

Master d'Informatique de Paris 6 **M1**  
Spécialité SAR

Architecture Avancée des Noyaux des Systèmes  
d'Exploitation

UE 4I401 : **Noyau**

2017/2018

Introduction Noyau  
Gestion de Processus

Pierre Sens



# Cours n° 1

## Introduction au Noyau



### PLAN DU MODULE

## Objectifs

### Mécanismes internes du noyau (UNIX)

Processus,  
Ordonnancement  
Fichiers,  
Mémoire virtuelle



**Début TD : La semaine du 25 septembre (sauf AFTI)**

**Notation : 40% Examen réparti 1 + 60 % Examen réparti 2**

**Equipe enseignante**

Cours : Pierre Sens

Pierre.Sens@lip6.fr

TD : Luciana Arantes, Philippe Cadinot,  
Swan Dubois, Jonathan Lejeune  
Julien Sopena  
{Prénom.Nom}@lip6.fr

**TD/TME : Etude noyau Unix V6 (~10 000 lignes de code C)**



**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

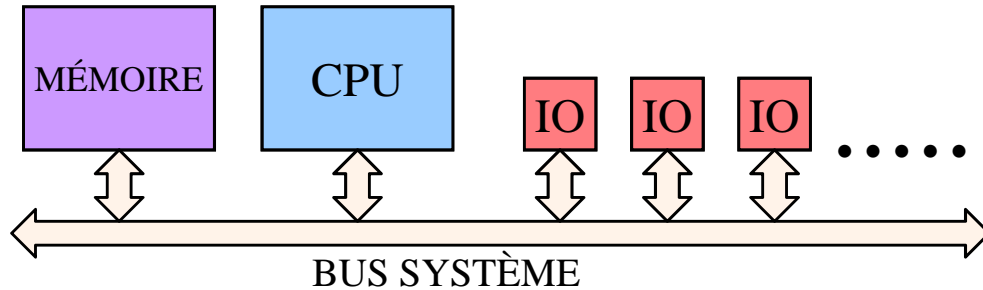
**Mécanismes internes Windows**

Windows® Internals, Fifth Edition

## Vue d'une machine

### Ensembles de composants organisés autour de bus

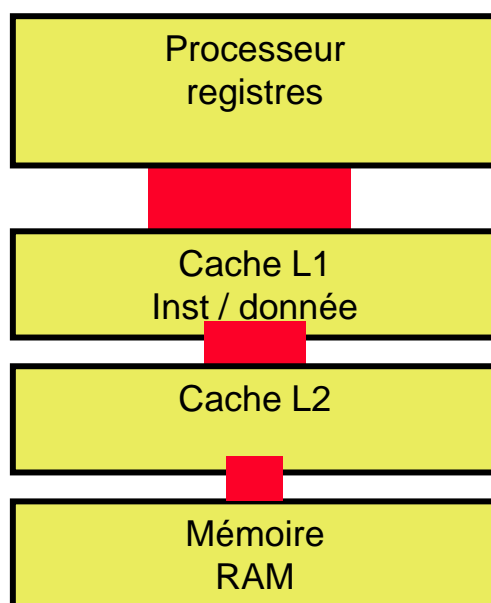
Composants de base: mémoire, CPU, I/O, bus système



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 : Les niveaux de caches



Temps accès < 1ns

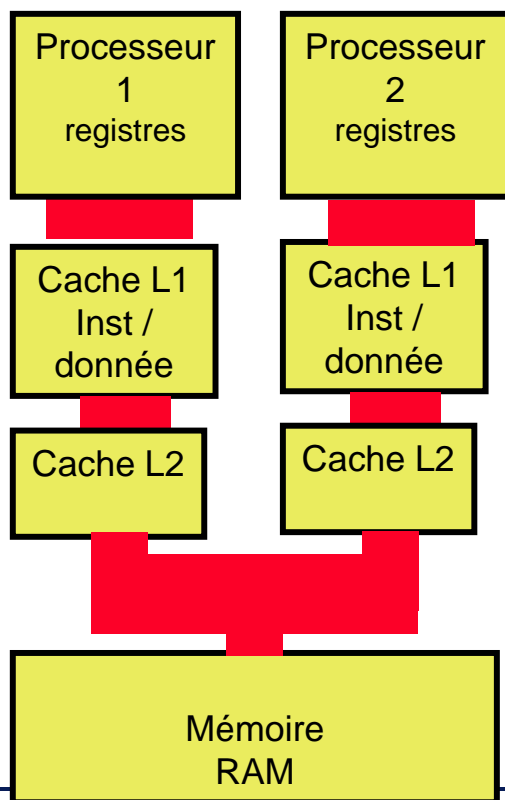
Temps accès ~1ns

Temps accès 2-3 ns

Temps accès 10-50 ns

Disque : 10 ms ( $\times 10^6$  RAM !)

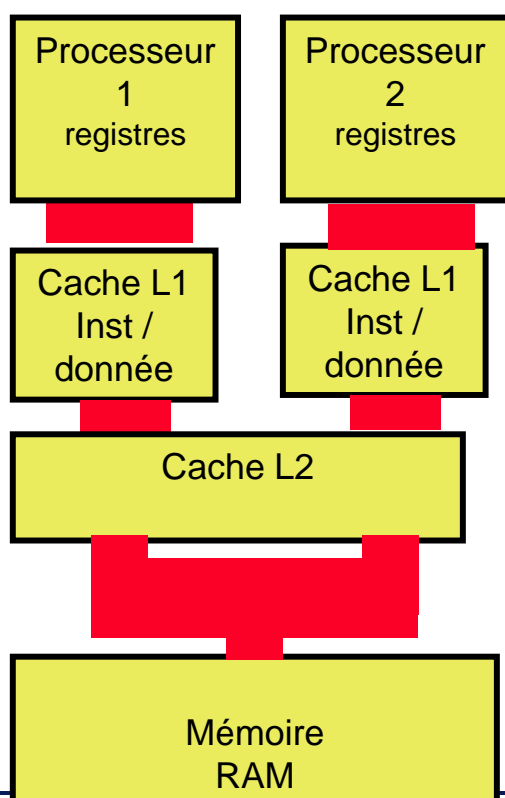
## Vue d'une machine : Multi-processeur SMP



### 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)

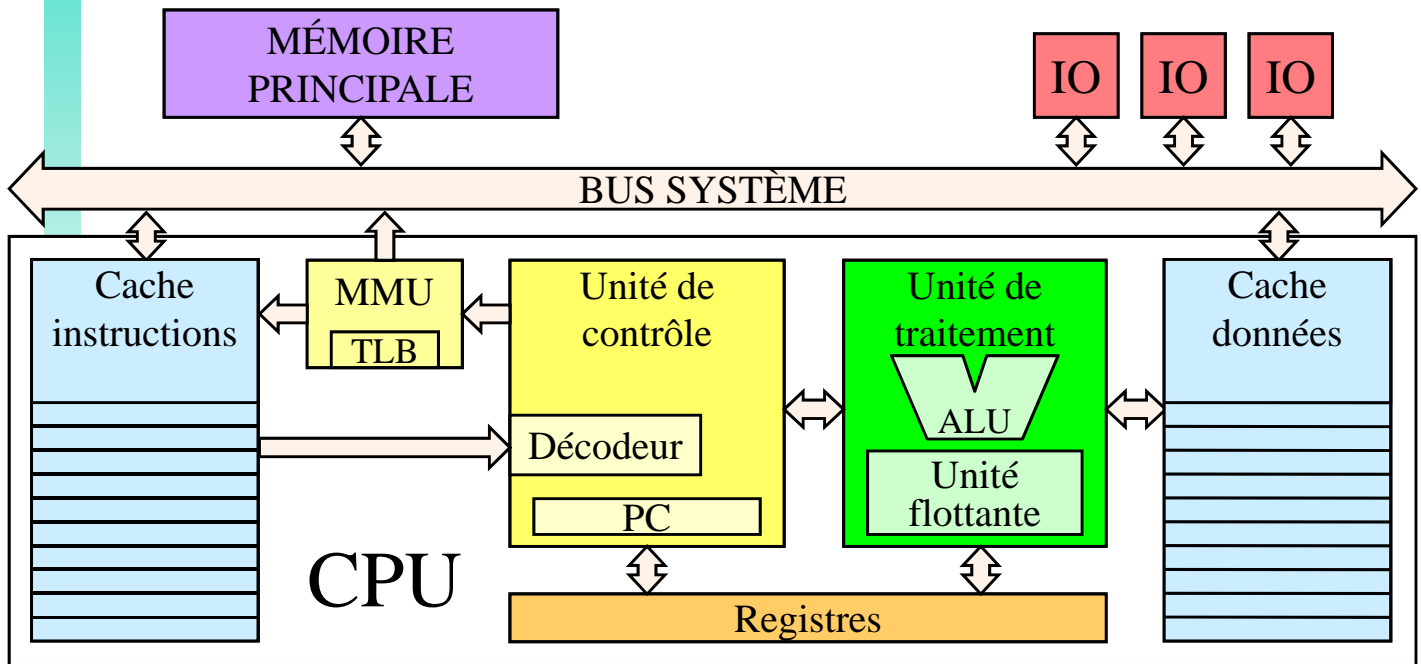
## Vue d'une machine : Multi-core



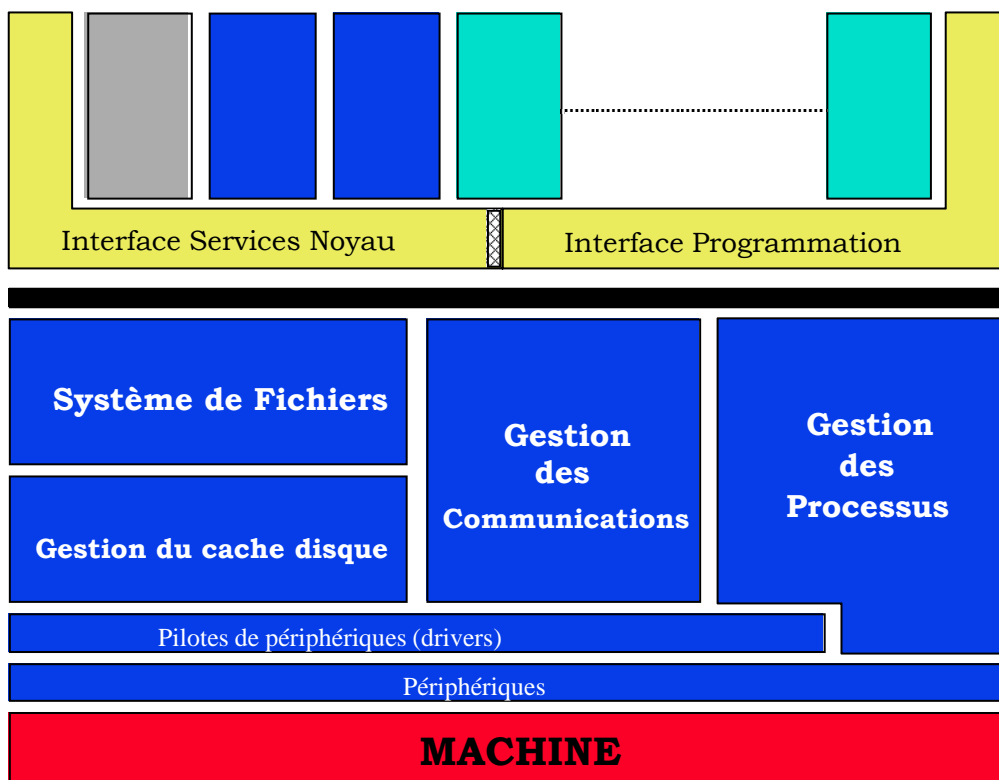
### Architecture *Multi-core* (ex. Intel Dual/Quad core, Cell)

- 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

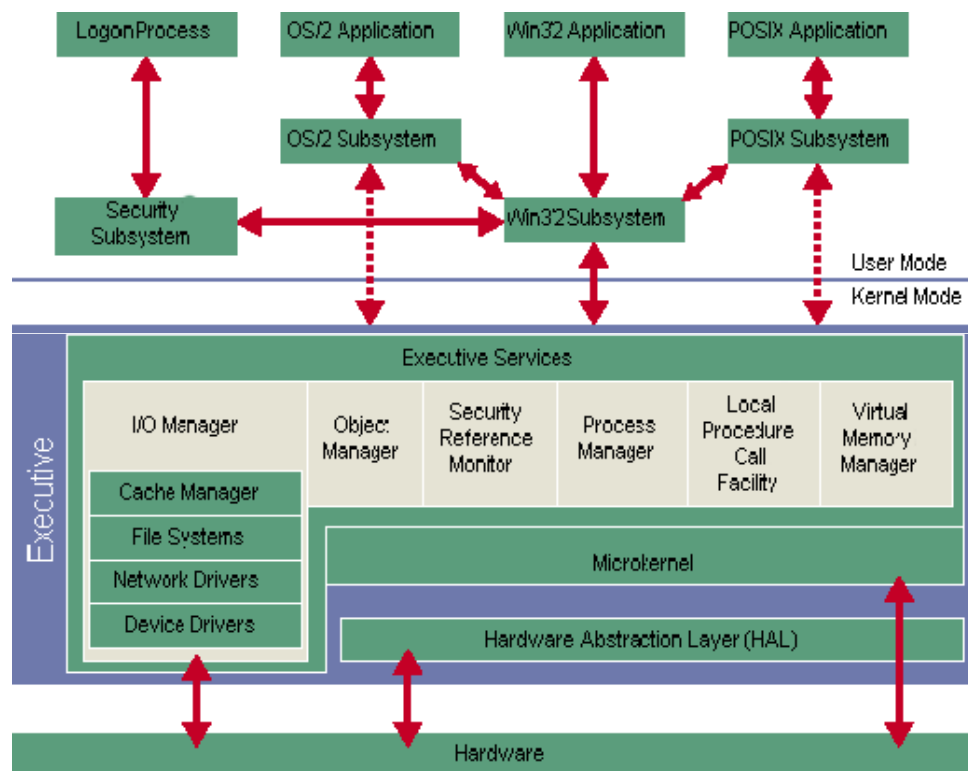
## Vue d'une machine



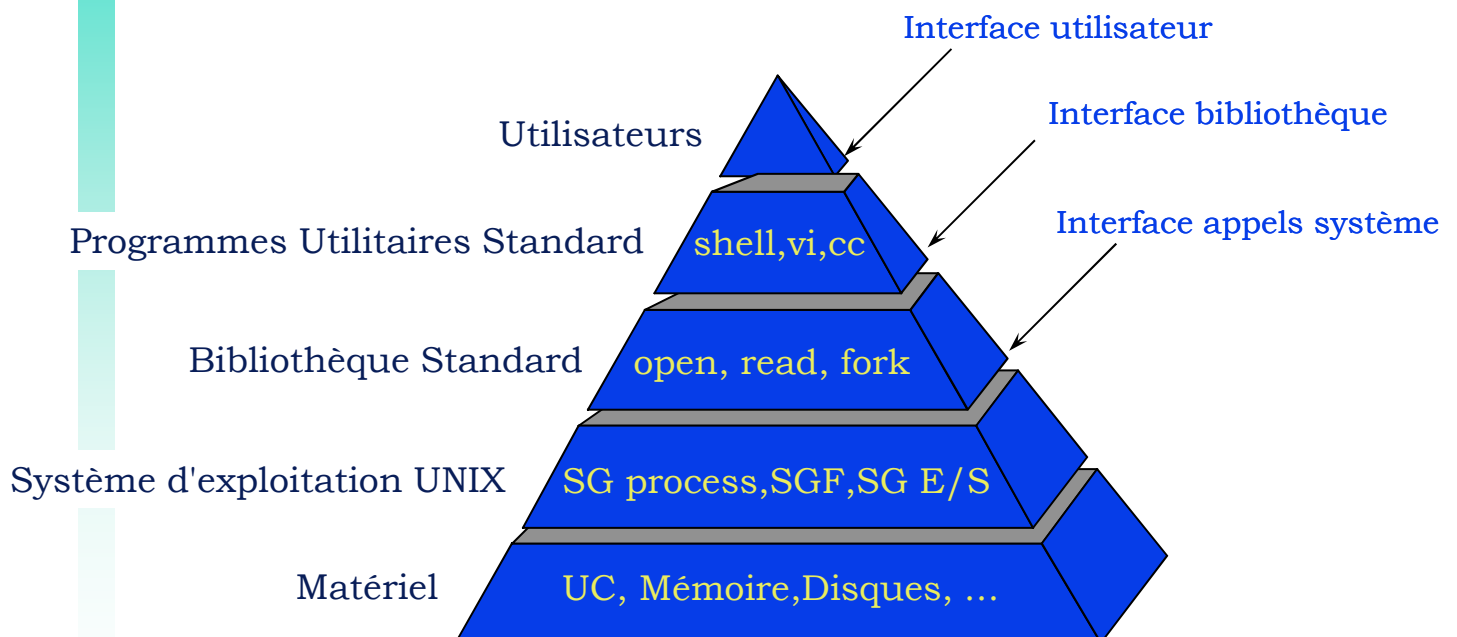
## Vue du système : architectures classiques



## Vue du système : architectures modulaire (Windows NT)



## UNIX - Généralités (1)



## UNIX - 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 Foundation) : 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)

À l'origine UNICS (UNIplicated information and Computing System)

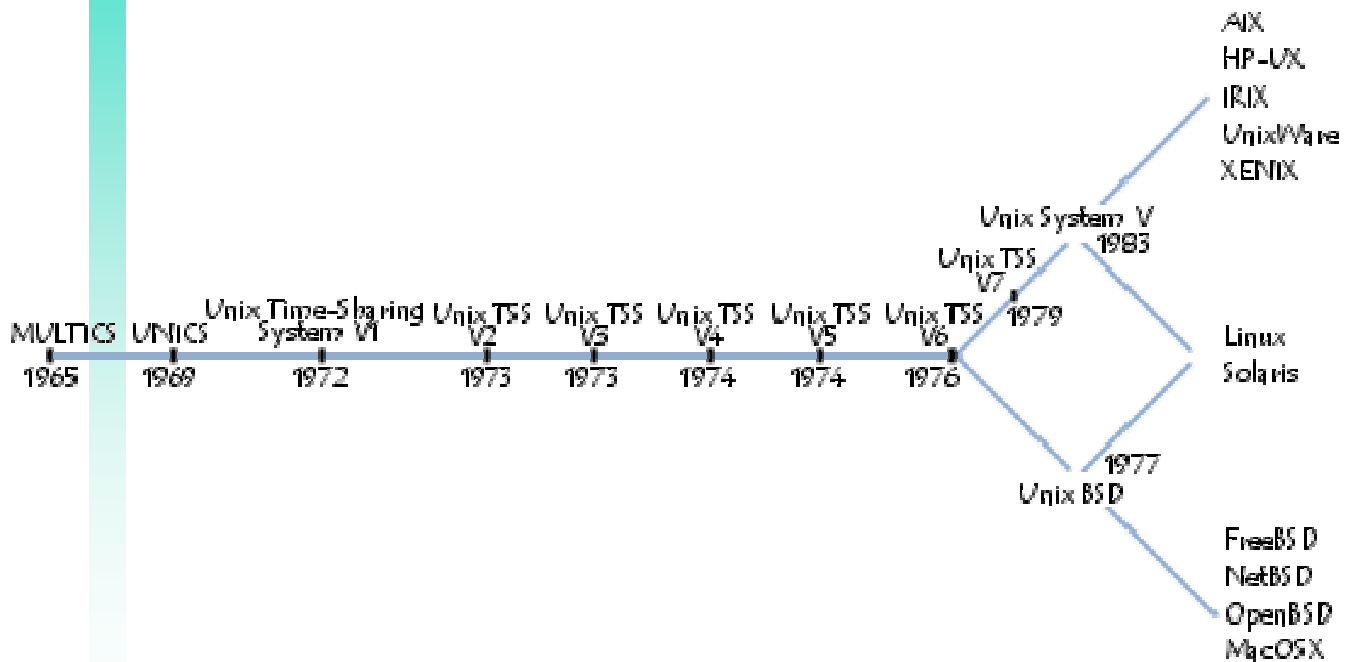
1969 UNIX/PDP-7 Ecrit en assembleur sur un PDP-7 pour développer un traitement de texte aux Bell Labs

1973 UNIX V5 Langage C (90%)

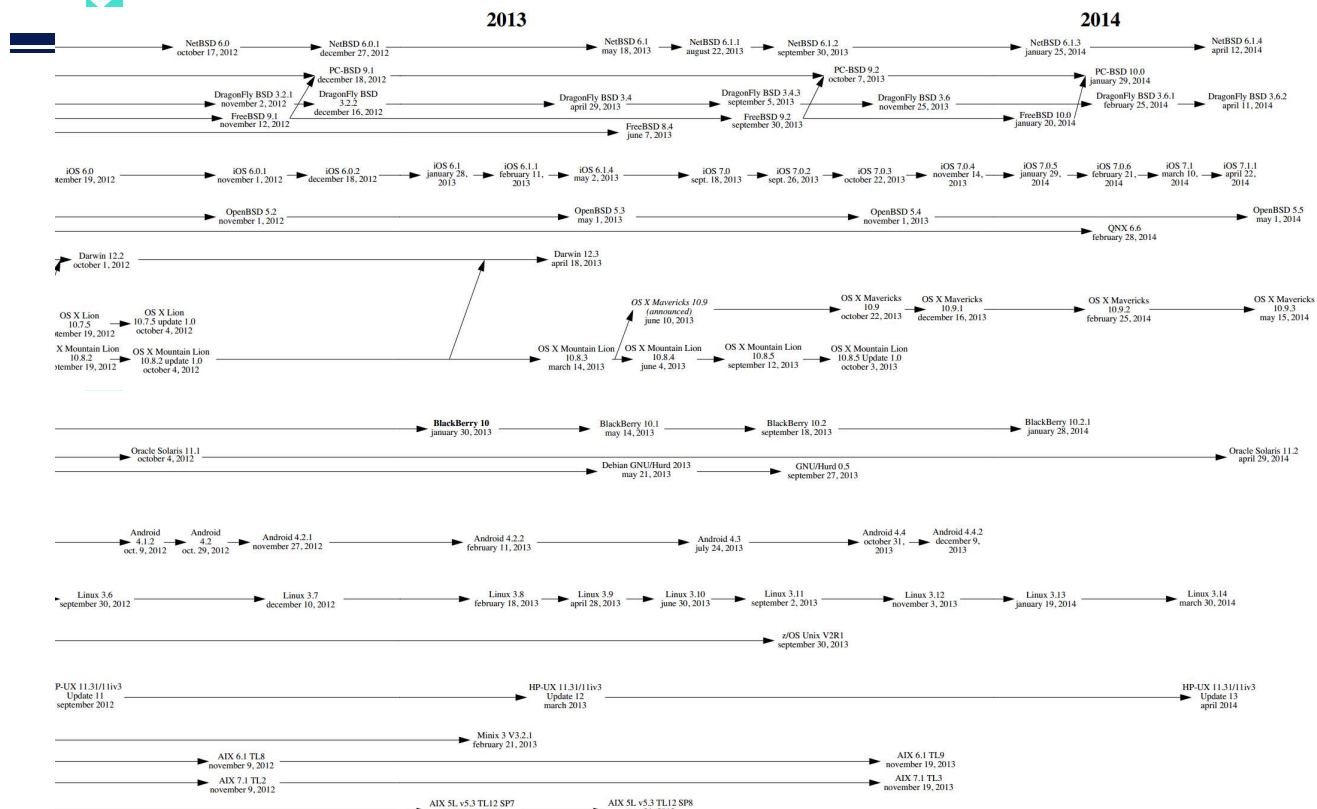
1976 UNIX V 6 (coopération de Bell Labs/universités américaines)

	<b>Berkeley</b>	<b>Bell Labs</b>	<b>AT&amp;T</b>
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
<b>1991</b>		<b>Linux</b>	

## Historique (2)



## Unix aujourd'hui



## Limites d'Unix

---

**Complexité de certaines versions => problèmes de robustesse**

**Interface utilisateur**

**Prolifération des versions => situation chaotique**

**Approche monolithique => difficilement extensible**

# Mécanismes Internes

---

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

1

## Architecture générale

---

Appels système						
Fichiers ouverts						Mémoire virtuelle
socket	VNODE					
	NFS	UFS (locaux)			fichier spéciaux	
		FFS	LFS	...	Gestion de terminaux	
Protocoles réseau		buffer cache				
pilote réseau		pilote bloc			pilote caractère	
Matériel						

2

# Accès au service du noyau

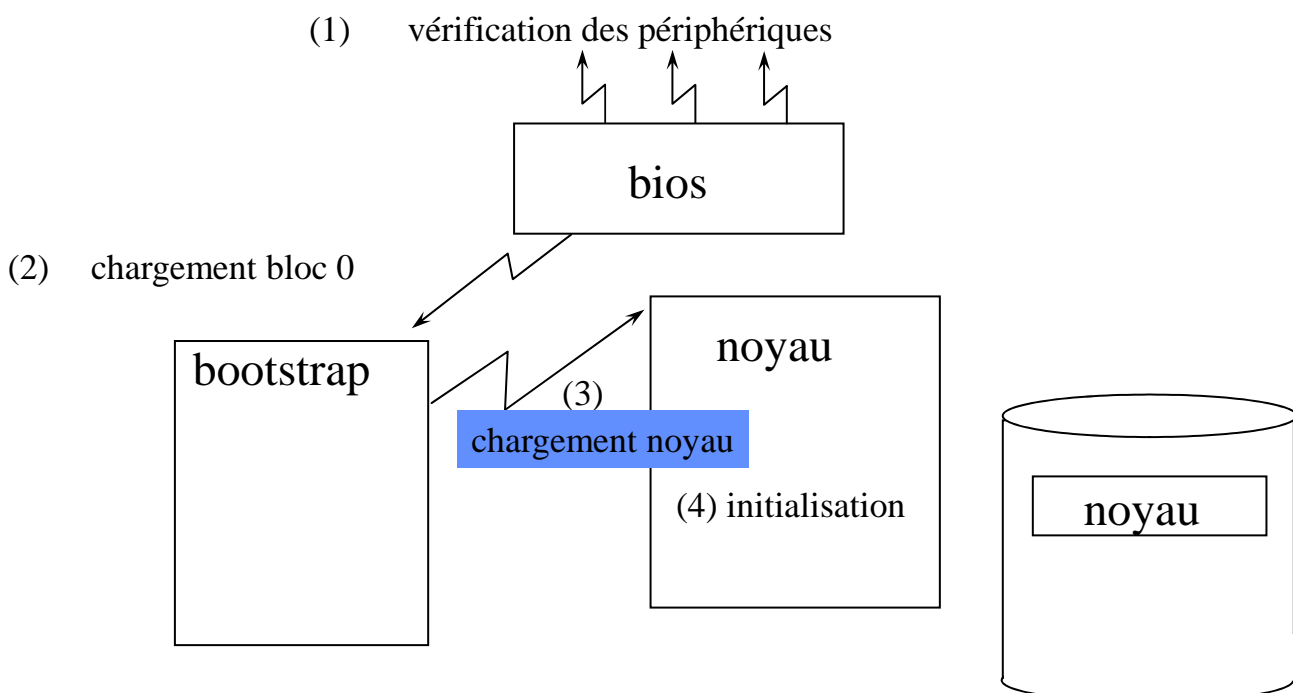
---

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

3

## Démarrage

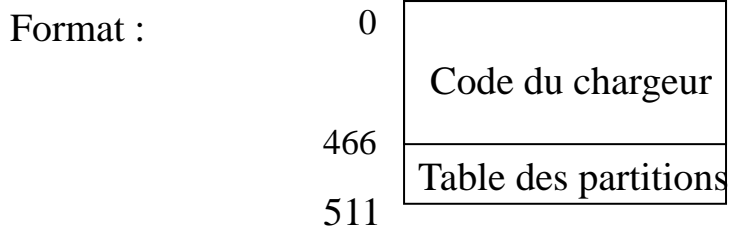
---



4

# boot bloc et multi-boot

---



## 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

5

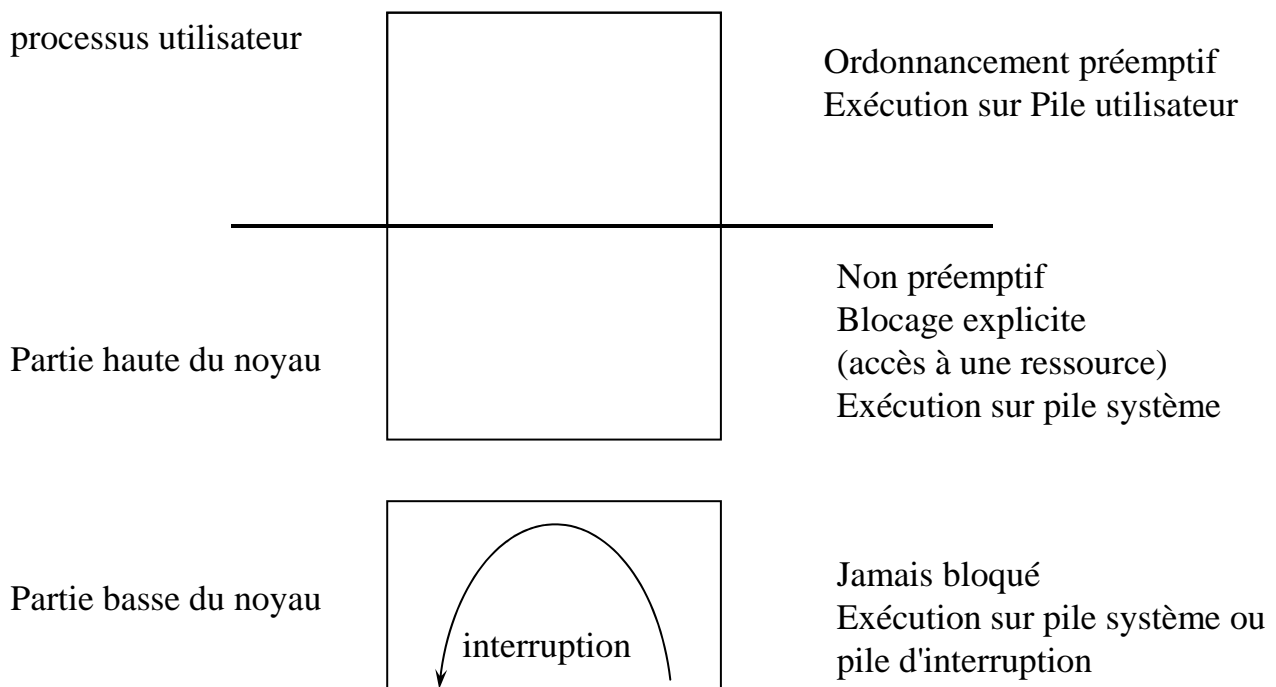
# Bios vs EFI

---

- 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 informations de boot **GUID Partition Table (GPT)**
  - GPT contient adresse programme d'armorage et la table des partitions
  - Entrée sur 64 bits => Taille max =  $2^{64}$  blocs

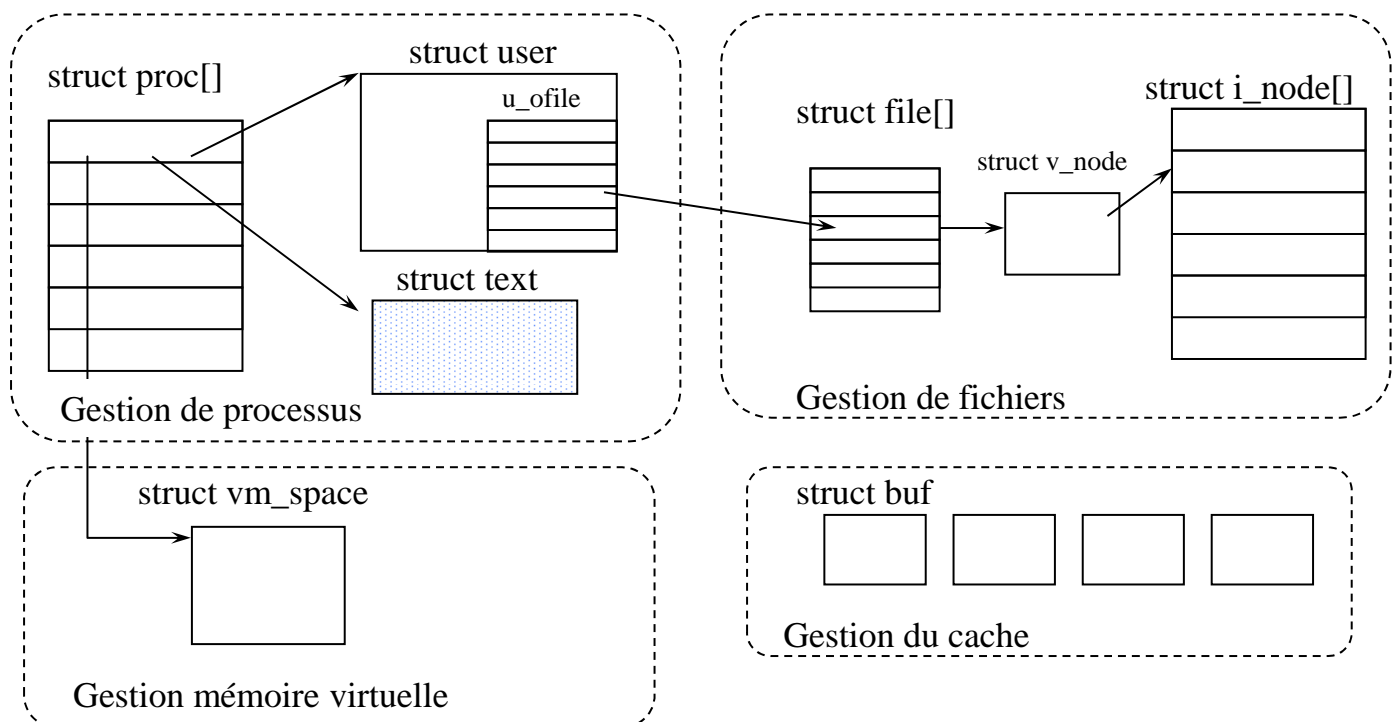
6

# Organisation et mode d'exécution



7

## Les principales structures



8

# Gestion de processus

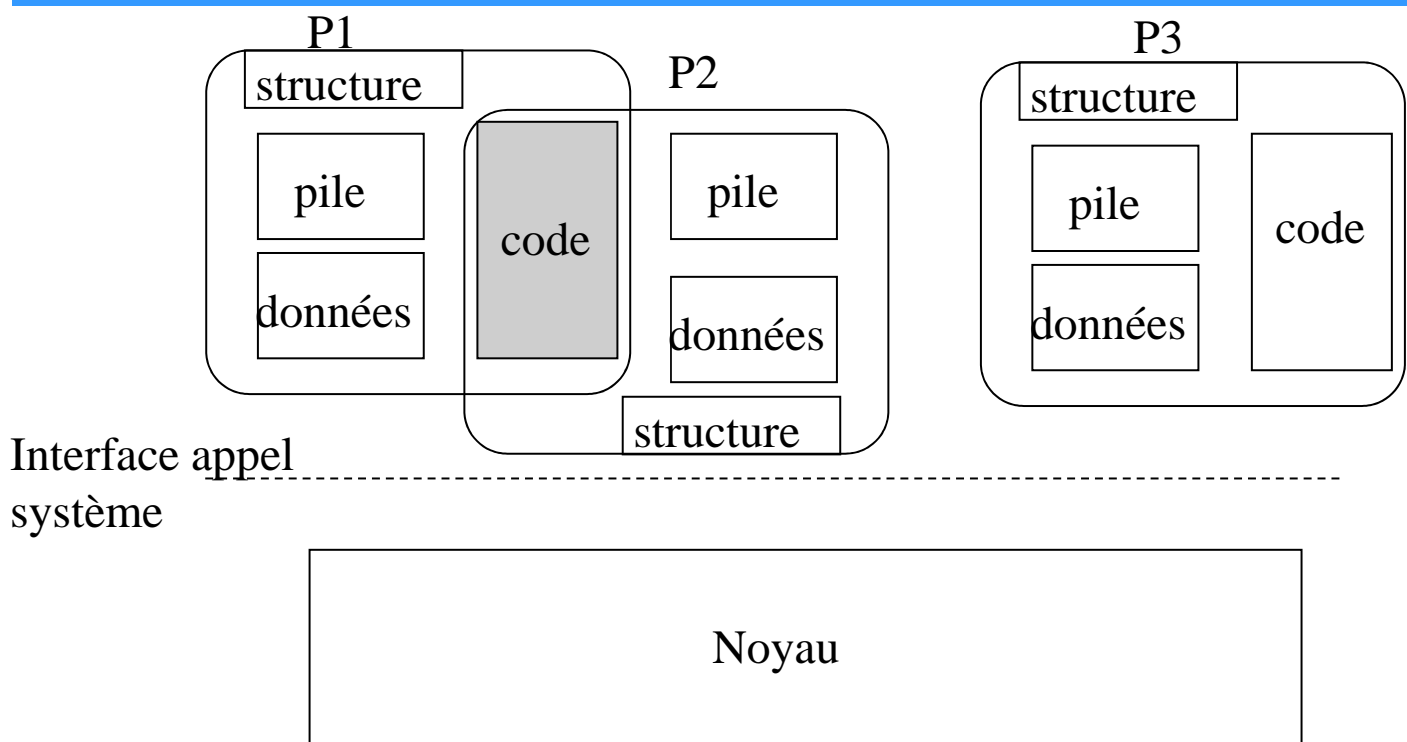
---

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

1

## Architecture

---

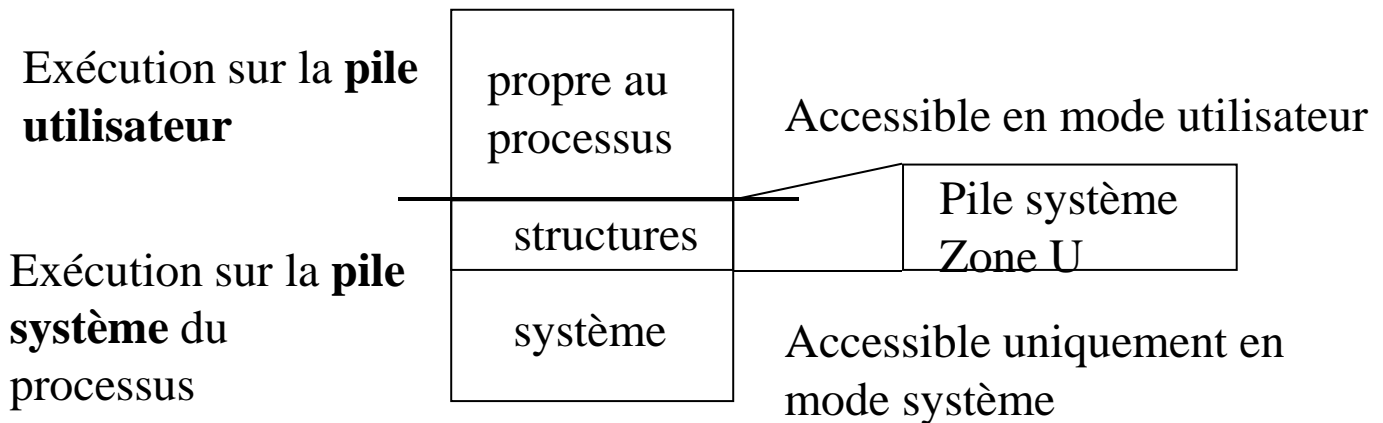


2



# 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 !



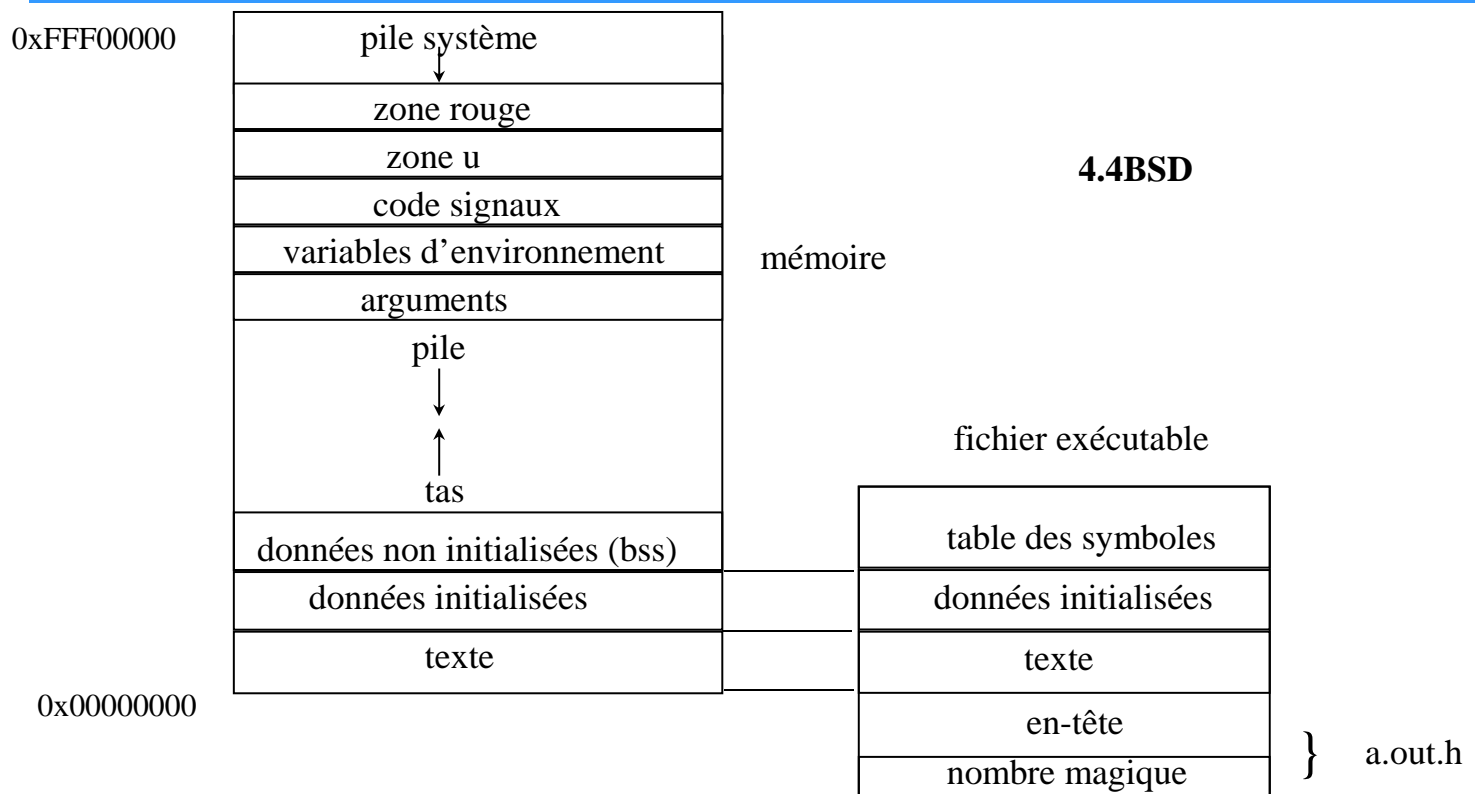
3

## Structure interne

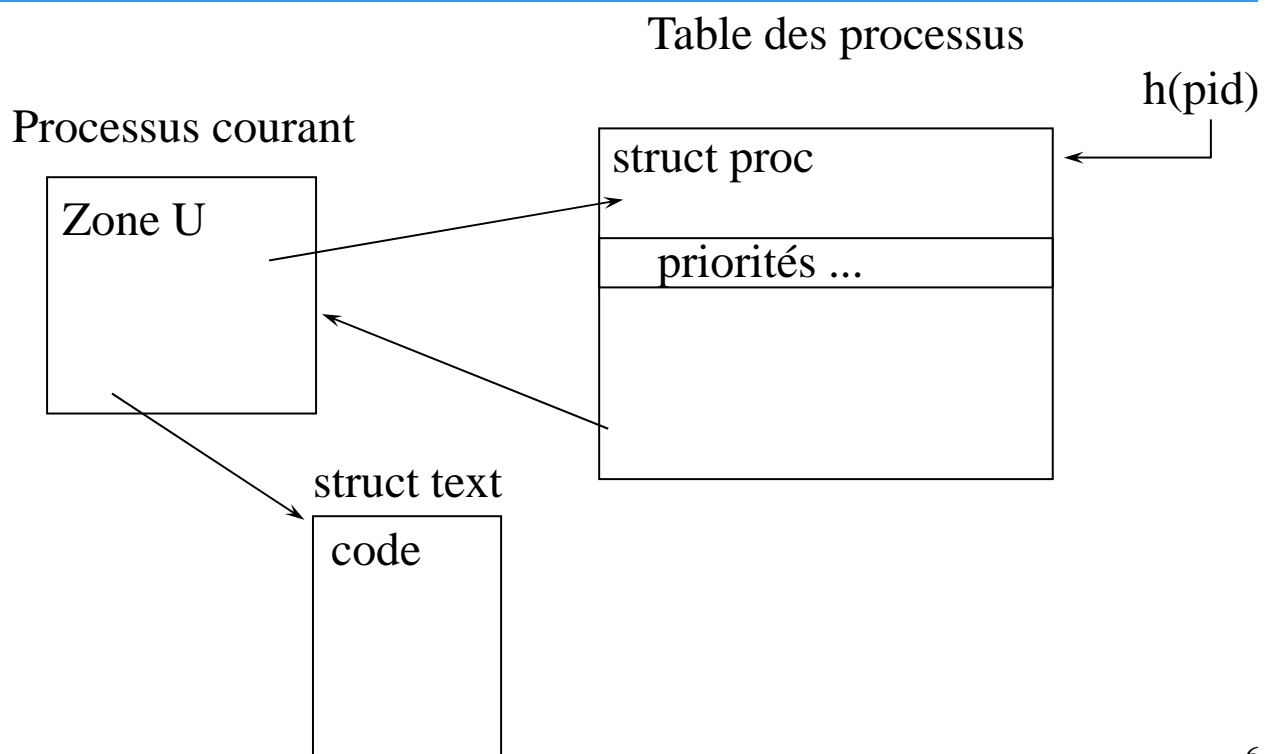
- Contexte :
  - Espace utilisateur (données, pile)
  - Information de contrôle (zone u, struct proc)
  - Variables d'environnement
- Contexte matériel :
  - Compteur ordinal
  - Pointeur de pile
  - Mot d'état (Process Status Word) : état du système, mode d'exécution, niveau de priorité d'interruption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

4

# 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 ]

7

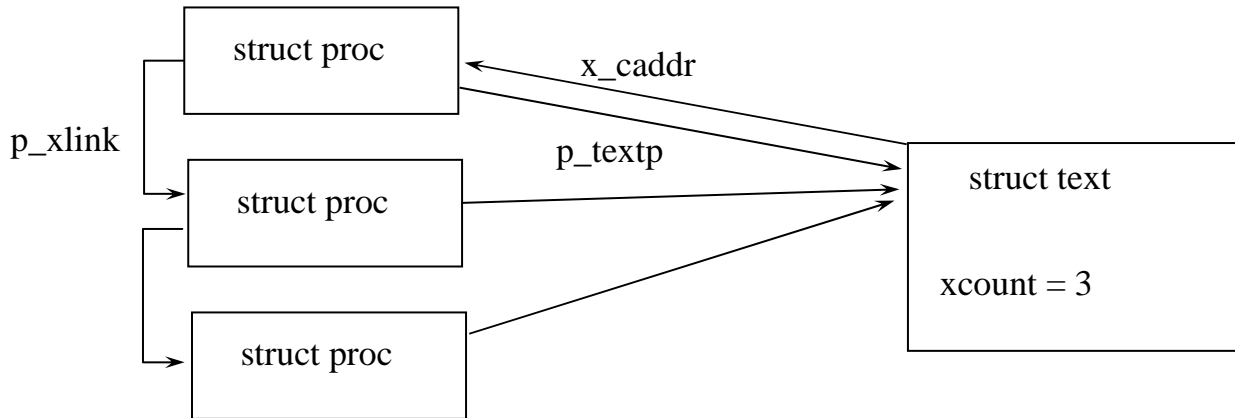
# Structure résidente

---

- 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

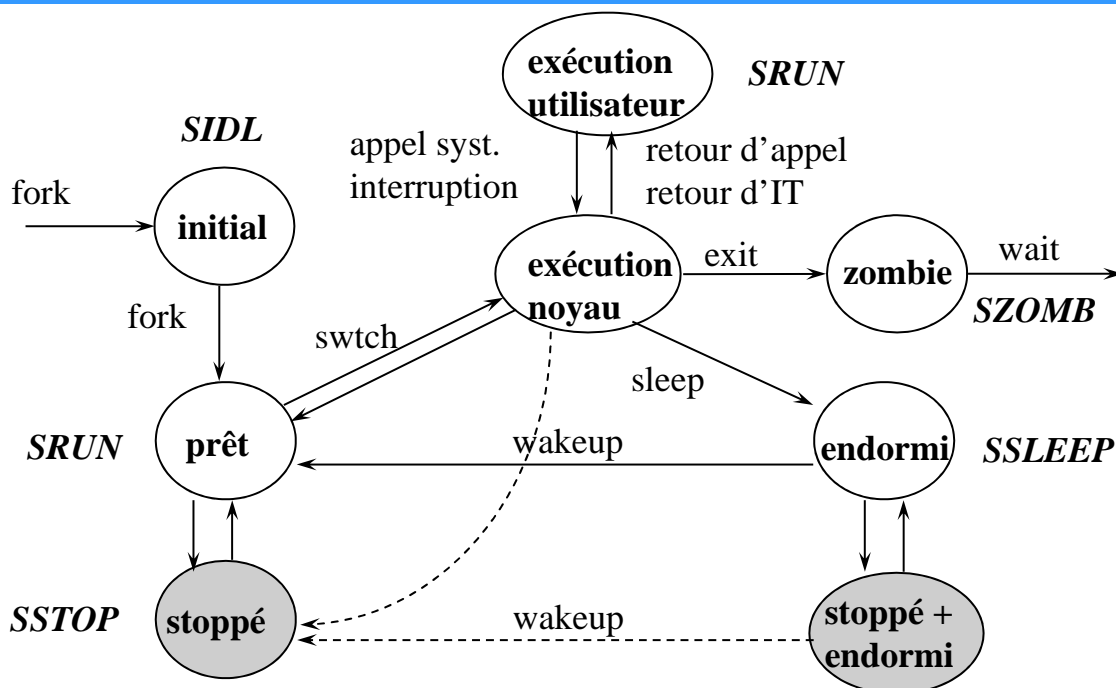
8

# Partage de code



9

# Etat d'un processus

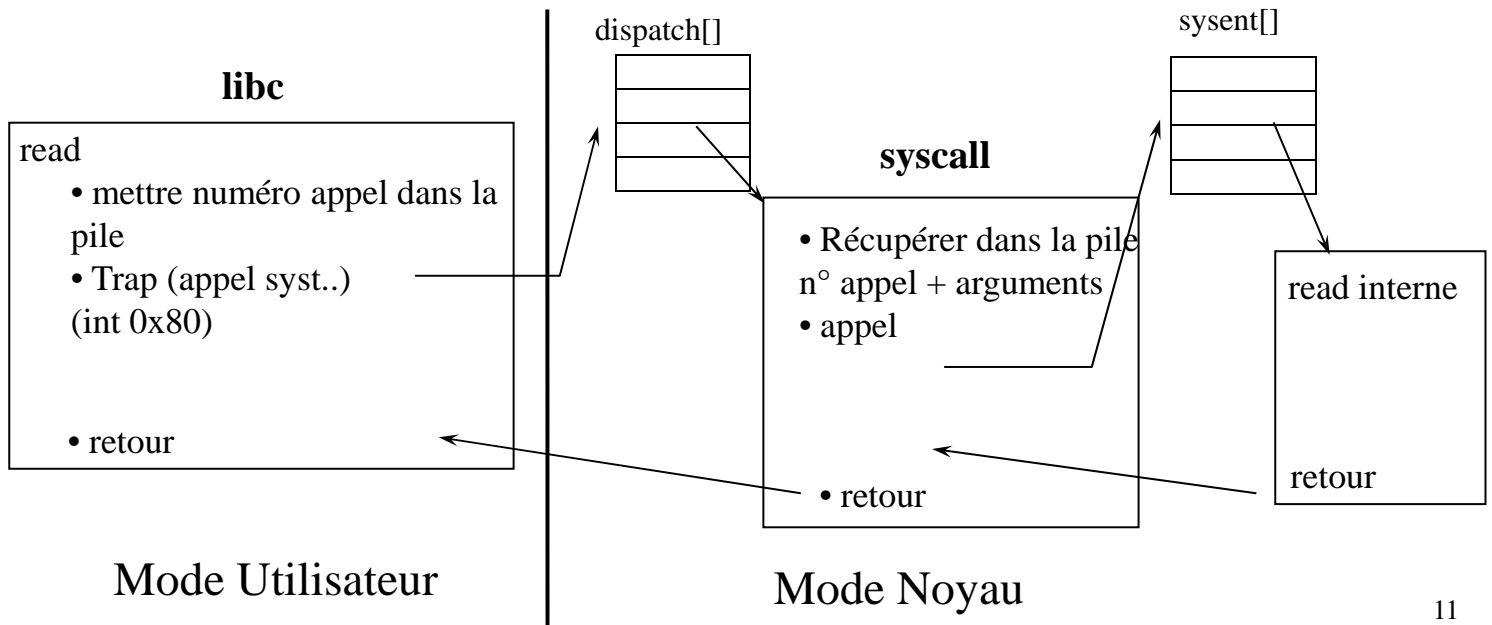


BSD

10

# Interface des appels systèmes

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



11

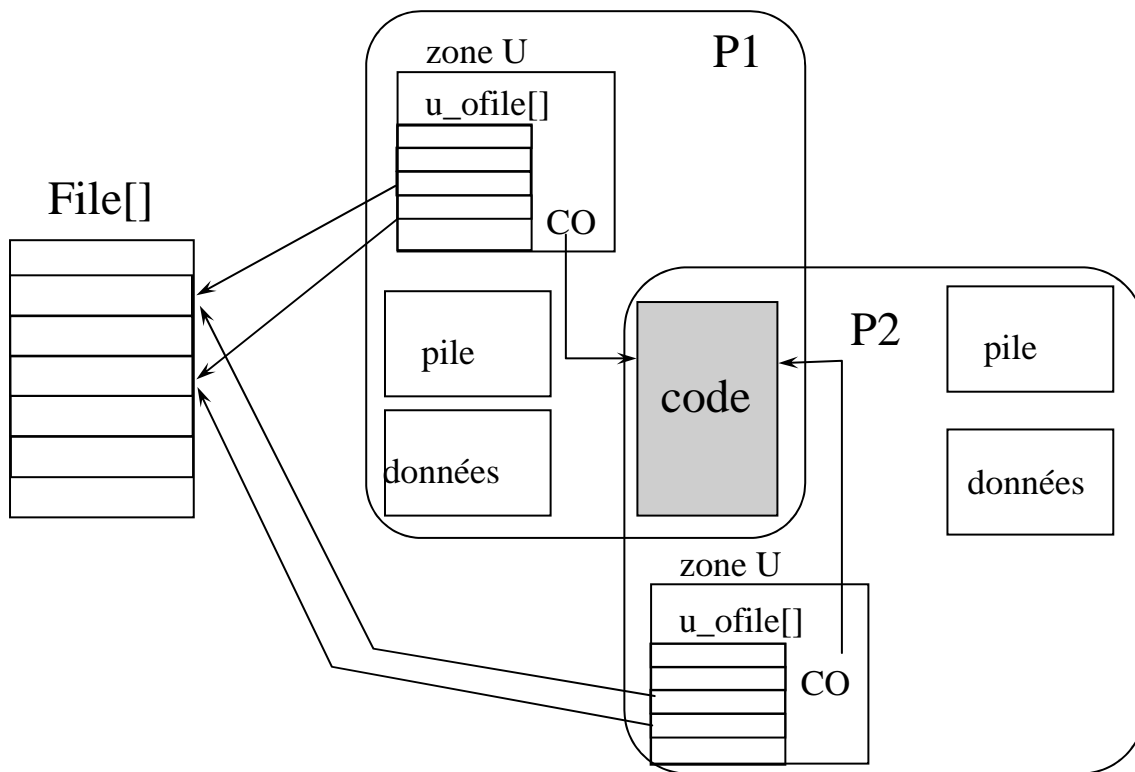
## Algorithme de syscall

- 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

12



## Création (2)



15

## 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)

16

# Exit: 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

17

# 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

18



# 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

19

## Signaux : traitement (1)

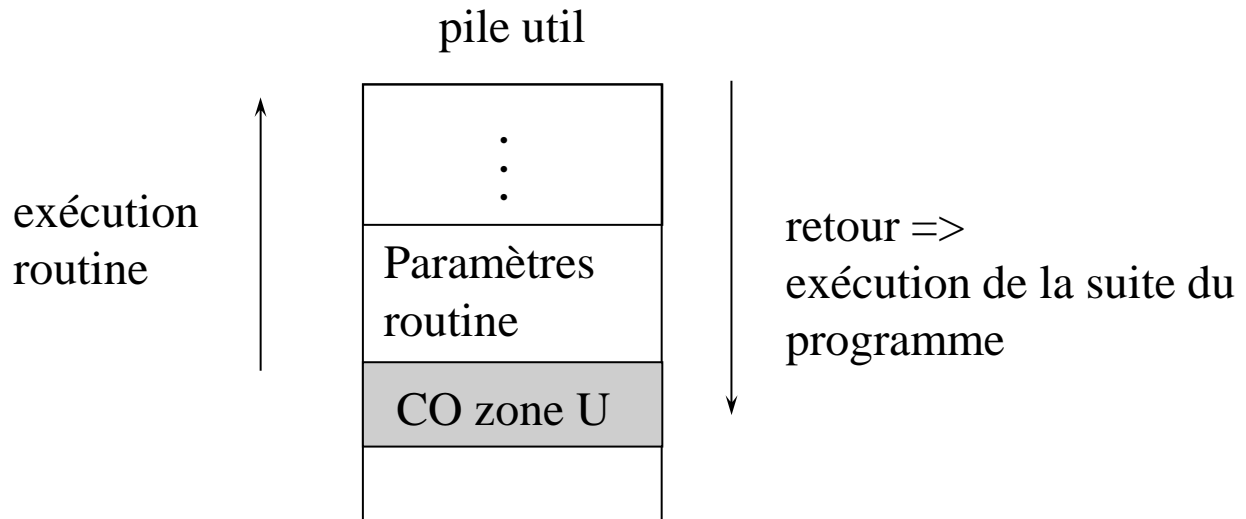
---

- Vérifier la présence de signaux : appel à issig
  - issig est appelée 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.

20

## Signaux : traitement (2)

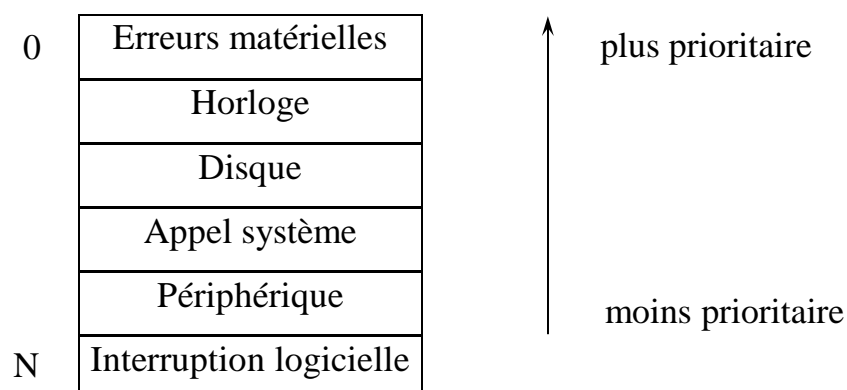
- sendsig : appel dépendant de la machine



21

## Les interruptions

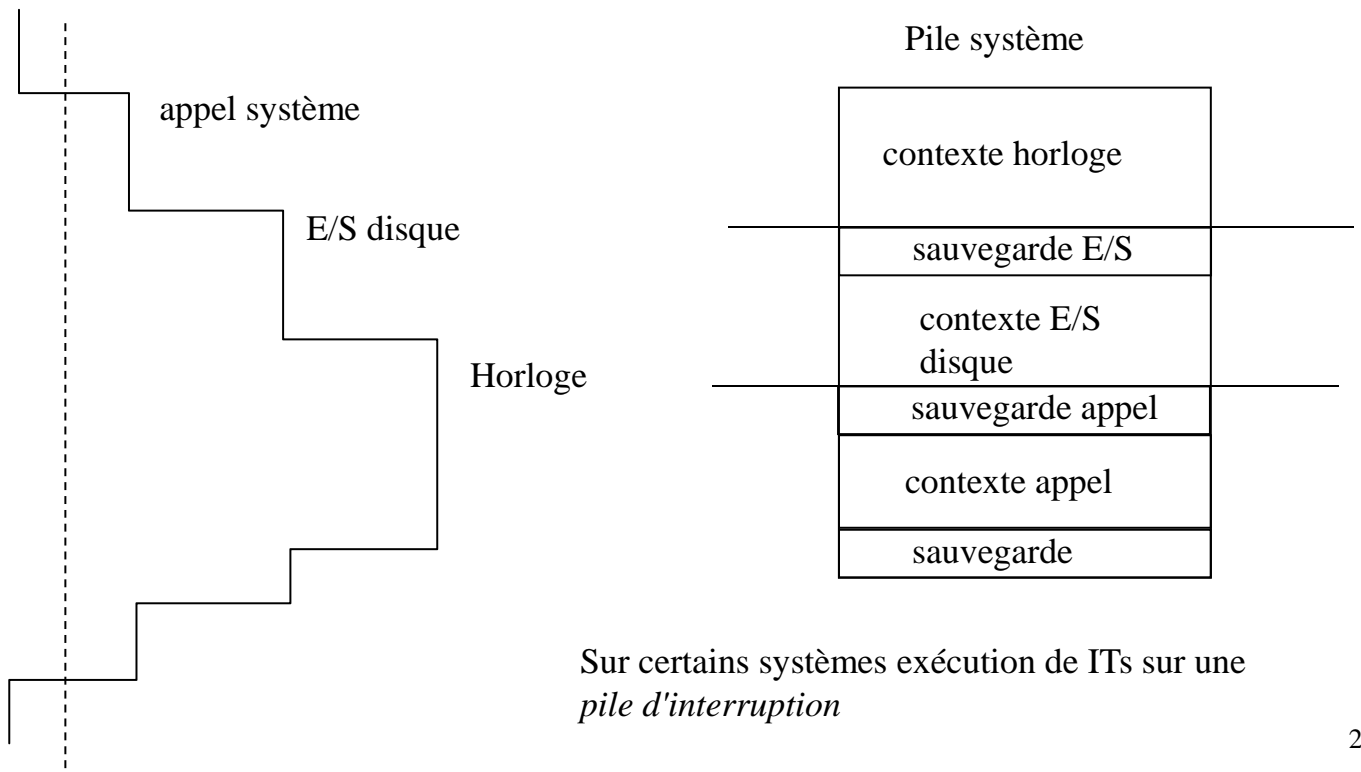
- Pour chaque interruption, un niveau de priorité (ipl: interrupt priority level)
- 7 niveaux Unix de base, 32 niveaux Unix BSD



- ipl stocké dans le mot d'état

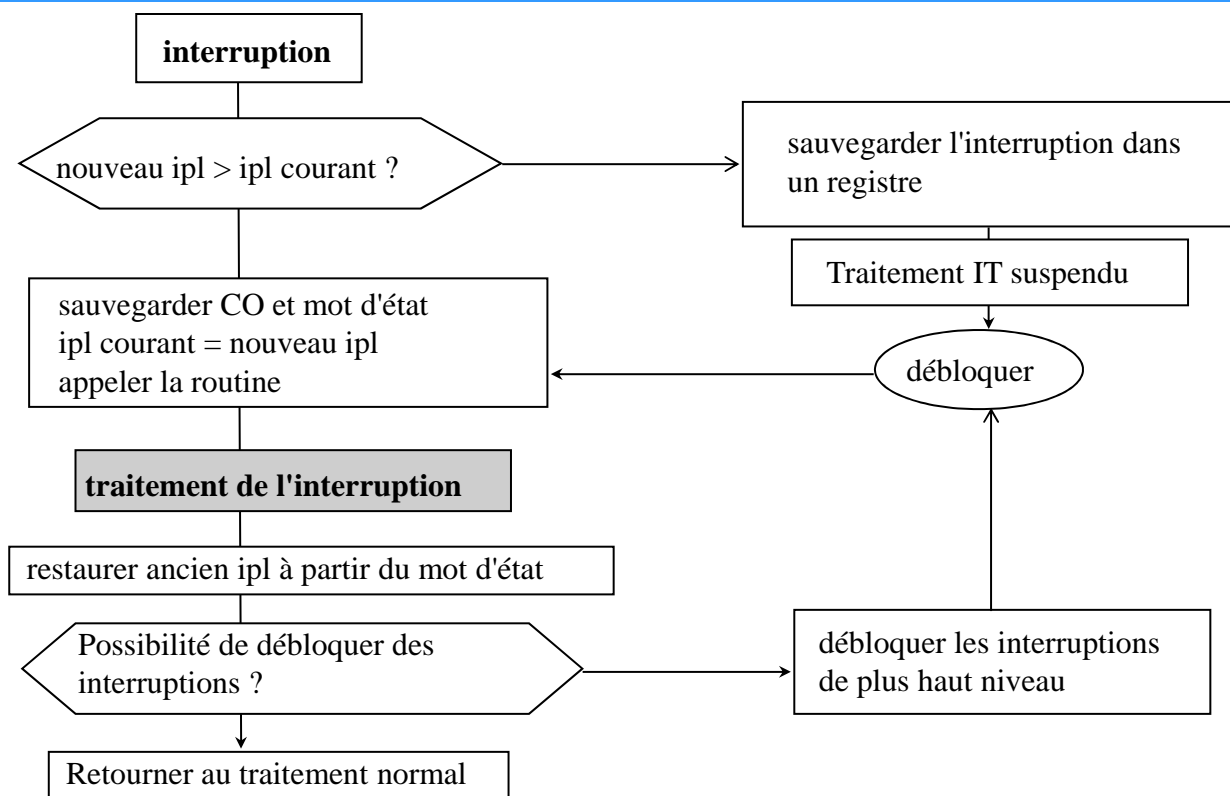
22

# Les niveaux d'exécution



23

# Traitement des interruptions



24

# 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

25

## Section critique

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

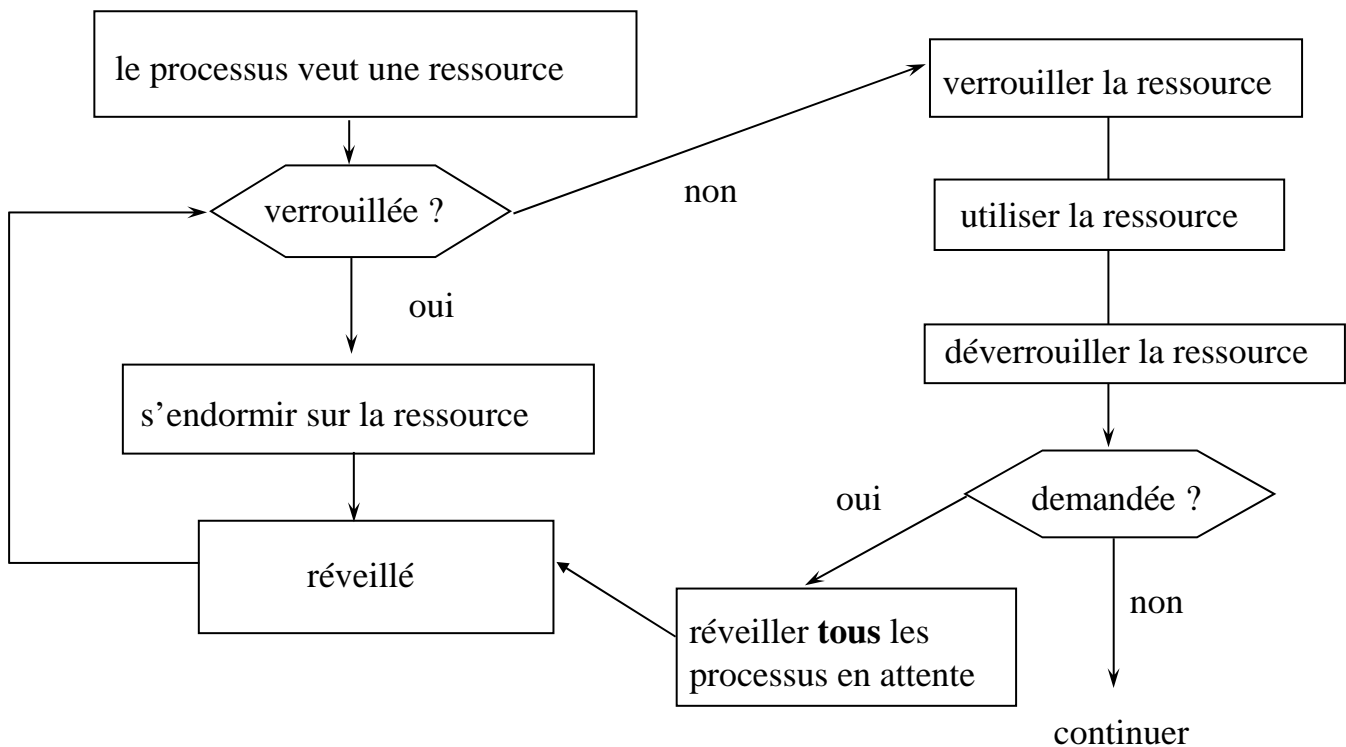
Primitive	Activité bloquée
slp0()	aucune
splsoftclock()	horloge faible priorité
splnet()	protocoles réseaux
spltty()	terminaux
splbio()	disques
splimp()	périphériques réseaux
splclock()	horloge
splhigh()	toutes les interruptions

- Exemple :

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

26

# Verrouillage de ressources



27

## Exemple de code

- Verrouillage

```

/* Attente d'une E/S */
iowait(bp) {
    ps = slpbio();
    while ( !bp->b_flags & B_DONE )
        sleep(bp, PRIBIO);
    slpx(ps);
    ...
}

/* Fin d'E/S */
iodone(bp) {
    ...
    bp->b_flags |= B_DONE ;
    if (bp->b_flags & B_ASYNC)
        brelse(bp) ;
    else wakeup(bp) ;
}
  
```

iodone exécuter ici => sleep inutile !

28

# Primitive sleep

- 2 paramètres :
  - adresse de l'obstacle
  - Priorité en mode noyau (priorité du processus endormi)
- Priorité (4.3BSD):

– PSWP	Swapper	↑ Non interruptibles par des signaux ↓
– PMEM	Démon de pagination	
– PINOD	Attente d'une inode	
– PRIBIO	Attente E/S disque	
– PZERO	Seuil	
– PPIPE	Attente sur tube (plein ou vide)	↑ Interruptibles par des signaux ↓
– TTIPRI	Attente entrée sur un terminal	
– TTOPRI	Attente écriture sur un terminal	
– PWAIT	Attente d'un fils	
– PSLEP	Attente d'un signal	

29

## 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)

30

# 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

31

# 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)

32

# 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....

33

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

34



# 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)

35

## 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
  - Décrémenté régulièrement, 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 certain linux 2.6

36

# 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

37

## Routine de traitement de IT horloge

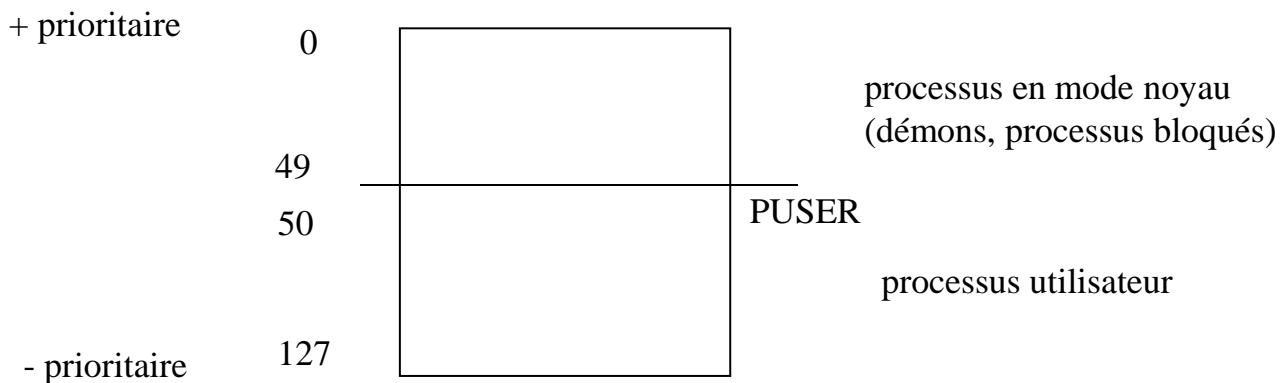
---

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

38

# 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



39

## 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 :
$$p\_usrpri = PUSER + p\_cpu/4 + 2*p\_nice$$
$$p\_cpu = p\_cpu * decay$$
$$decay = 1/2$$
$$decay = (2 * load) / (2*load + 1)$$

System V Release 3BSD
- Un processus qui a eu un accès récent => p\_cpu élevé => p\_usrpri élevé.

40

# 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 un nouveau processus

41

## Exemple - System V Release 3

---

- Quantum = 60 tics, PUSER = 60
- Fin Quantum :  $p\_cpu = p\_cpu / 2$ ;  $p\_pri = PUSER + p\_cpu/2$

42

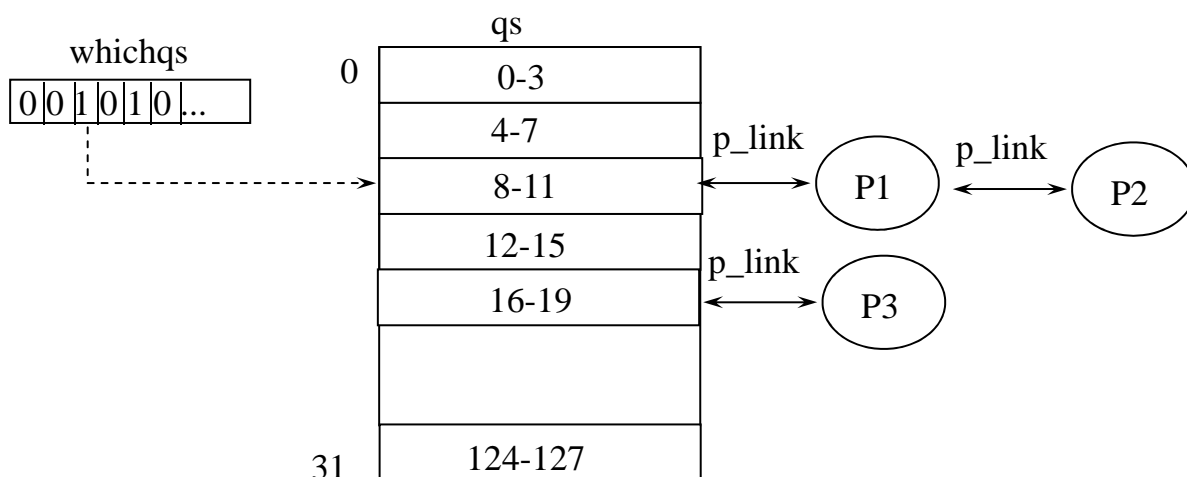
# Priorité des processus bloqués

- Les processus sont bloqués avec une haute priorité  $\neq$  priorité utilisateur ( $p\_pri \neq p\_usrpri$ )  
=> 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 ( $p\_pri = p\_usrpri$ )

43

## Implémentation

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



44

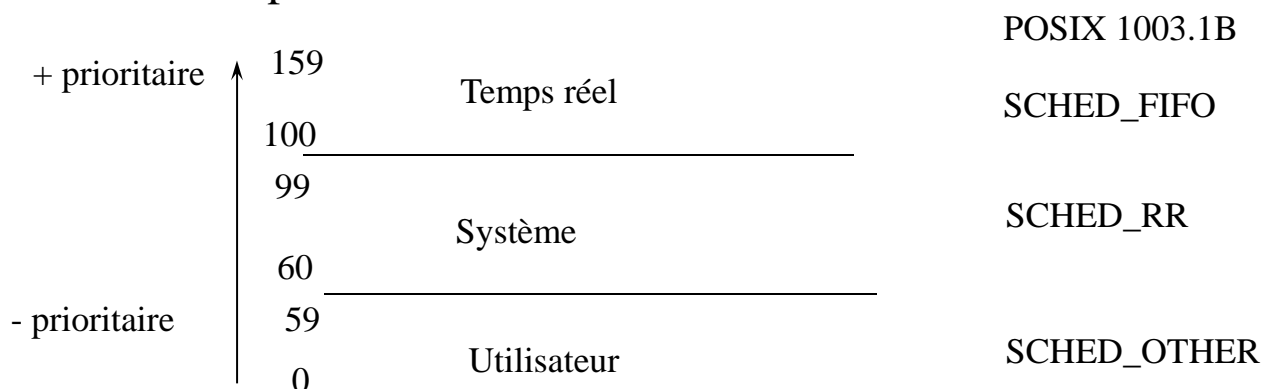
# Algorithme de switch

- 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

45

## 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`)

46

# 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) ...

47

## Classe "temps partagé"

---

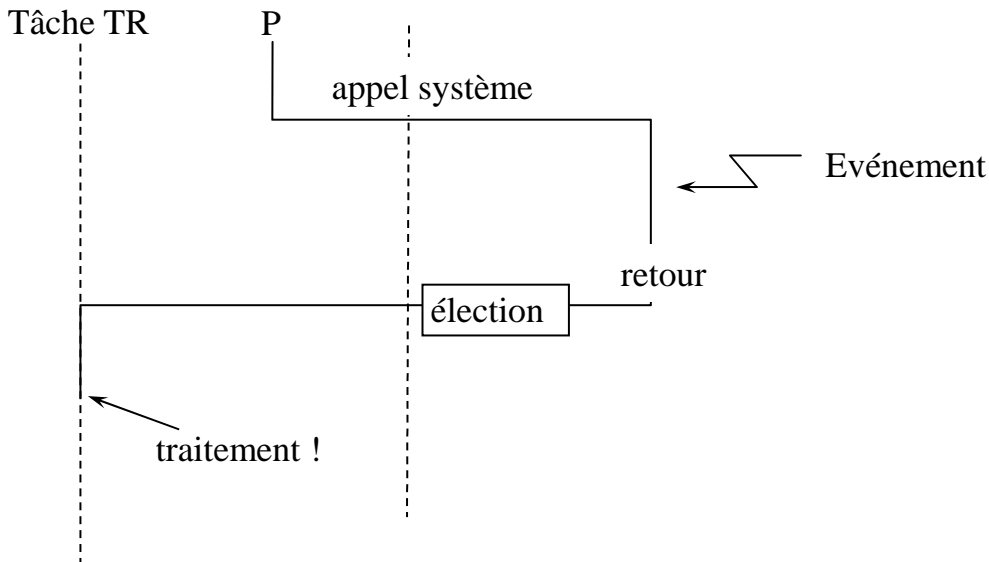
- Quantum variable d'un processus à l'autre
- inversement proportionnel à la priorité !
- Définis statiquement pour chaque niveau de priorités

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

48

# Classe "temps réel"

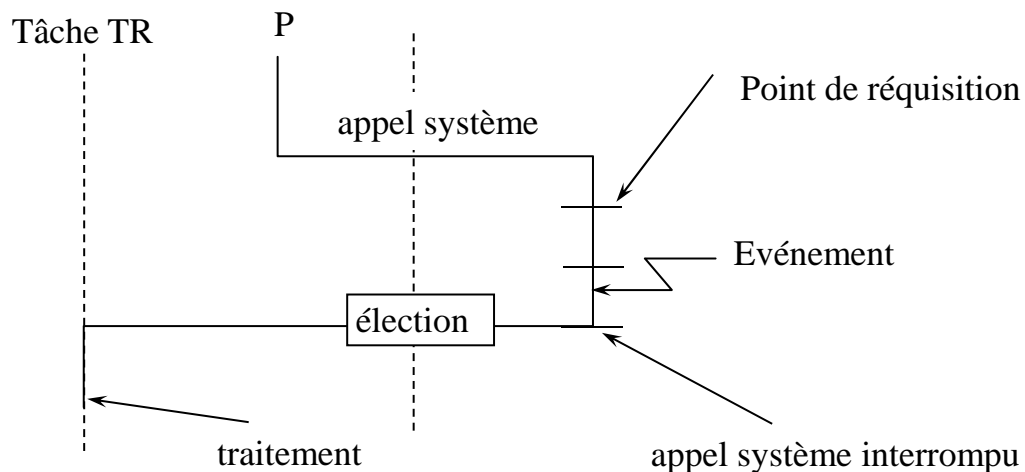
- Objectif : satisfaire des contraintes de temps
  - Processus temps réel très prioritaire en attente d'événement
- Impossible dans la plupart des Unix car noyau non-préemptif !



49

## 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

50



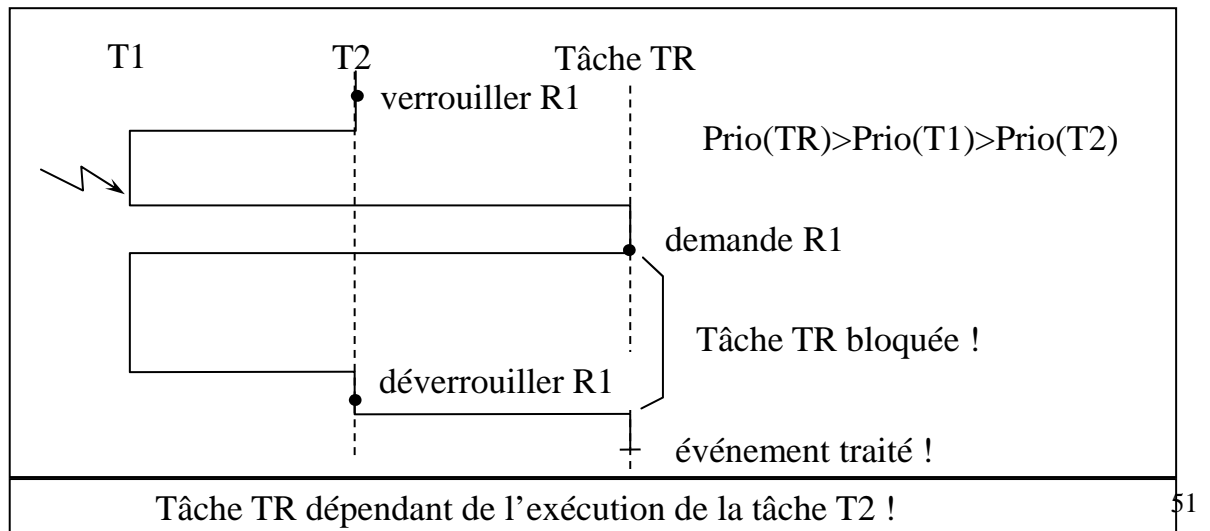
# Noyaux Préemptifs (Solaris 2.x)

- 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)

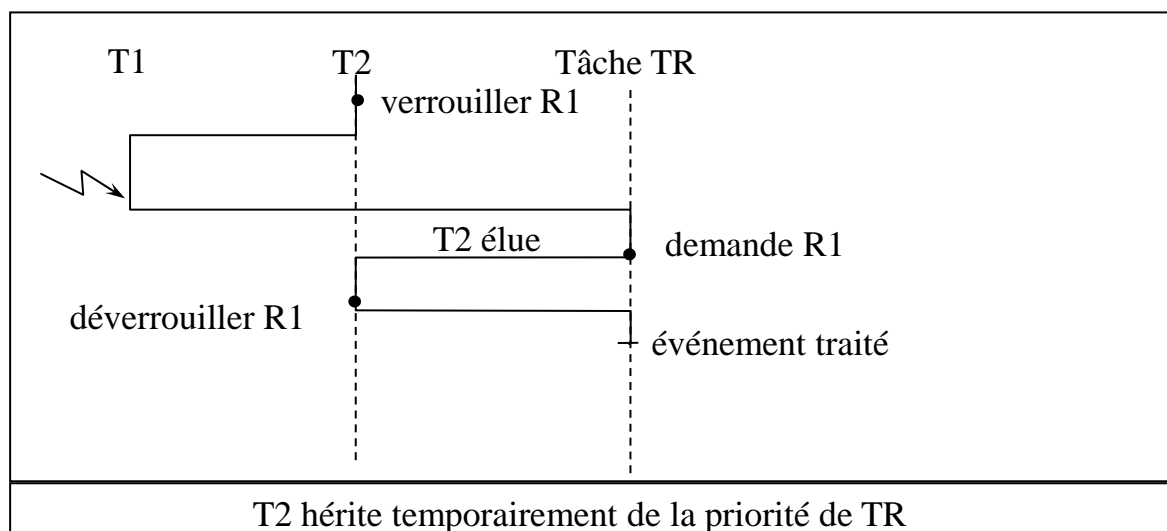
**Inversion de  
priorité**



51

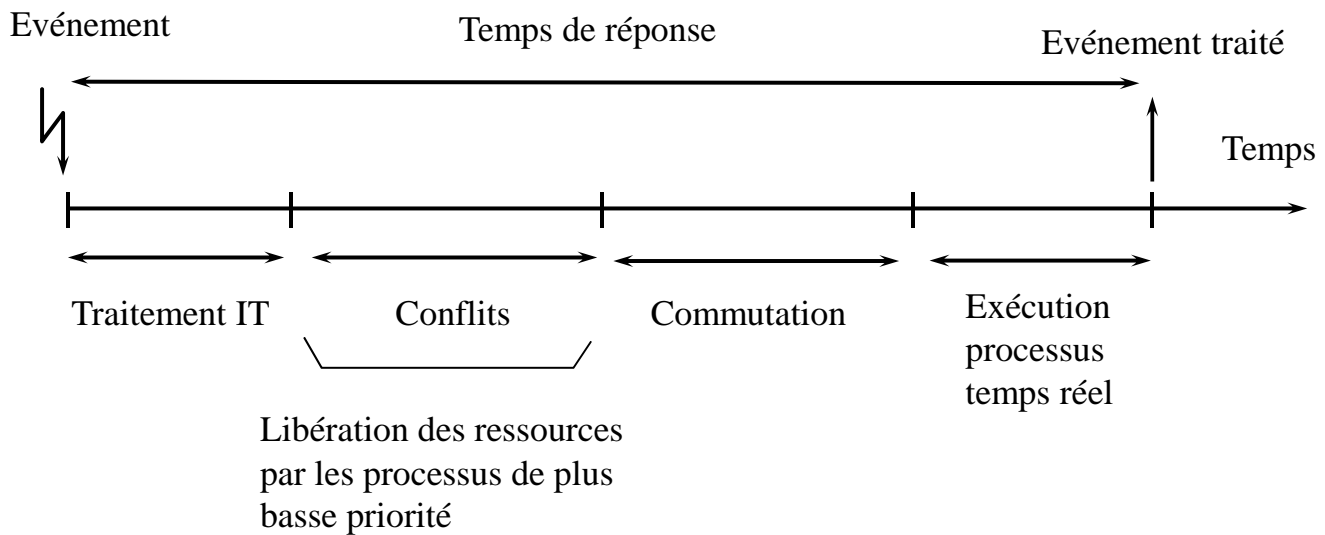
## Héritage de priorité

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



52

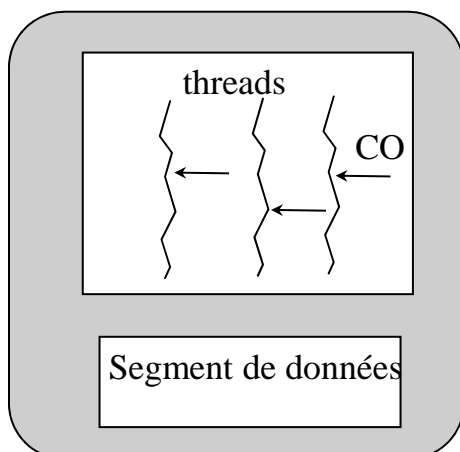
# Temps de réponse



53

## 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

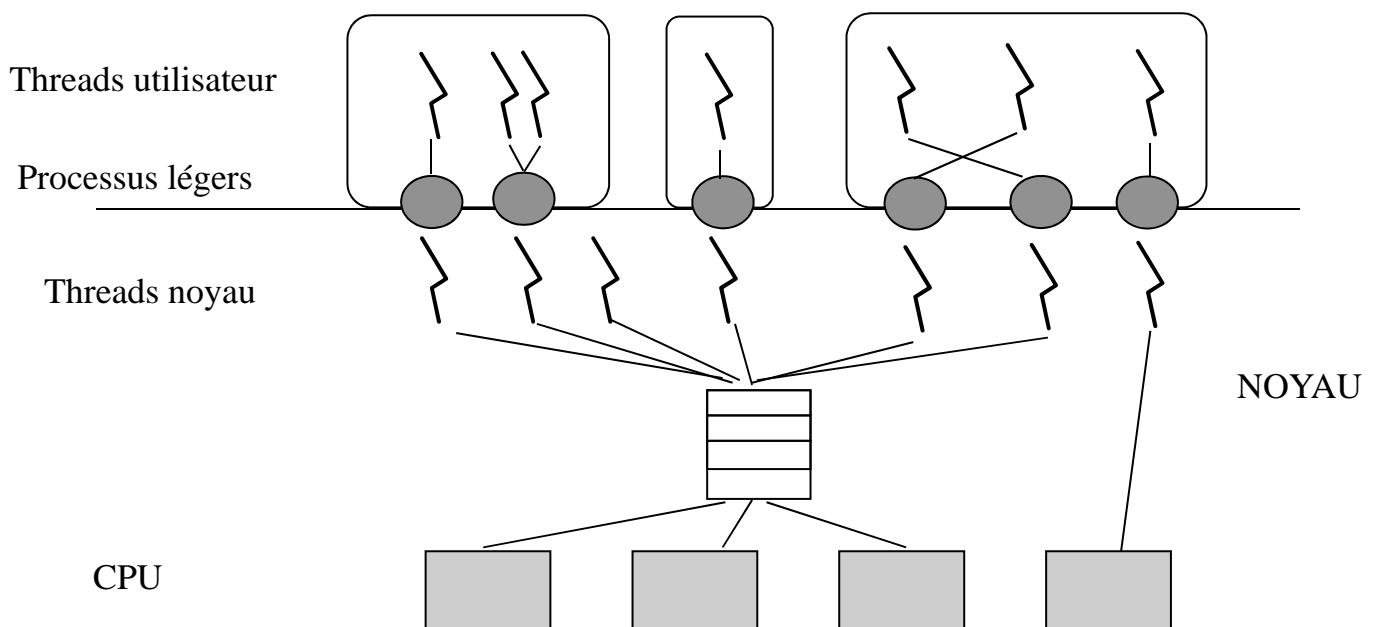
54

# 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

55

## Exemple Solaris 2.x



56

# Comparaison

---

	<b>Temps de création (microsecondes)</b>	<b>Temps de synchronisation en utilisant des sémaphores (microsecondes)</b>
Thread utilisateur	52	66
Processus léger	350	390
Processus	1700	200

Solaris sur Sparc2

57

## Gestion des processus dans LINUX

---

- Structures
- Ordonnancement
- Nouveautés depuis 2.6

58

# Structure de données : struct task

---

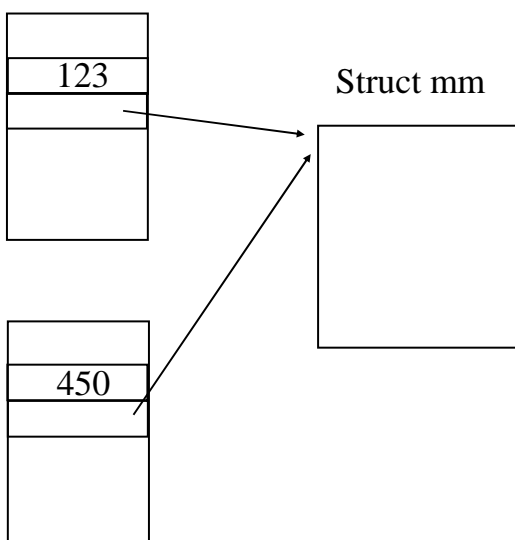
- 1 table des processus de type struct task (équivalent à proc + user)
- 1 entrée par processus
- Première entrée réservée au processus **init**
- **Task\_struct** :
  - **policy** (SCHED\_OTHER, SCHED\_FIFO, SCHED\_RR) : stratégies d'ordonnancement
  - **state** : running, waiting, stopped, zombie
  - **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
  - ...

59

## Threads

---

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



60

# Processus Linux ( $\leq 2.4$ )

---

- **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

61

## Stratégies d'ordonnancement ( $\leq 2.4$ )

---

- 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

62

# 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

63

## Fonction schedule()

---

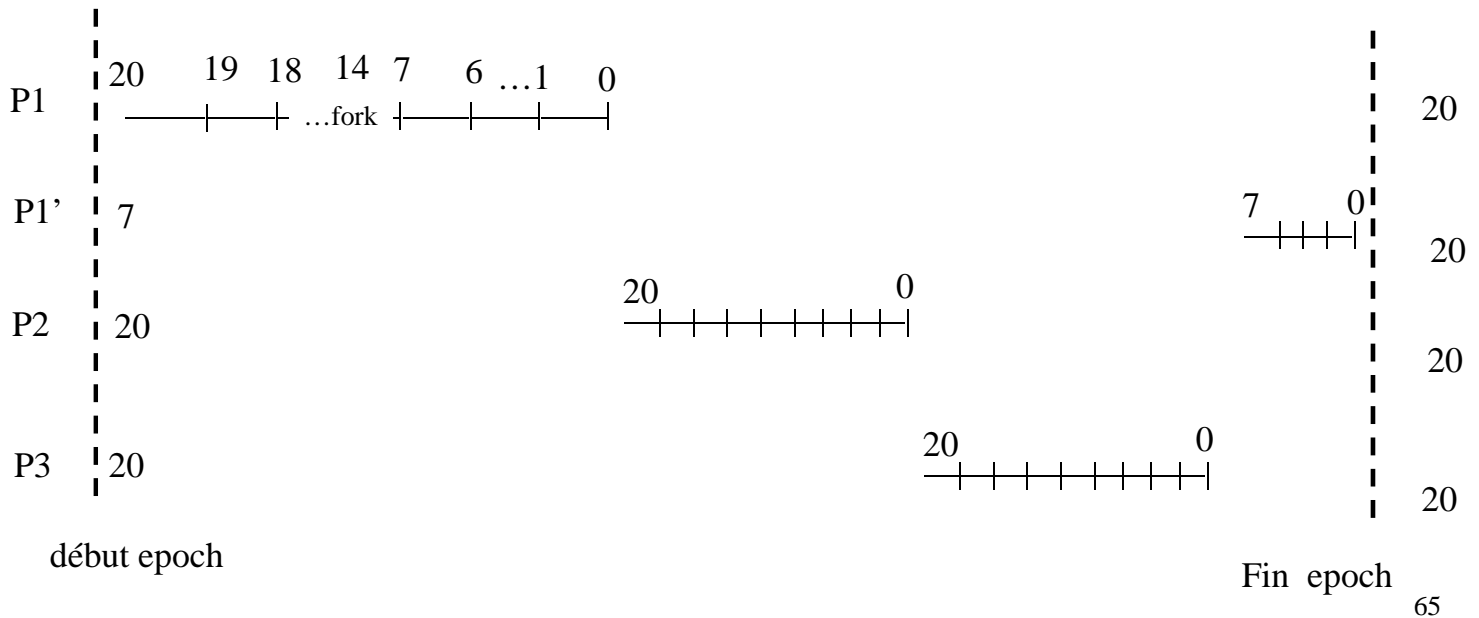
- Implémente l'ordonnancement
  - Invoquée directement en cas de blocage
  - Invoquée « paresseusement » au retour en mode U
- Schedule
  1. Choisir le meilleur candidat : celui ayant le poids le plus élevé (fonction goodness) :

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 :
      - $p \rightarrow \text{counter} = (p \rightarrow \text{counter} \gg 1) + p \rightarrow \text{priority}$
      - Rem : la priorité des processus en attente augmente

64

# 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 précédente 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

67

# Nouveautés Linux 2.6

---

- Ordonnancement
  
- Noyau préemptif

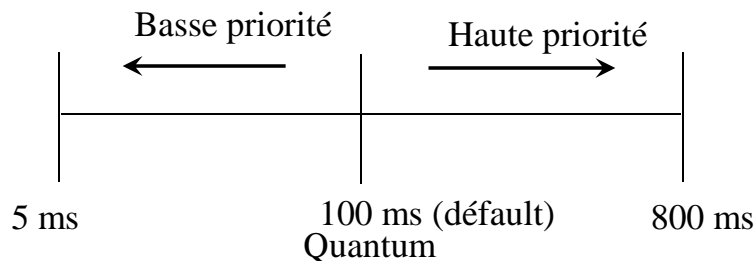
68

## Ordonnancement 2.6.X

- Objectif : diminuer les temps de réponses et une gestion plus fine des temporisateurs pour applications 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 quantum variables



69

## Algorithme d'ordonnancement 2.6 : Priorité

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

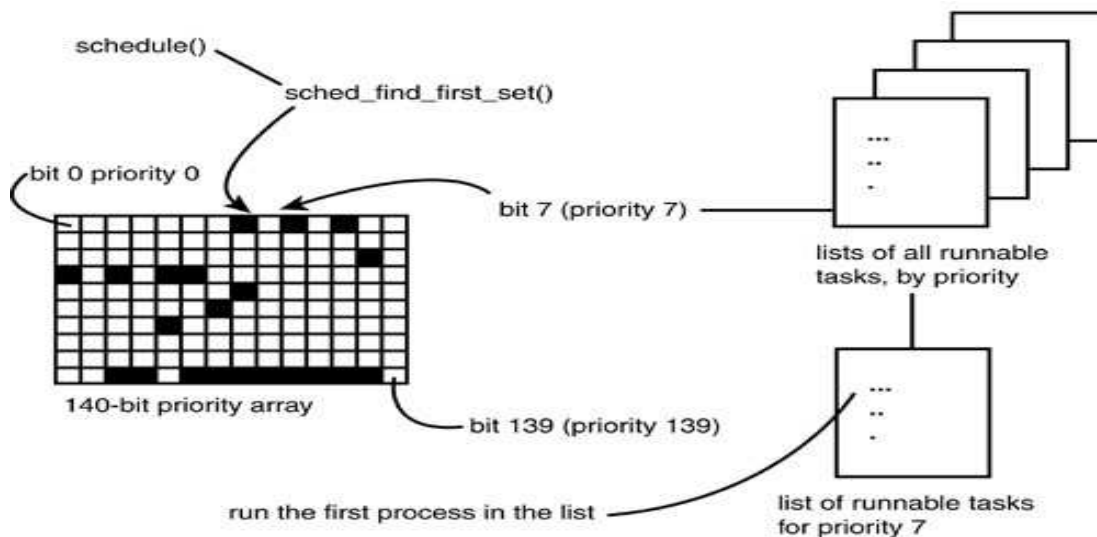
Bonus max = -5, Penalité max= +5

- 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'exécution : sleep-avg-- à chaque tic
- Fonction effective\_prio() : correspondance entre sleep\_avg et bonus [-5,+5]

70

# Algorithme de choix du processus

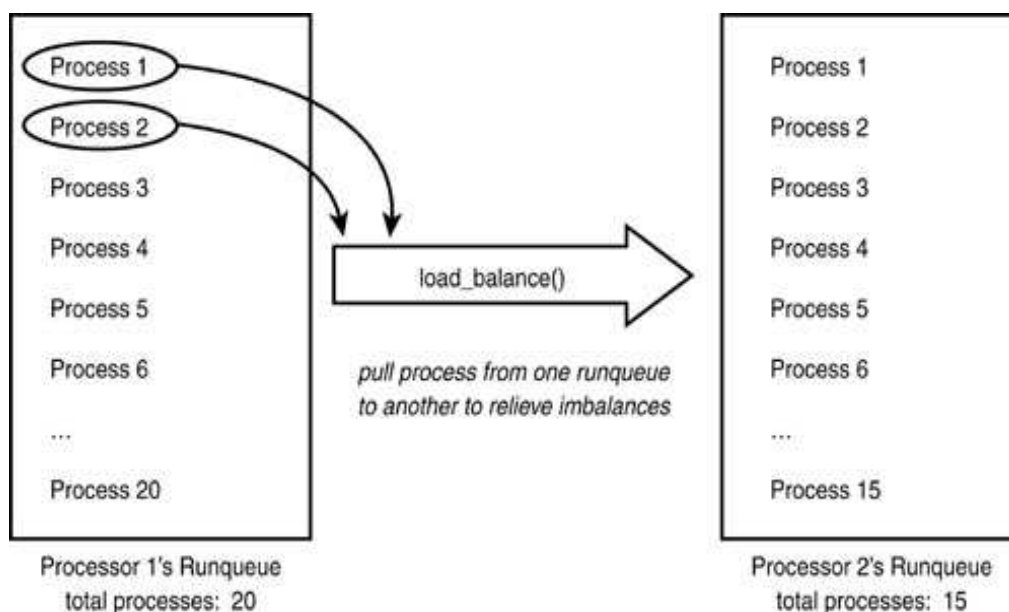
- Algorithme en  $O(1)$
- 140 niveaux de priorité, 1 file par niveau



71

## Equilibrage de charge

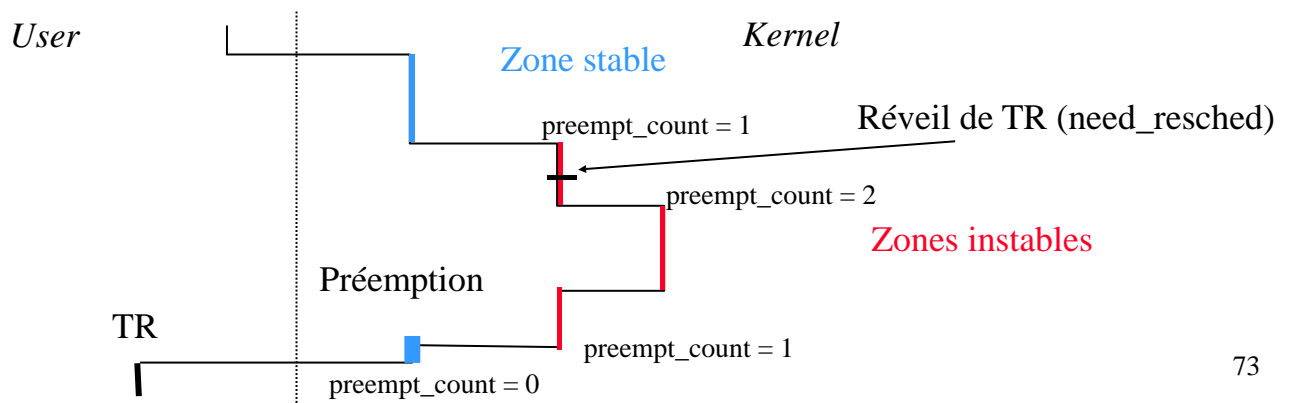
- Fonction `load_balance()` appelée par `schedule()` lorsqu'une file est vide ou périodiquement (toutes les ms si aucune tâche, toutes les 200 ms sinon)



72

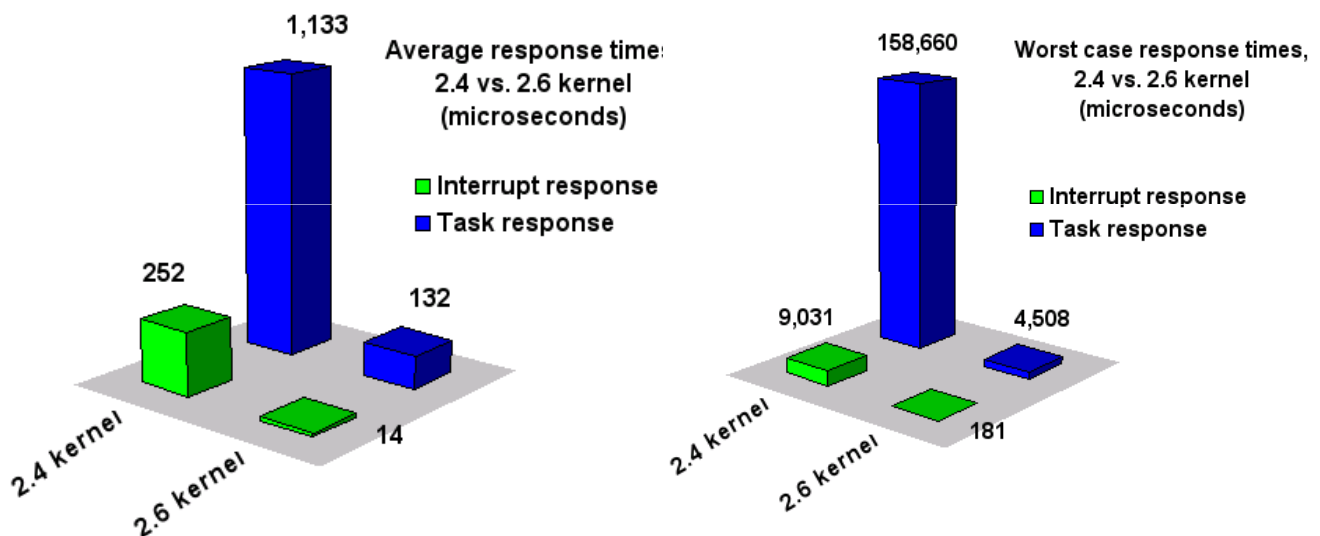
# Noyau Préemptif – Linux 2.6.x

- 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 → Prémption



73

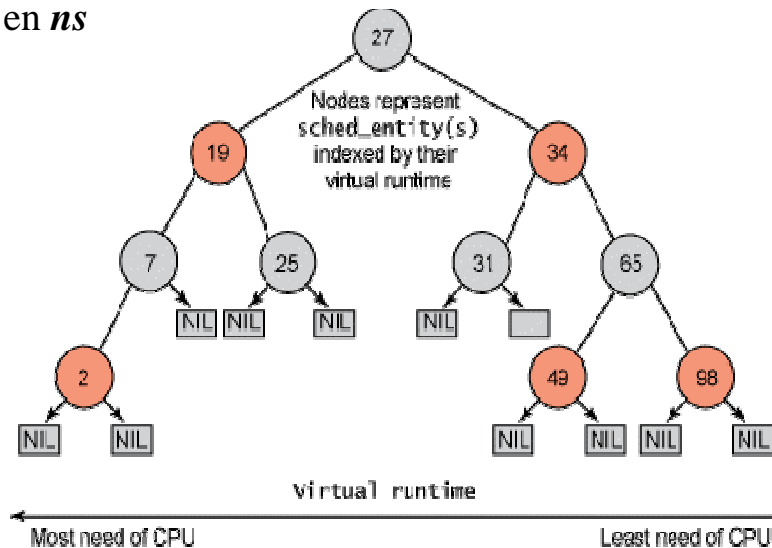
## Linux 2.4 vs. 2.6



74

# Linux 3.x - “Completely Fair Scheduler”

- Depuis Linux 2.6.23 un nouvel ordonnanceur : CFS
- Tâches prêtes stockées dans un arbre rouge-noire (~équilibré) => Temps de mise à jour en  $O(\log n)$
- Tâches ordonnées en fonction de leur temps virtuel d'exécution (*virtual runtime*) exprimé en *ns*



<http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>

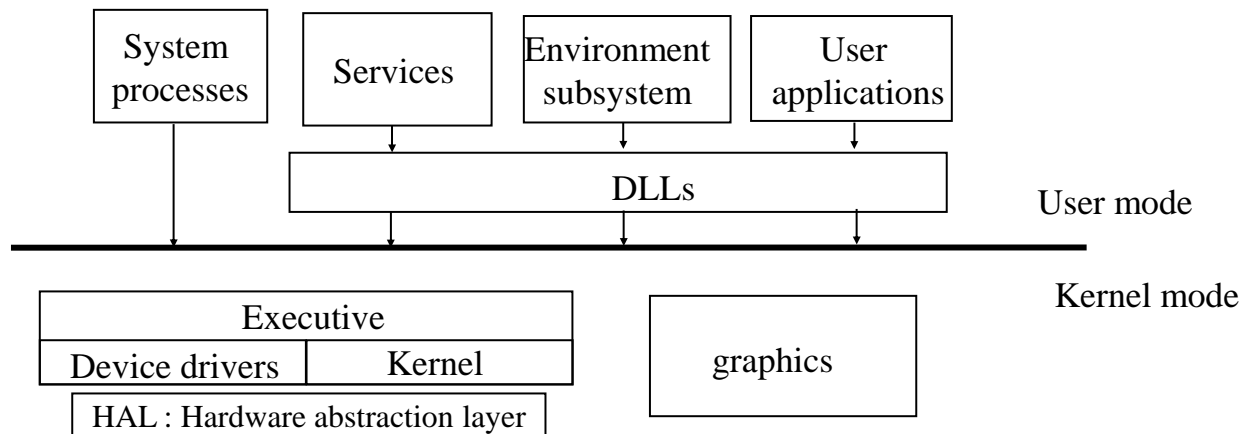
75

## CFS

- A l'invocation de l'ordonnanceur :
  - Choisir la tâche ayant le plus petit virtual runtime : (la tâche la plus à gauche dans l'arbre) => Tâches courtes ou qui ont fait des E/S sont avantagées
  - Calculer le temps d'exécution maximum pour le processus élu (fonction du nombre de tâches et de son temps d'attente)
  - Si le processus a atteint son maximum : mise à jour de son virtual runtime et réinsertion dans l'arbre, choisir un nouveau processus ....
- Intégration des priorités :
  - Augmentation de la priorité => diminution d'un certain facteur (decay) du virtual runtime
  - Virtual runtime est relatif

76

# Gestion de processus dans Windows NT

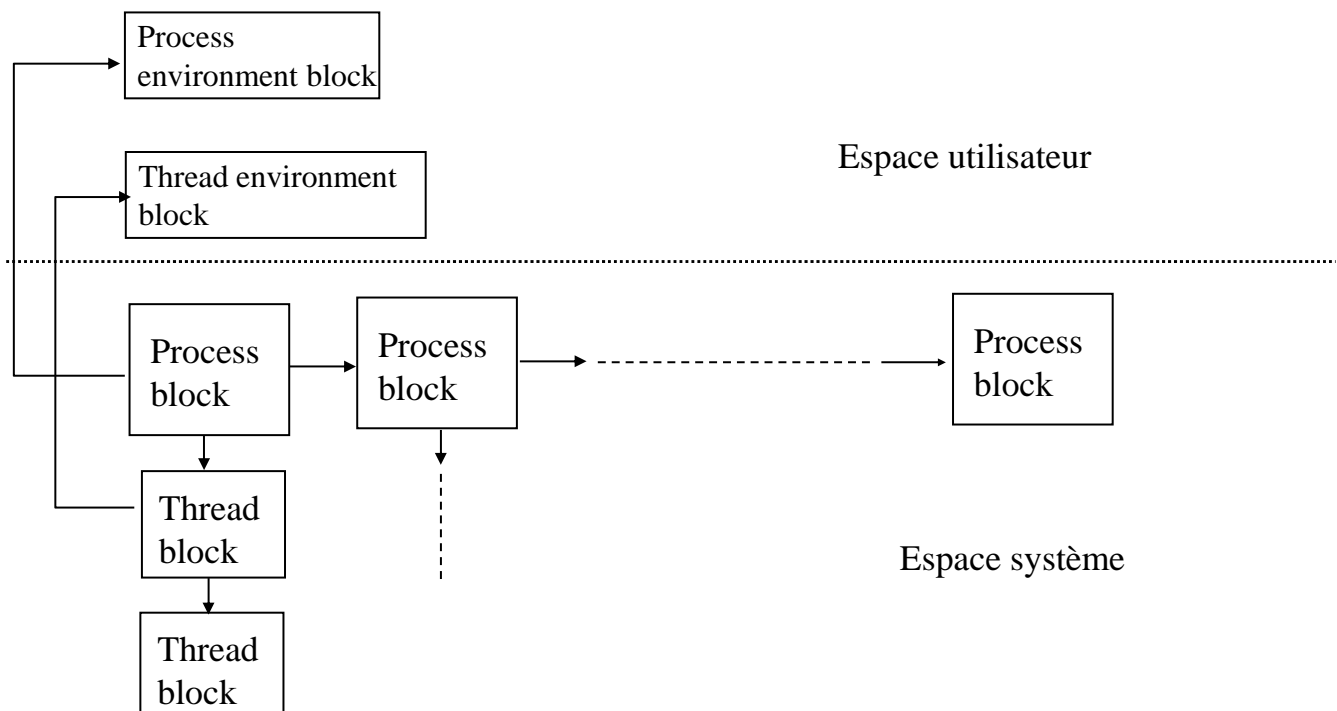


## 1. Processus et Threads

## 2. Ordonnancement

77

# Les structures



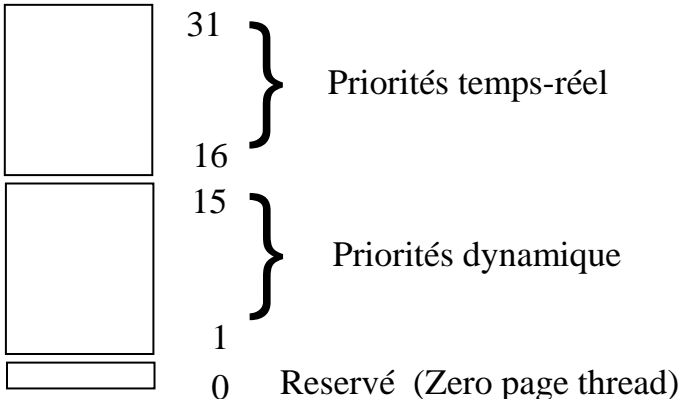
78

# 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)

79

## Ordonnancement

- Priorités
- 
- The diagram illustrates the priority ranges for thread scheduling. It consists of three stacked rectangular boxes. The top box is labeled with '31' at its top right corner. The middle box is labeled with '16' at its top right corner and '15' at its bottom right corner. The bottom box is labeled with '1' at its top right corner and '0' at its bottom right corner. To the right of the top box, a curly bracket groups the range from 31 down to 16, labeled 'Priorités temps-réel'. To the right of the middle box, a curly bracket groups the range from 15 down to 1, labeled 'Priorités dynamique'. To the right of the bottom box, the value '0' is labeled 'Reservé (Zero page thread)'.

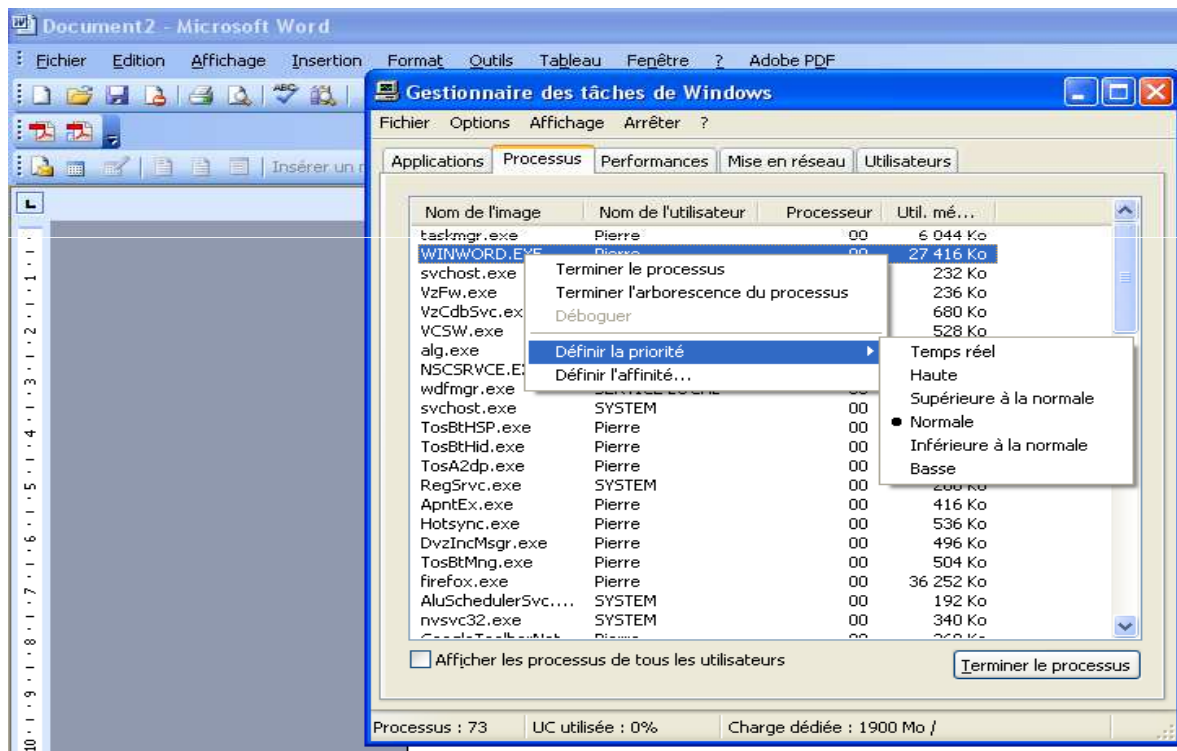
Ordonnancement temps partagé (quantum)

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

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

80

# Classes de priorité



81

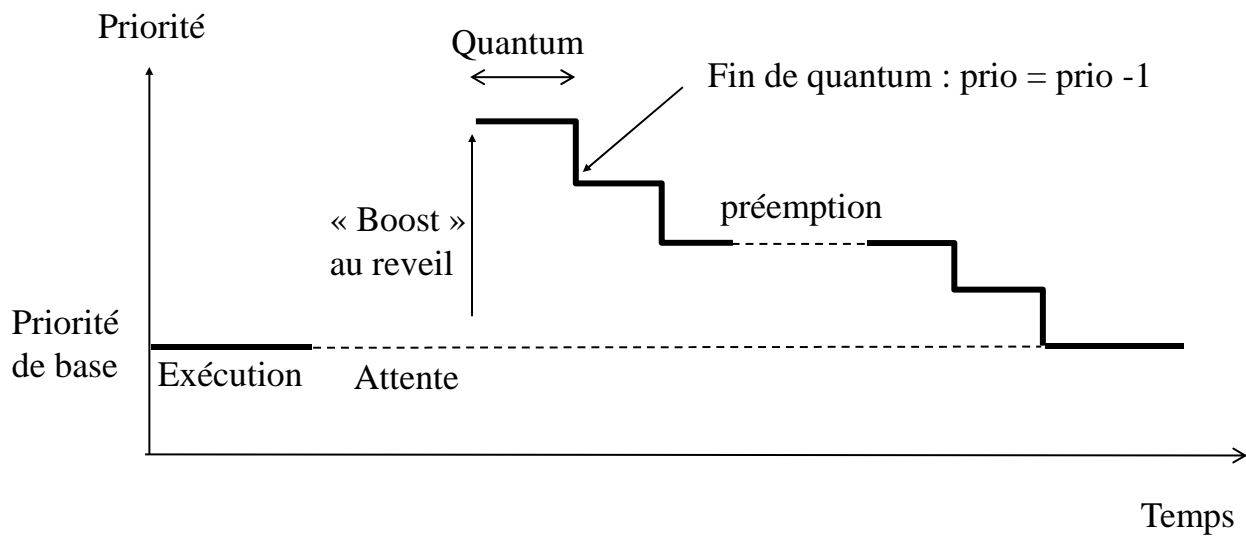
## Ajustement des priorités et du quantum

- 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

82



# Exemple



83

## Ordonnancement SMP

- Définition d'**affinités** pour chaque thread : liste des CPU sur lesquels peuvent 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

84