancement **preemptif (quantum)**
- **Tic = 10ms (paramètre HZ = 100 défini dans param.h)**
- **Deux types de priorité correspondant à 2 classes d'ordonnancement :**
  - **Priorité statique** : processus temps-reel (1 à 99), priorité fixe donnée par l'utilisateur
  - **Priorité dynamique :** somme de la priorité de base et du nombre de tics restants (counter) avant la fin de quantum

Algorithme d'ordonnancement

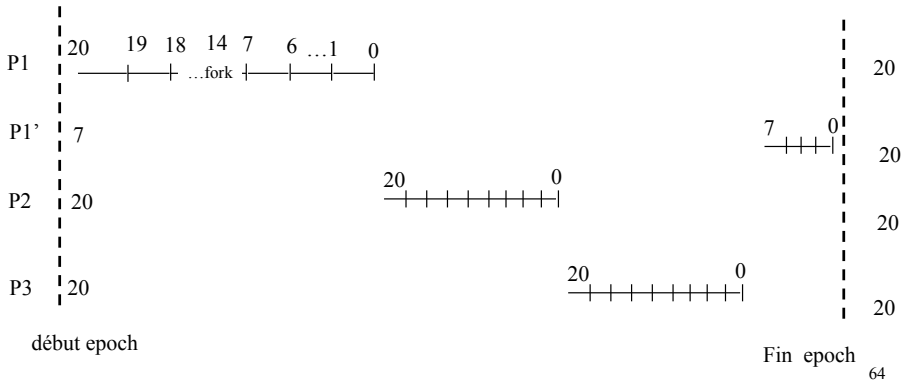
- Temps divisé en **périodes (epoch)**
- Début période :
  - Un quantum associé à chaque processus prêt
- Fin période :
  - Tous les processus ont terminé leur quantum
- Calcul du quantum :
  - 1 quantum de base = 20 tics (200 ms)
    #define DEF_PRIORITY (20*HZ/100) (=20)
  - priority = DEF_PRIORITY
  - Counter : temps restant (nb tics)
  - Création : le processus hérite de la moitié du quantum restant du père
- Champs priority et counter pas utilisés pour les processus de classe SCHED_FIFO

Fonction schedule()

- Implémente l'ordonnancement
  - Invoquée directement en cas de blocage
  - Invoquée « paresseusement » au retour en mode U
- Schedule
  1. Choisir le meilleur candidat : celui ayant le poids le plus élevé (fonction goodness) :
     Poids = 1000 + priorité base pour processus temps réel
     Poids = counter + priorité base pour autre processus
     Poids = 0 si counter = 0
  2. Si tous les processus prêts ont un poids de 0 => **Fin de période**
     1. Ré-initialisation des counter de TOUS les processus :
     - p->counter = (p->counter >> 1) + p->priority
     - Rem : la priorité des processus en attente augmente

Exemple

- Evolution du champs *counter*



Ordonnancement SMP (1)

- Critère supplémentaire pour l'ordonnanceur :
  - Moins coûteux de ré-exécuter un processus sur le même processeurs (exploitation des caches internes)
  - Maximiser l'utilisation des différents processeurs
- Exemple :
  - 2 processeurs (CPU1, CPU2) et 3 processus (P1, P2, P3)
  - Priorité P1 < Priorité de P2 < Priorité de P3
  - CPU1 exécute P1
  - CPU2 exécute P3
  - P2 exécution précedent sur CPU 2 devient prêt
  - Question : P2 « prend » CPU1 (préemption) => perte du cache de CPU2 ou attendre que CPU2 deviennent disponible ?
- => Heuristique qui prend en compte la taille des caches

Ordonnancement SMP (2)

- P2 préempte P1 sur CPU1 si :
  - Le quantum restant de P3 sur CPU2 (counter) est supérieur au temps estimé de remplissage des caches de CPU1

Nouveautés Linux 2.6

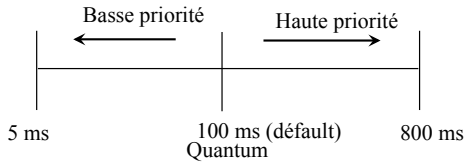
- Ordonnancement
- Noyau préemptif

Ordonnancement 2.6.X

- Objectif : diminuer les temps de réponses et une gestion plus fine des temporisateur pour application multi-média

=> diminution de la valeur du tic (jiffy) = 1 ms (HZ =1000)

- 2 types de processus
  - I/O Bound (E/S) : processus faisant beaucoup d'E/S
  - Processor Bound : processus de calcul
- Objectif : avantager les processus I/O Bound avec des quantums variables



Algorithme d'ordonnancement 2.6 : Priorité

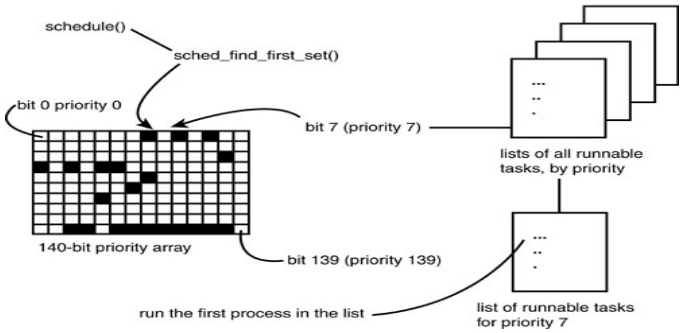
- Priorité de base = valeur du nice [-20,+19]
- Système de pénalité/bonus en fonction du type de processus

Bonus max = -5, Penalité max= +5

- Pour déterminer les types de processus : ajout d'un champs sleep_avg dans structure task
- Sleep_avg = temps moyen à l'état bloqué
  - Au reveil d'un processus : sleep-avg augmenté
  - A l'éxecution : sleep-avg-- à chaque tic
- Fonction effective_prio() : correspondance entre sleep_avg et bonus [-5,+5]

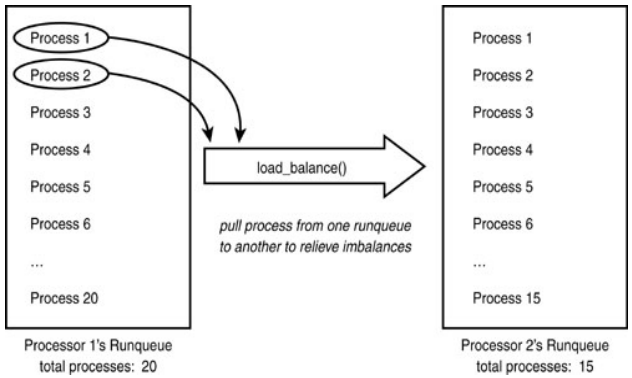
Algorithme de choix du processus

- Algorithme en o(1)
- 140 niveaux de priorité, 1 file par niveau



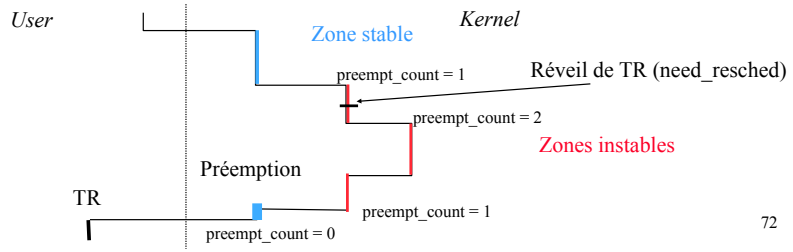
Equilibrage de charge

- Fonction load_balance() appelée par schedule() lorsqu'une file est vide ou périodiquement (toutes les ms si aucune tâche, toutes les 200 ms sinon)

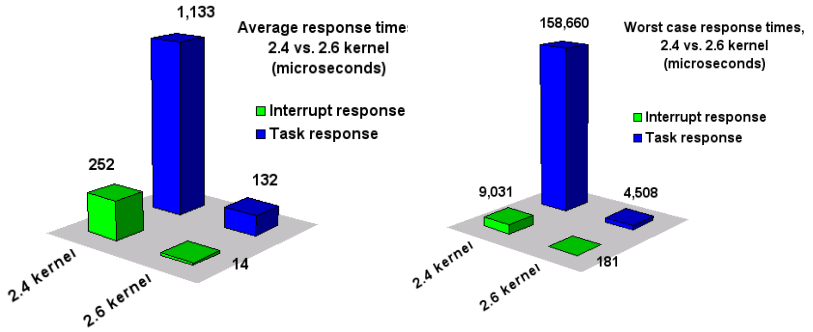


Noyau Préemptif – Linux 2.6.x

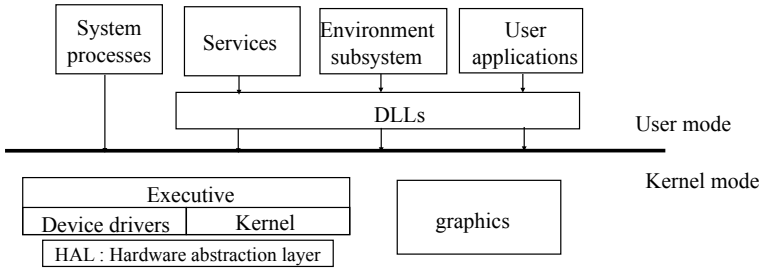
- Proche de la notion de points de réquisition :
  - Quitter le noyau uniquement à des points stables
- Verrouillage pour protéger les régions instables :
  - => un compteur (preempt_count) incrémenté à chaque verrouillage
- Retour d'IT :
  - si need_resched et preempt_count == 0 ➔ Préemption



Linux 2.4 vs. 2.6

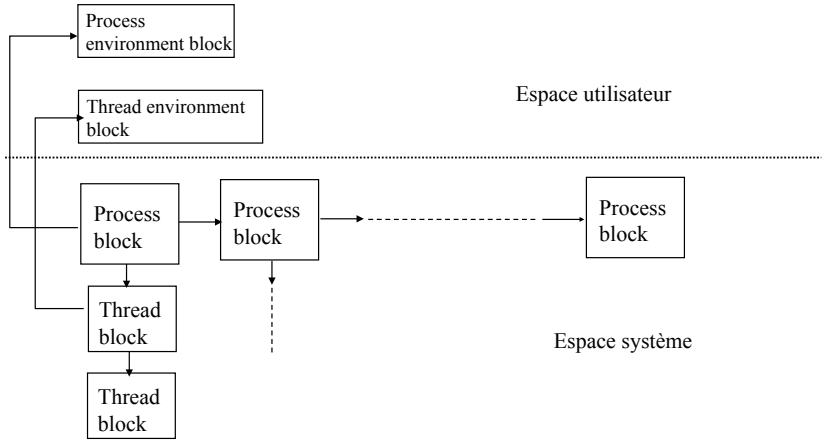


Gestion de processus dans Windows NT



1. Processus et Threads
2. Ordonnancement

Les structures

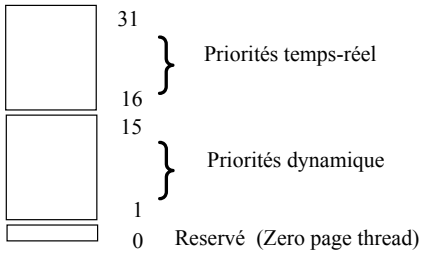


Les structures

- Process block (EPROCESS) : similaire à struct proc Unix
  - PID, PPID
  - Valeur de retour
  - Kernel process block (PCB) : statistiques, priorité, état, pointeur table des pages
- Thread block (ETHREAD) :
  - Statistiques, adresse de la fonction, pointeur pile système, PID …
  - Kernel thread block (KTHREAD) : synchronisation, info ordonnancement (priorité, quantum …)
- Process Env. block (PEB) :
  - Informations pour le « chargeur », gestionnaire de pile (modifiable par DLLs)
- Thread Env. block (TEB) :
  - TID, information pile (modifiable par DLLs)

Ordonnancement

- Priorités

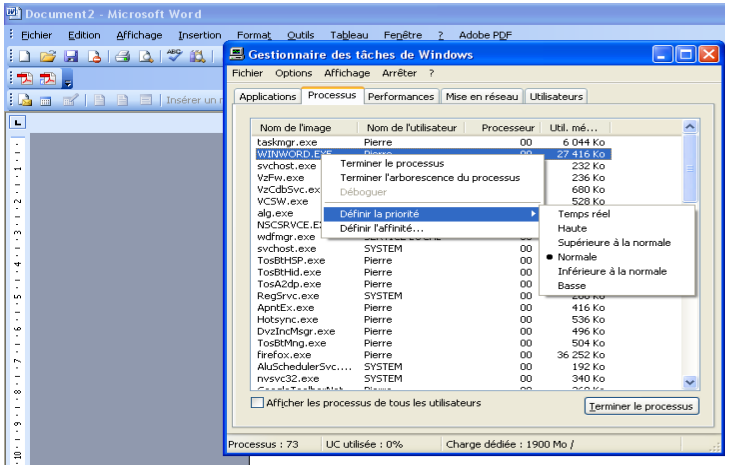


Ordonnancement temps partagé (quantum)

Par thread : priorité de base (processus), priorité courante

Choisir le thread le plus prioritaire (structure similaire à « whichqs » 4.4 BSB)

Classes de priorité

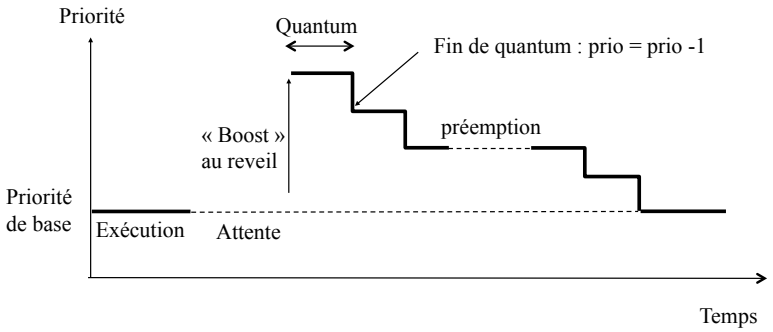


Ajustement des priorités et du quantum

- Création = priorité thread = priorité base (dépendant de la classe)
- 4 mécanismes
  - Augmentation du quantum des threads de processus en « arrière plan »
  - Augmentation de la priorité des processus endormis (*boost*)
    - +1 = sémaphore, disque, CR-ROM, port parallèle, vidéo
    - +2 = réseau, port série, tube
    - +6 = clavier, souris
    - +8 = son
  - Augmentation de la priorité des threads prêts en attente (éviter famine)
    - Thread en attente depuis 300 tics (~ 3 sec)

    => priorité = 15, quantum x 2

Exemple



Ordonnancement SMP

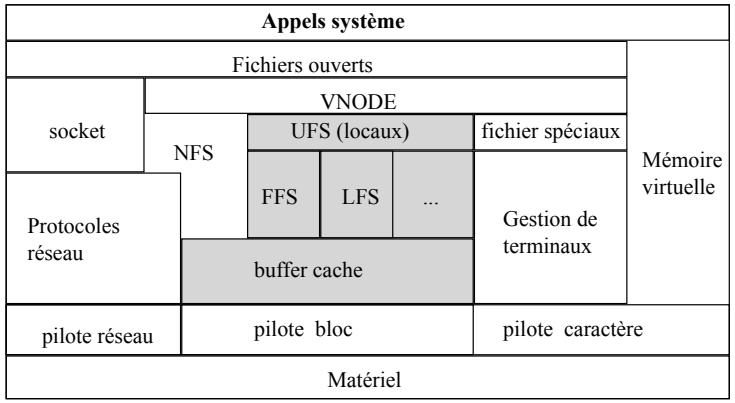
- Définition d'**affinités** pour chaque thread : liste des CPU sur lequel peut s'exécuter la tâche
- Chaque thread a un processeur "idéal"
- Quand un thread devient prêt, il s'exécute :
  - Sur le processeur idéal si il est libre
  - Sinon sur le processeur précédent si il est libre
  - Sinon rechercher un autre thread prêt

Système de gestion des Entrées/ sorties

1- Le sous système d'entrées/sorties

2- Les systèmes de fichiers locaux

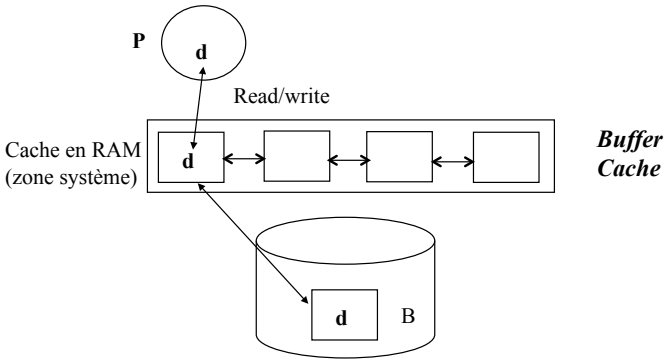
I - Système de fichiers locaux



PARTIE 1 : Cache

Principe du cache

Accès donnée **d** dans bloc B

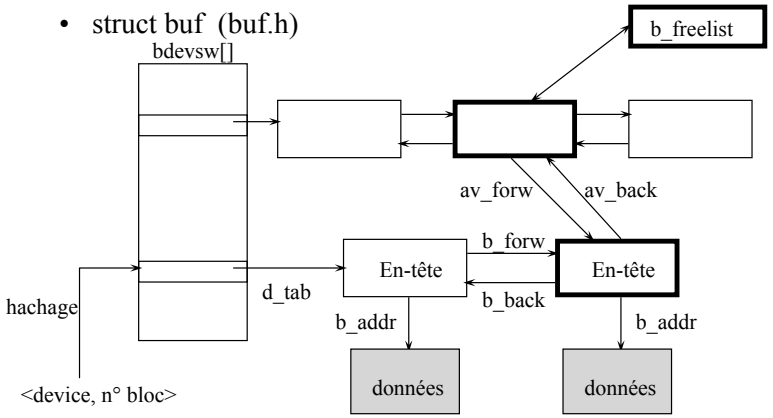


1 - Gestion du cache - le buffer cache

- 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»)

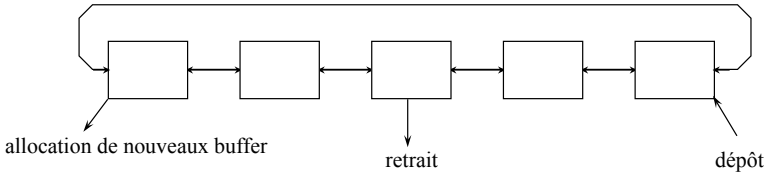
Les primitives

- 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

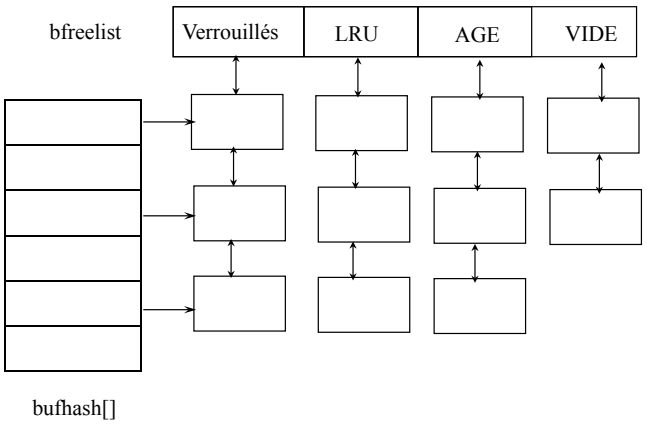
Lecture d'un buffer (bread)

- **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

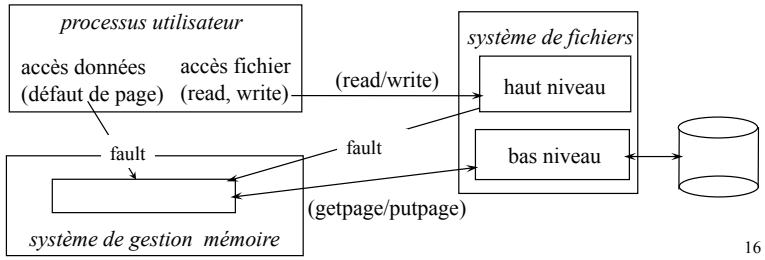
- Bdwrite
  Positionnement de B_DELWRI pour E/S asynchrone
- Besoin de place
  Dans getblk: Si le bloc n'est pas dans le cache
  => allouer un nouveau buffer
  Si B_DELWRI => Ecriture
- Régulièrement **sync** parcourt la liste des buffers
  Si B_DELWRI => Ecriture

Organisation du buffer cache (BSD)



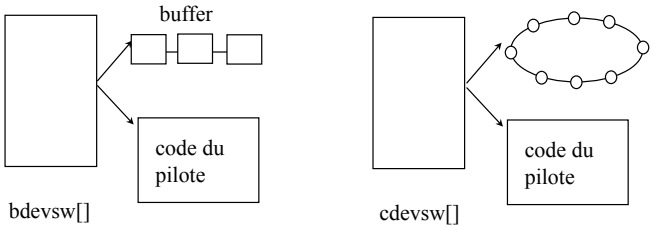
Mémoire virtuelle et buffer cache

- 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**



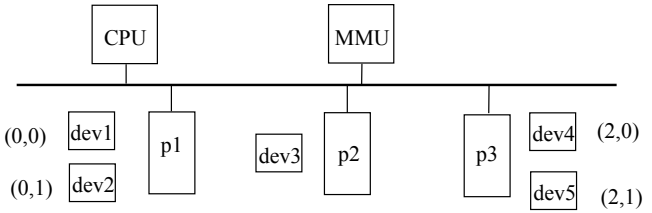
2 - Les Entrées/sorties

- 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

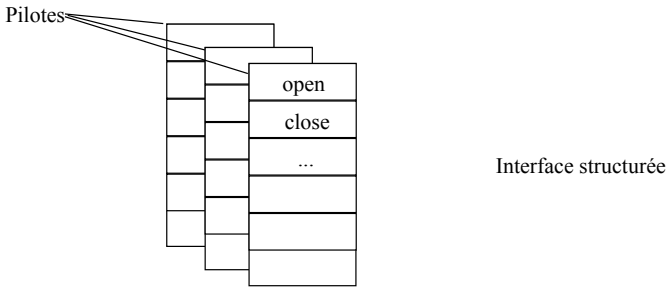
- Configuration typique



- 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)



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

Pilotes en mode bloc

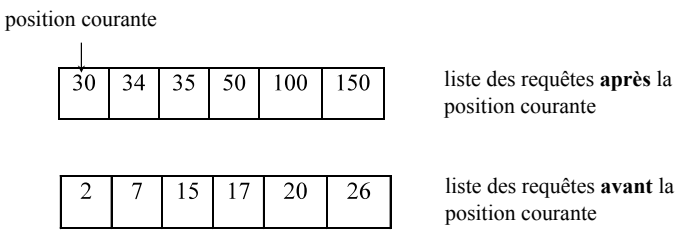
- Table bdevsw :

```
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



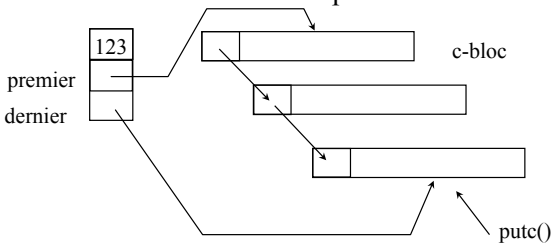
Pilotes en mode caractère

- table cdevsw

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

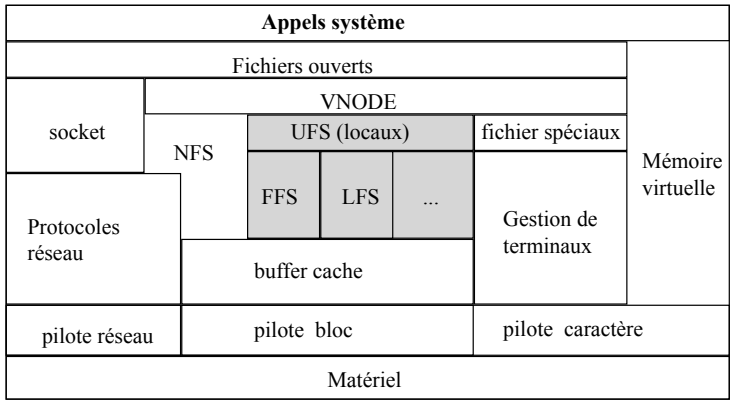

Tampon

- 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)

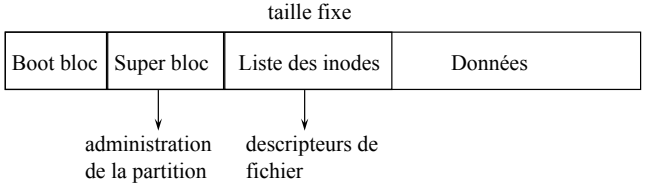
  Fast File System (FFS)
  - Introduit dans 4.2BSD
- Système de fichiers générique
  - Virtual File System (Sun 86)

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

Organisation générale du disque (s5fs)

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

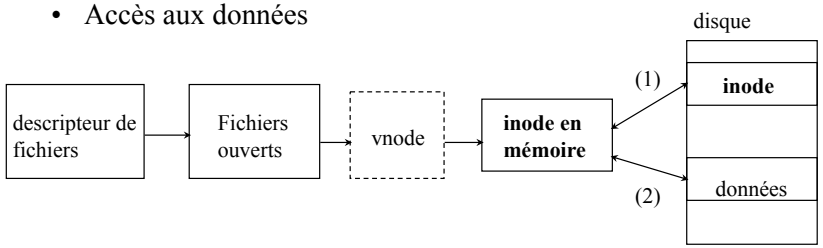
- Organisation d'un partition :



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

Les structures

- Accès aux données



- Allocation de bloc

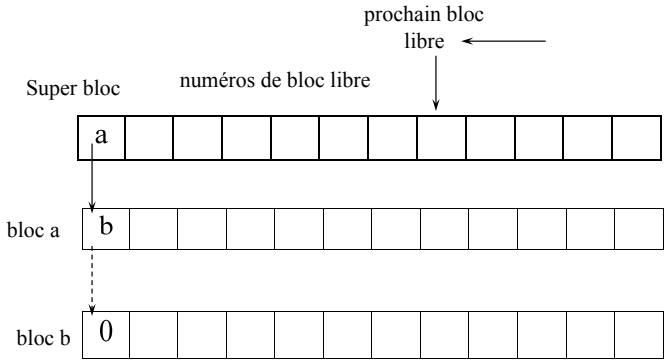
  - Le superbloc contient la liste des blocs libres, des inodes libres

Gestion des blocs libres

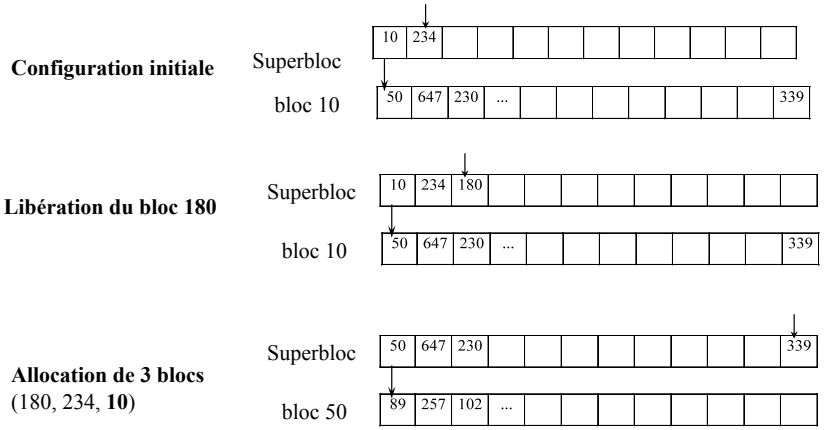
- Le superbloc : struct fs (fs.h)

  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

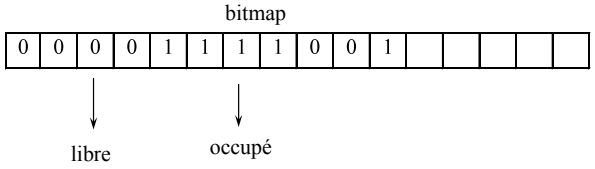


Allocation/libération de bloc : exemple



Allocation dans les nouveaux systèmes de fichiers

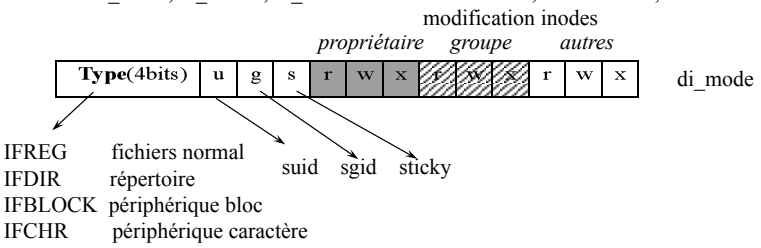
- Pb de la stratégie "classique" : pas de prise en compte de la contiguïté des blocs libres
- => vecteur binaire



- 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



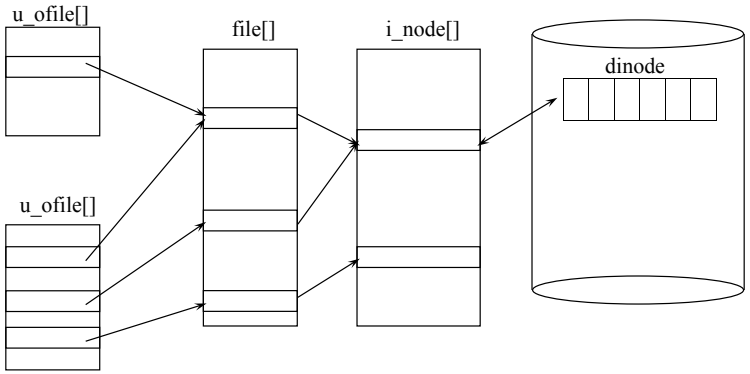
Structure inode

- En mémoire : struct **inode**
  - dinode avec en plus :
  - i_dev : device (partition) } accès à l'inode sur disque (mises à jour)
  - i_number : numéro d'inode
  - i_flags : Flags (synchronisation, cache)
  - pointeurs sur la freelist (liste des inodes libres)

Les autres structures

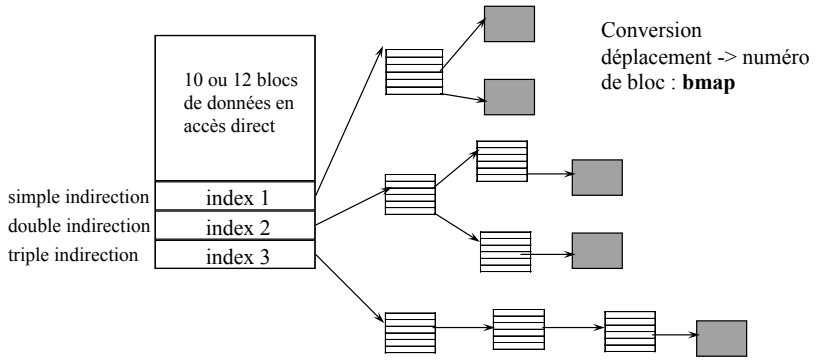
- file[] : table globale des fichiers ouverts
  - f_flag : mode d'ouverture (Lecture, Ecriture, Lecture/Ecriture)
  - f_offset : déplacement dans le fichier
  - f_inode : numéro d'inode
  - f_count : nombre de références
- u_ofile[] : Table locale des ouverts ouverts par un processus
  - entrée dans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

Résumé des structures



Où trouver les blocs ?

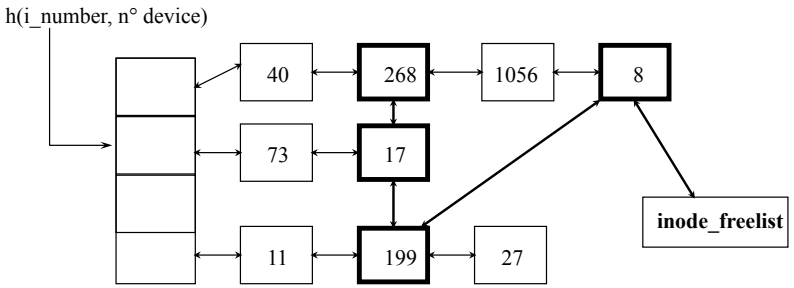
- Liste de blocs dans l'inode



- Vision de l'utilisateur :

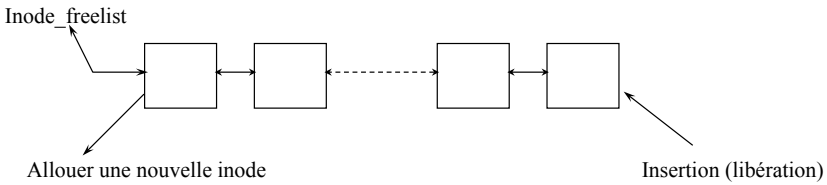
Les inodes en mémoire

- 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

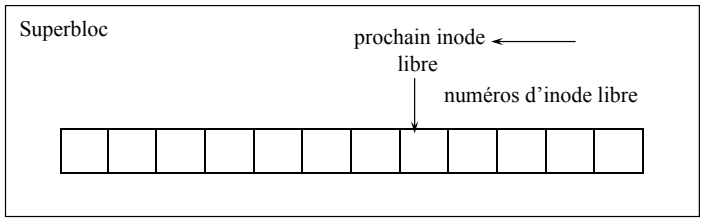
- Si une inode n'est plus utilisée par aucun processus => insertion dans inode_freelist.
- Inode_freelist = cache des anciennes inodes



- 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

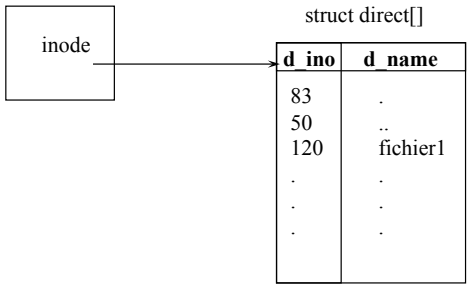
- Vérifier si aucun autre processus n'a verrouillé le superbloc (sinon sleep)
- Verrouiller le superbloc
- Si liste des inodes libres sur disque non vide
  - Prendre l'inode libre suivante dans superbloc
  - attribuer une inode en mémoire (iget)
  - mise à jour sur disque (inode marquée prise)
- Si liste 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

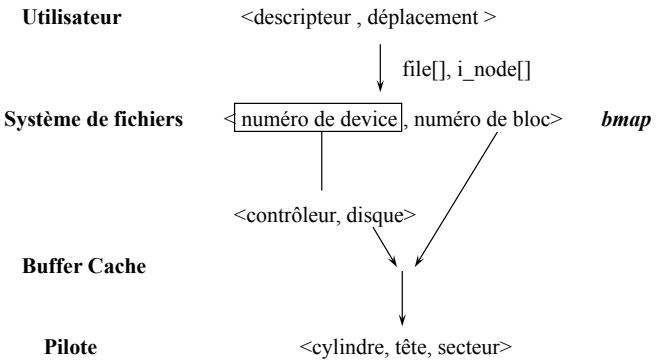
ln /users/paul/f2 /users/pierre/lhf2

rm /users/pierre/lhf2

Droits : droits sur le fichier

Implémentation des appels système

- Systèmes d'adressage :

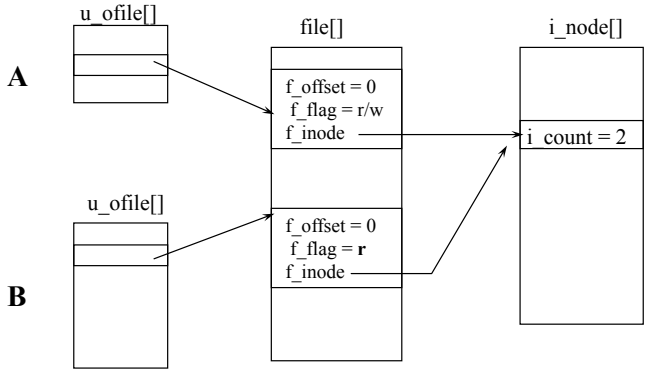


Algorithme de open

- 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

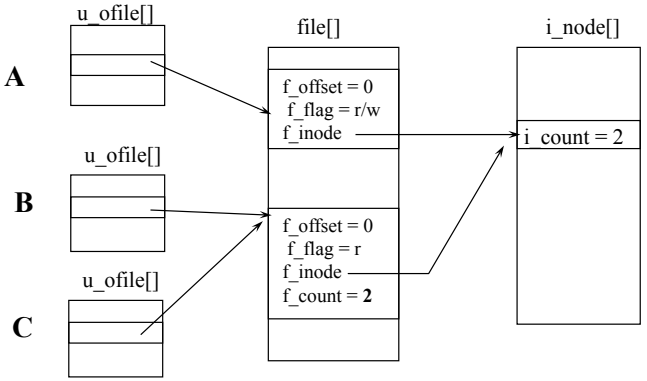
Exemple

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

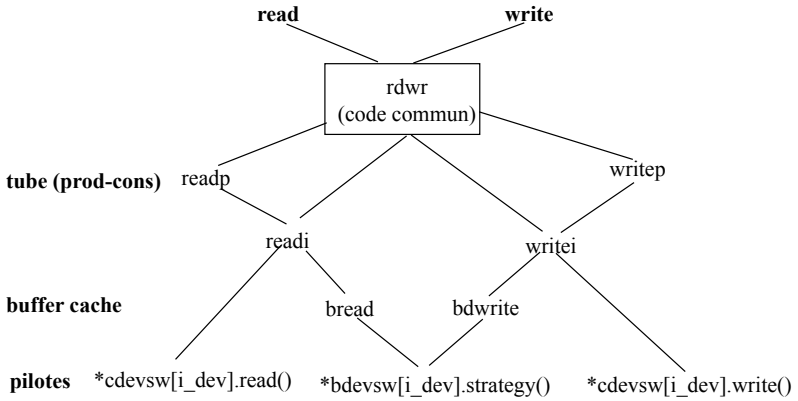


Exemple (2)

Processus B : fork()



Appels read et write

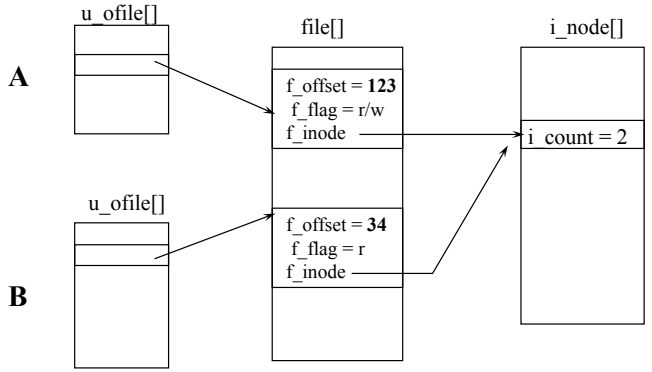


Algorithme de read

- 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);



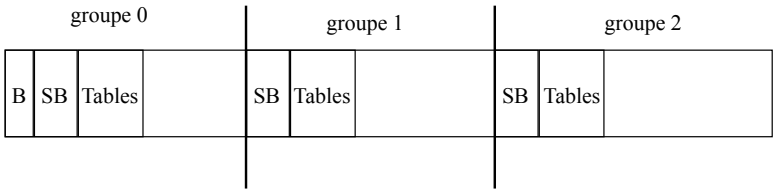
Les optimisations : fast file system (ffs)

- 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)

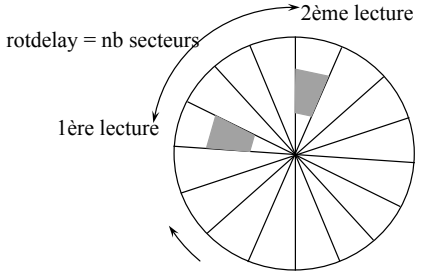
- Idée : partager les blocs entre plusieurs fichiers
- => Blocs divisés en fragment
  - 2,4,8 fragments par bloc
  - Taille "classique" : blocs 8Ko, fragment 512 octets
- Unité d'allocation = fragment
  - => perte réduite
  - => plus de structures de données

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

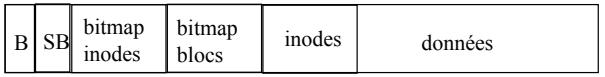


Performances

- 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

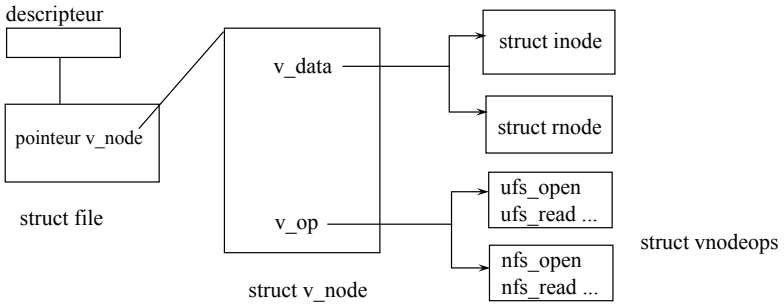
D'autres organisations

Exemple Ext2fs

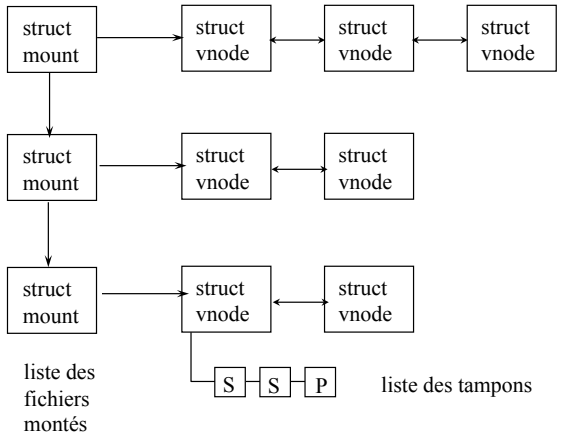


Systèmes génériques : VFS

- 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

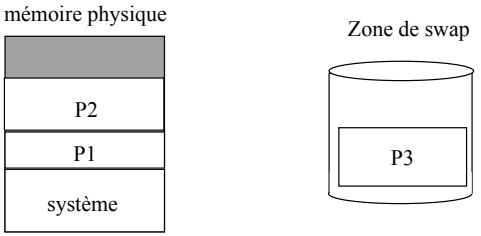


Mémoire virtuelle

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

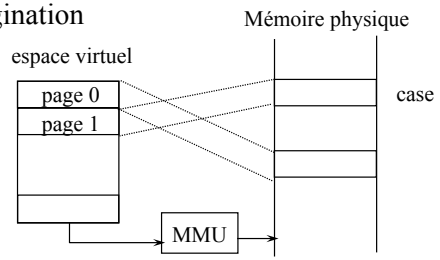
Notions de base

- 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)

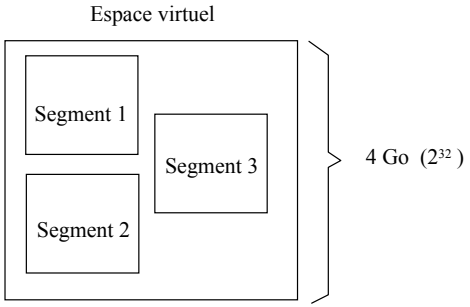
Historique

- 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

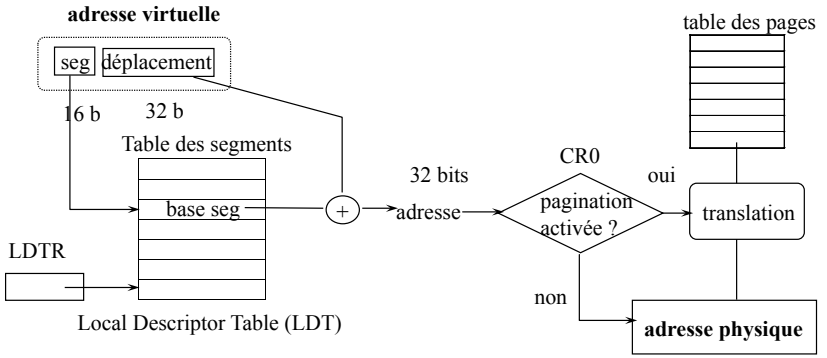
- Dans Unix, segmentation cachée à l'utilisateur, utilisée uniquement pour le partage et la protection

Support matériel :
Exemple Pentium

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



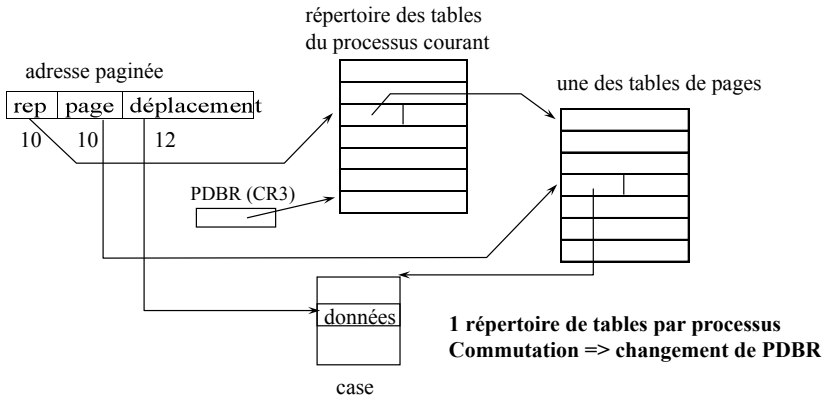
Architecture



- **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 !)



1 répertoire de tables par processus
Commutation => changement de PDBR

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

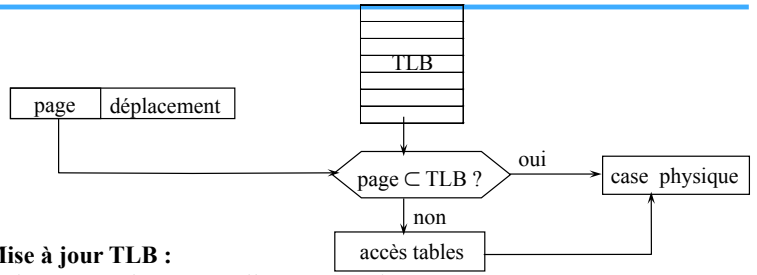
Mémoire associative : Translation Lookaside Buffer

Page	case

= cache des adresses

- TLB nombre d'entrées limité

Gestion de la TLB

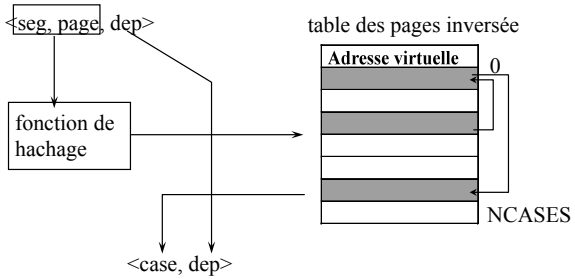


Mise à jour TLB :
- Chargement d'une nouvelle page pour le processus courant
- Commutation => invalidation de **toute** la TLB
  (automatique Intel x86 avec mise à jour PDBR)
- Déchargement page => invalidation entrée TLB
- Recouvrement (exec)

Autre approche : RS/6000

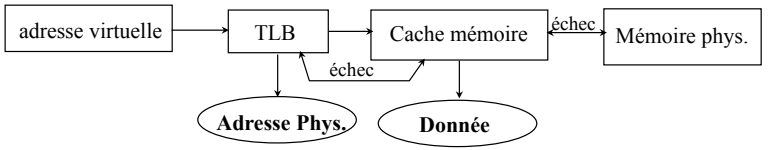
- Architecture RISC base pour AIX
- Utilisation d'une table des pages inversée = 1 entrée par **case** => taille réduite (page 4Ko, 32Mo de RAM => 128 Ko)
  1 seule table globale

adresse virtuelle du processus courant

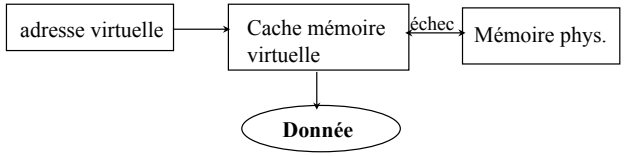


Architecture récente : Cache virtuel

- Architecture «classique» = 2 niveaux de cache mémoire

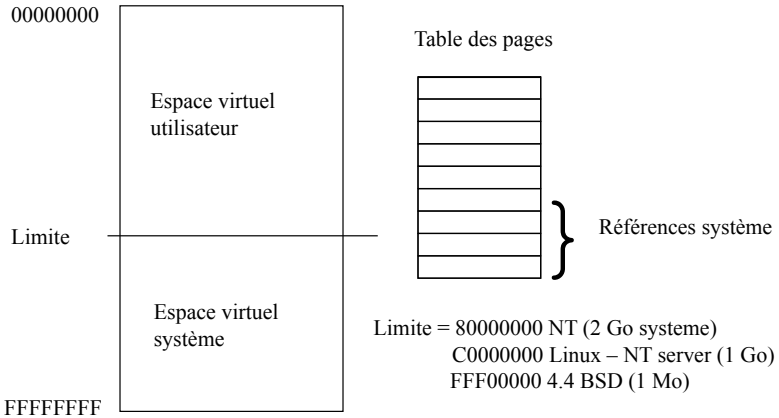


- Architecture à cache virtuel



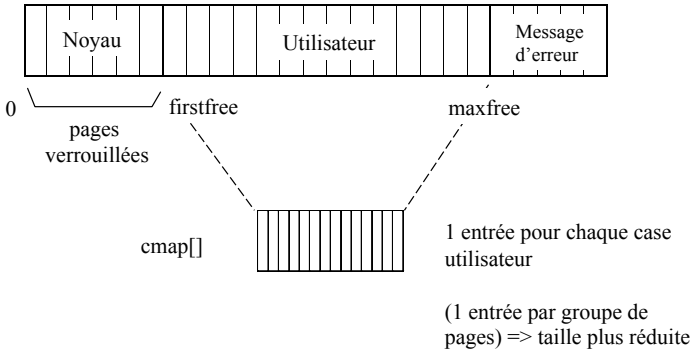
- Avantage : 1 seul niveau + pas de «flush» à la commutation

Les processus en mémoire



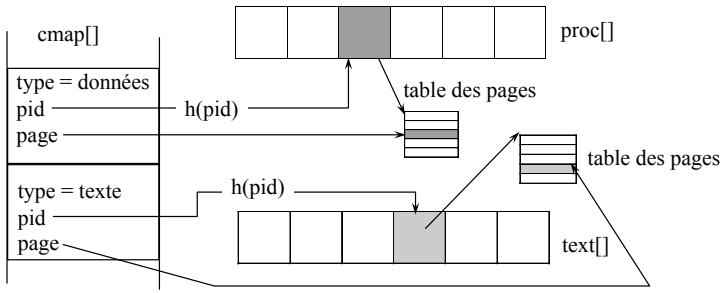
Etude de cas : 4.3BSD

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

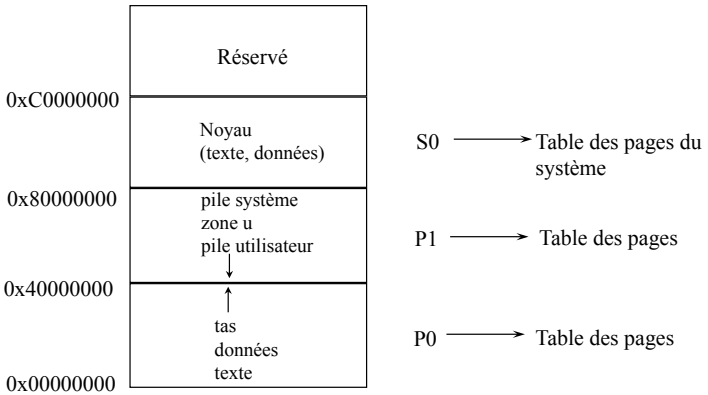


Etat d'une page

- 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)

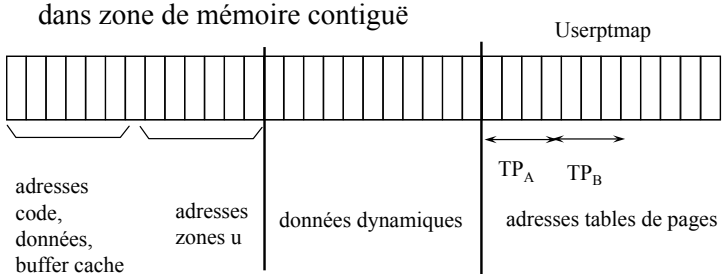
Structure de l'espace d'adressage

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



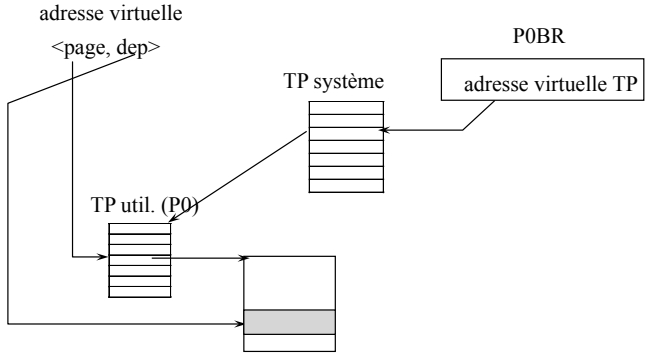
Organisation de l'espace virtuel noyau

- 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

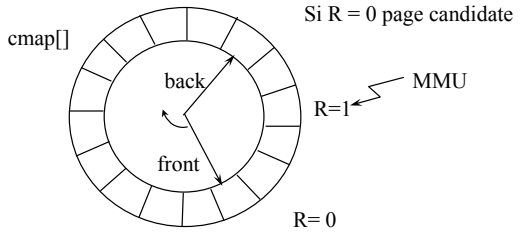
```
Pagein(adresse virtuelle) {
    Verrouiller la table des pages
    Si (adresse non valide) {
         envoyer SIGSEGV au processus;
         aller fin;
    Si  (page dans le cache des pages) { // page de code
         extraire page du cache
         mise à jour de table des pages;
         tant que (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

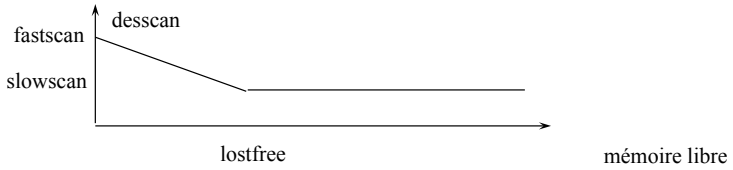
Implémentation du NRU

- 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

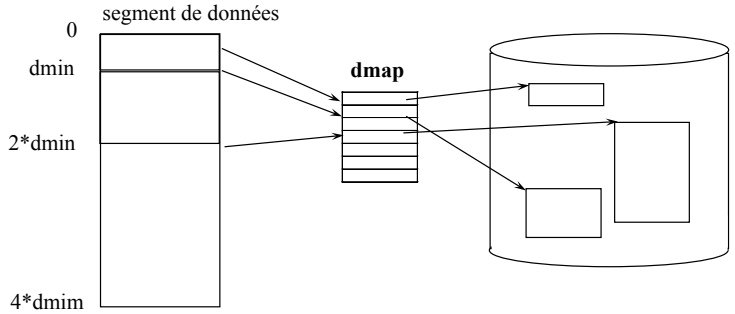
```
boucle
    recherche processus SRUN et non SLOAD le plus ancien
    si non trouvé
        alors sleep (&runout, PSWP); continuer
    sinon
        si swapin(p); continuer
        /* place insuffisante en mémoire */
        Si existe processus endormis ou en mémoire depuis longtemps
          alors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

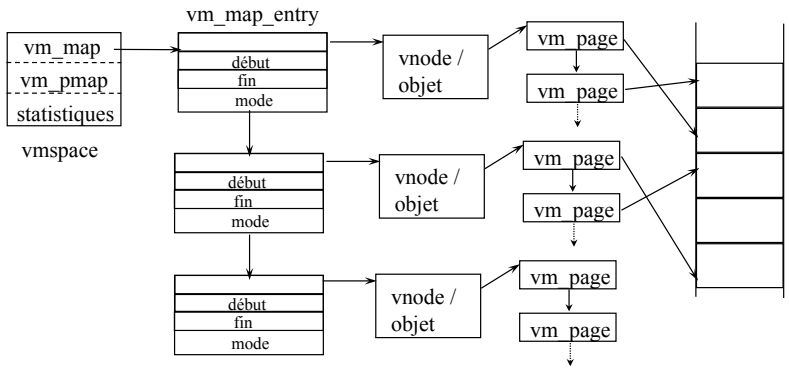
Les nouveaux systèmes

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

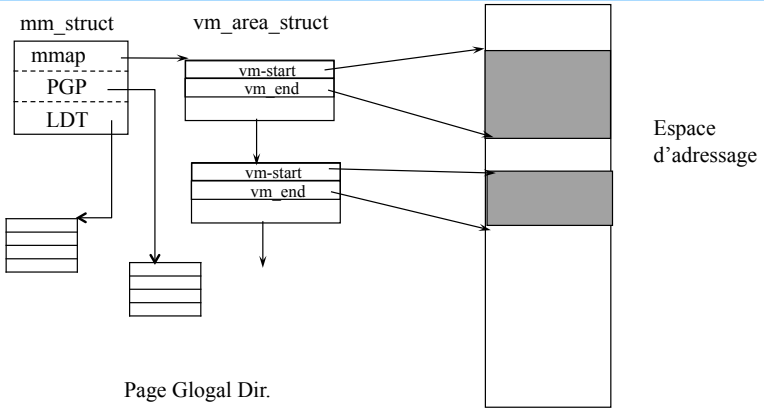
} Mémoires virtuelles très proches

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

Structure d'un espace 4.4BSD

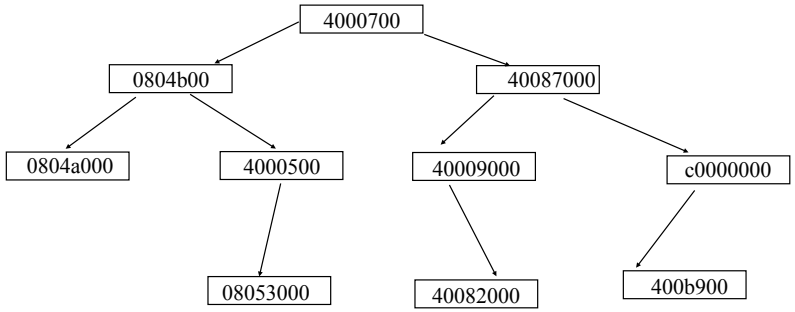


Structure dans Linux



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

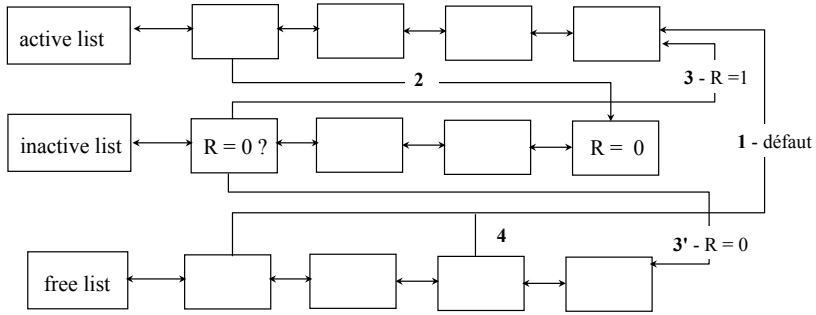
```
# more /proc/1/maps
08048000-0804f000 r-xp 00000000 03:06 80252      /sbin/init
0804f000-08051000 rw-p 00006000 03:06 80252      /sbin/init
08051000-08055000 rwxp 00000000 00:00 0
40000000-40012000 r-xp 00000000 03:06 69906      /lib/ld-2.1.3.so
40012000-40013000 rw-p 00011000 03:06 69906      /lib/ld-2.1.3.so
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 ..

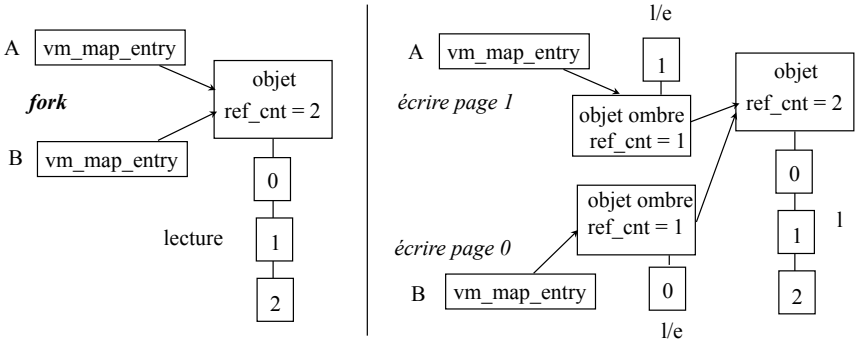
Remplacement de pages



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



Etude des Micro-noyaux

Plan

- Présentation générale des micro-noyaux
- Mach
- Chorus
- Amoeba, V system
- Comparaison des différentes approches

Les architectures de Noyau

- Monolithique
- Extensible
- **Micro-noyau**
- **Exo-noyau**

Introduction Micro-Noyau

Objectifs :

Portabilité

Informatique répartie et coopérative

Environnement facilitant l'intégration de nouvelles fonctions

Principe :

Terme micro-noyau introduit par Ira Galdstein de l'OSF (Open Software Foundation)

Issue de travaux sur les systemes répartis des années 80

Division du système d'exploitation en deux parties :
1°) Le micro-noyau
2°) Un ensemble de modules serveur

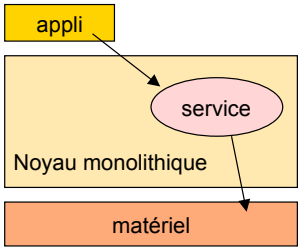
Historique :

Carnegie Mellon (1980) isole les fonctions élémentaires du système

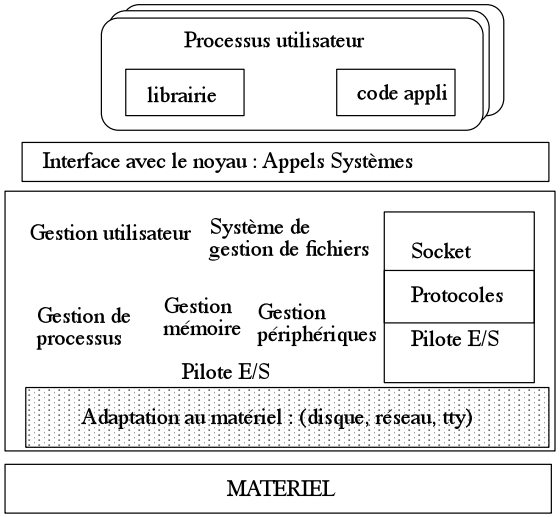
A partir de 1979 l'INRIA implémente des fonctions principales sous forme de modules indépendants.

Noyau monolithiques

- 1eres générations d'Unix, Linux, AIX…
- Performant et relativement sécurisé.
- Peu extensible, maintenance délicate.

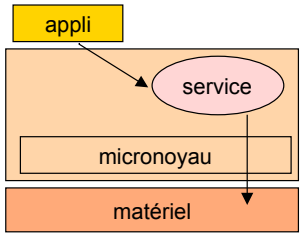


Système Monolithique



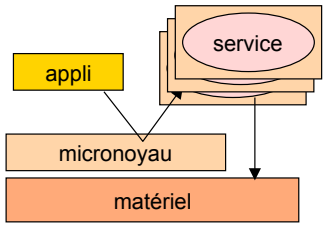
Noyau extensible

- Linux, AIX, Solaris…
- Chargement dynamique de code dans le noyau (module)
- Manque de sécurité



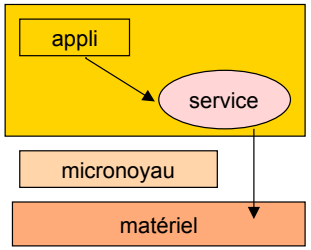
Micro-noyau

- Mach(BSD, MacOSX), GNU-HURD
- Très sécurisé
- Peu performant (Génération actuelles optimisées)



Exo-noyau

- Exokernel
- Performance Extensible
- Perte de contrôle des ressources par le noyau
- Difficile à mettre en oeuvre



Comparaison Monolithique/ Micro-noyau

Monolithique

Principes :
Un seul programme constitue le système

Avantages :
bonnes performances
(partage de données dans le noyau)

Inconvénients :
programme gisgantesque
difficile à maintenir et à faire évoluer

Exemples :
Locus, Sprite, Unix

Micro-noyau

Principes :
Un noyau minimal fournit des fonctionnalités de bas niveau
Les fonctionnalités du système réalisées par un ou plusieurs serveurs au-dessus du micro-noyau

Avantages :
Facilité de mise au point
Evolution facile
Modèle client-serveur adapté aux systèmes répartis

Inconvénients :
Performances

Exemples :
Amoeba, Mach, Chorus, V Kernel

Historique

Dans les années 80 : Recherche

Grapevine (Xerox)

Accent (CMU)

Amoeba (Amsterdam 84)

Chorus (Inria) 79

Mach (CMU) 86

V-System (Stanford) 83

Sprite (Berkeley)

Fin 80 debut 90 : Systèmes commerciaux

Chorus Systèmes 86

Mach-OSF-1

Amoeba (ACE)

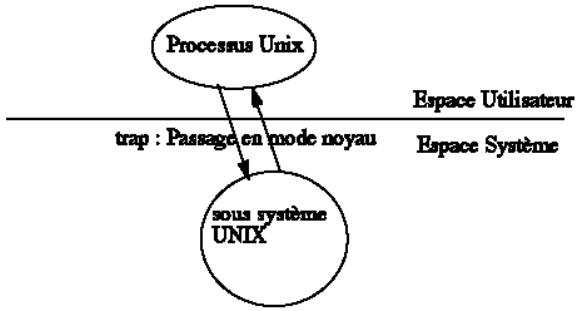
Windows NT (Microsoft)

Organisation

. Micro-noyau (base logicielle)

Serveurs
- de type système (dans l'espace du système)
- de type utilisateur

Sous systèmes : machines virtuelles (assuser la compatibilité binaire)



Micro-noyau

Conception de systèmes au-dessus de micro-noyaux

2 approches d'implémentation :

Un serveur unique

Avantage : Implémentation rapide à partir de l'existant (portage),
Inconvenient : Pas modulaire => non extensible

Multi-serveurs

Avantages : Modularité, extensibilité, mise au point
Inconvénients : Reconception totale,
Difficulté pour connaitre l'état global,
Communication entre les serveurs,
Performances

Mach

Le micro-noyau Mach

Objectifs :

Base pour la conception de systèmes d'exploitation
Support d'espace d'adressage
Accès transparent aux ressources distantes
Exploitation maximal du parallèlisme
Portabilité

Mach

Fonctionnalités de Mach

Mach fournit :

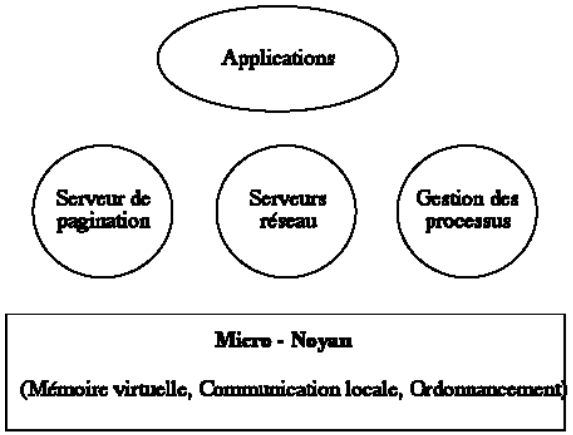
Gestion de tâches et d'activités (thread)
Communication entre tâches
Gestion de la mémoire physique et virtuelle
Gestion des périphériques physiques

Mach ne fournit pas :

Gestion d'un système de fichier hiérarchique
Gestion de processus
Gestion de terminaux
Chargement et écriture de pages
Gestion du réseau

Mach

Architecture du Système Mach



Abstraction de Mach

Abstractions de base :

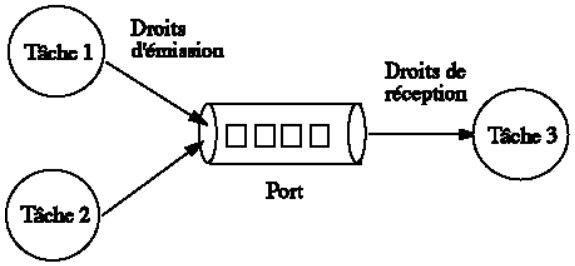
- Tâches (Tasks)
  Environnement d'exécution des activités (espace d'adressage)
- Activités (Thread)
  Unité d'exécution pour l'ordonnancement
- Ports
  Canal de communication (file de messages)
- Messages
  données typées

Abstractions secondaires :

- Ensemble de ports (Port Sets)

Communication : Les ports

- Un canal de communication
- Communication unidirectionnelle
- Un récepteur
  Un ou plusieurs émetteurs
- Droits d'accès transmissibles : en émission, en réception



Création : *Port_allocate*

Primitives de communication

Communication Asynchrone :

msg_send(msg, option, timeout)
msg_receive(msg, option, timeout)

Communication synchrone :

msg_rpc(message, option, ...)

Les options d'envoi (nombreuses) :

Specifier l'action si la file est pleine:
(attendre indéfiniment, délai, pas d'attente, transmettre le message à Mach)

Droits sur les ports

Accès à un port via un droit :

droit de réception
droit d'émission
droit d'émission unique (send_one) utilisé pour les RPC

Gérés par le noyau :

Table interne à Mach

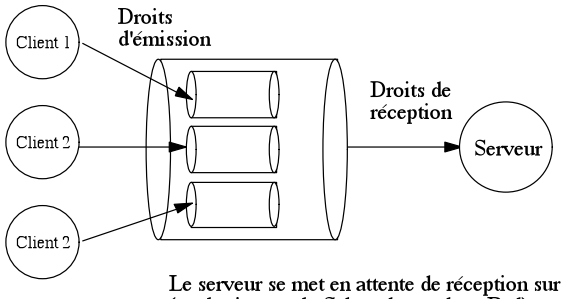
Les droits sur les ports sont transmis par msg_send

Transmission sur les droits d'émission :
l'émetteur et le récepteur ont les droits

Transmission sur les droits de réception :
l'émetteur perd les droits et le récepteur les récupère

Ensembles de ports

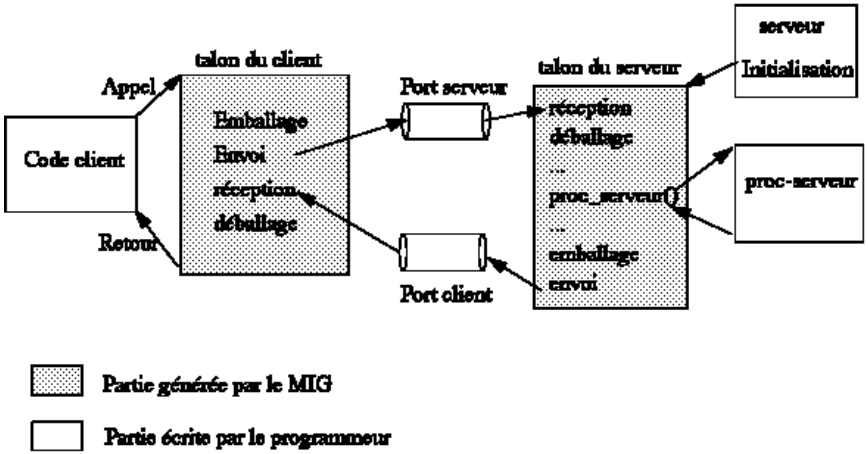
Regroupement de ports : Permet de définir plusieurs points d'entrée à un serveur



Le serveur se met en attente de réception sur un des port
(analogie avec le Select des sockets Bsd)

Opérations :
port_set_allocate, port_set_add, port_set_remove

Utilisation de MIG



Un outils : Le Mach Interface Generator (MIG)

Objectif : Simplifier l'écriture d'applications réparties

MIG utilisé pour générer automatiquement les fonctions de communications de type RPC (de manière similaire au rpcgen de Sun)

A partir d'un **fichier de spécification** 3 programmes sont générés :

- "User Interface Module": code de la partie cliente (le "stub" client)
- "User Header Module" : définitions des types et prototype
- "Server Interface Module" : code de la partie serveur ("stub" serveur)

MIG - Exemple d'utilisation

Exemple pour la fonction :
Ma_procedure(int a, int *b)

Fichier de spécification (mfonc.defs) :

```
routine Ma_fonction (
    server_port : port_t;
    arg : int;        /* parametre d'entree */
out   reponse : int; /* parametre de sortie */
)
```

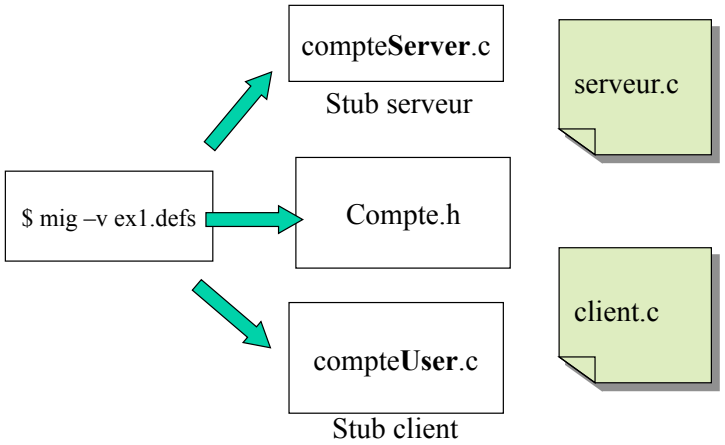
mig -v mfonc.defs

Exemple - MIG

```
/* Exemple : gestion compte */
subsystem compte 32768;

userprefix compte_;
serverprefix do_;

#include <mach/std_types.defs>

routine ajout
(
        serveur : mach_port_t;
        in val : int;
        out nouv : int
);

routine retrait
(
        serveur : mach_port_t;
        in val : int;
        out nouv : int
);
```

Ex1.def

Exemple



MIG : serveur.c

-
```
#include <stdio.h>
#include <mach/mach.h>
#include <mach/mach_error.h>
#include <mach/mig_errors.h>
#include <mach/message.h>
#include <mach/notify.h>
#include "compte.h"

#define MAX_MSG_SIZE 512

extern boolean_t        compte_server();
int val_compte = 0; /* Le compte */

/* implementation des 2 fonctions */

kern_return_t do_ajout(mach_port_t s, int val, int *nouv) {
  val_compte += val;
  *nouv = val_compte;

  return KERN_SUCCESS;
}

kern_return_t do_retrait(mach_port_t s, int val, int *nouv) {
  val_compte -= val;
  *nouv = val_compte;
  return KERN_SUCCESS;
}
```

Mig: serveur.c (2)

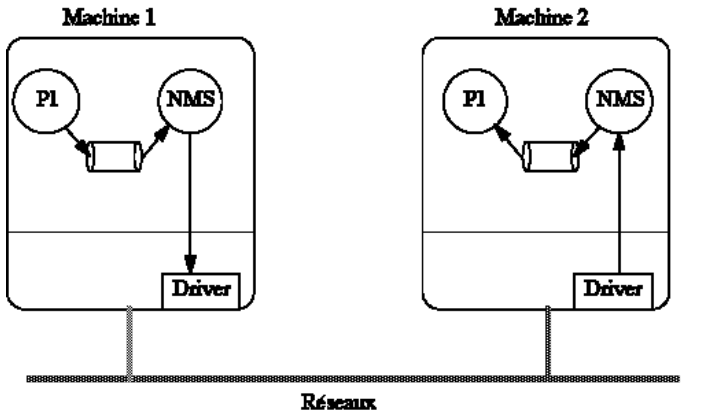
```
int main()
 {
   port_t ServerPort;
   kern_return_t retcode;

           /* Allouer un  port au serveur  */
   retcode = mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE,
 &ServerPort);

   if (retcode != KERN_SUCCESS){
     printf("mach_port_allocate %s\n",
              mach_error_string(retcode));
     exit(1);
   }

   /* Enregister le port dans le serveur de nom */
   retcode  = netname_check_in(bootstrap_port,"Essai", mach_task_self(),ServerPort);

   if (retcode != KERN_SUCCESS){
           printf("netname_check_in : %s\n",
                                   mach_error_string(retcode));
           exit(1);
   }

   /* Attente de message */
   retcode = mach_msg_server(compte_server, MAX_MSG_SIZE, ServerPort,0);

   printf ("(* !!!!! Server exited !!!!! : %s\n", mach_error_string(retcode));
   return 0;
 }
```

Mig : client.c

```
#include <stdio.h>
#include <mach/mach.h>
#include <mach/message.h>
#include <mach/mach_error.h>
#include "compte.h"

int main() {
  mach_port_t serveur;
  kern_return_t ret;
  int c;

  /* Rechercher le port du serveur */
  task_get_bootstrap_port(mach_task_self(), &name_server_port);
  ret = netname_look_up(name_server_port, "", "Essai", &serveur);

  if (ret != KERN_SUCCESS) {
    printf("Pb look_up : %s\n", mach_error_string(ret));
    exit(1);
  }

  /* Appel des stub clients */
  compte_ajout(serveur, 10, &c);
  printf("Valeur du compte = %d\n", c);

  compte_retrait(serveur, 5, &c);
  printf("Valeur du compte = %d\n", c);
  return 0;

}
```


Communications distantes

Première approche : Un serveur réseau sur chaque machine



Les serveurs réseau

Netmsgserver

- **Développé par Carnegie Mellon en 1986**
- **Composé d'une tâche multithreadée en mode utilisateur**
- **Chaque serveur à une vue cohérente de toutes les tâches s'exécutant sur le réseau (cohérence assurer à base de diffusion)**
- **Offre un service de communication transparent à l'utilisateur**

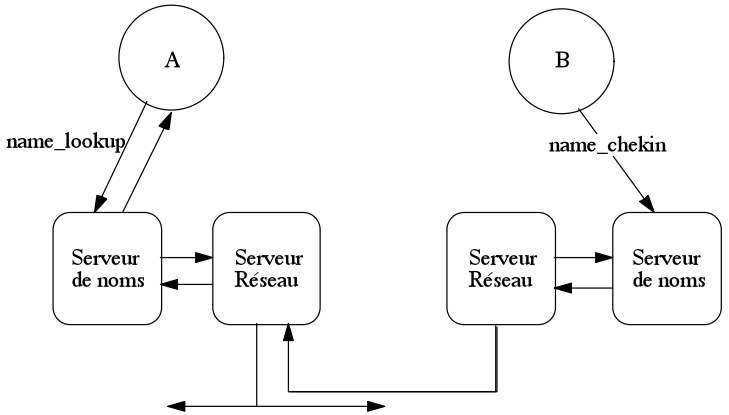
Mais problème de performances

Le serveur réseau générique (Masix)

- **Développer au Masi en 1995**
- **Garantit la transparence des communications**
- **Optimisation dans le protocole de résolution (gestion des caches)**
- **Limite les changements de contexte (deporter une partie du traitement directement dans l'espace des applications)**

Gestion des accès transparents

Exemple de Masix



Communication Distantes (2)

2ème approche : Gestion des communications intégrée dans le noyau

- **NORMA (OSF) : Définition de port "NORMA" globaux uniques dans le système**
- **Chaque site maintient dans le noyau une table globale des ports NORMA**

Avantage : les performances

Inconvénient : modification du micro-noyau => problème de portabilité

Les systèmes existants au-dessus de Mach

Différentes catégories :

Système UNIX : Lites, OSF1, BSD SS, MachOS X (NextStep), Mklinux, Hurd

Multi-environnements : MASIX, Windows

3 approches de conception :

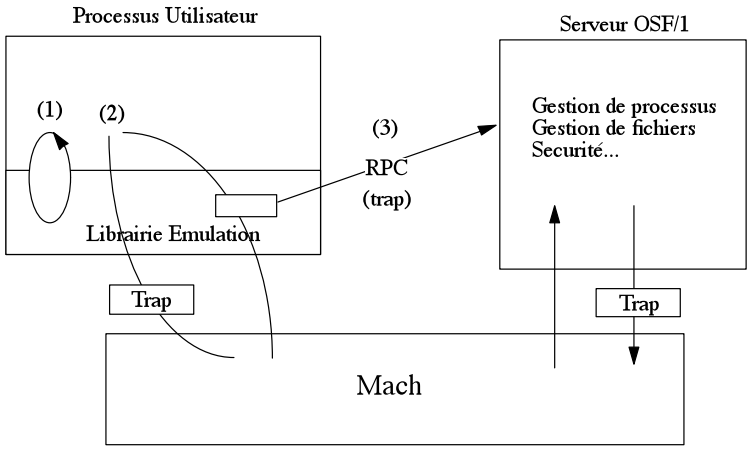
Monolithiques : OSF/1 IK

Serveur unique : OSF/1 MK, BSD SS, Sprite, MKlinux

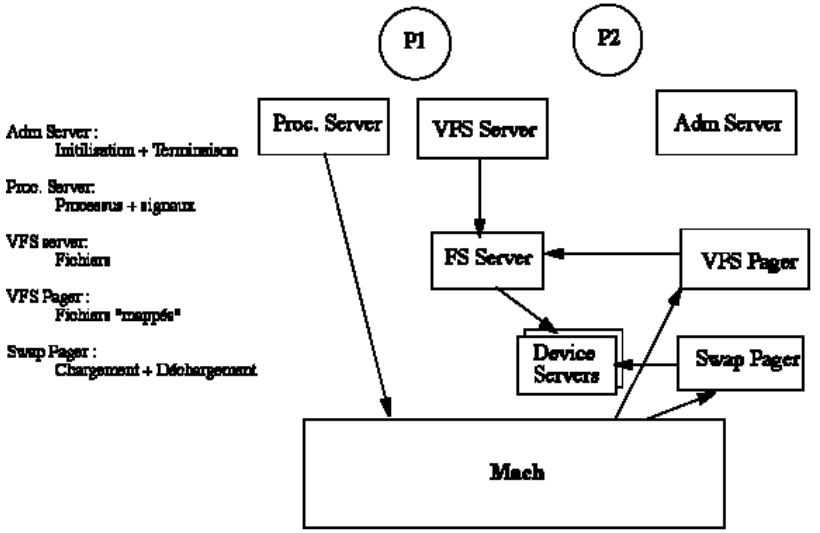
Multi-serveurs : **Hurd**, Unix Multi server (CMU), Guide, Masix

Approche Mono-serveur : OSF/ 1 MK

Réalisation des appels systèmes



Approche Multi-serveusr : MASIX



Introduction Chorus

Propriétés :

Temps réel

Répartition transparente des traitements et des données

Modularité

Structure :

Petit noyau temps réel (100Ko) intégrant :

- la gestion de la mémoire
- les communications entre tâches

Un ensemble de sous systèmes indépendants

Le système Chorus : Abstraction

- **Acteurs :**
  Unité d'encapsulation de ressources
  (espace d'adressage, activités, communications)

  Acteurs utilisateurs :
  espace d'adressage privé

  Acteurs système :
  réaliser certains appels système

  Acteurs superviseurs :
  espace d'adressage du noyau

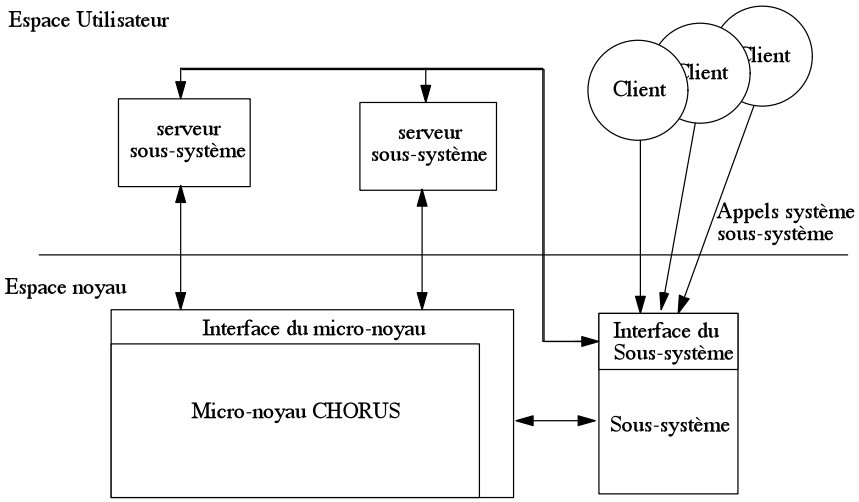
- **Activités :**
  Unités d'exécution appartenant à un acteur
  entités indépendantes pour l'ordonnancement

  Activité de supervision

  Activités utilisateurs

- **Désignation :**
  Identificateurs Uniques (UI) <site de création, type, n° incarnation+compteur >
  (64 bits)

Architecture Générale



Nommage des objets

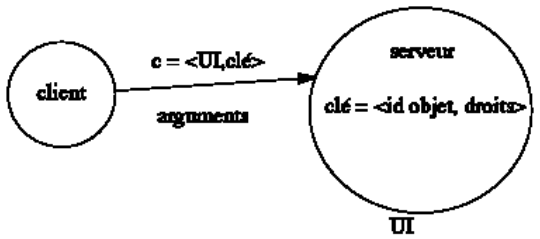
3 niveaux :

1°) Local Identifier (LI) : A l'intérieur d'un acteur

2°) Unique Identifer (UI) : Global et idenpendant du site (64 bits)
Site de création (13 bits), type de l'objet (3 bits), incarnation + compteur (48 bits)

3°) Capacité (groupes, acteurs, segments) : Global + Protection
UI + Clé (64 bits)

=> Gestion plus fine de l'objet



IPC - Chorus

Communication locale à un acteur

partage de données dans l'espace d'adressage
contrôle de concurrence : sémaphores

Communication entre activités d'acteurs différents

Communication par message (locale ou distante)
Communication synchrone / asynchrone
Transparence vis-à-vis de la localisation

Communication - Les portes (2)

- Identification : UI
- Localisation :
  - Une cache sur chaque site
  - Diffusion réquête de localisation`
- Droits d'émission vers une porte :
  - Connaissance de l'UI
  - Protection assurée par sous-système
- Droits de réception :
  - Appartenance à l'acteur possédant la porte

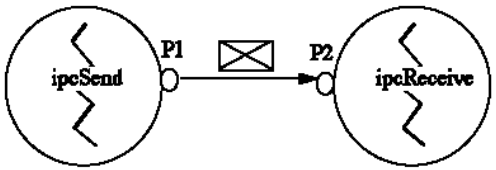
Création :
  portCreate : UI générée par le noyau
  portDeclare : UI construite
Suppression :
  portDelete
Migration :
  portMigrate (option, K_WITHMSGS, K_KILLMSGS)

Modes de communication

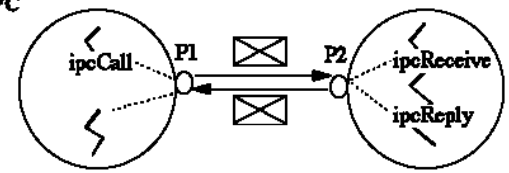
Messages

Forme libre

Asynchrone



Synchrone / RPC



sémantique au plus une fois (at most once)

Groupe de Portes

Objectif : Permettre de garantir une certaine stabilité du système en cas de défaillance d'un serveur

- Identification par un **UI de groupe**
- Capacité associée :
  - <UI du groupe, clé>
  - La clé sert à modifier la composition du groupe

Groupe statique :

Nom de groupe connu (utilisé pour générer une capacité)
(analogue aux ports des sockets Bsd ou aux clés des IPC système V)

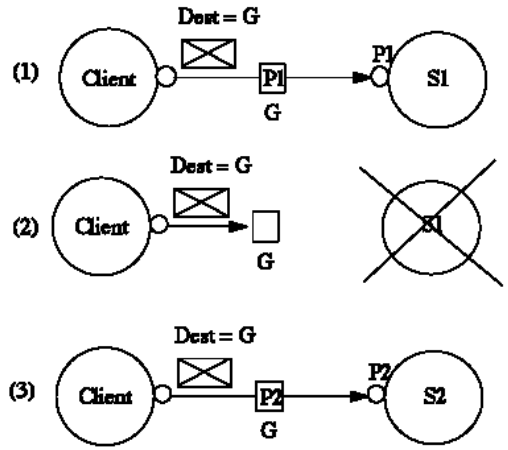
Groupe dynamique :

Une nouvelle capacité générée par le système

grpAllocate(K_DYNAMIC | K_STATUSER,...)

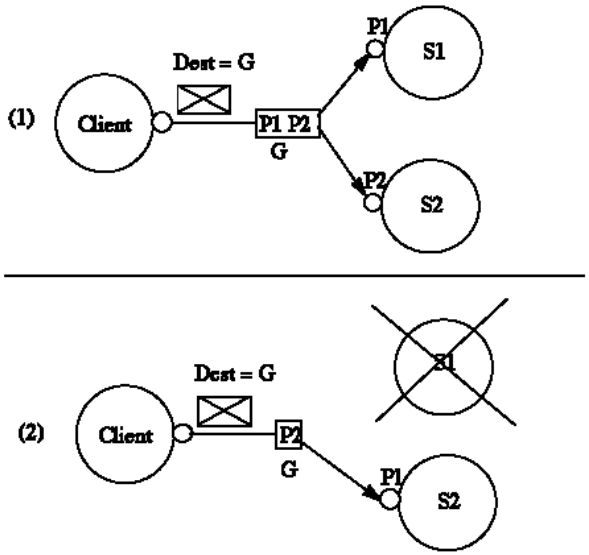
Utilisation des groupes

1) Reconfiguration



Utilisation des groupes

2) Réplication



Opérations et Modes d'adressage

Opérations sur les groupes :

Insertion : grpPortInsert

Extraction : grpPortRemove

Modes d'adressage:

- Diffusion (Broadcast) : vers toutes les portes du groupe
- Fonctionnel simple : vers une des portes du groupe
- Fonctionnel indicé : vers une porte du groupe sur le même site
- Fonctionnel exclusif : vers une porte du groupe sur un site différent

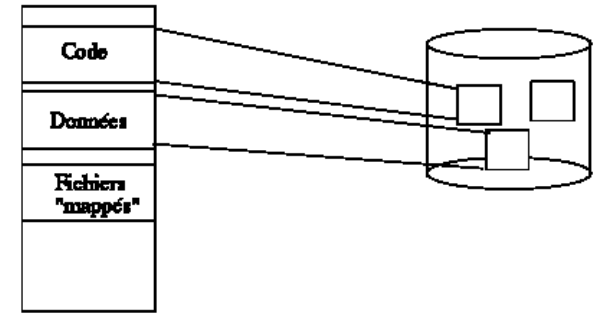
Exemple

```
KnUniqueId    myportUI;
int           myPort;
char *        message = "Hello world"
KnMsgDesc     msg;
KnIpcDest     dest;
int           result;

main()
{
     /* création de mon port */
     myPort = portCreate(K_MYACTOR, &myPortUi);

     /* Emission d'un message à partir de myPort vers dest/
     msg.flags = 0;
     msg.annexAddr = 0;
     msg.bodySize = strlen(message);
     msg.bodyAddr = (VmAddr) message;
     ipcSend(&msg, myport, dest);

     /* Réception d'un message sur myPort/*
     msg.flags = 0;
     msg.annexAddr = 0;
     msg.bodySize = 80;
     msg.bodyAddr = (VmAddr) message;
     result = ipcReceive(&msg, &myPort, K_NODELAY);
     if (result < 0)
         printf("Erreur réception");
     else
         printf("réception de %d octets", result);
}
```

Gestion de la mémoire

Espace d'adressage d'un acteur est découpé en **région**

Segment : abstraction associée à un objet qui pourra être projeté dans une région d'un acteur



Gestion de la mémoire (2)

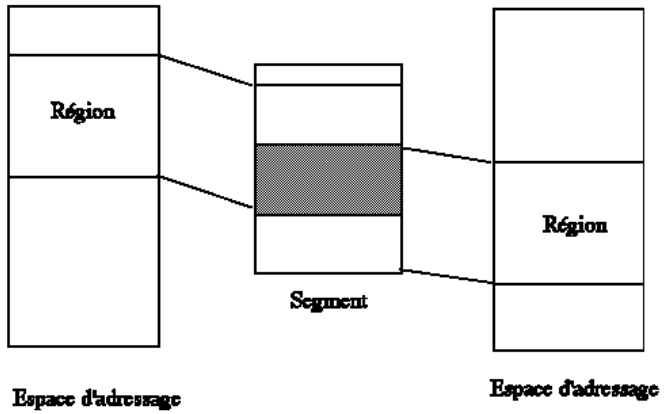
Segment :

- Désigné par une capacité => indépendant de la localisation
- Utilisé pour implémentation de fichiers mappés, mémoire paginée, mémoire partagée
- Géré par un serveur (mapper)
- Chaque serveur possède sa propre gestion de la cohérence, protection et représentation

Région :

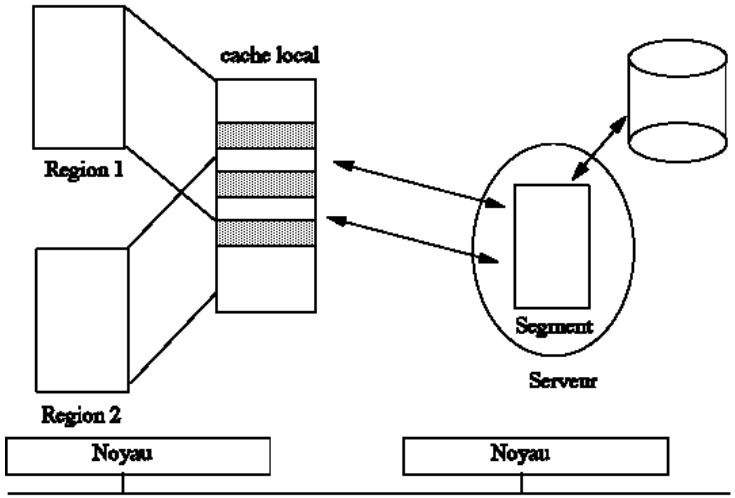
Associé à un portion de segment
Attributs :
- Position
- Protection
- Héritage (copie ou partage)
- Pagination à la demande autorisé

Partage

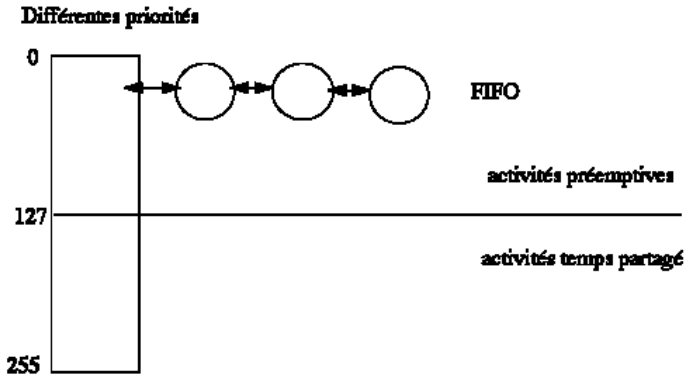


Cohérence gérée uniquement pour un partage entre acteurs locaux

Représentation des segments en mémoire physique



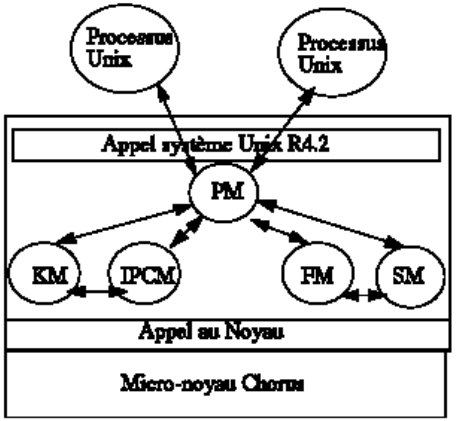
Ordonnancement Chorus



Modularité : Implanter plusieurs politiques d'ordonnancement
1 classe par défauts + 1 classe pour le sous-système Unix

Sous-système : Exemple Chorus MIX

SUSI : Single Unix System Image



PM = Process Manager
FM = File Manager
SM = Streams Manager
IPCM = IPC Manager (System V)
KM = Key Manager

Amoeba

- Micro-noyau multi-serveurs utilise le modèle de "pool de processeurs"
- Orienté objet
- Objets référencés par des capacités
- Intègre la notion de threads
- Pas de gestion de mémoire virtuelle !

Communication :

- Utilise des ports (similaire à Mach)
- Possibilité de diffuser des messages à des groupes logiques de ports

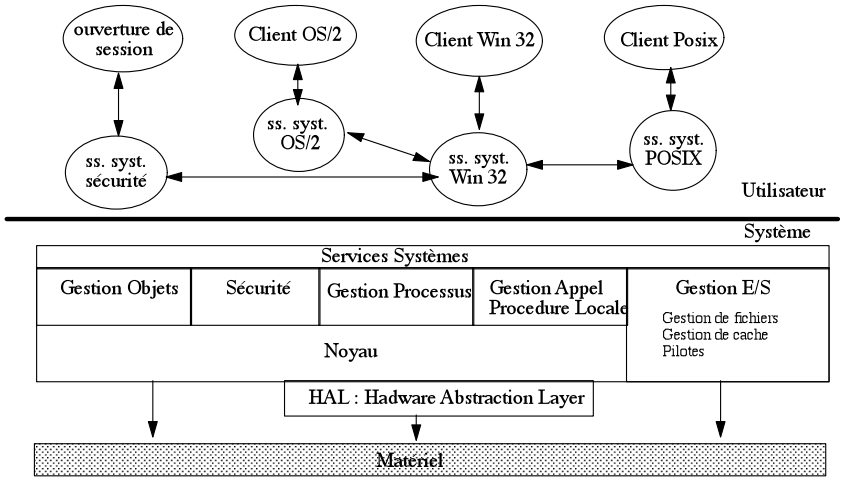
V - Kernel

- Notion de processus legers
- Défintion de groupe de processus légers
- Ordonnancement à deux niveaux :
  Micro-noyau, Externe

Communication :

- Pas de port (on nomme le processus destinataire)
- Communication par RPC uniquement
- Diffusion vers un ensemble de processus
- Messages de 32 bits !

Architecture de Windows NT



Synthèse sur les micro-noyaux

Points communs :

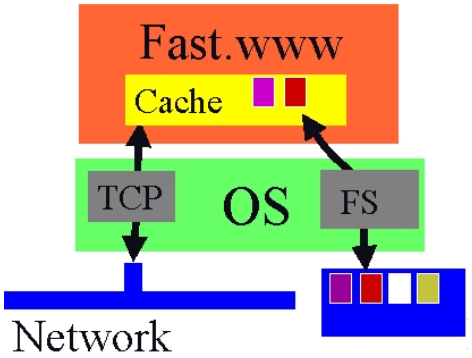
- **Tâches et activités**
- **Communications**

Spécificités :

- **Amoeba : pas de mémoire virtuelle**
- **Chorus : Migration de portes, Message handler**
- **Mach : Ecoute sur un ensemble de ports**
- **V kernel : pas de ports, messages de 32 bits**

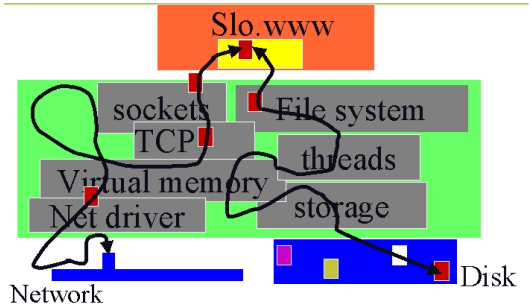
Exokernel (MIT)

• Lourdeur des systèmes traditionnel :
Exemple un serveur web



Exokernel : Motivations

Système traditionnel

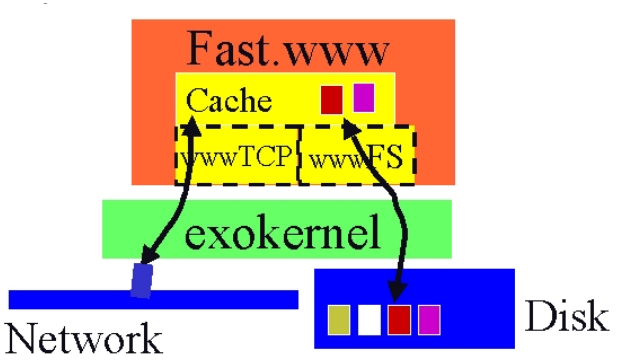


La solution Exokernel

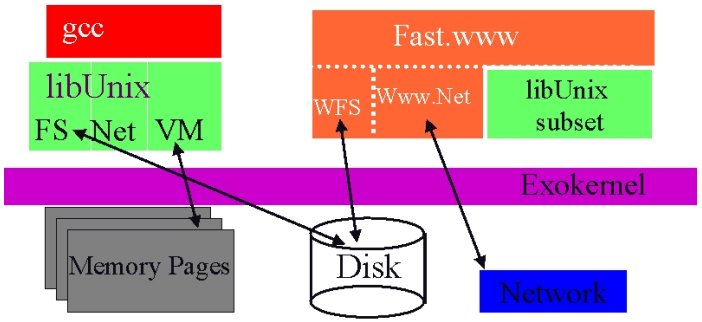
- Noyau externalisé dans l'espace utilisateur
- Noyau minimum
- Toutes les fonctions dans des bibliothèques : les libOS

=> moins de recopies plus de partages

Architecture Exokernel

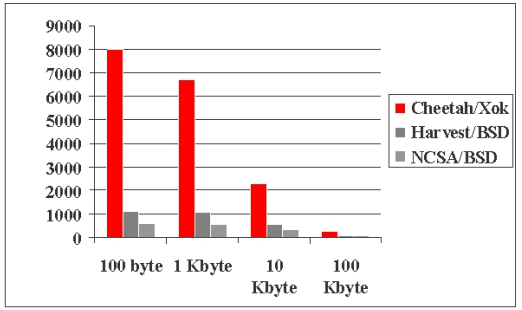


Architecture (2)



Performances Exokernel

The Cheetah Web Server



Exokernel - Conclusion

- Approche alternative performante
- Complexité de mise en œuvre (développement des libOS)
- Sécurité

L4 Microkernel

- 1995 - German National Research Center for IT
- Nouveau micro-noyau :
  - Objectif améliorer les performances des micro-noyaux existants (Mach)
  - => Améliorer les échanges entre serveurs (IPC)

L4 Abstractions

- Espace d'adressages
  - Map, Grant, Unmap
- Threads
  - Ordonnancement de type RR avec 256 niveaux de priorités
  - Flexible : possibilité d'émuler une stratégie FIFO pour le temps réel
- IPC
  - Messages courts (registres)
  - Copie limitée de grands messages (partage de l'espace de l'émetteur)

L4 Performance

	8 Byte IPC	512 Byte IPC
L4	5 μs	18 μs
MACH	115 μs	172 μs