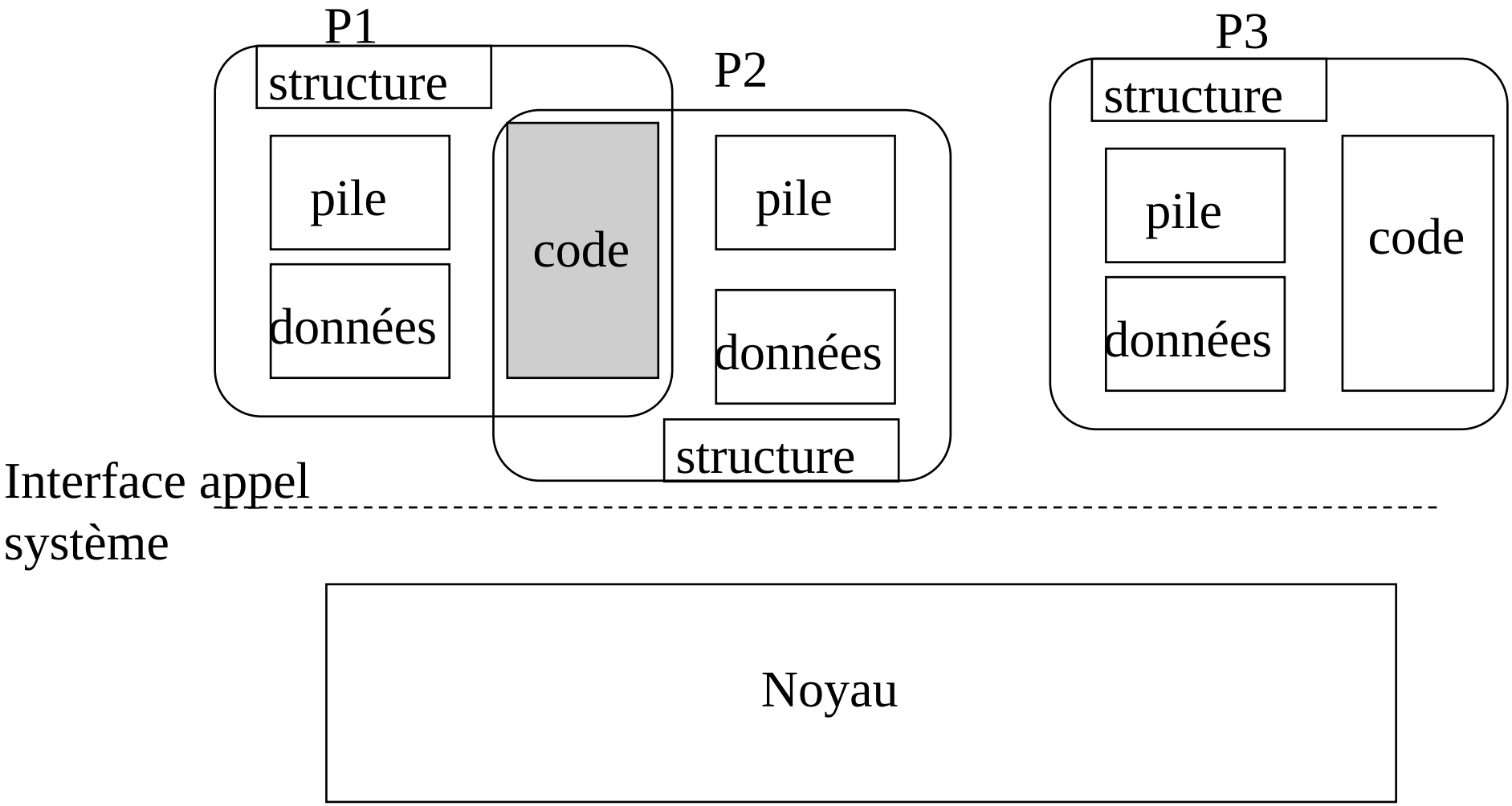
Gestion de processus

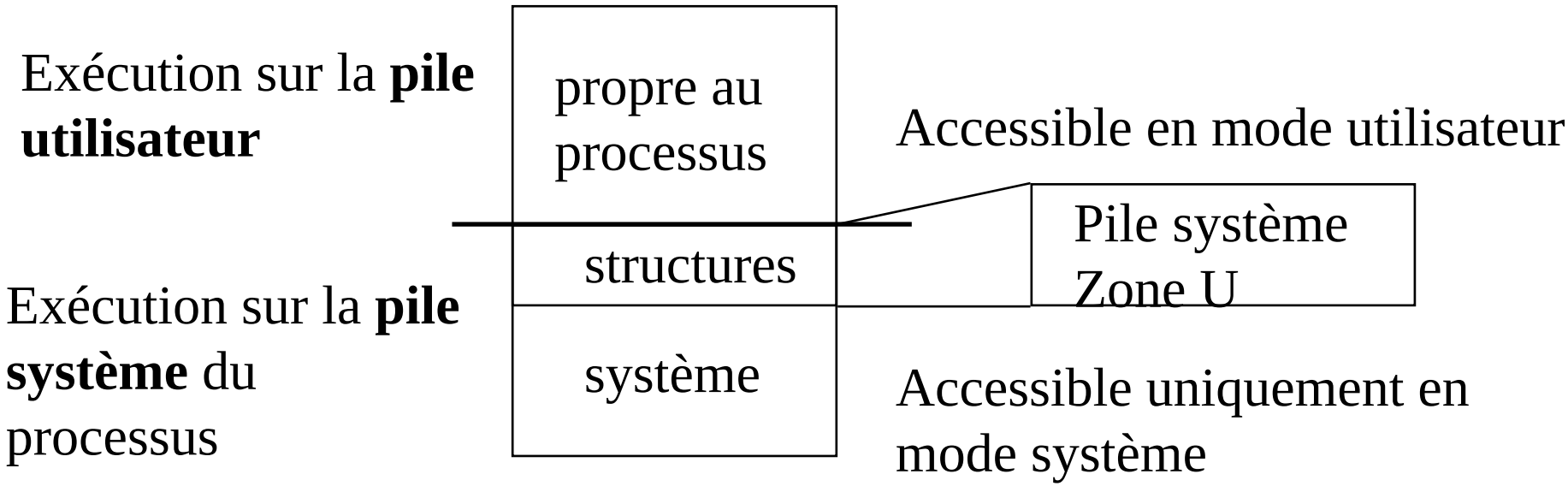
1. Architecture - Mode d’exécution- Etats
2. Création/terminaison
3. Signaux
4. Les processus du système/ Initialisation du système
5. Ordonnancement
6. Processus légers – threads
7. Linux
8. Windows

Architecture



Mode d’exécution

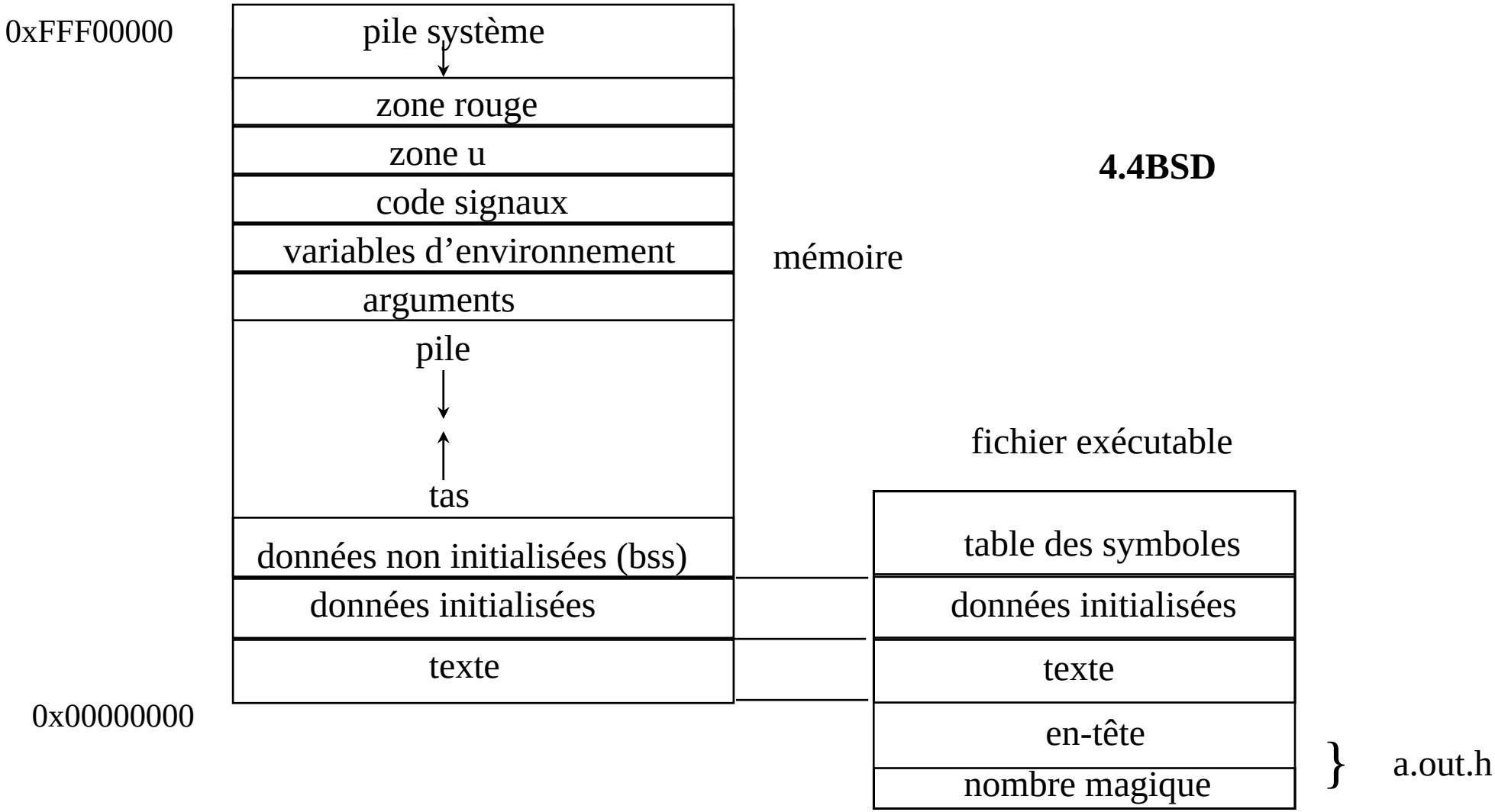
- Deux modes : utilisateurs / systèmes
  - OS/2 3 niveaux, Multics 7
- => 2 zones de mémoire virtuelle :
  - le noyau fait partie de l’espace virtuel du processus courant !



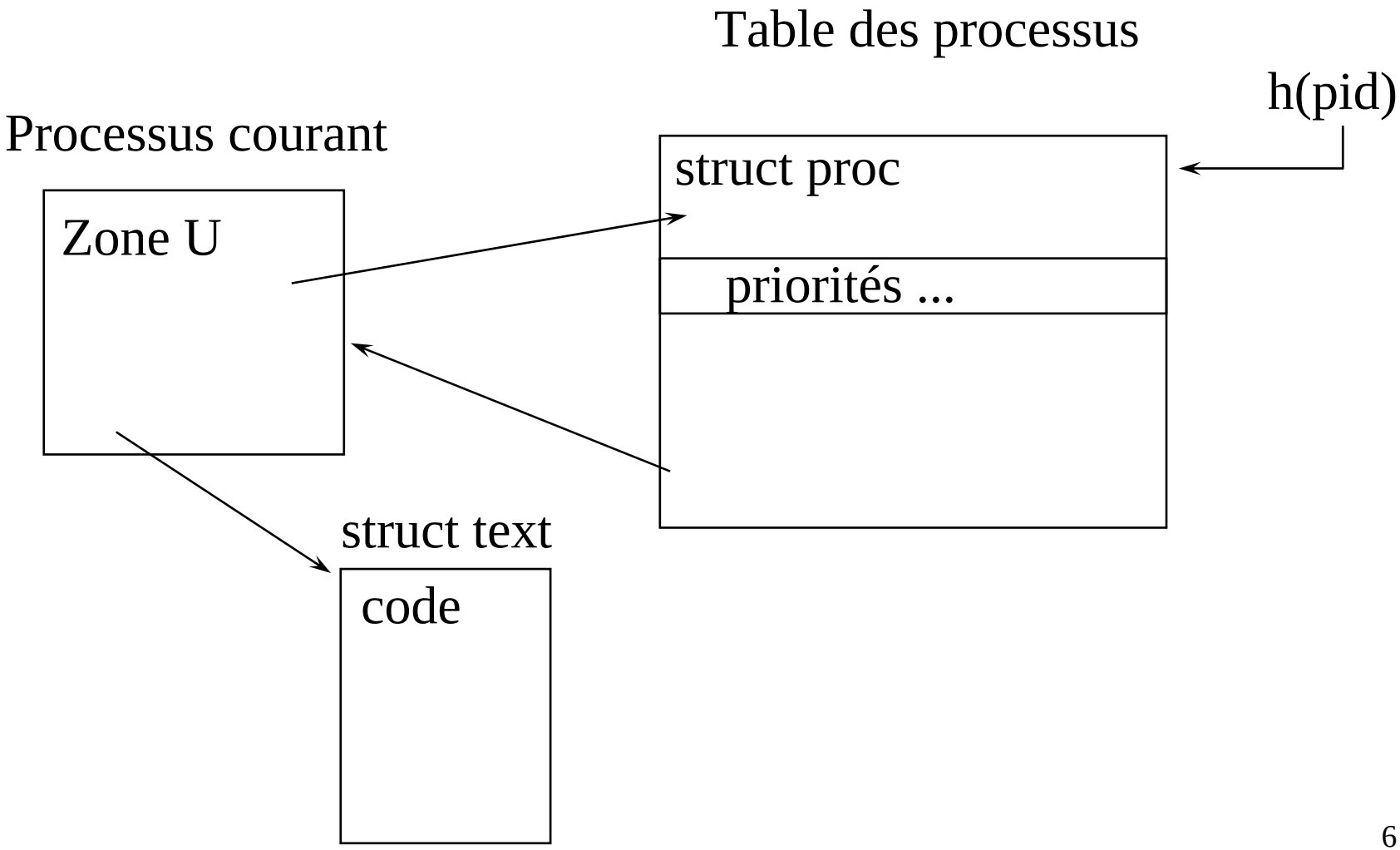
Structure interne

- 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



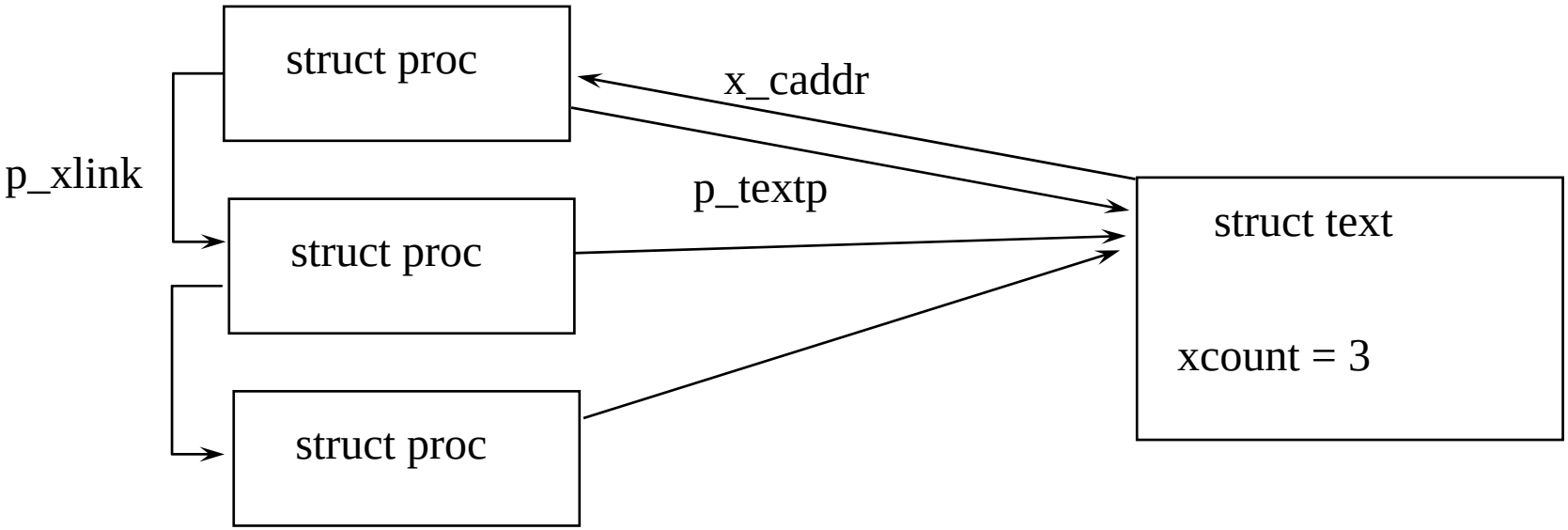
Structure - Zone U

- 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]

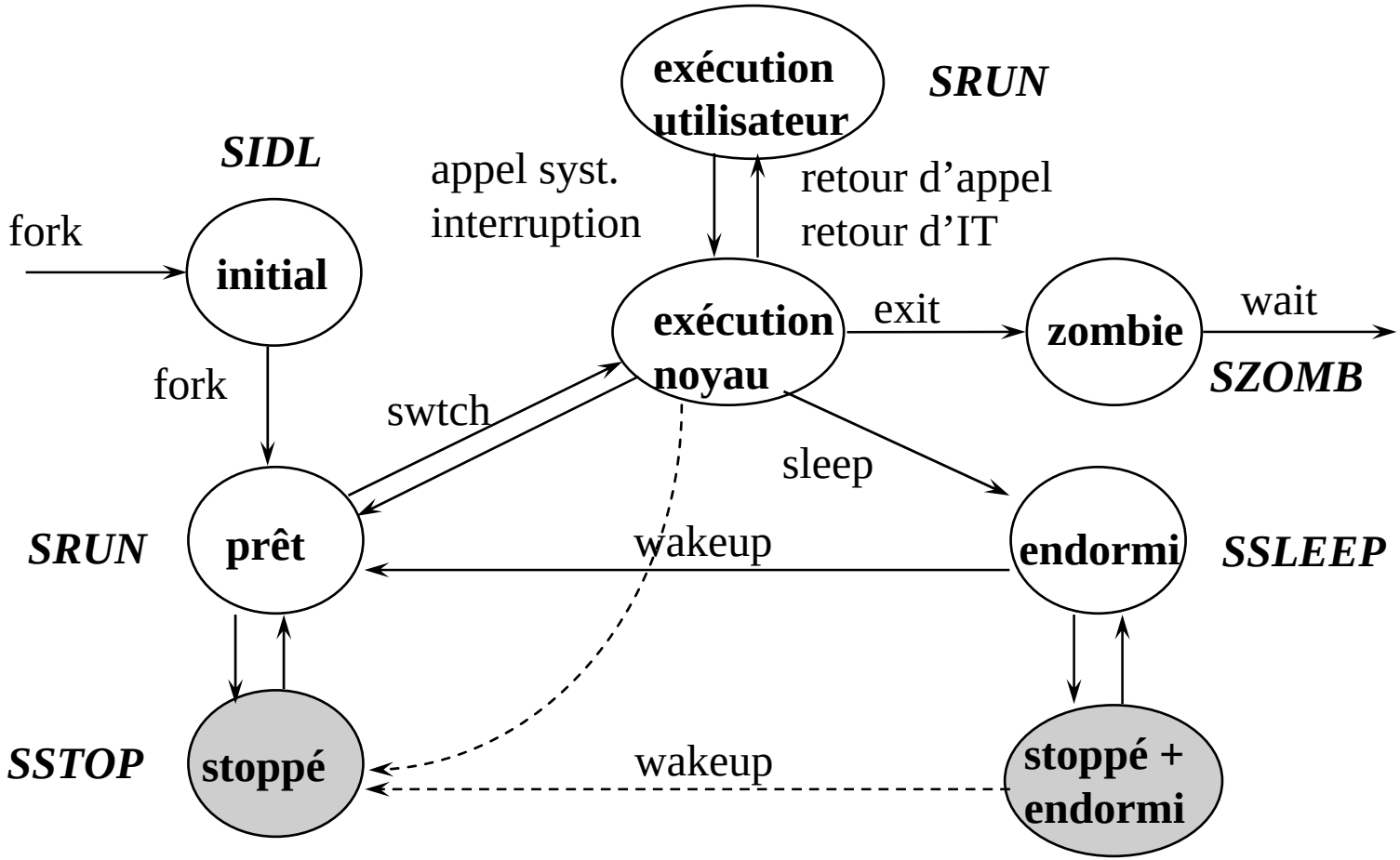
Structure résidente

- 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



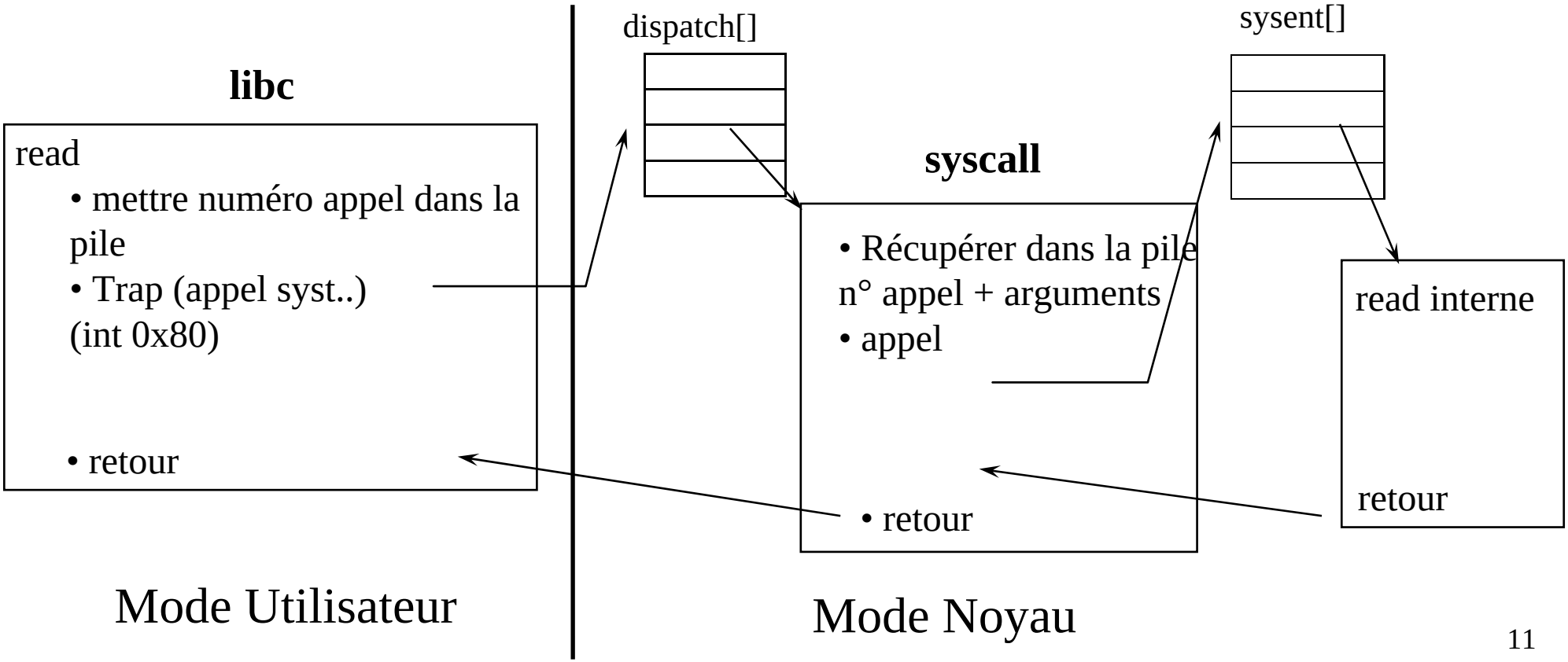
Etat d'un processus



BSD

Interface des appels systèmes

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



Algorithme de syscall

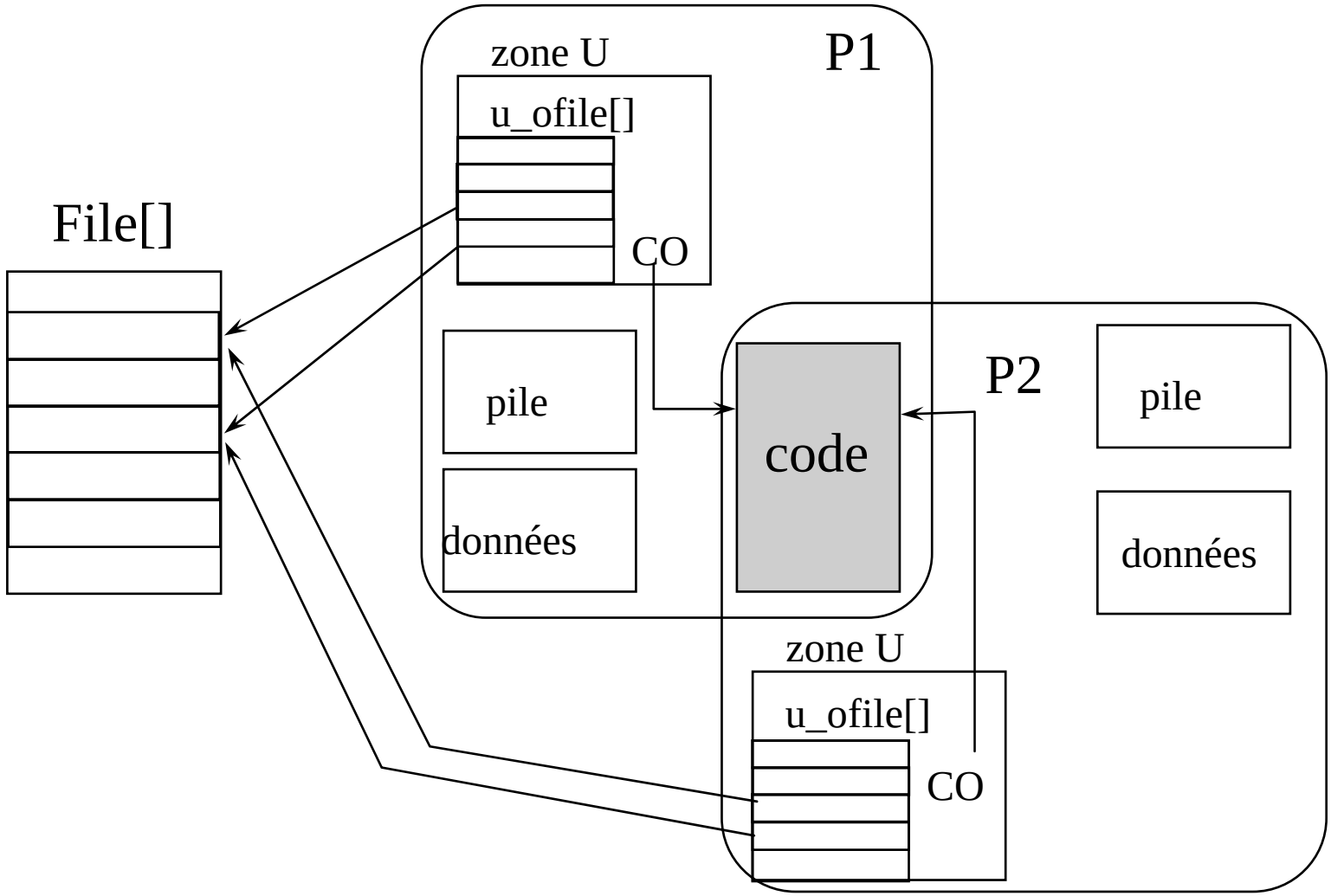
- Trouver les paramètres dans la pile du processus
- Copier les paramètres dans zone U (champs u_arg)
- Sauvegarder le contexte en cas de retour prématuré (interruption par des signaux)
- Exécuter l'appel
- Si erreur : positionner le bit report du mot d'état
  mettre le numéro d'erreur dans un registre

- 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)


zone U
P1
u_ofile[]
File[]
CO
pile
code
P2
pile
données
données
zone U
u_ofile[]
CO

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)

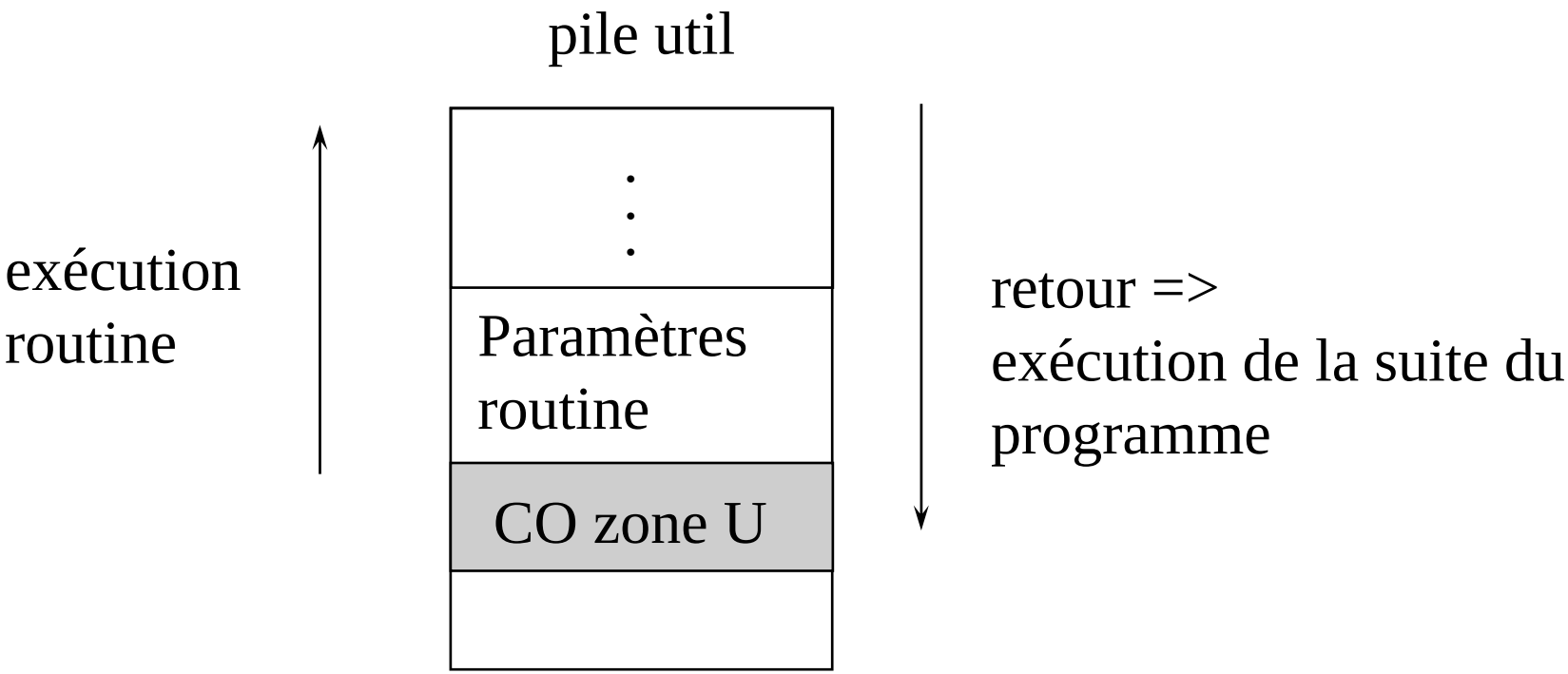
- => 1 seul traitement pour plusieurs instances du même signal
- Le signal ne sera traité que lorsque le processus cible passera sur le processeur

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.

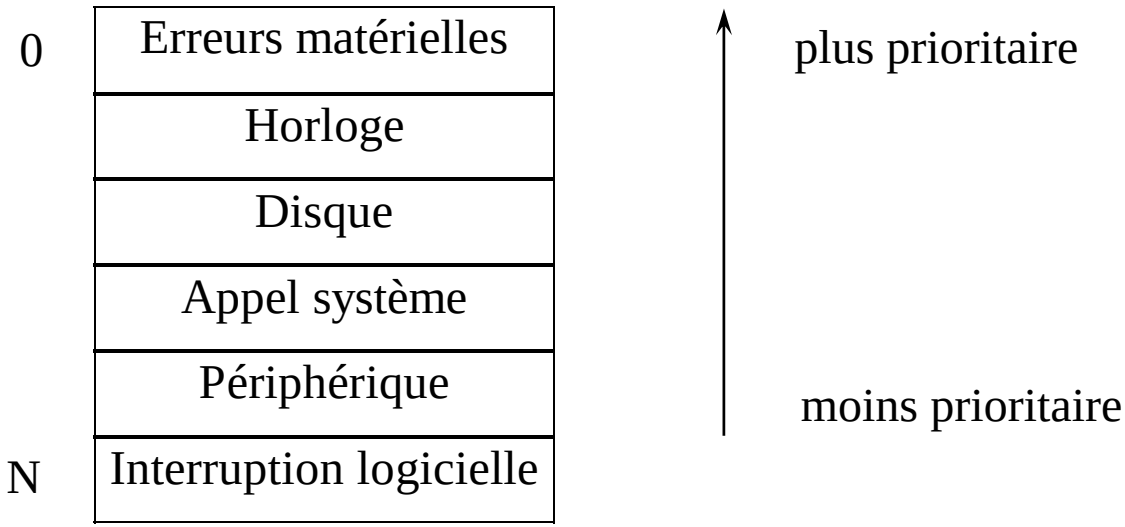
Signaux : traitement (2)

- sendsig : appel dépendant de la machine



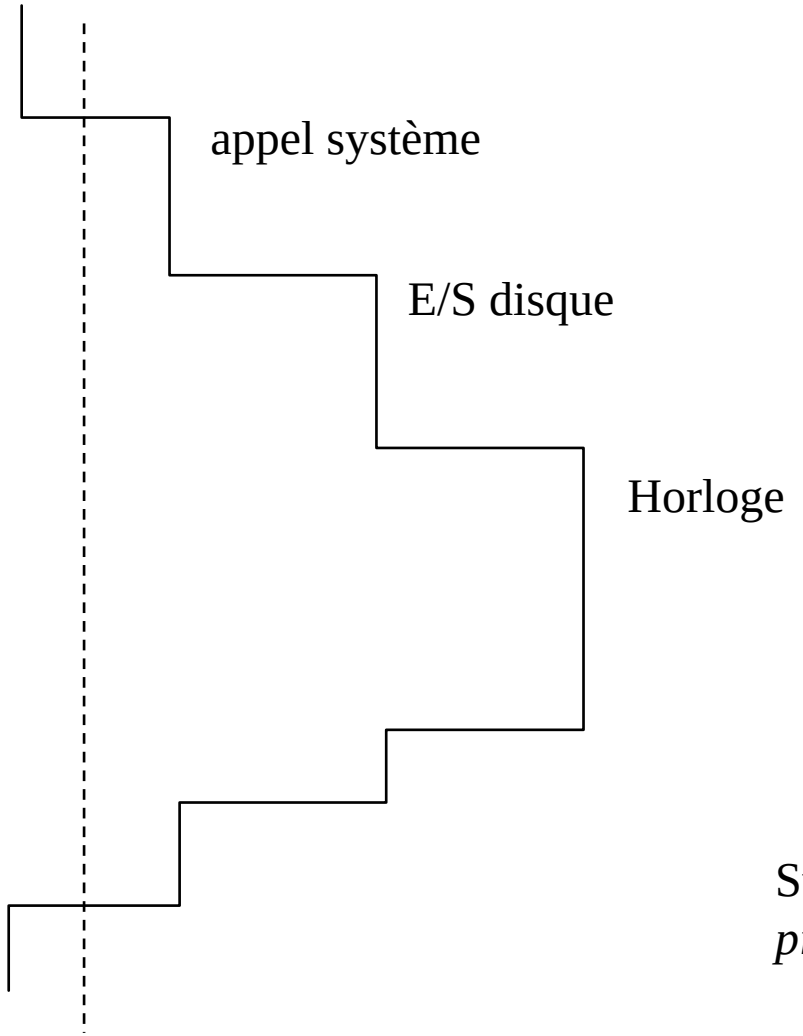
Les interruptions

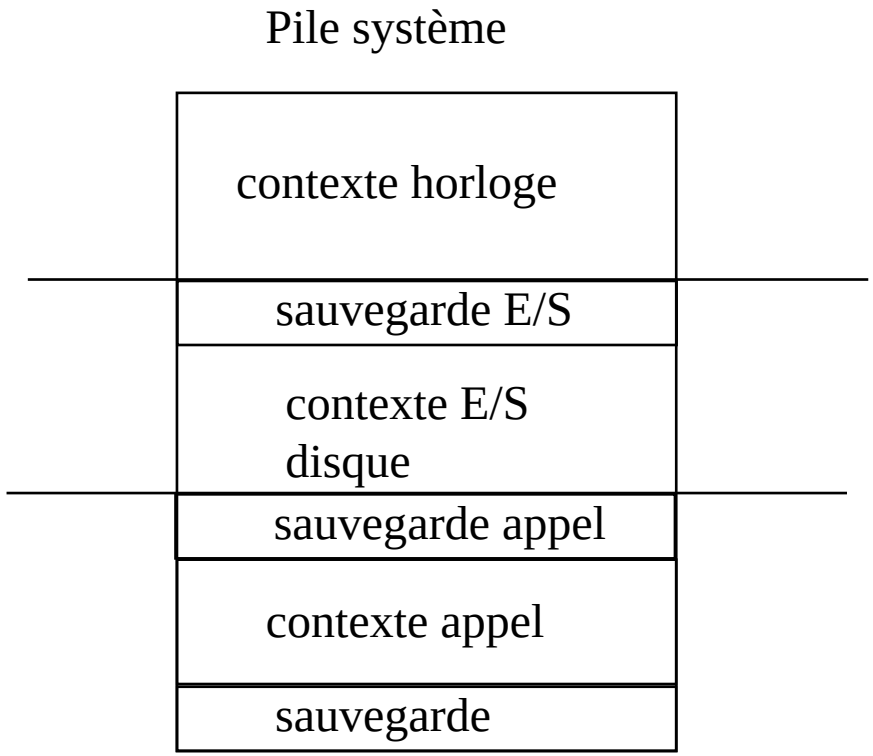
- Pour chaque interruption, un niveau de priorité (ipl: interrupt priority level)
- 7 niveaux Unix de base, 32 niveaux Unix BSD



- ipl stocké dans le mot d'état

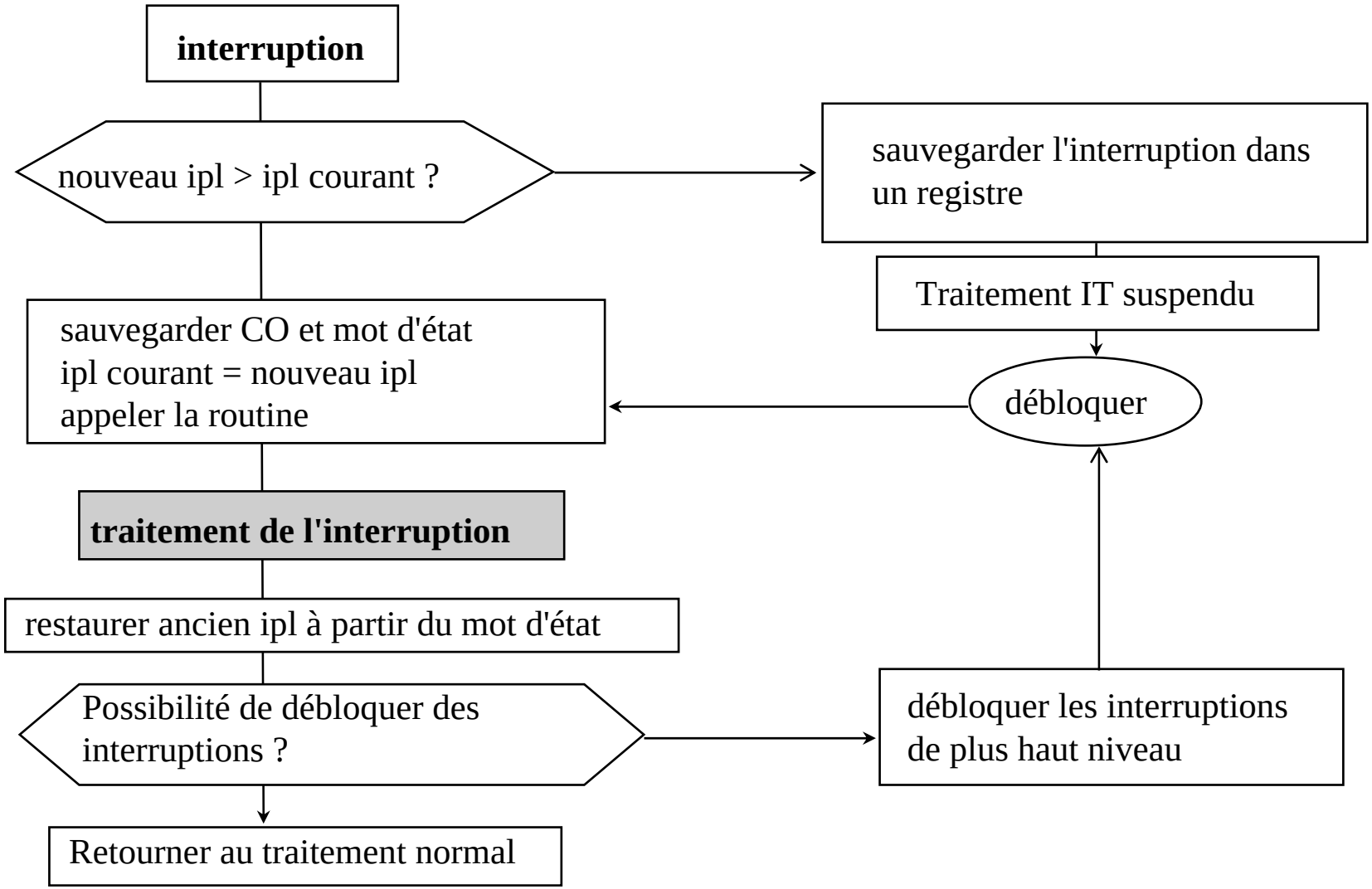
Les niveaux d'exécution





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

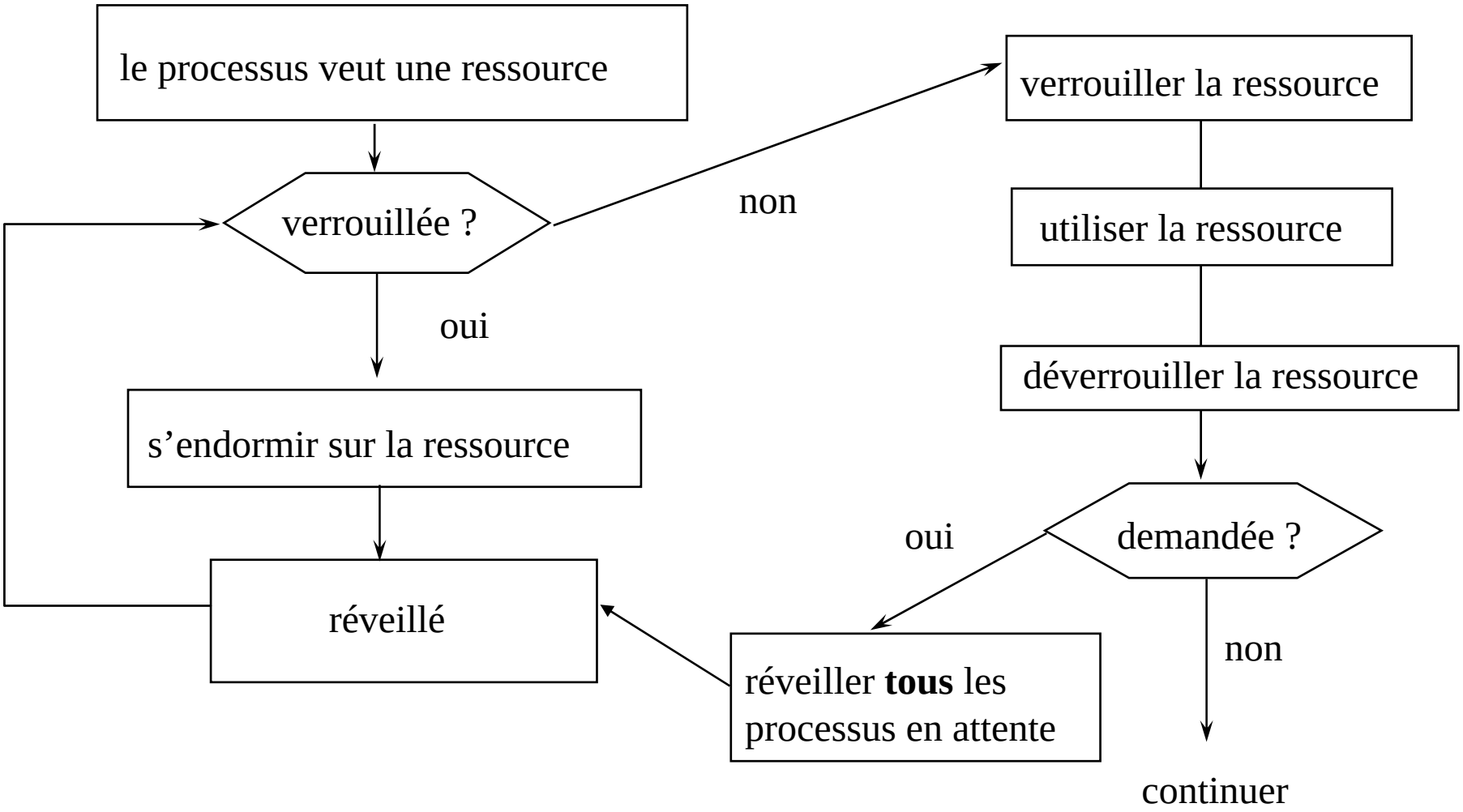
- Appel à set-priority-level pour bloquer les interruptions

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 */

iowait(bp) {
        ps = slpbio();
        while ( !pb->b_flgags & B_DONE )
                sleep(bp, PRIBIO);
        slpx(ps);
        ...
}
```

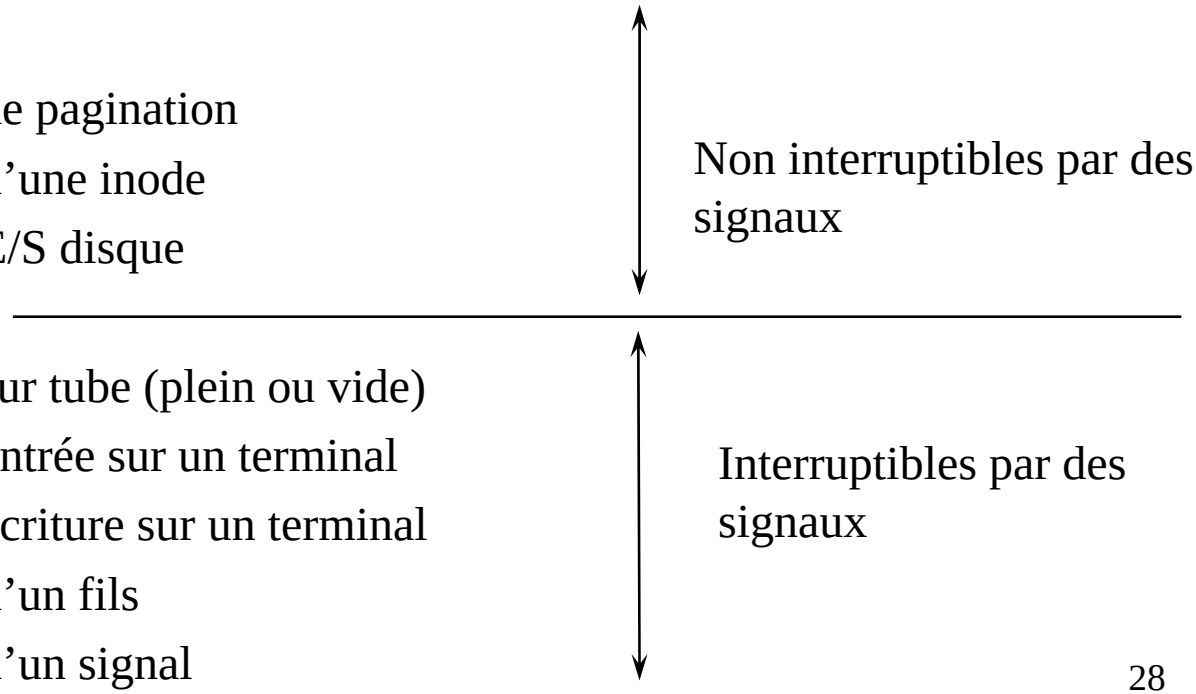
```
/* Fin d'E/S */

iodone(bp) {
        ...
        bp->b_flags |= B_DONE ;
        if (bp->b_flags & B_ASYNC)
                brelse(bp) ;
        else wakeup(bp) ;
}
```

iodone exécuter ici => sleep inutile !

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



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()

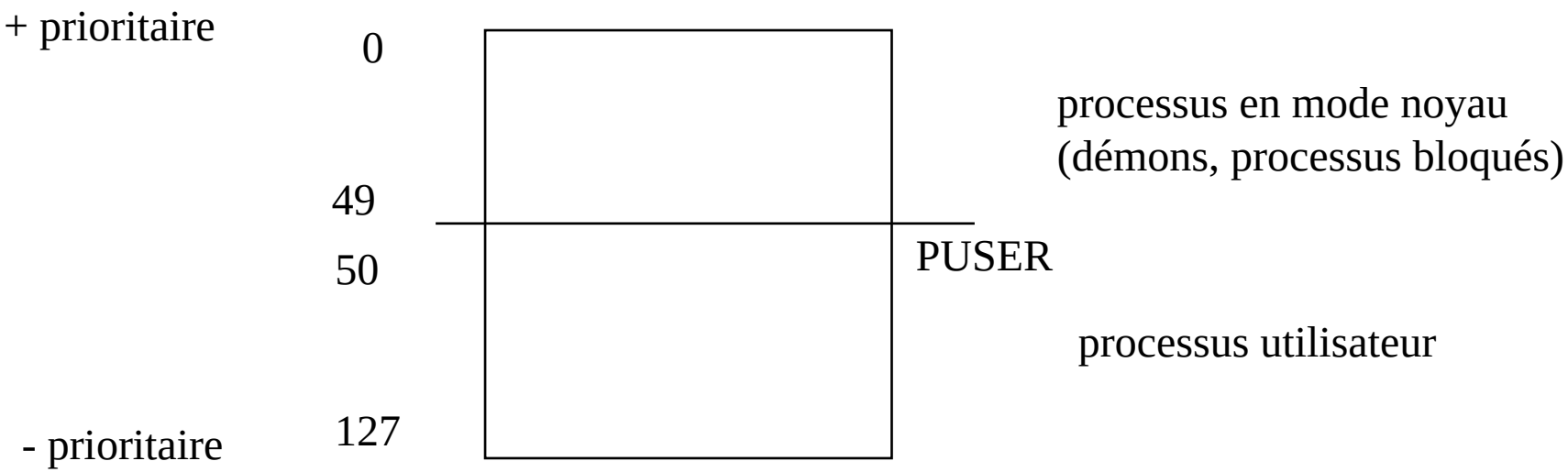
- Horloge matérielle interrompt le processus à des intervalles de temps fixes = tics horloge
- tic - 10 ms
  - HZ dans param.h indique le nombre de tics par seconde (par ex. 100)
- Routine de traitement dépendant du matériel
- Doit être courte !
- Très prioritaire
- 1 quantum = 6 ou 10 tics

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)

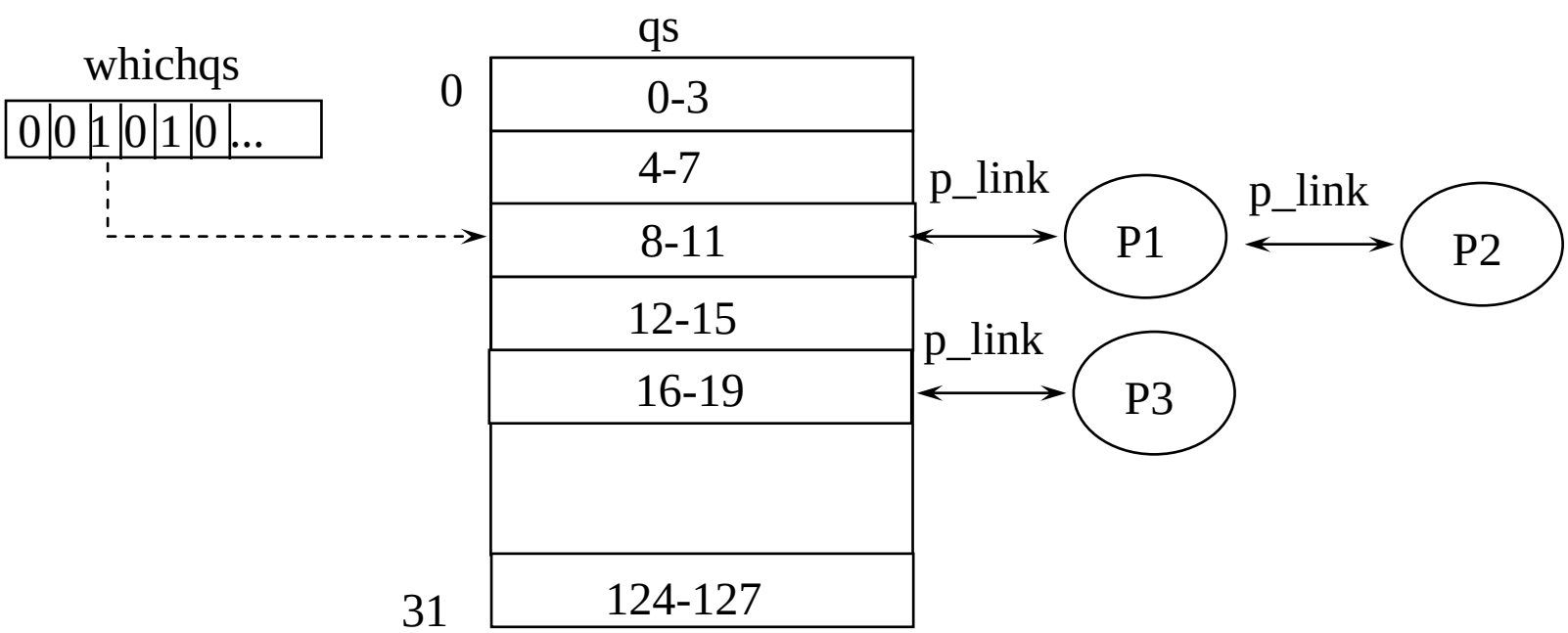
=> Au réveil le processus a une plus grande probabilité d'être élu

=> 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

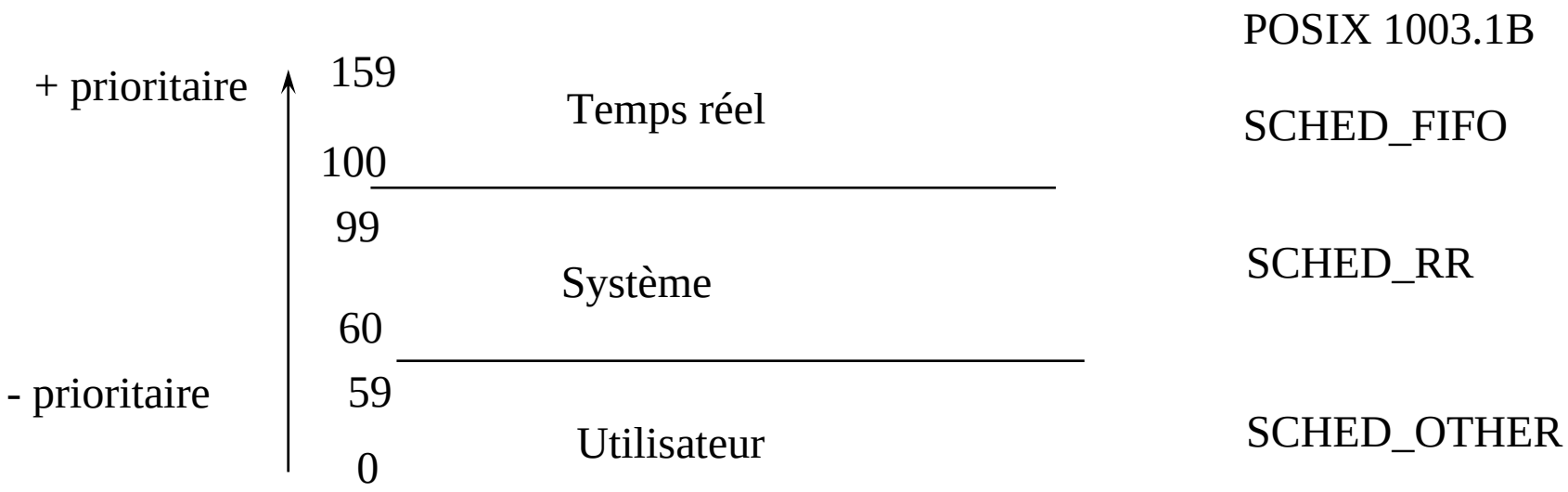


Algorithme de swtch

- 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) ...

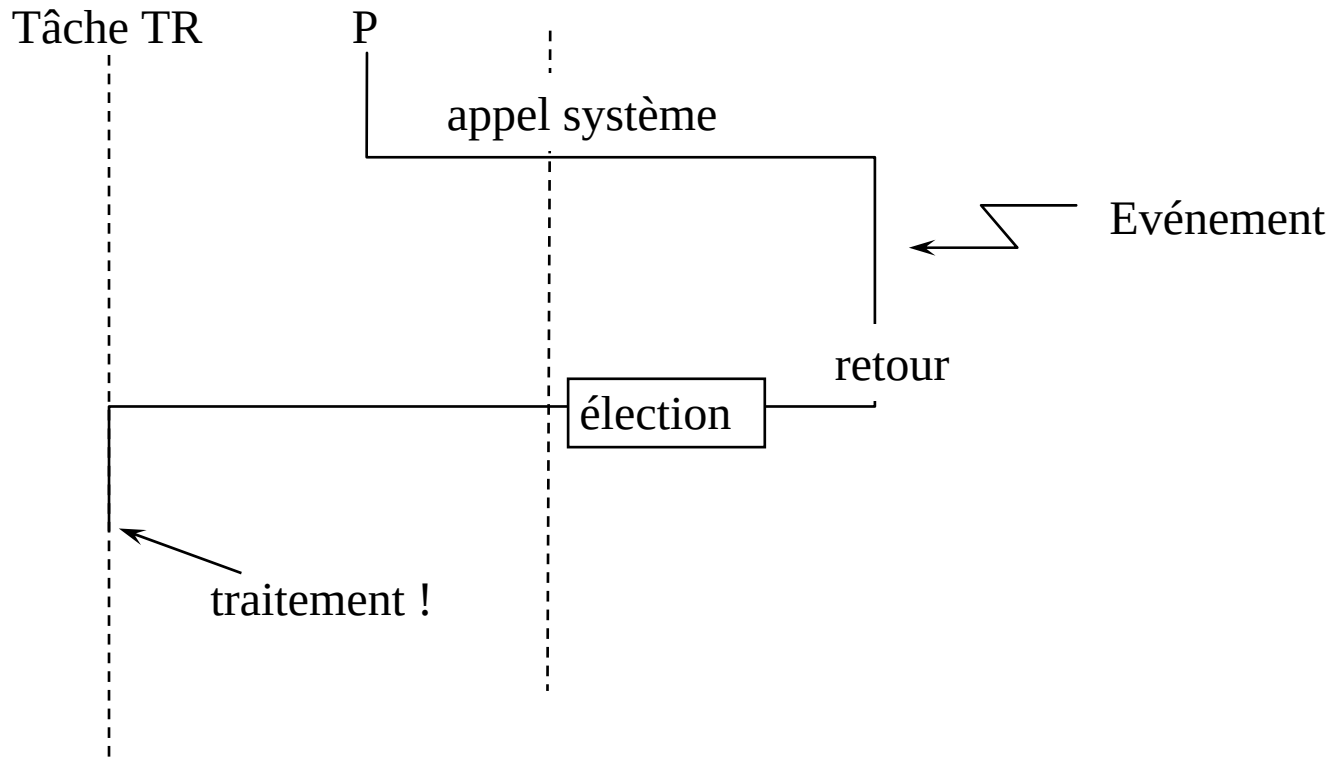
Classe "temps partagé"

- 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

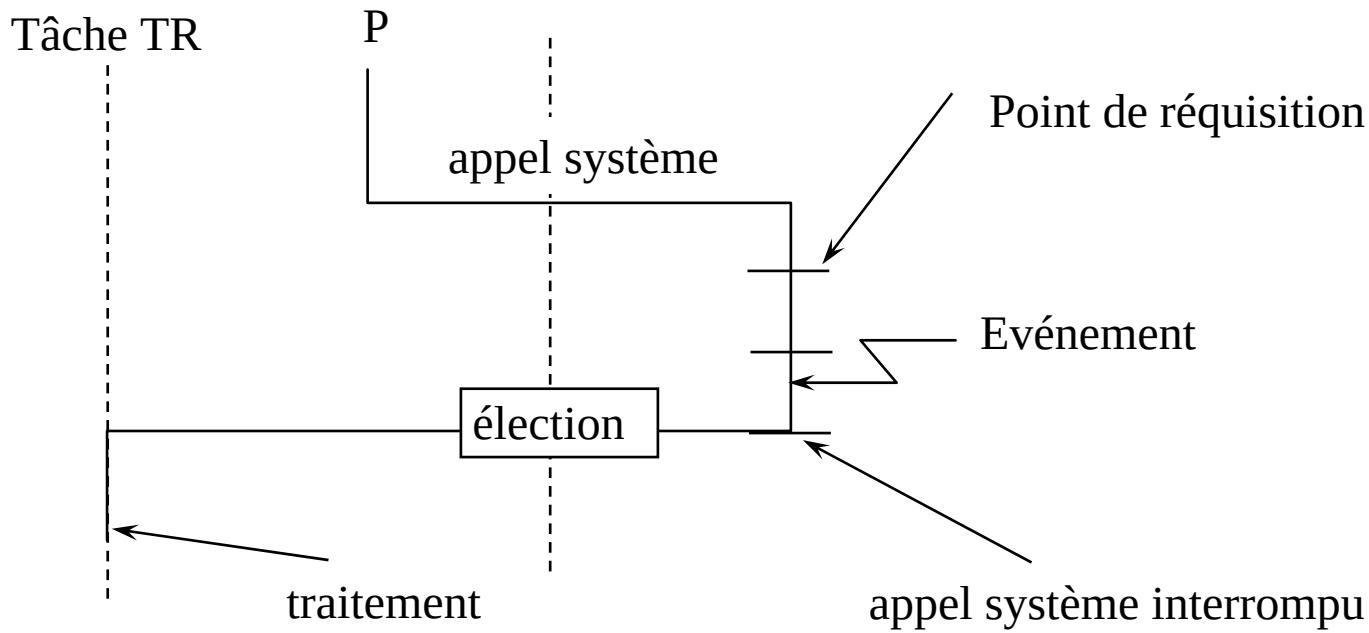
Classe "temps réel"

- 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

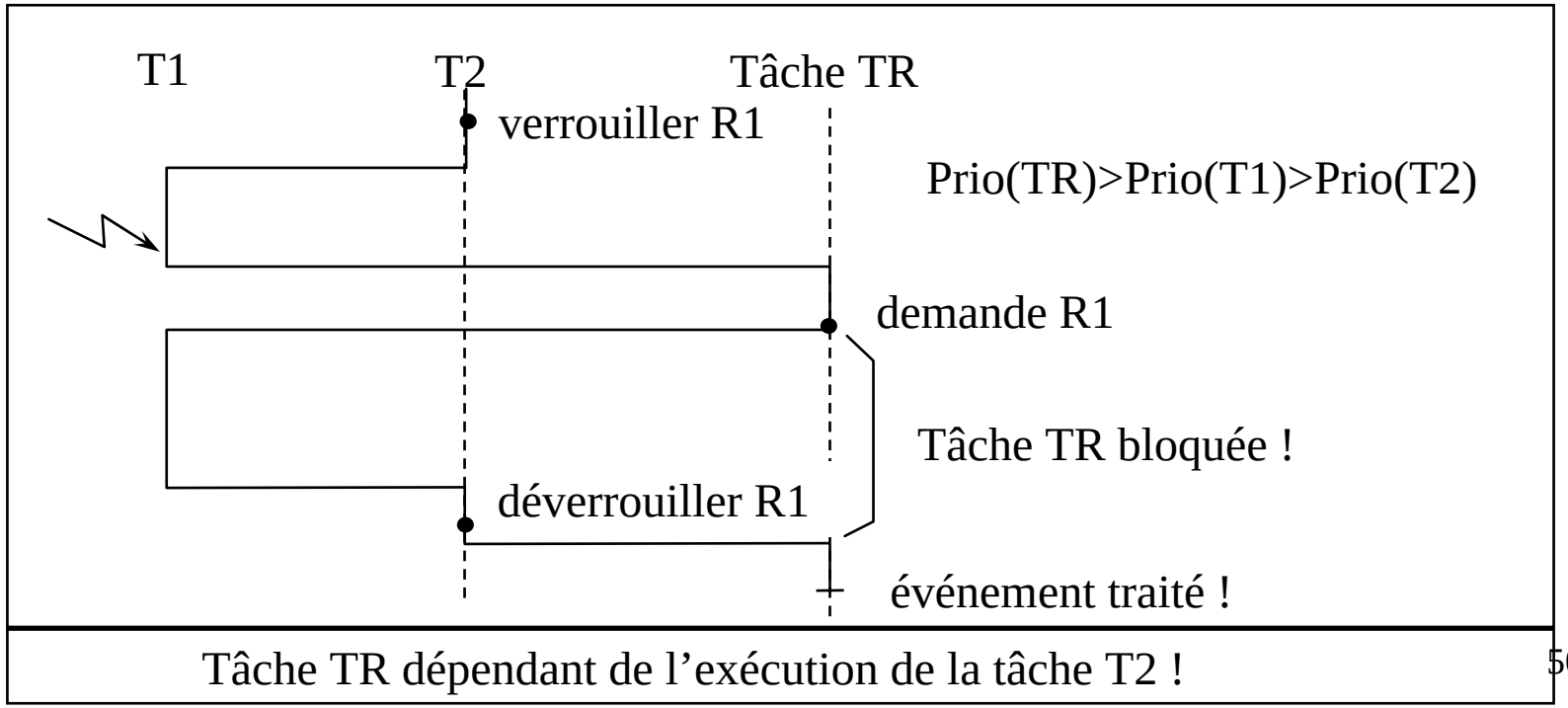
Noyaux Préemptifs (Solaris 2.x)

- Solution 2 : rendre le noyau préemptif

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

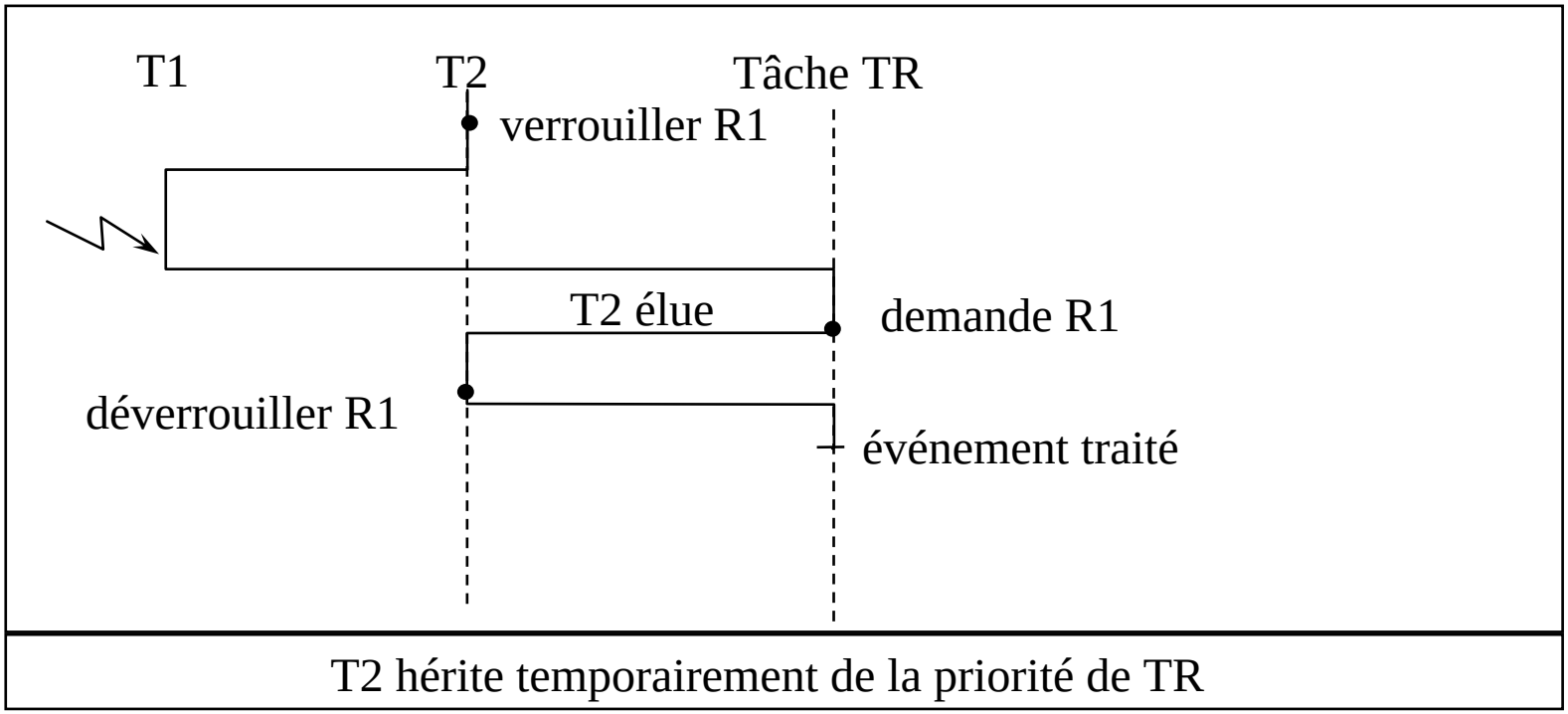
- Protéger toutes les structures de donnée du noyau par des verrous (~ sémaphores)

Inversion de priorité

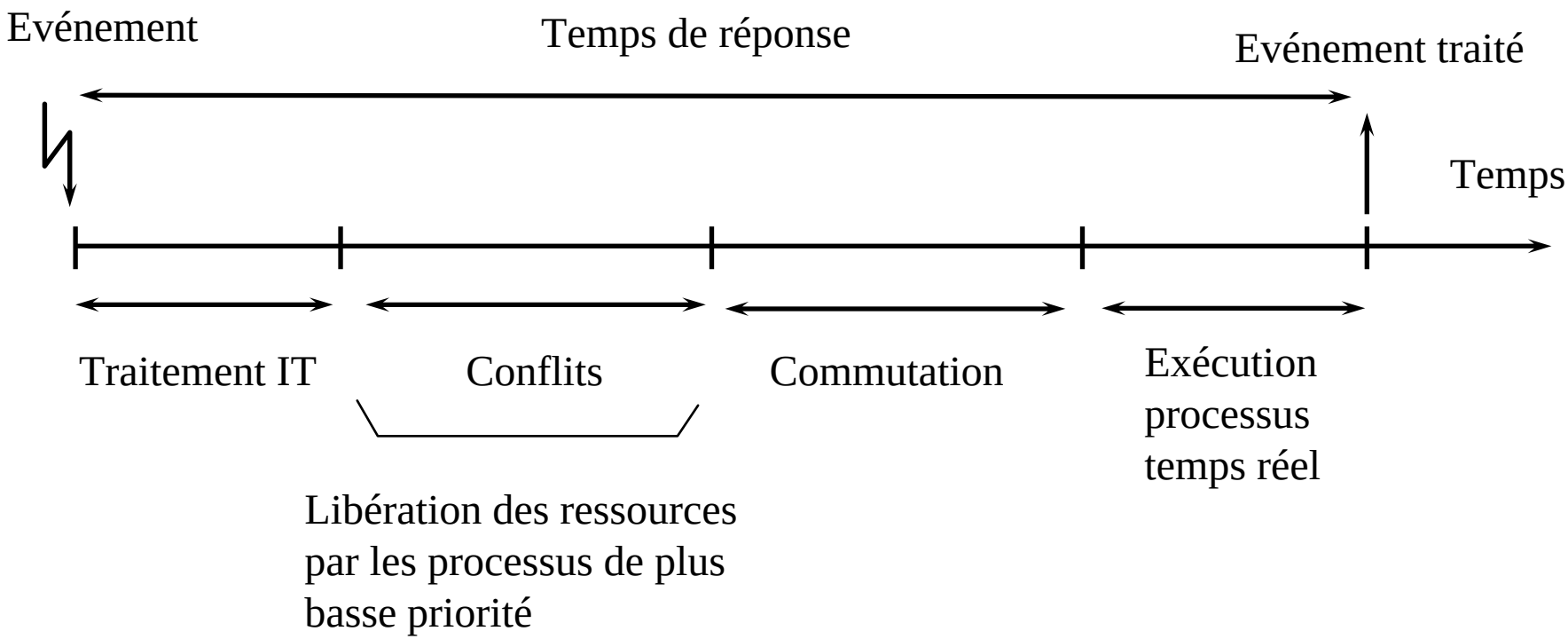


Héritage de priorité

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

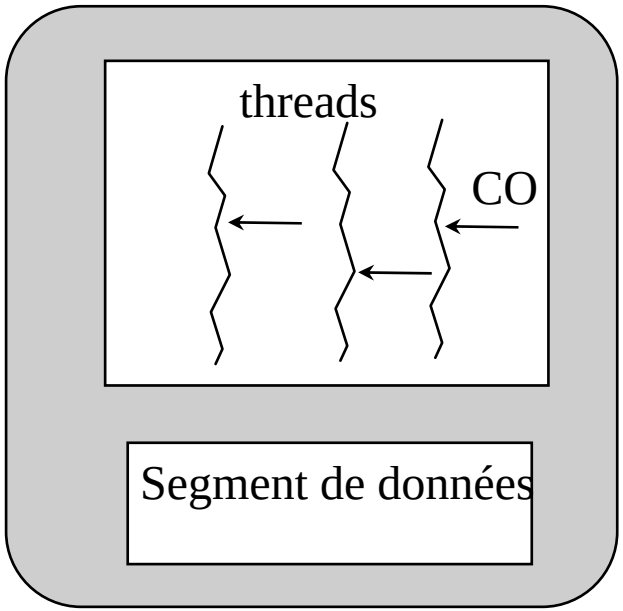


Temps de réponse



Processus légers

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



thread = code + pile + registres

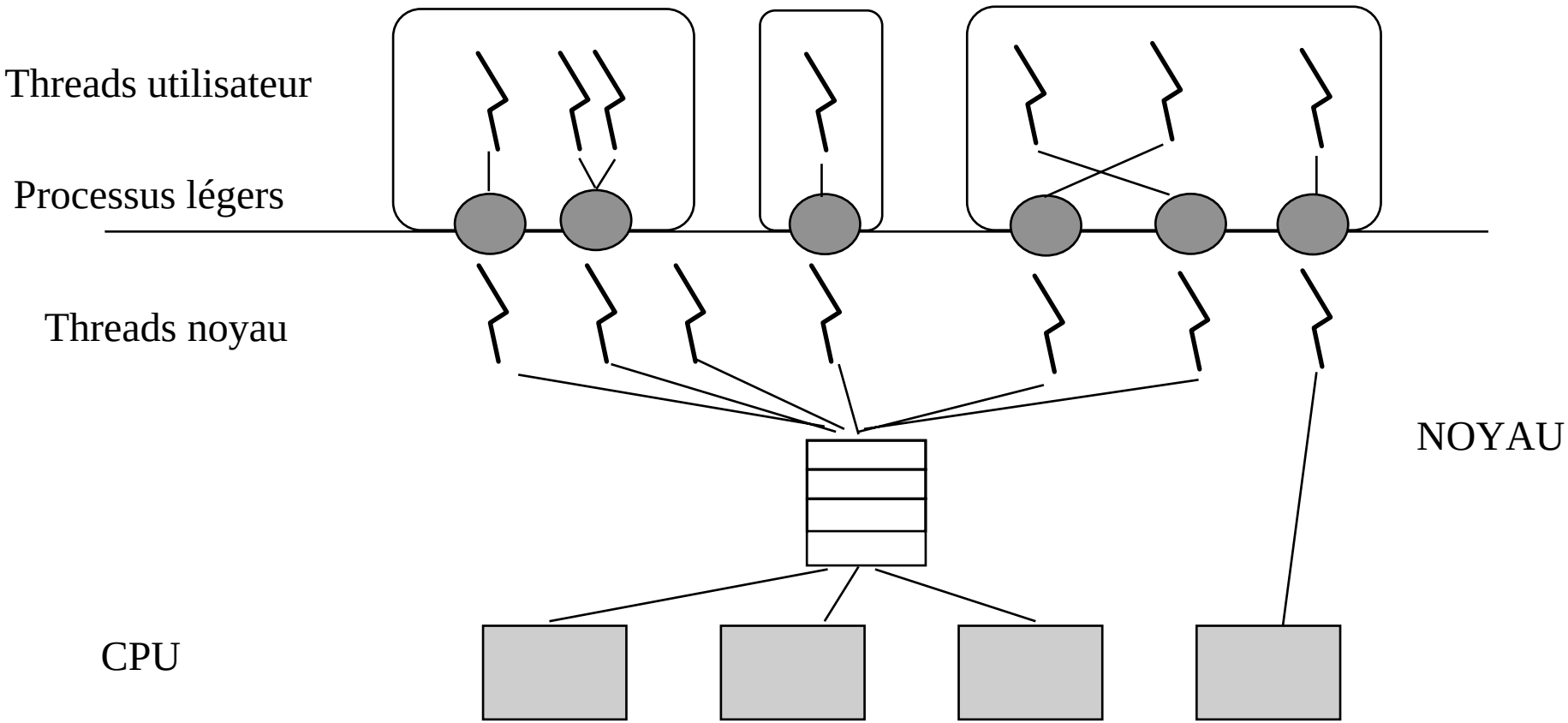
- thread (processus léger) : unité d'exécution

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 synchronisation en utilisant des sémaphores (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

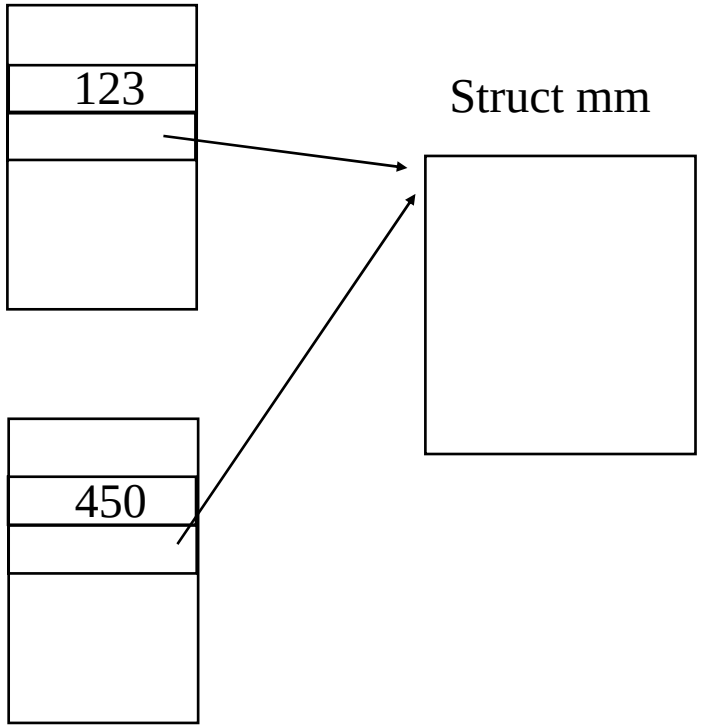
Structure de données : struct task

- 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)
  - 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) :

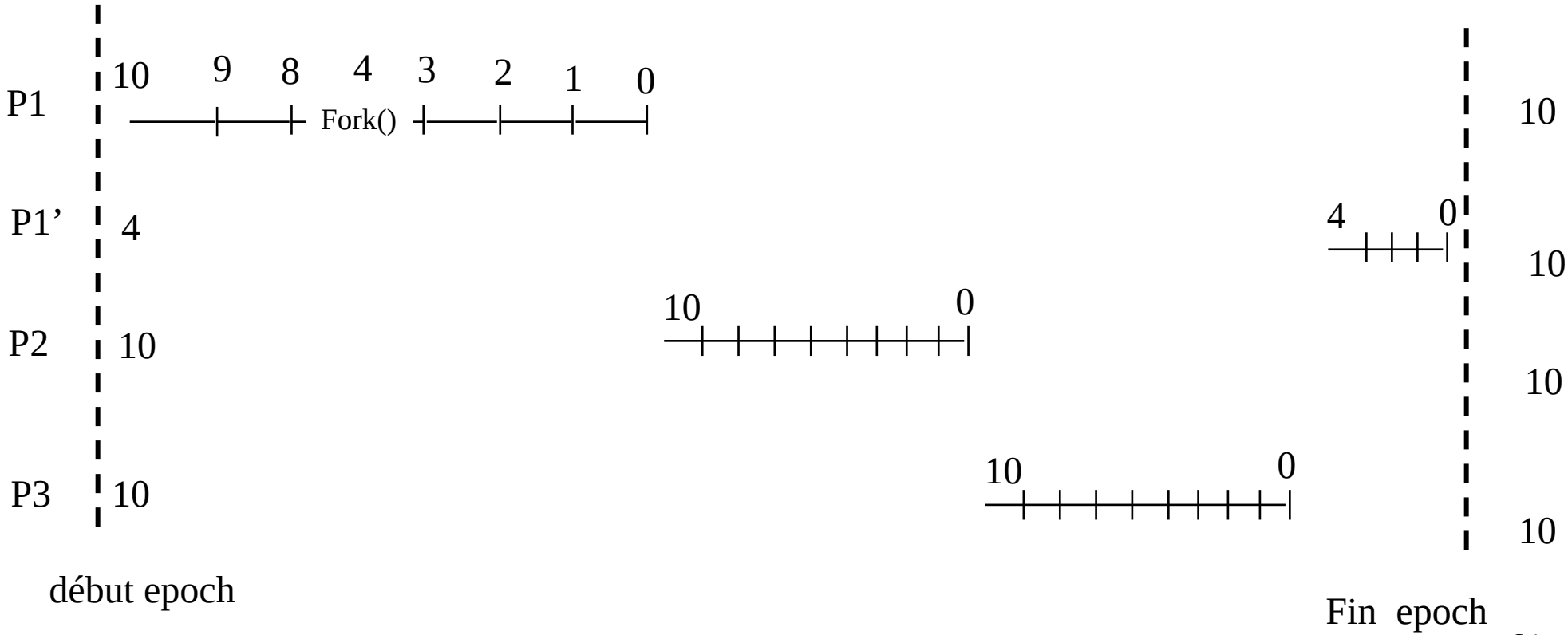
     Poids = 1000 + priorité base pour processus temps réel

     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 P1 sur CPU1 (counter) est supérieur au temps estimé de remplissage des cache de CPU1

Nouveautés Linux 2.6

- Ordonnancement

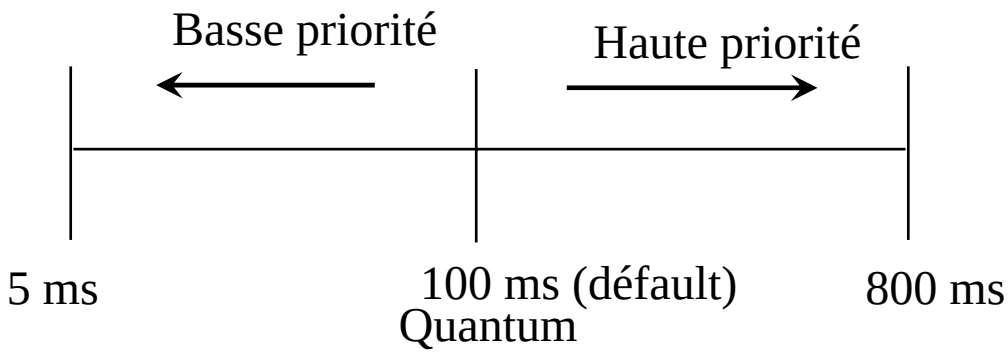
- Noyau préemptif

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

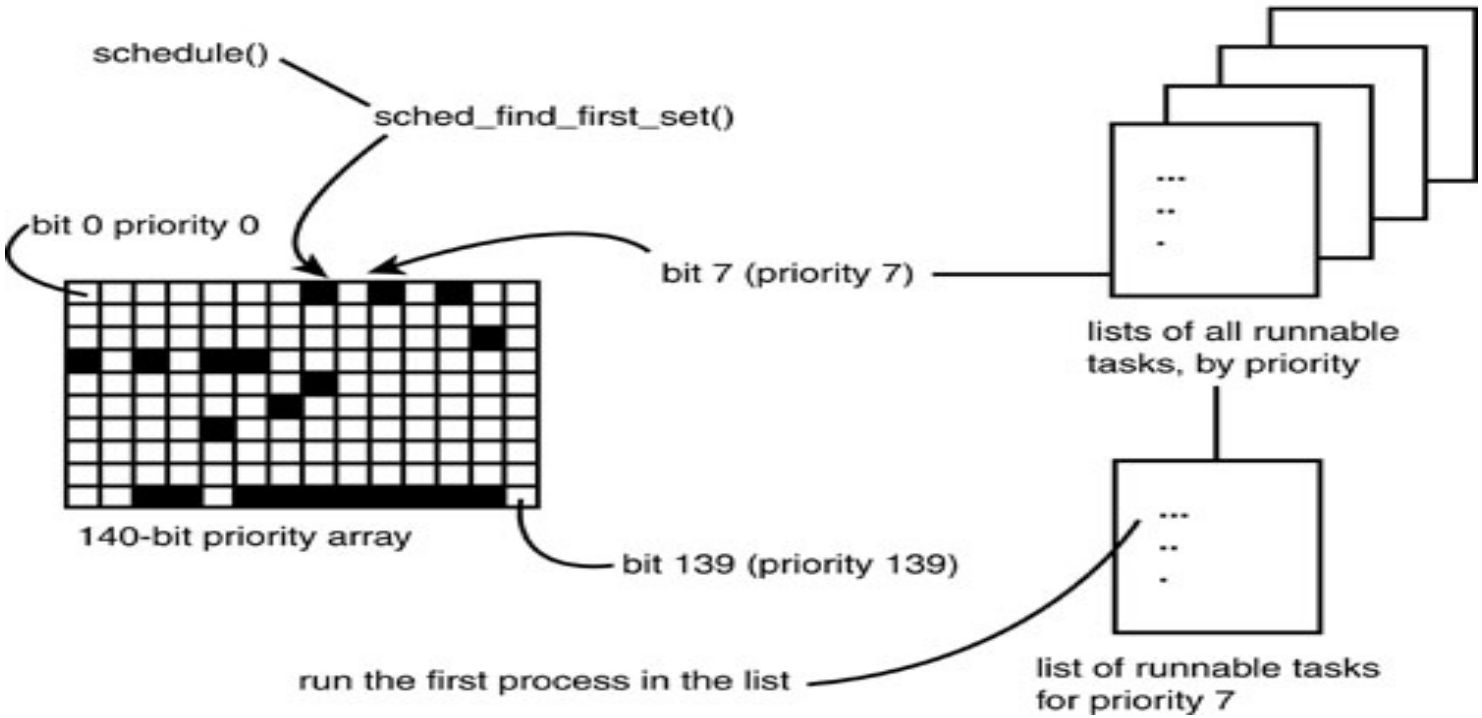
- Priorité de base = valeur du nice [-20,+19]
- Système de pénalité/bonus en fonction du type de processus

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]

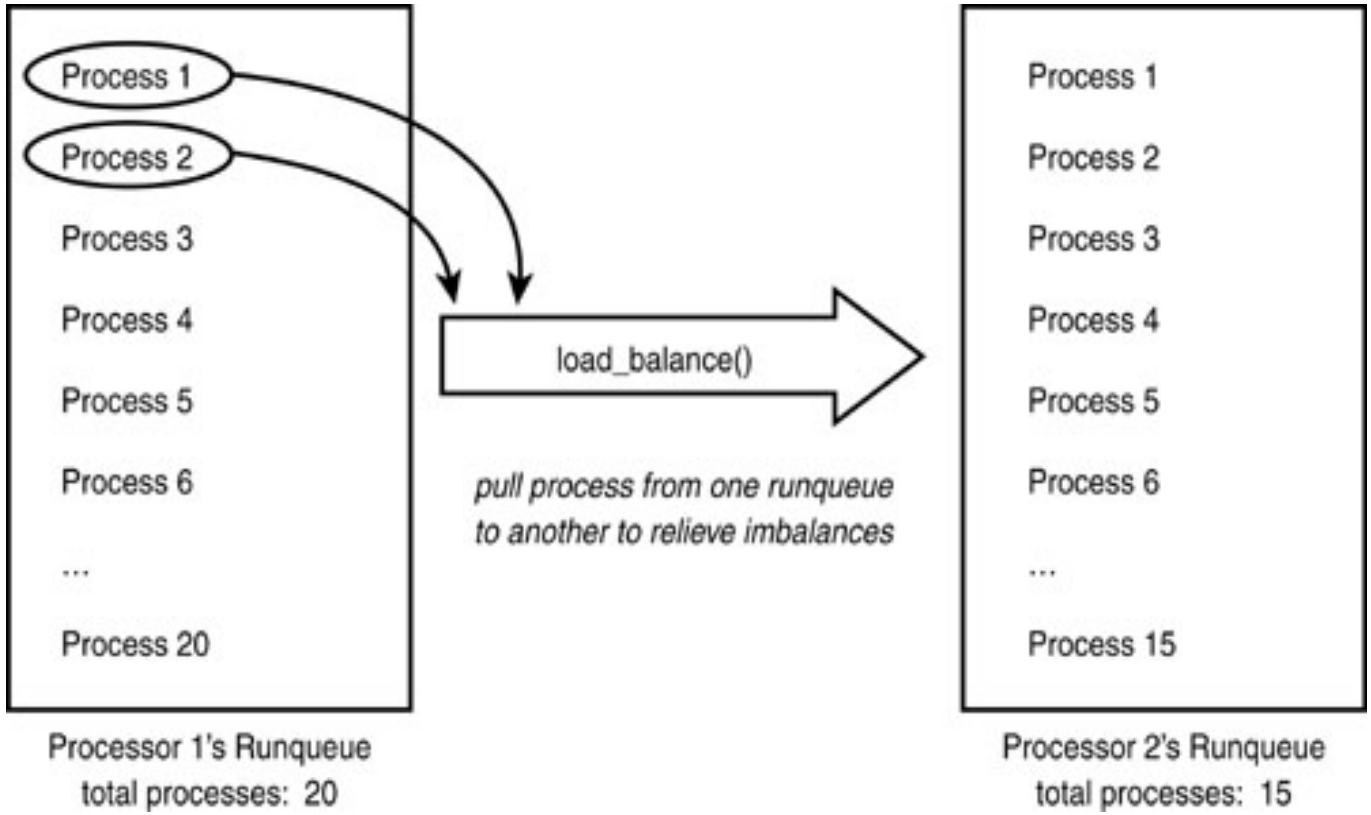
Algorithme de choix du processus

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



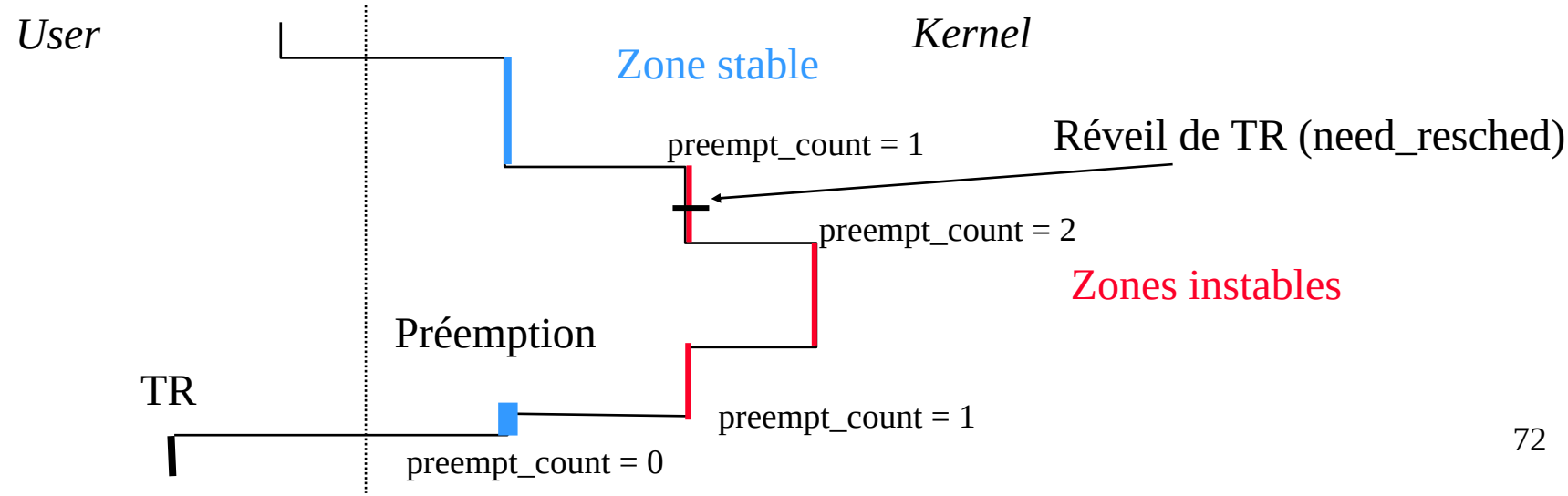
Equilibrage de charge

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

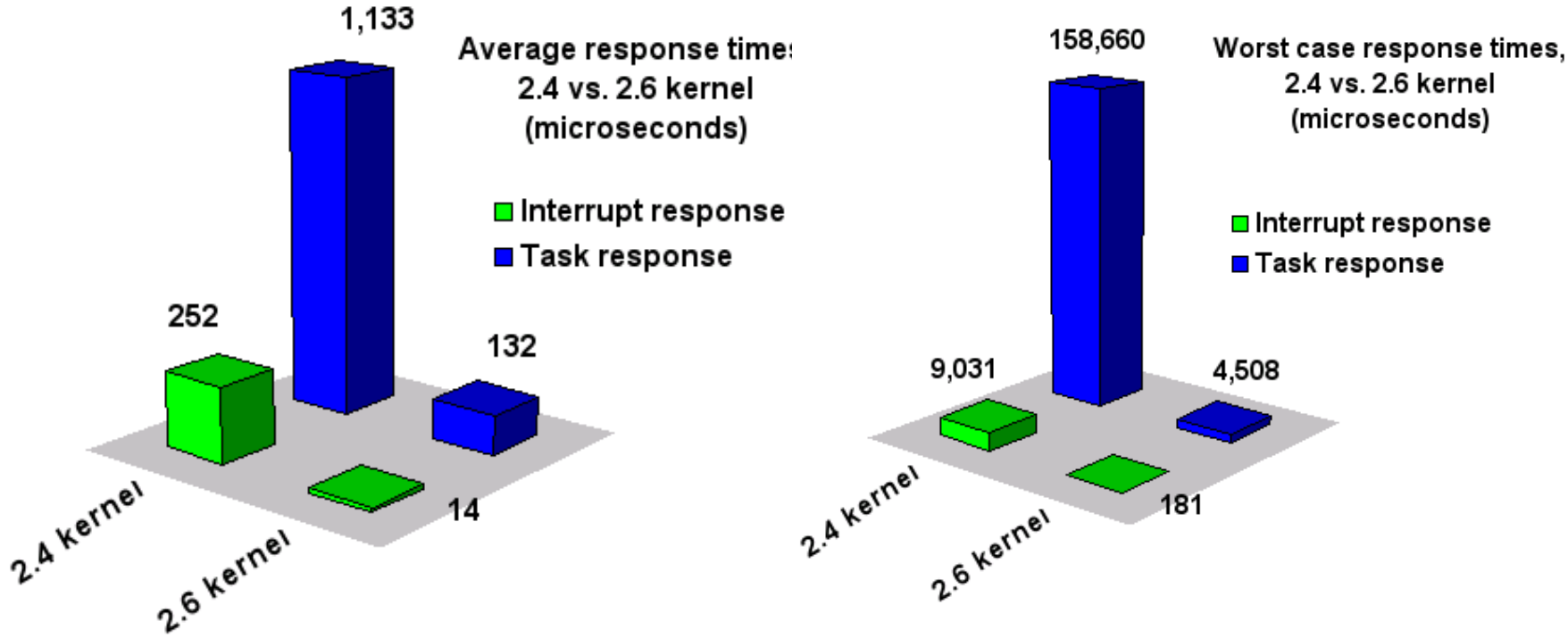


Noyau Préemptif – Linux 2.6.x

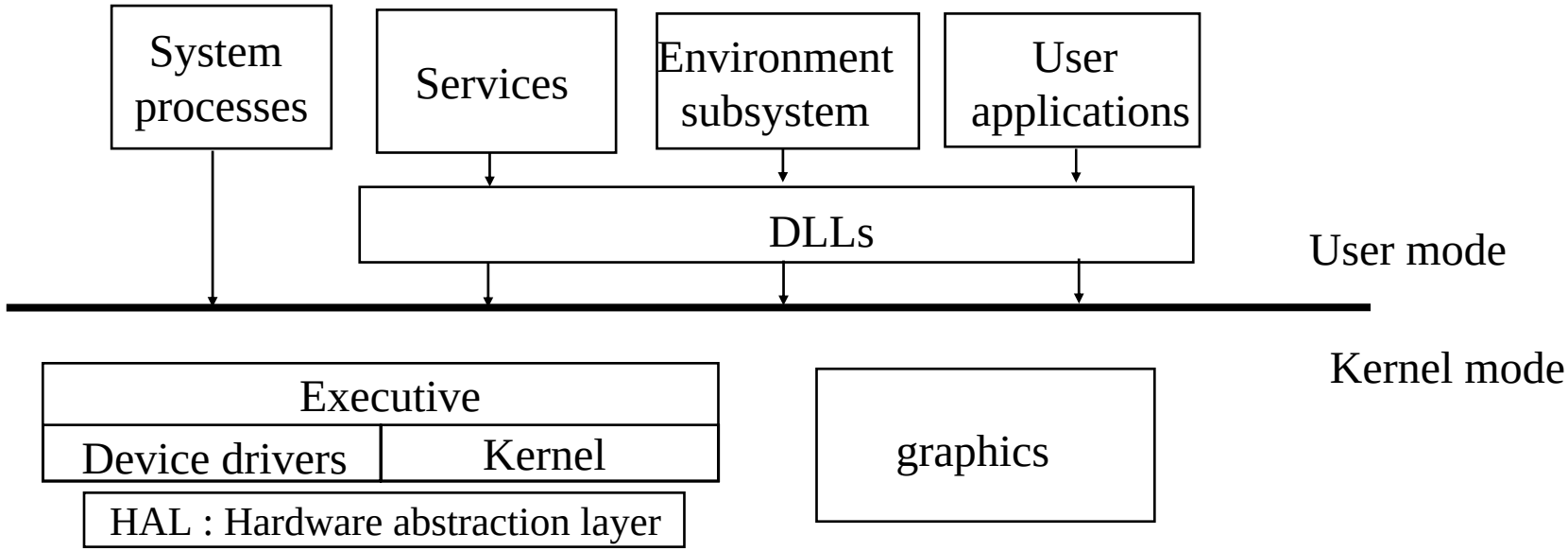
- Proche de la notion de points de réquisition :
  - Quitter le noyau uniquement à des points stables
- Verrouillage pour protéger les régions instables :
  - => un compteur (preempt_count) incrémenté à chaque verrouillage
- Retour d'IT :
  - si need_resched et preempt_count == 0 ➔ Préemption



Linux 2.4 vs. 2.6

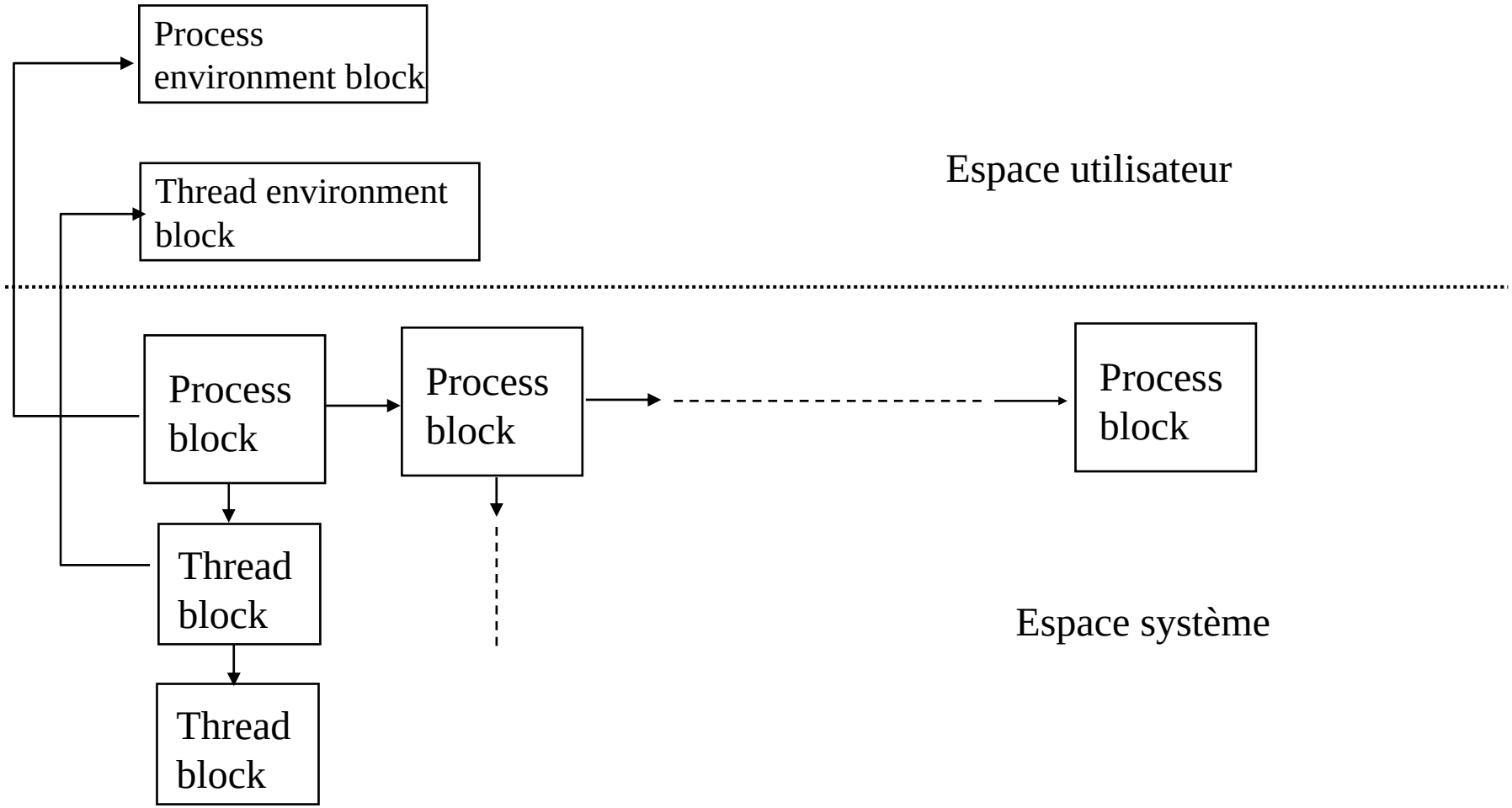


Gestion de processus dans Windows NT



1. Processus et Threads
2. Ordonnancement

Les structures

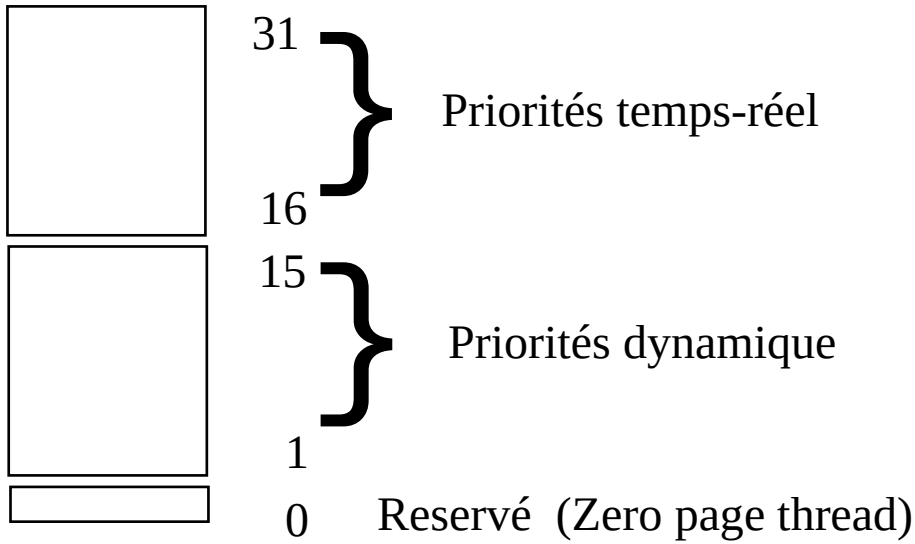


Les structures

- 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

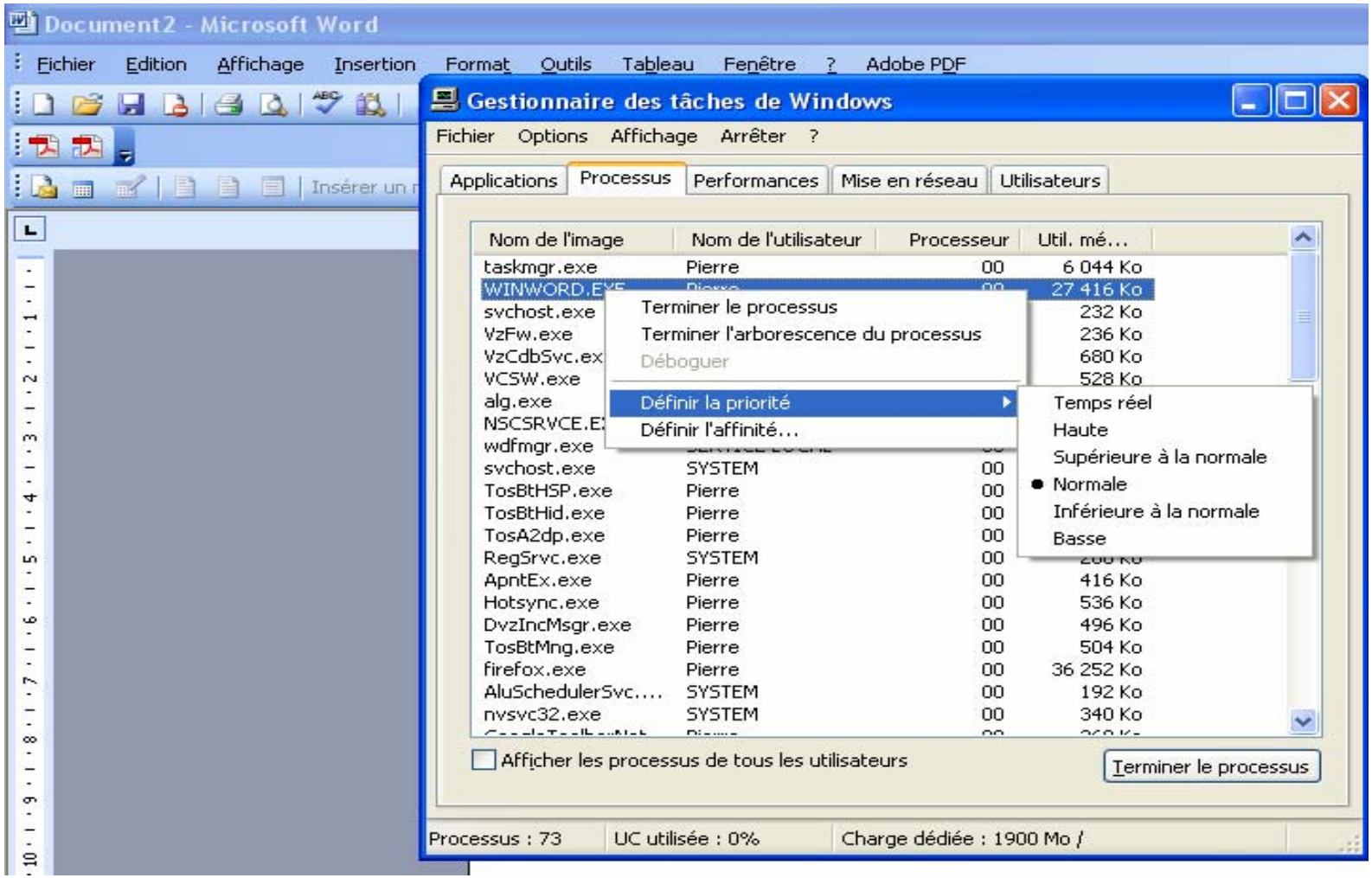


Ordonnancement temps partagé (quantum)

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

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

Classes de priorité

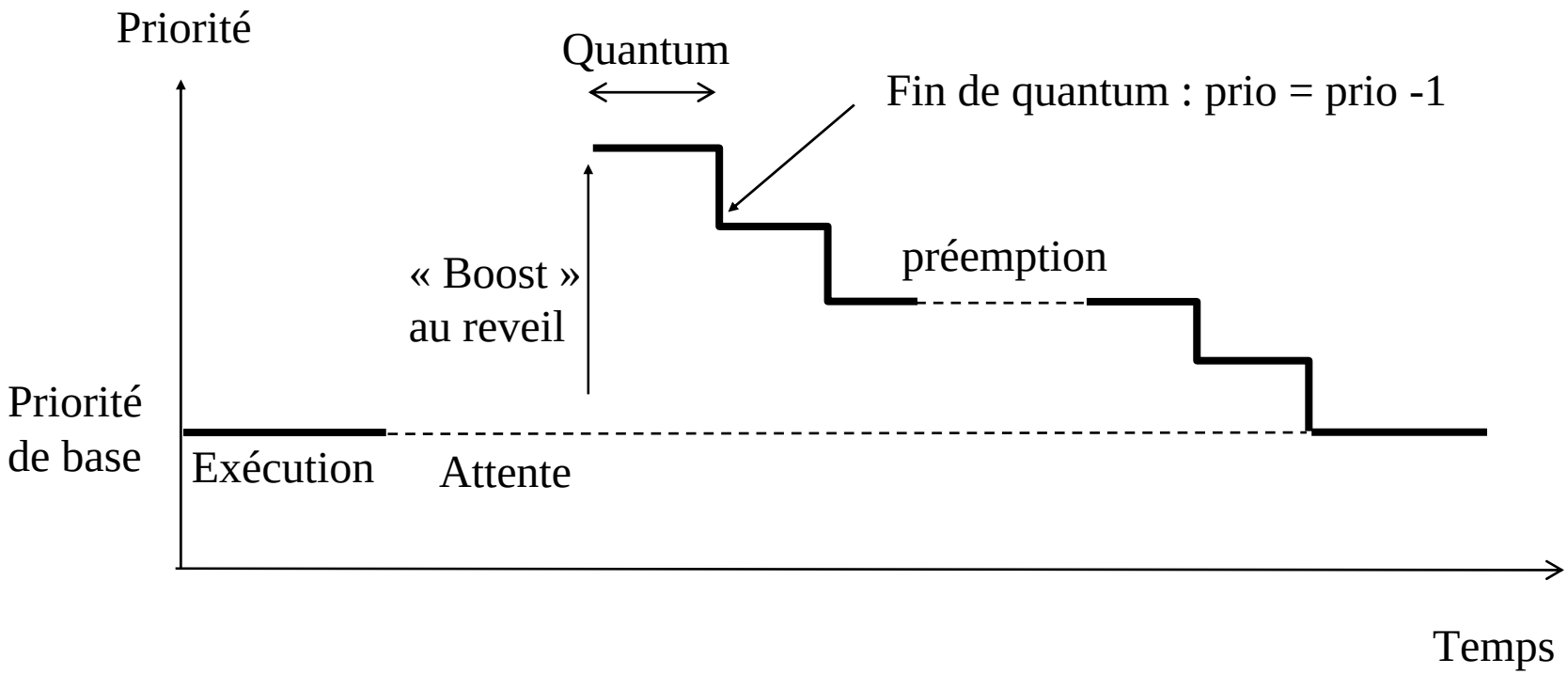


Ajustement des priorités et du quantum

- 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


Priorité
Quantum
Fin de quantum : prio = prio -1
« Boost »
au reveil
préemption
Priorité
de base
Exécution
Attente
Temps

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