Troisième partie

Gestion des processus et ordonnancement

- Implémentation des processus
- Implantation des processus Linux 2.6
  - Descripteur de processus
  - Processus légers
  - Le cas de l'architecture x86
- Ordonnancement des processus (scheduling de l'UC)
  - Introduction
  - Algorithmes de scheduling
  - Étude de cas : SOLARIS, HP-UX, 4.4BSD et Linux 2.6

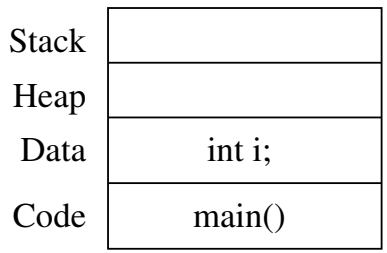
Les processus : rappel

- **Processus :** l'entité dynamique qui exécute un programme sur un processeur
- **Processus != Programme**

- **Programme :**
  Code + data (**passif**)

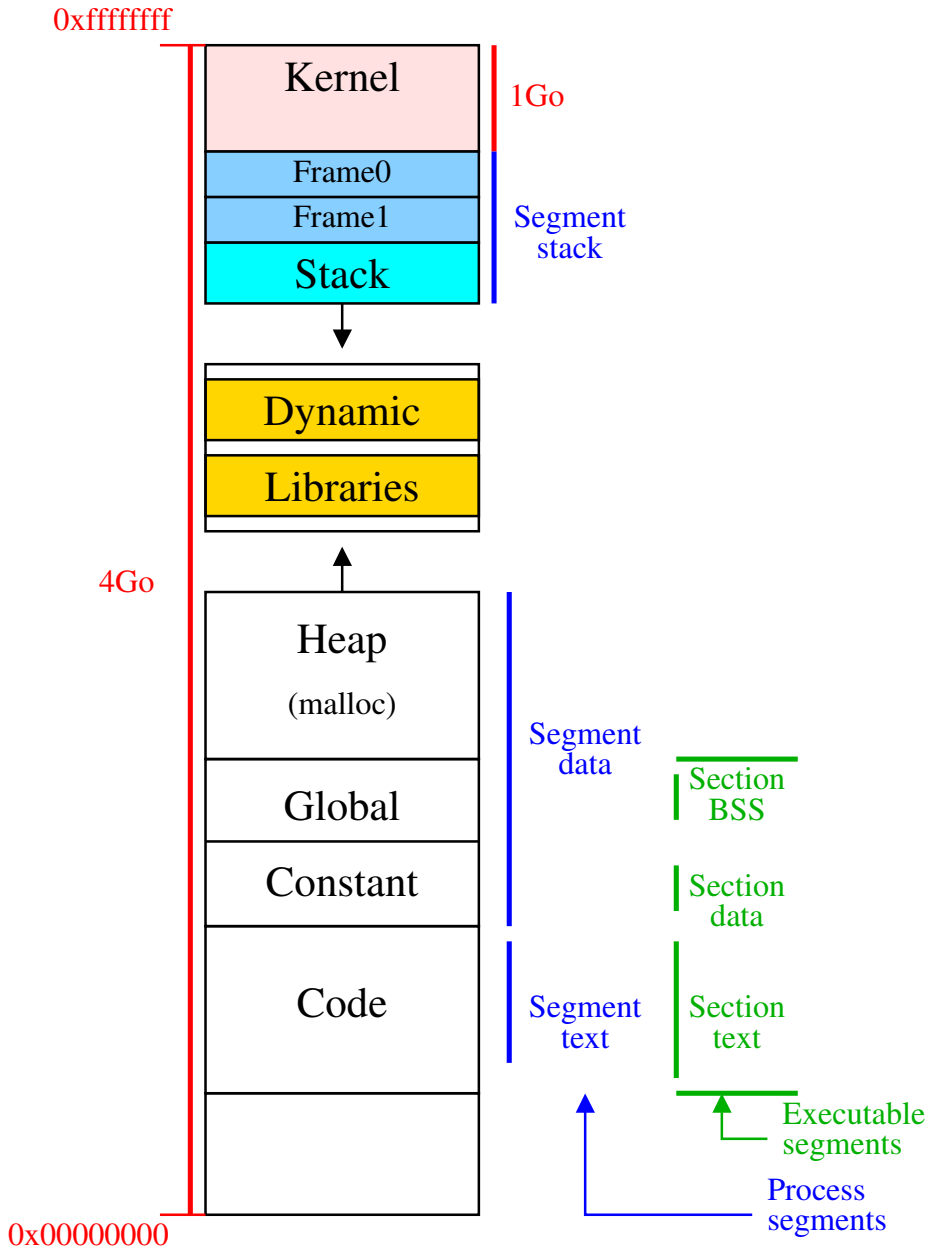
```
int i;
int main() {
printf("Salut\n");
}
```

- Processus :
  Programme **en cours d'exécution**



- Pour 2 personnes qui exécutent le même programme, il y aura 2 processus différents.

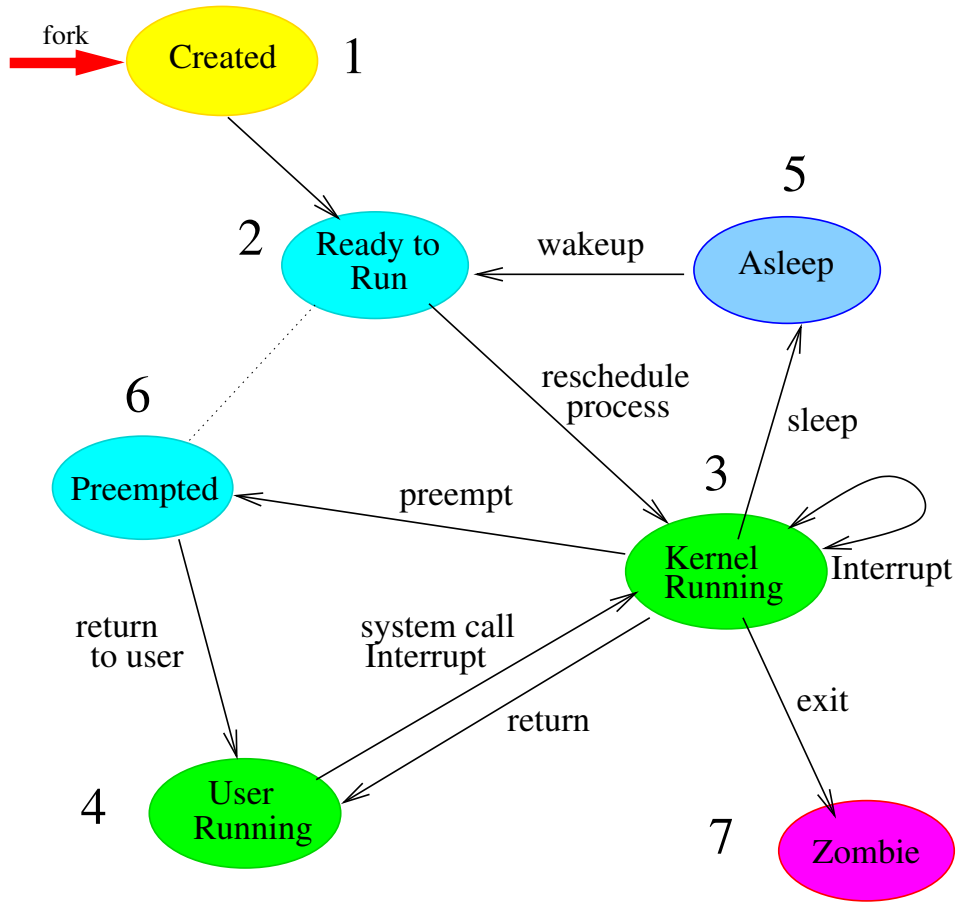
La mémoire virtuelle d'un processus



- En mode utilisateur, un processus manipule des zones privées pour la pile, les données et le code.
- En mode noyau, un processus utilise les zones de données et de code du noyau et une pile noyau privée (réentrance du noyau)
- la pile utilisateur (user stack) contient les frames des fonctions appelées en mode utilisateur (en plus des variables locales)
- la pile noyau (kernel stack) contient les frames des fonctions appelées en mode noyau (en plus des données des appels système)

États d'un processus

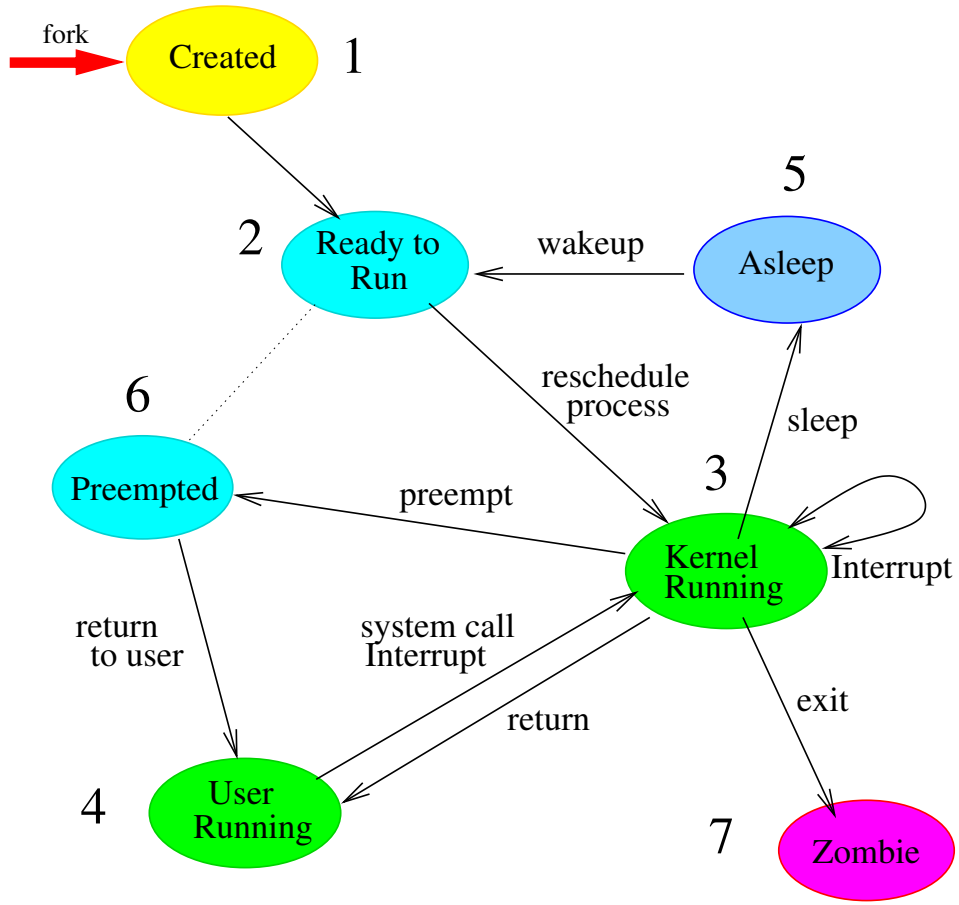
1. Created (Nouveau) : en cours de création
2. Ready to Run (Prêt) : le processus est prêt en attente de l'attribution du processeur
3. Kernel Running : le processus exécuté en mode noyau
4. User Running : le processus exécuté en mode utilisateur
5. Asleep (Bloqué) : le processus en atente d'un événement
6. Preempted (Réquisitionné) : le noyau lui a réquisionné le processeur lors du passage du mode noyau au mode utilisateur.
7. Zombie (Terminé) : le processus a terminé



États d'un processus

Exemples de transitions (1/2)

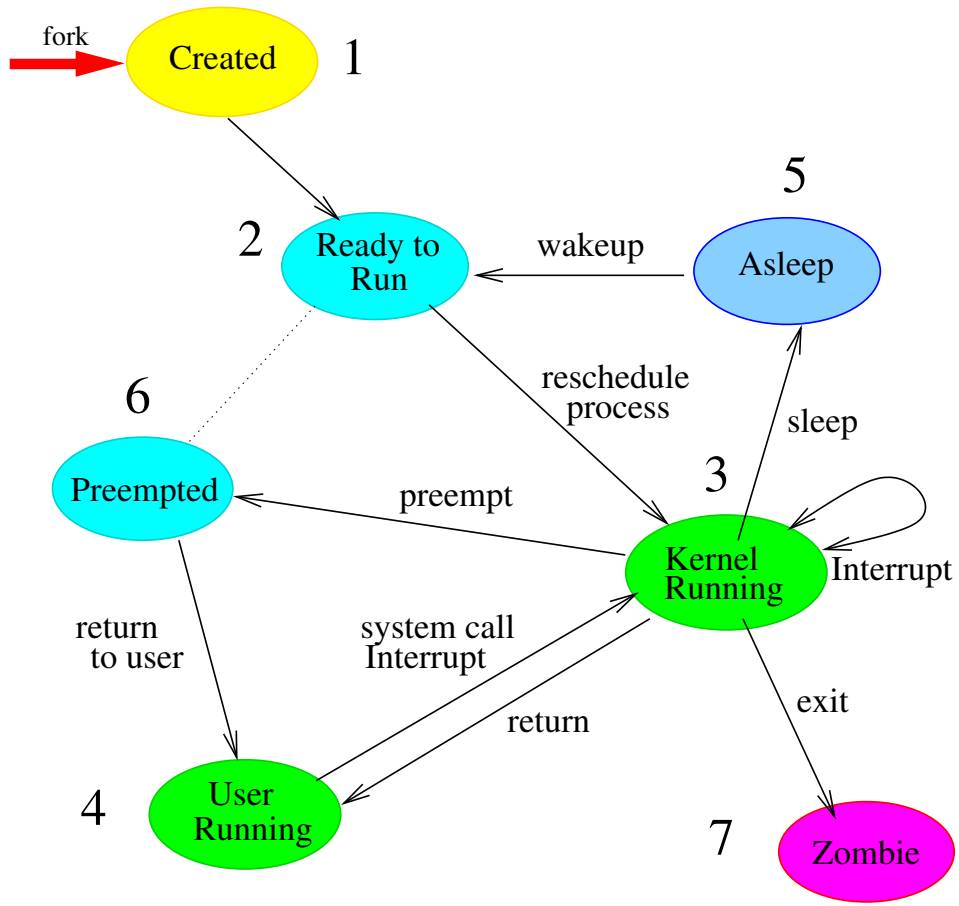
- 1 → 2 : processus admis dans le système
- 2 → 3 : l'ordonnanceur a choisi le processus et s'exécute en mode noyau pour terminer le fork()
- 3 → 4 : le processus termine l'appel système et passe en mode utilisateur
- 4 → 3 : interruption d'horloge et retour au mode noyau
- 3 → 6 : interruption traitée et l'ordonnaceur décide de passer le processeur à un autre processus
- 6 → 4 : l'ordonnanceur a choisi ce processus



États d'un processus

Exemples de transitions (2/2)

- 4 $\rightarrow$ 3 : le processus exécute un appel système (E/S par exemple)
- 3 $\rightarrow$ 5 : le processus est en attente de l'E/S
- 5 $\rightarrow$ 2 : une interruption signale la fin de l'E/S et le handler réveille le processus qui passe à l'état "Prêt"



Contexte d'un processus

Le contexte d'un processus est l'ensemble des données qui permettent de reprendre son exécution si jamais interrompu.

- Contexte utilisateur zones texte, données et pile utilisateur
- Contexte matériel l'ensemble des registres du processeur
- Contexte système les structures de données du système pour l'implantation des processus.

Le changement de contexte consiste en la sauvegarde du contexte d'un processus et à restaurer celui d'un autre

- nécessaire en cas de blocage sur une E/S, une interruption, une exception, . . .
- recommandé pour assurer la réactivité, le partage et l'équité dans un système multiprogrammé

Changement de contexte

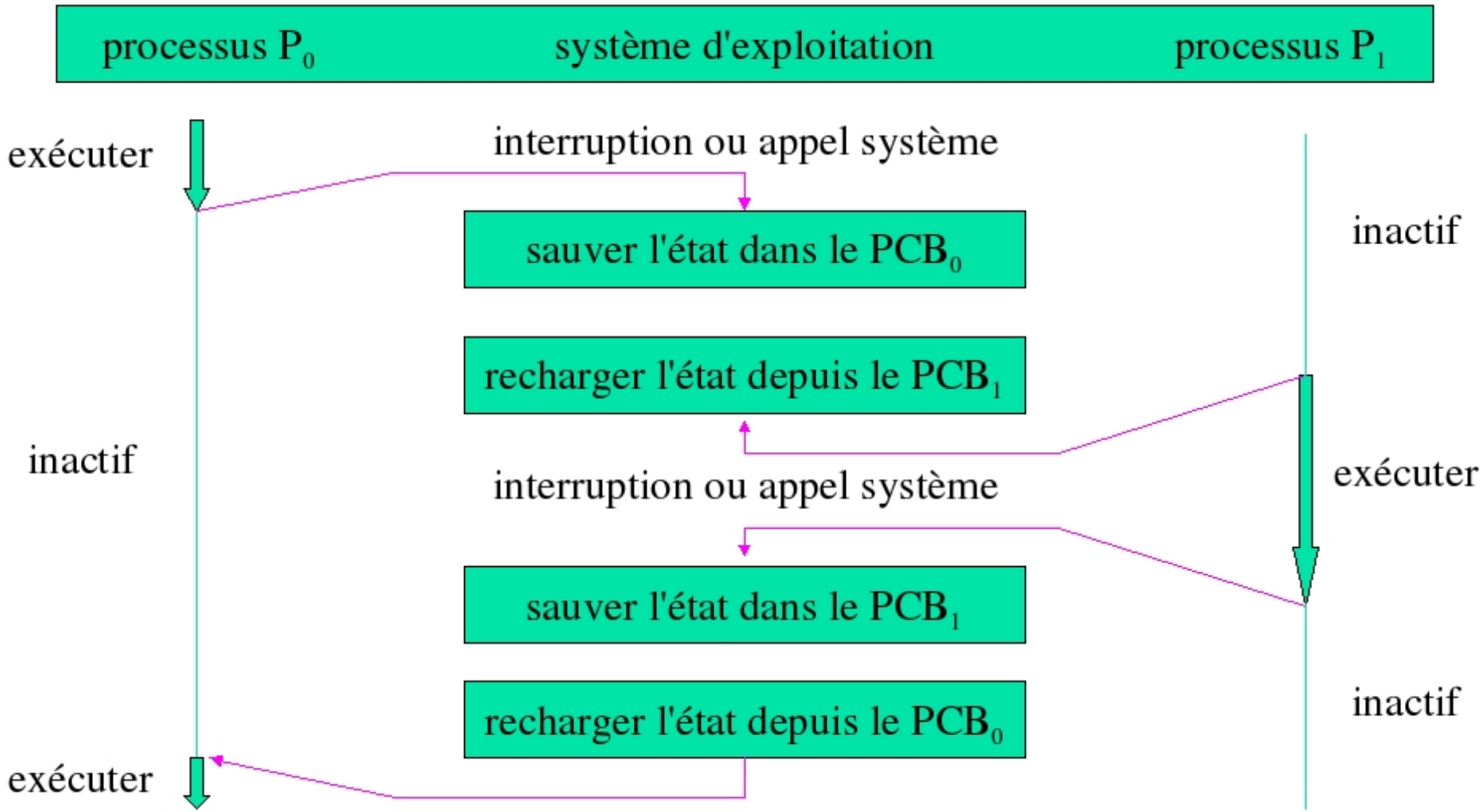
Synchrone

- involontaire : le processeur est alloué à un autre processus
  - interruption d'horloge
- volontaire : un processus se bloque (appel système sleep())

Asynchrone

- cas d'une interruption matérielle
- la commutation est généralement prise en charge par le matériel

Changement de contexte illustré



Troisième partie

Gestion des processus et ordonnancement

- Implémentation des processus
- Implantation des processus Linux 2.6
  - Descripteur de processus
  - Processus légers
  - Le cas de l'architecture x86
- Ordonnancement des processus (scheduling de l'UC)
  - Introduction
  - Algorithmes de scheduling
  - Étude de cas : SOLARIS, HP-UX, 4.4BSD et Linux 2.6

Linux 2.6 : Descripteur de processus (task_struct)

- le descripteur de processus (**task_struct**) contient toutes les informations relatives à un processus

État d'un processus (champ state)

- TASK_RUNNING le processus est prêt à être exécuté ou en cours d'exécution.
- TASK_INTERRUPTIBLE le processus est suspendu en attendant qu'une condition soit réalisée :
  - une interruption matérielle,
  - libération d'une source que le processus attend
  - réception d'un signal, . . .
- TASK_STOPPED processus arrêté à cause d'un signal SIGSTOP, SIGTSTP, SIGTTIN ou SIGTTOU.
- TASK_ZOMBIE l'exécution du processus est terminée alors que son père n'a pas encore utilisé un appel système de type wait() pour obtenir des informations à propos du processus mort.
- . . .

Linux 2.6 : task_struct

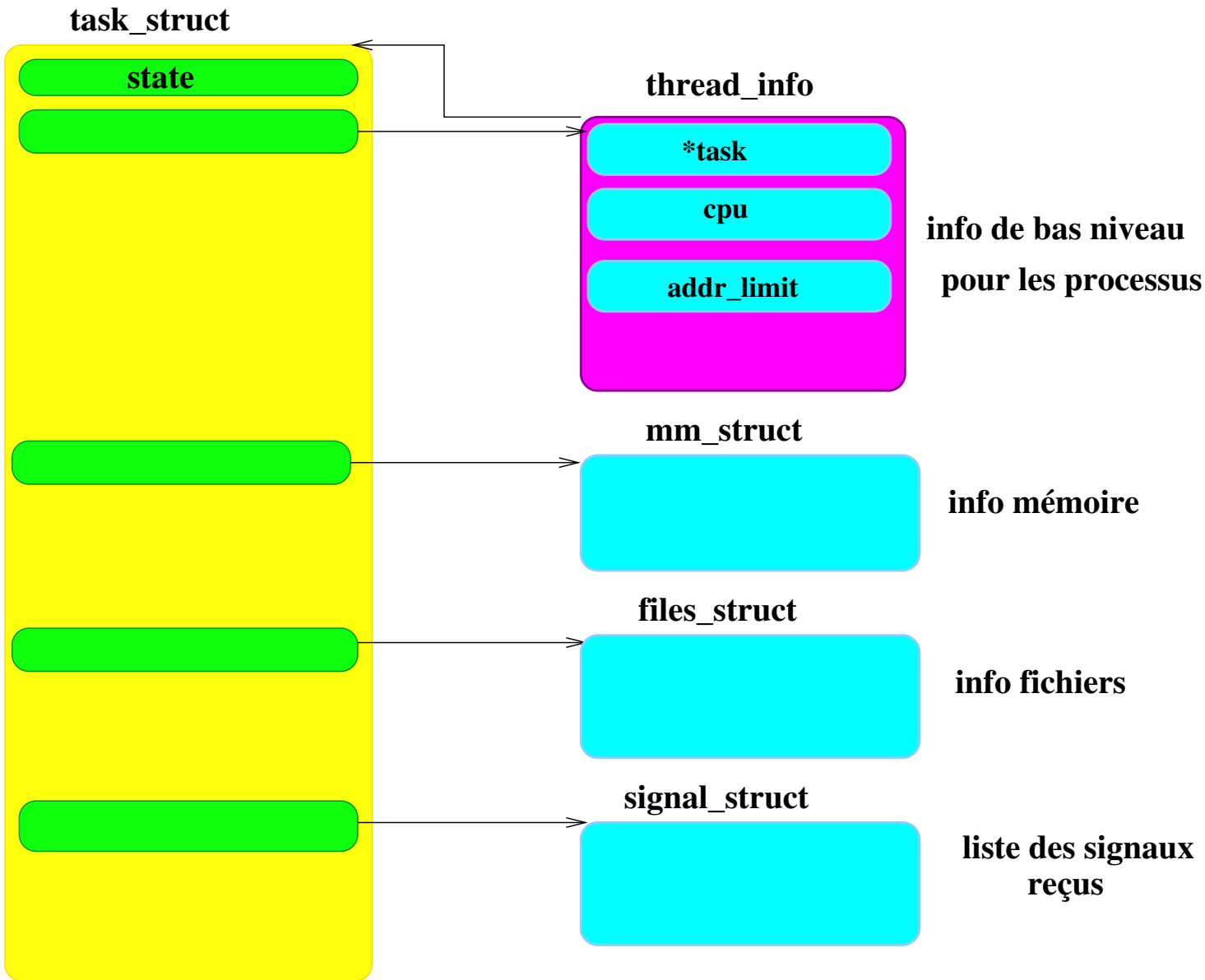
Lightweight processes (processus légers)

- Un processus léger correspond à un thread
- Un groupe de threads est un ensemble de processus légers qui implante une même application multithreadée :
  - partagent l'espace d'adressage
  - agissent comme un tout vis-à-vis de certains appels système : getpid(), kill(), ...
  - peuvent être schedulés séparément
  - chacun son pid, mais un seul pid de groupe : le pid du premier thread du groupe

Identification d'un processus

- pid : identifiant du processus de 0 à 32767 $=$ PID_MAX_DEFAULT -1
  /proc/sys/kernel/pid_max
- tgid (thread group leader pid) : pid du premier processus léger du groupe
  getpid() retourne tgid et non pas pid (POSIX compatible)

Linux 2.6 : task_struct

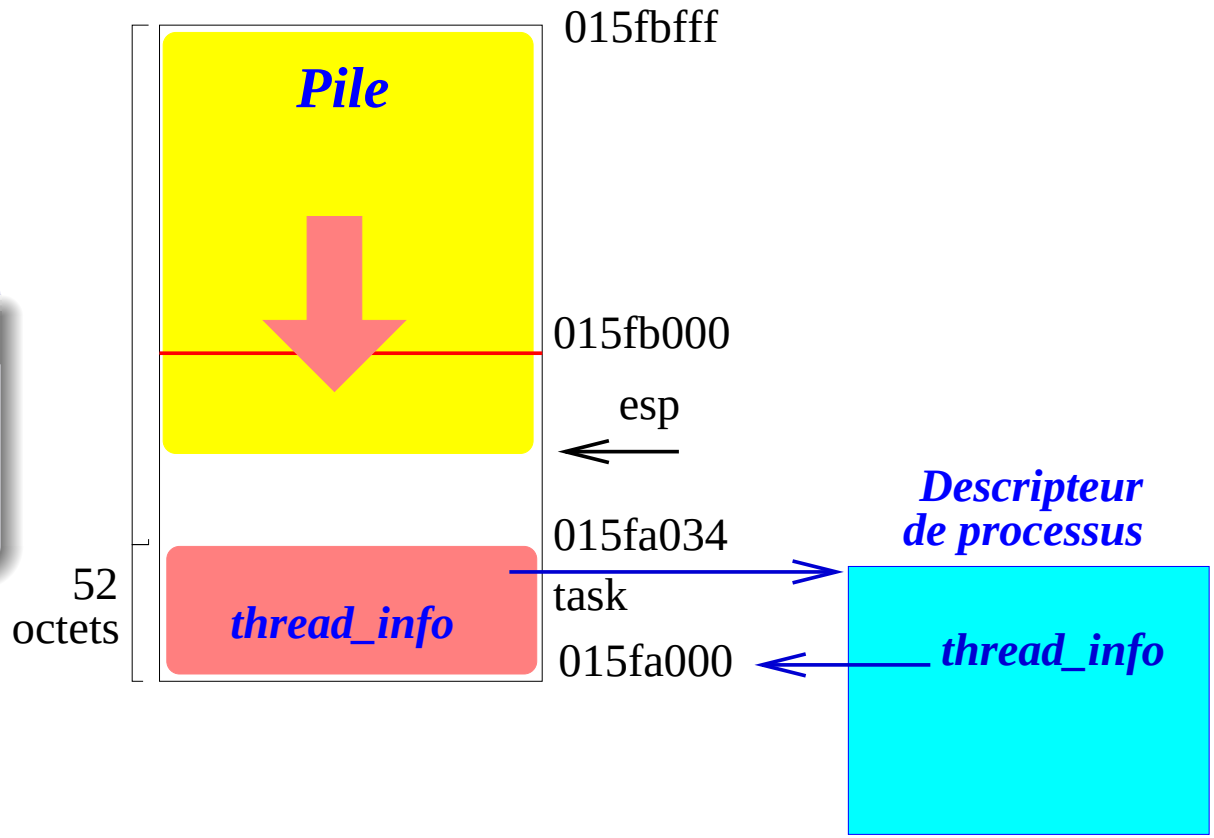


Linux 2.6 : task_struct

Cas du x86 (1/2)

thread_union

```
union thread_union {
struct thread_info thread_info;
unsigned long stack[2048];
}
```



Linux 2.6 : task_struct

Cas du x86 (2/2)

- À partir de esp, le noyau peut trouver, pour le processus en cours :

- l'adresse de la structure "thread_info"

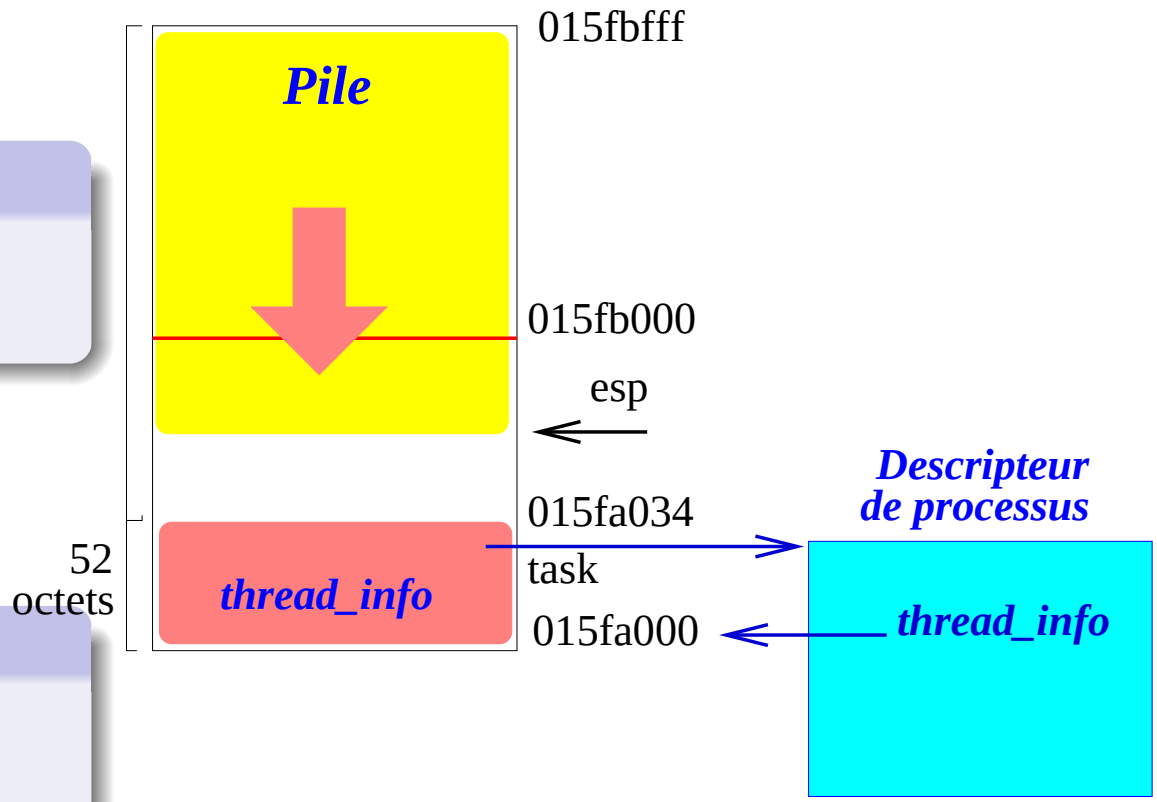
current_thread_info

```
movl $0xffffe000,%ecx
andl %esp,%ecx
movl %ecx,p
```

- l'adresse de son descripteur

current

```
movl $0xffffe000,%ecx
andl %esp,%ecx
movl (%ecx),p
```



Troisième partie

Gestion des processus et ordonnancement

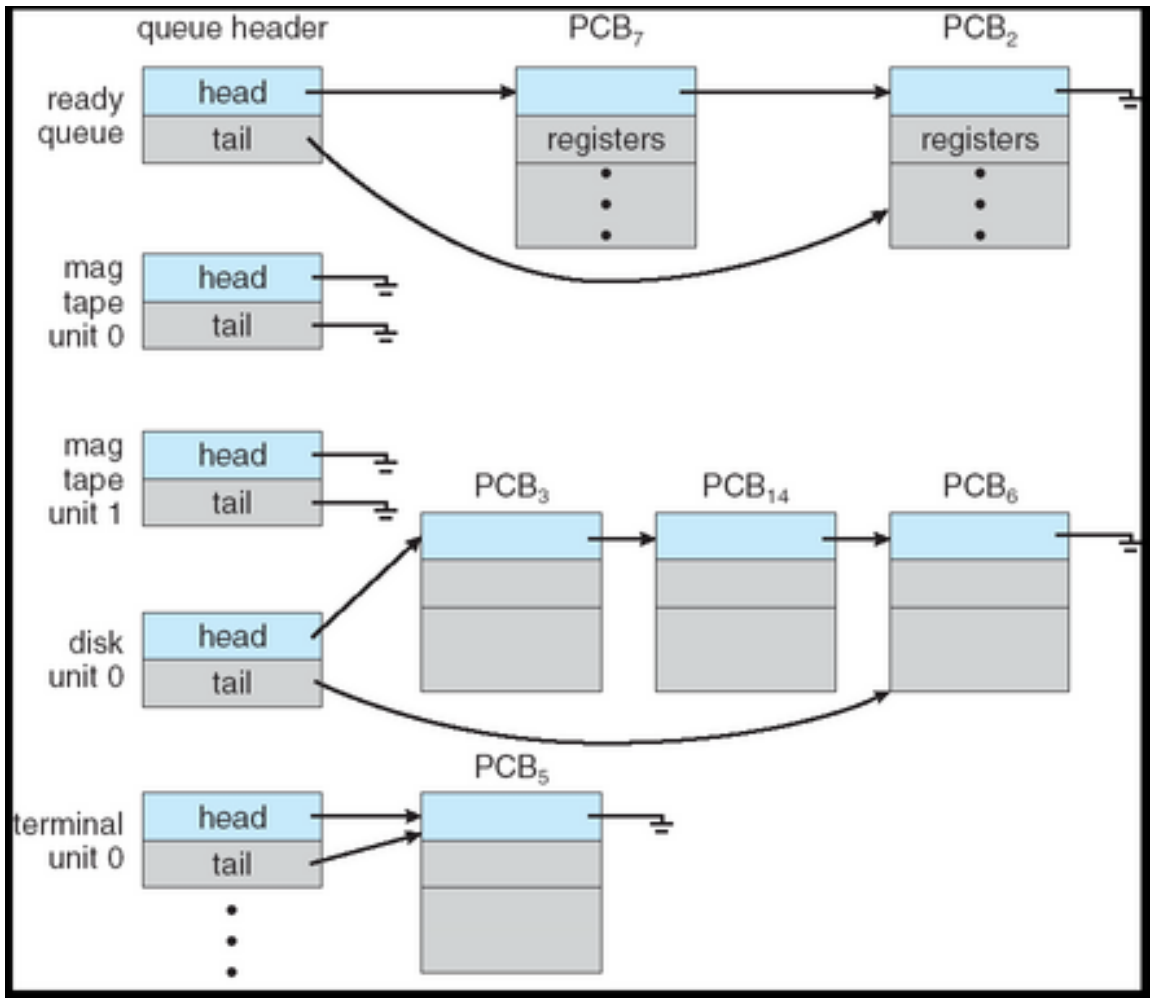
- Implémentation des processus
- Implantation des processus Linux 2.6
  - Descripteur de processus
  - Processus légers
  - Le cas de l'architecture x86
- Ordonnancement des processus (scheduling de l'UC)
  - Introduction
  - Algorithmes de scheduling
  - Étude de cas : SOLARIS, HP-UX, 4.4BSD et Linux 2.6

Ordonnancement : introduction

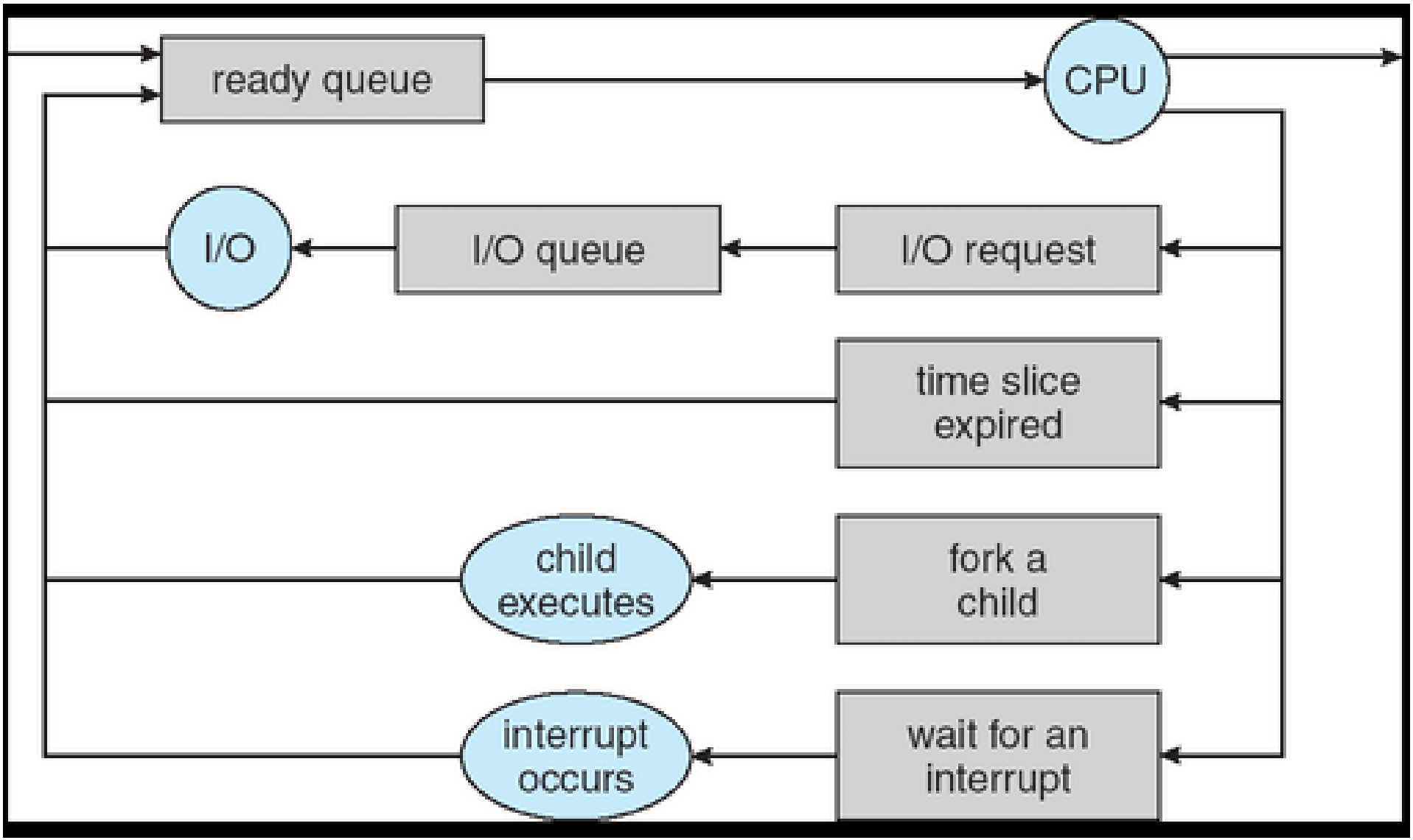
- Optimiser l'utilisation de l'UC :
  - multiprogrammation et temps partagé
  - multiprogrammation : plusieurs processus par processeur
  - utiliser l'UC pendant qu'un processus est bloqué : recouvrement des E/S
- Problème d'allocation des ressources : un seul processeur
- la partie du SE qui s'occupe de l'allocation du processeur s'appelle l'ordonnanceur ou le scheduler
- Le scheduling nécessite une commutation de contexte qui doit être la moins lourde possible
- Le module qui donne le contrôle de l'UC au processus sélectionné par le scheduler s'appelle le dispatcher :
  - commutation de contexte (en mode noyau suite à l'interruption d'horloge)
  - commutation au mode utilisateur
  - branchement au bon emplacement dans le programme utilisateur pour redémarrer ce programme
- Le dispatcher doit être le plus rapide possible $\Rightarrow$ latence de dispatching

Files d'attente d'ordonnancement (1/2)

- une file d'attente des travaux (job queue) contient l'ensemble des processus du système.
- une file d'attente des processus prêts (Ready queue) contient l'ensemble des processus résidents en mémoire principale prêts à être exécutés
- les files d'attente des périphériques d'E/S (Device queues), une file (par périphérique d'E/S) contient l'ensemble des processus en attente sur ce périphérique.



Files d'attente d'ordonnancement (2/2)



Les ordonnanceurs

l'ordonnanceur à long terme (long-term scheduler) ou l'ordonnanceur des travaux (job scheduler)

- choisit lequel parmi les processus de la file d'attente des travaux à mettre dans la file des processus prêts
- invocation non fréquente (secondes, minutes) et donc peut être lent

l'ordonnanceur à court terme (short-term scheduler) ou l'ordonnanceur de l'UC (CPU scheduler)

- choisit un processus parmi ceux de la file d'attente des processus prêts à être exécuter
- invoqué fréquemment (millisecondes) et donc doit être rapide

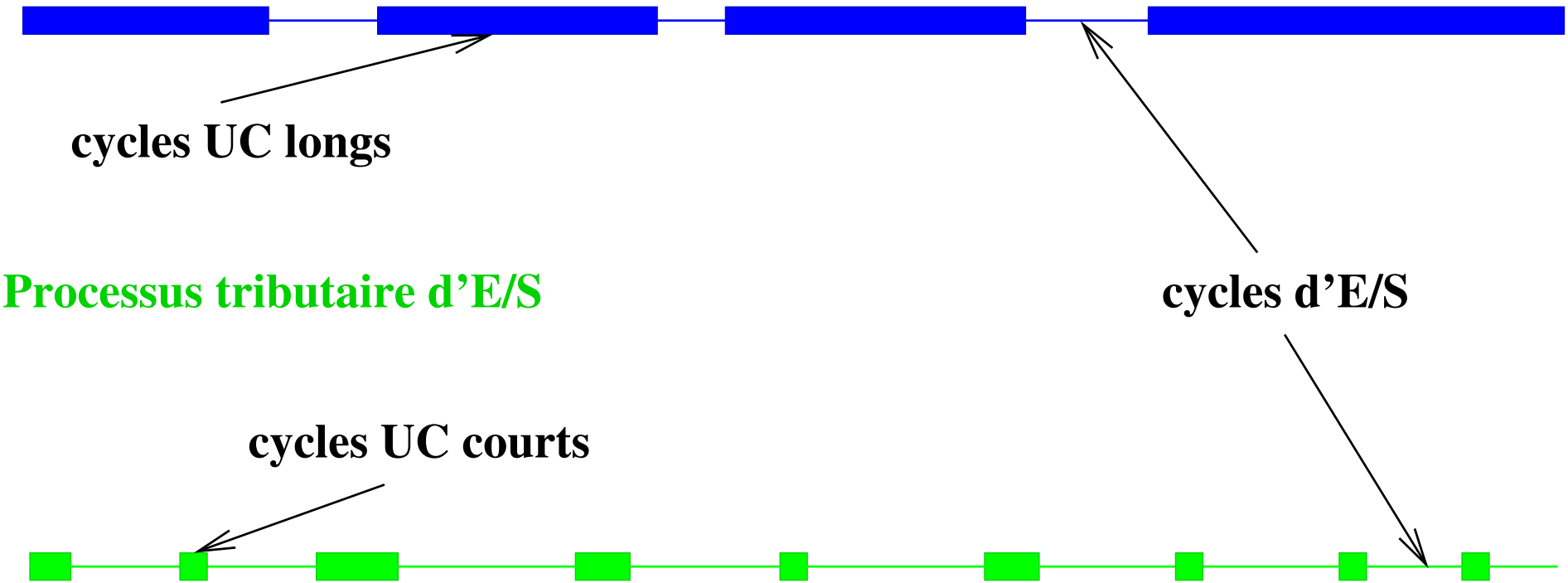
Ordonnancement : quand ?

1. “en cours d’exécution” $\rightarrow$ “en attente” (attente d’un évènement)
2. “en cours d’exécution” $\rightarrow$ “prêt” (suite à une interruption)
3. “en attente” $\rightarrow$ “prêt” (terminaison d’une E/S par exemple)
4. Terminaison d’un processus

- 1 et 4 correspondent à un ordonnancement sans réquisition ou non-preemptive scheduling
- 2 et 3 correspondent à un ordonnancement avec réquisition ou preemptive scheduling
  $\Rightarrow$ problèmes d’exclusion mutuelle à résoudre

Comportement des processus (1/2)

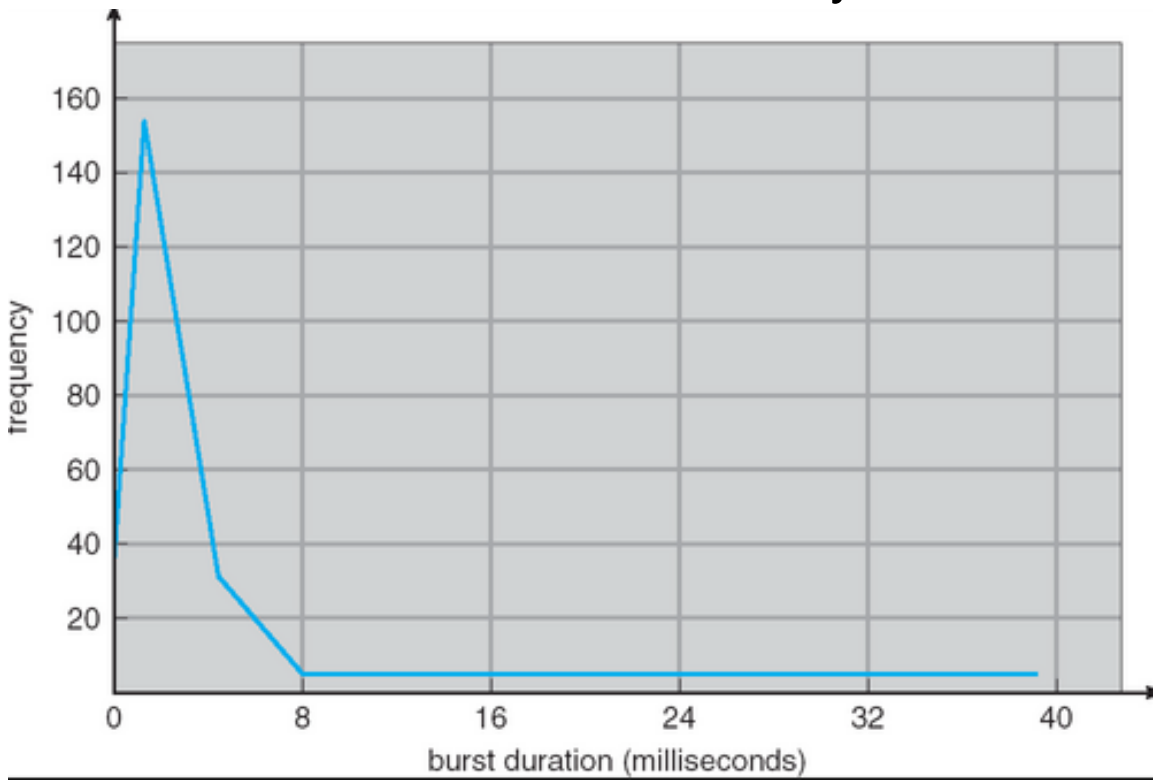
Processus tributaire de l'UC



- Avoir une idée sur le type de processus sur un système donné permet de choisir l'algorithme d'ordonnancement le plus approprié

Comportement des processus (2/2)

Distribution de la durée d'un cycle CPU



- Les cycles CPU ont tendance à être plus fréquents et plus courts dans un système interactif.

Catégories d'algorithmes d'ordonnancement

- systèmes de traitement par lots
  - un algorithme sans réquisition (non préemptif) est suffisant
  - les commutations de contextes sont réduites ce qui augmente les performances
- systèmes interactifs
  - un algorithme avec réquisition est nécessaire
- systèmes temps réel
  - la réquisition est parfois inutile

Métriques

- Utilisation de l'UC : taux d'utilisation à maintenir au maximum possible
- Capacité de traitement : quantité de processus terminés par unité de temps
- Délai de rotation : l'intervalle entre le moment de soumission d'un travail jusqu'au moment de sa terminaison
- Temps d'attente : la somme du temps passé à attendre dans la file d'attente des processus prêts
- Temps de réponse : temps écoulé entre la soumission d'une requête et l'arrivée de la première réponse
- Equité : un partage équitable entre processus

Objectifs de l'ordonnancement

Tous les systèmes

- application de la politique
- équité
- maximiser la capacité de traitement

Systèmes interactifs

- minimiser le temps de réponse
- proportionnalité (répondre aux attentes des utilisateurs)

Systèmes de traitement par lots

- minimiser le délai de rotation
- maximiser l'utilisation du processeur

Systèmes temps réel

- respecter les délais (éviter la perte de données)
- prévisibilité : éviter la dégradation de la qualité dans les systèmes multimédia par exemple

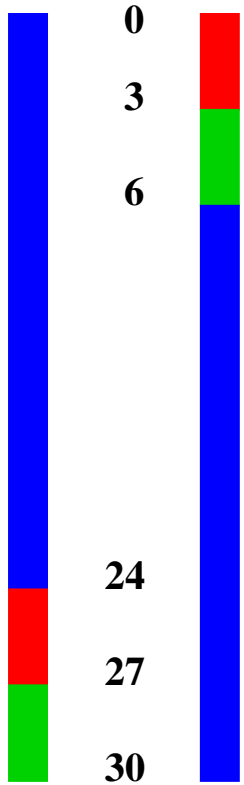
Algorithmes de scheduling

Premier arrivé, premier servi (First Come First Served)

- simple à réaliser à l'aide d'une file d'attente FIFO des PCBs
- l'UC est alloué au premier processus de la file d'attente
- pas de réquisition

Processus	Temps du cycle
P_1	24
P_2	3
P_3	3

	temps d'attente
P_1	0
P_2	24
P_3	27
Moyenne	**17**



	temps d'attente
P_2	0
P_3	3
P_1	6
Moyenne	**3**

Algorithmes de scheduling

Premier arrivé, premier servi

- le temps moyen d'attente peut varier beaucoup si le temps des cycles UC varie trop
- pénalise les processus d'E/S par rapport aux processus de traitement
- pas adapté aux systèmes interactifs
- et si on servait en premier le processus avec le prochain cycle le plus court ?

Algorithmes de scheduling

Le programme le plus court d'abord

(Shortest-Job-First)

- Associer à chaque processus la longueur de son prochain cycle UC
- L'UC est assigné au processus qui a le prochain cycle UC le plus court
- SJF est optimal : minimise le temps d'attente moyen. À condition que tous les travaux (cycles UC) sont disponibles en même temps
- Problème : pénalise les gros travaux (les processus tributaires d'UC)
- Comment peut-on connaître le temps des cycles prochains ?

Algorithmes de scheduling

Scheduling avec des priorités

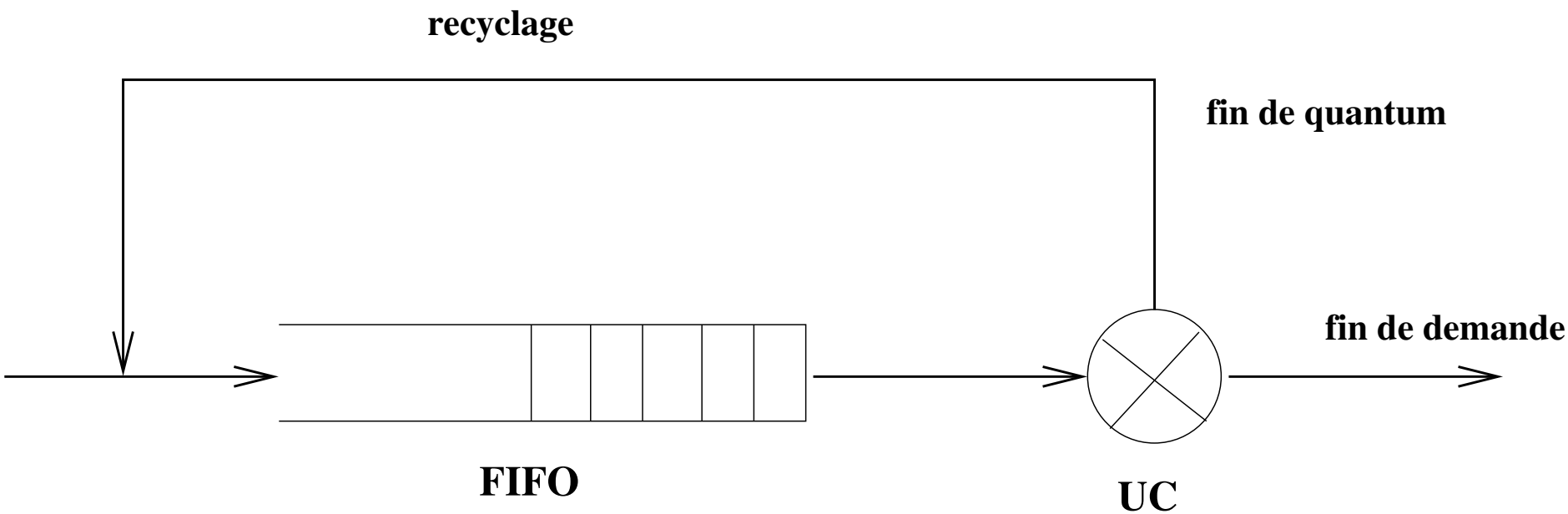
- associer à chaque processus une priorité
- l'UC est allouée à celui qui a la plus haute priorité
- peut être avec ou sans réquisition

- Problème : la famine. Une solution : le viellissement(AGING)
- Cas particulier = FCFS : même priorité pour tous
- Cas particulier = SJF : la priorité d'un processus est inversement proportionnelle au temps de son prochain cycle.

Algorithmes de scheduling

Le tourniquet (round-robin)

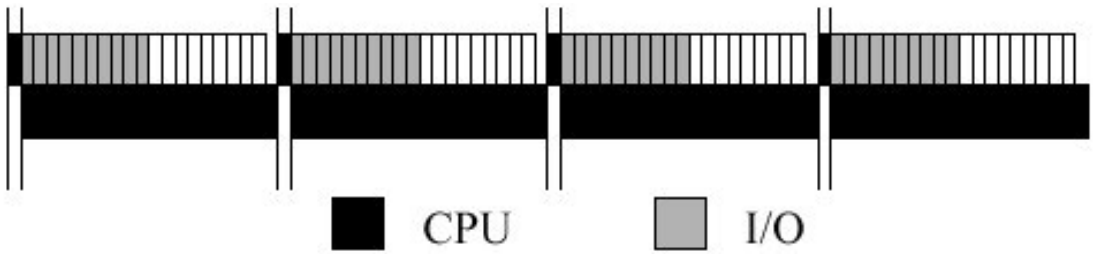
- L'UC est allouée par tranches de temps appelés quantum
- Si au bout du quantum, un processus n'a pas terminé, il est interrompu et inséré à la fin de la file des processus prêts
- Implémentation : une file FIFO avec recyclage des demandes



Algorithmes de scheduling

Le tourniquet (round-robin) : choix du quantum

Le quantum est grand

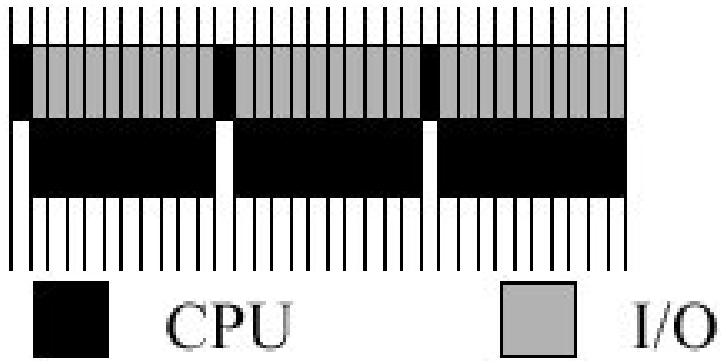


- processus 1 pénalisé (tourne à 11/21)
- mauvaise utilisation du périphérique d'E/S (10/21 au lieu de 10/11)
- en général : un quantum long augmente le débit global du système
- Remarque : lorsque $q = \infty$, RR devient FCFS

Algorithmes de scheduling

Le tourniquet (round-robin) : choix du quantum

Le quantum est petit

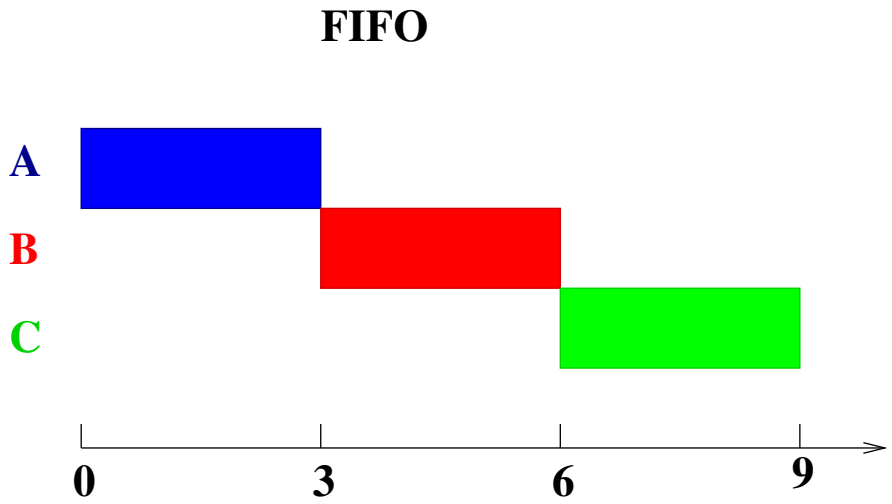


- le processus 2 est interrompu 9 fois pour rien
- en général : un quantum très court favorise les traitements interactifs mais trop court diminue les performances à cause de l'overhead dû aux changements de contexte

- RR suppose que tous les processus sont équivalents chacun reçoit une part égale de CPU. Ceci produit des mauvais résultats

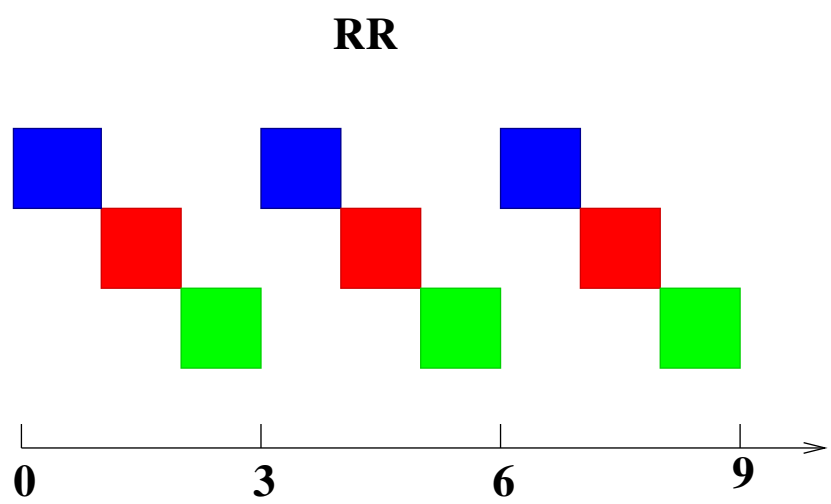
Algorithmes de scheduling

Le tourniquet (round-robin)



Délai de rotation moyen

(3+6+9)/3=6



Délai de rotation moyen

(7+8+9)/3=8

- Le délai de rotation moyen est long dans RR
- Solution : introduire des priorités
  - choisir le processus avec la priorité la plus élevée
  - faire du RR pour les processus à priorité égale

Algorithmes de scheduling

Multilevel Scheduling

- Les processus sont divisés en classes de processus
  - Critères : temps de réponse, taille mémoire, priorité ou type de processus . . .
- Chaque classe de processus est assignée à une file d'attente séparée
- Chaque classe est schedulée avec un algorithme différent
- On peut associer une priorité différente à chaque file
- Un processus est toujours associé à la même file d'attente

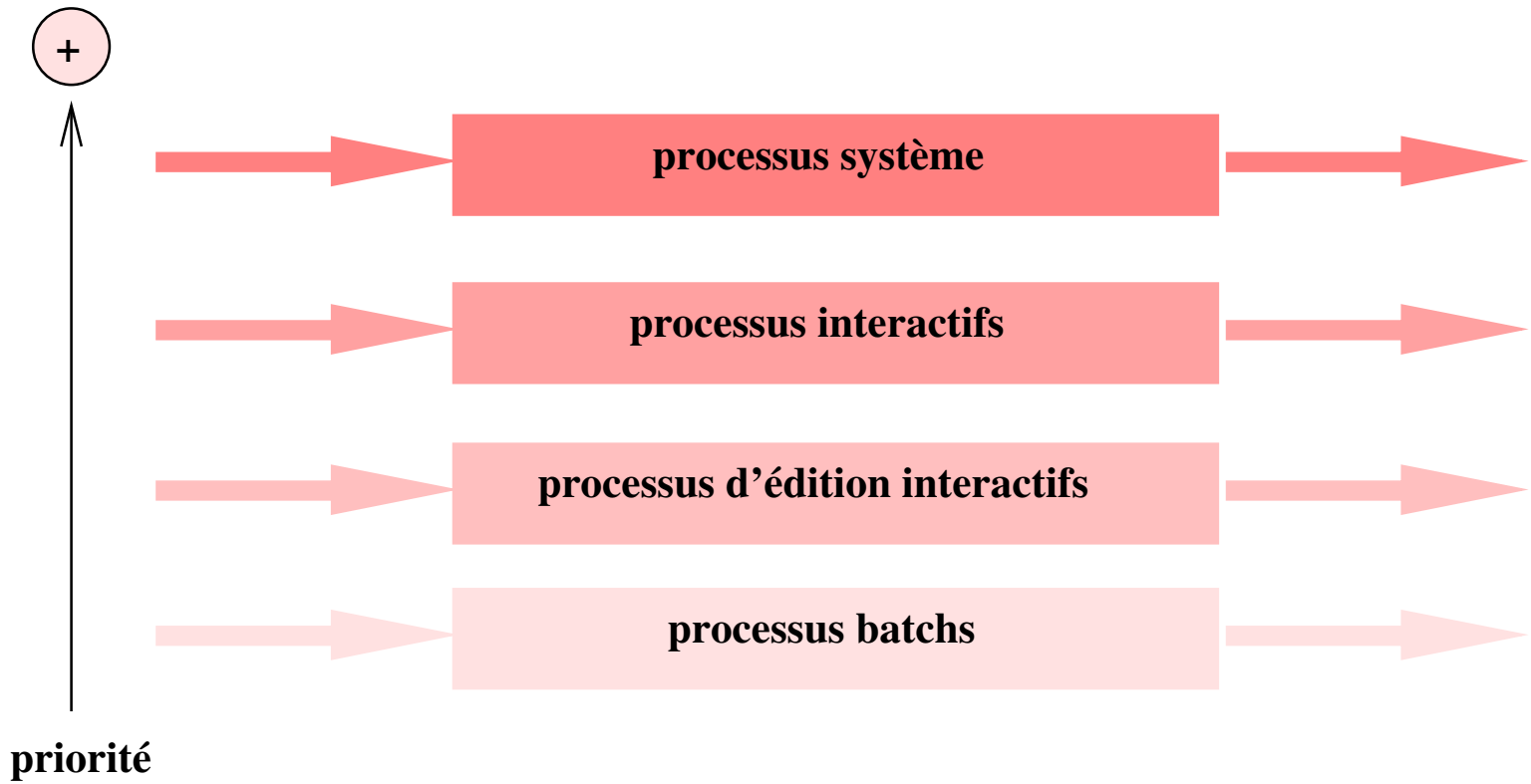
Exemple

- 2 classes :
  - RR pour les processus interactifs
  - FCFS pour les processus batch
- Un scheduling entre les files d'attente est nécessaire :
  - avec réquisition et priorités fixes.Exemple : la 1ère classe est plus prioritaire
  - associer un quantum différent par file. Exemple : 80% du temps CPU à la 1ère classe et 20% pour la 2ème

Algorithmes de scheduling

Feedback Multilevel Scheduling

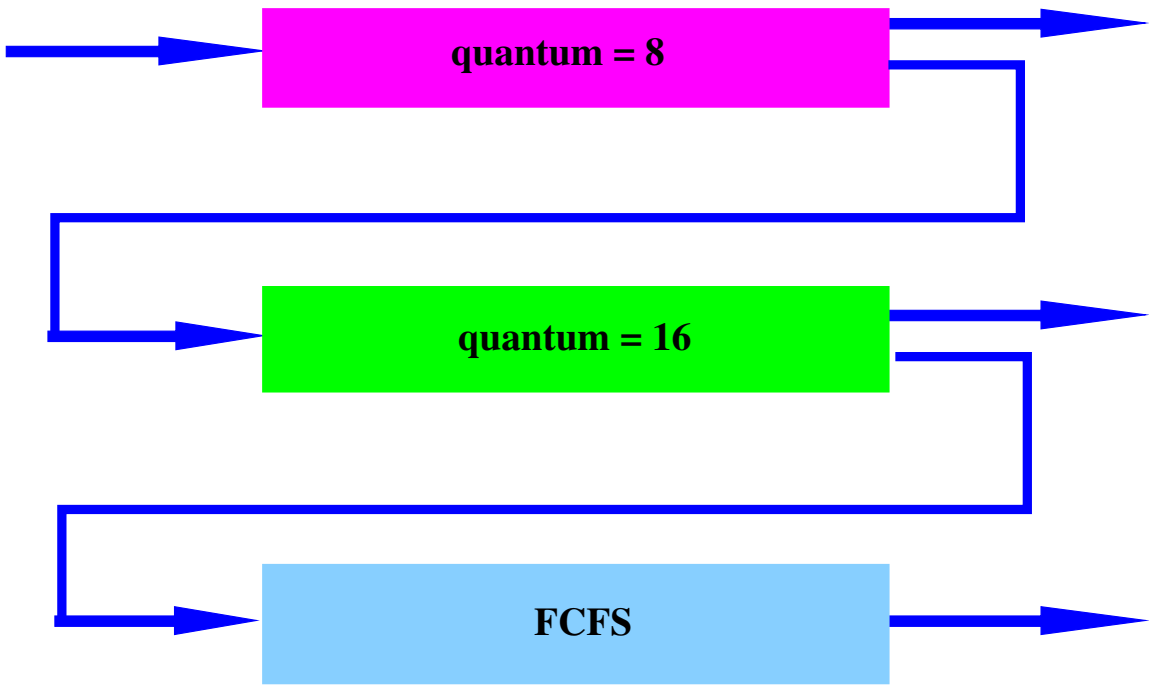
Exemple 2



Algorithmes de scheduling

Feedback Multilevel Scheduling

- Ici, les processus peuvent changer de files, ce qui permet de séparer les processus ayant des caractéristiques différentes en termes de cycles UC
- Si un processus a des cycles UC longs, le déplacer dans une file d'attente moins prioritaire $\Rightarrow$ les processus interactifs finissent par avoir la priorité la plus élevée
- Si un processus attend longtemps, il est déplacé dans une file plus prioritaire

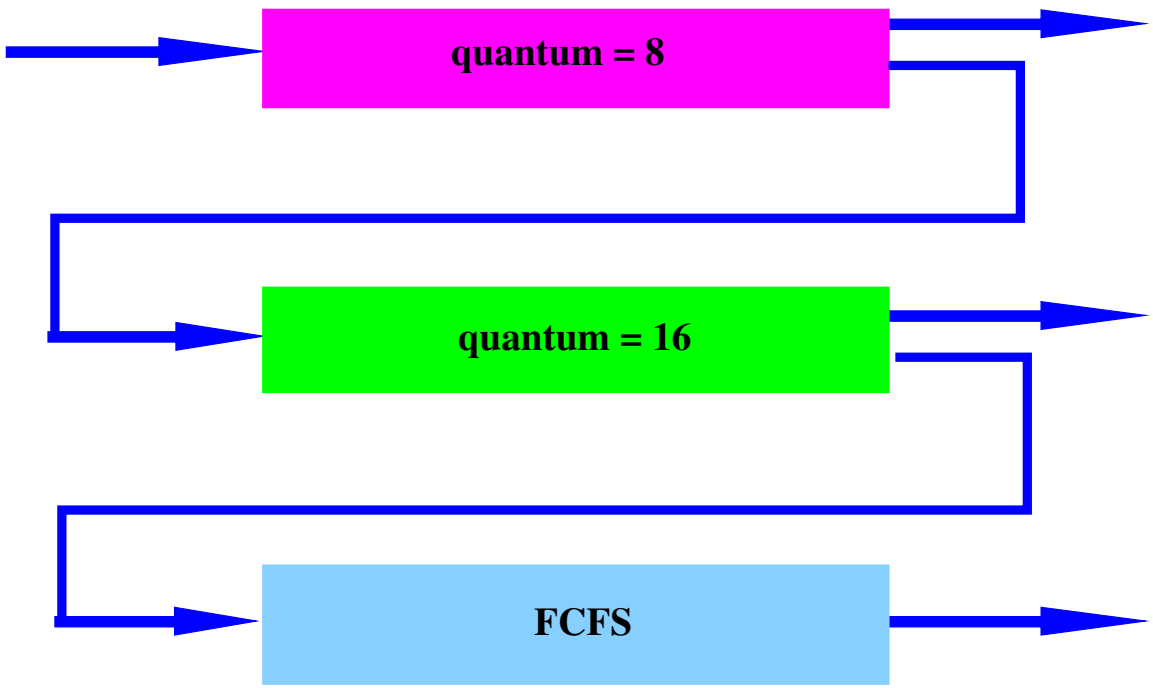


Algorithmes de scheduling

Feedback Multilevel Scheduling

est défini par

- le nombre de files d'attente
- l'algorithme de scheduling pour chaque file
- la méthode utilisée pour déterminer le moment de changer la priorité d'un processus



Troisième partie

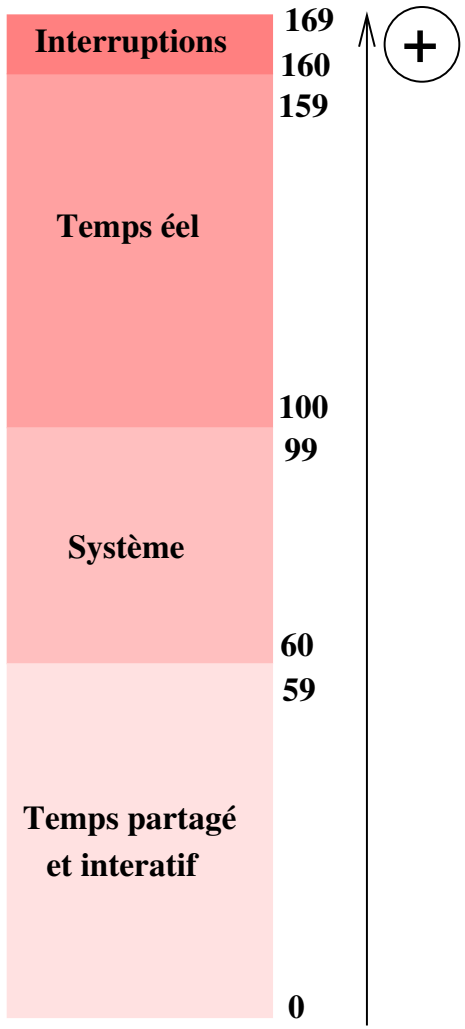
Gestion des processus et ordonnancement

- Implémentation des processus
- Implantation des processus Linux 2.6
  - Descripteur de processus
  - Processus légers
  - Le cas de l'architecture x86
- Ordonnancement des processus (scheduling de l'UC)
  - Introduction
  - Algorithmes de scheduling
  - Étude de cas : SOLARIS, HP-UX, 4.4BSD et Linux 2.6

Ordonnancement SOLARIS

3 classes d'ordonnancement

- Timesharing and interactive (TS & IA) : RR avec priorité (plus de priorité aux processus les plus interactifs)
- System(SYS) : FCFS avec préemption et priorités fixes
- Realtime (RT) : RR avec priorité où un processus (RT) a une priorité fixe durant sa vie



Ordonnancement SOLARIS

Dispatch table : processus interactifs

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

- time quantum : la longueur par défaut du quantum assigné au processus
- time quantum expired : la nouvelle priorité pour un processus qui utilise la totalité de son quantum
- return from sleep : la nouvelle priorité pour un processus qui se bloque avant d'utiliser la totalité de son quantum

Ordonnancement HP-UX

2 types d'ordonnanceurs :

Temps-réel (RealTime ou RT)

- FIFO ou RR
- priorités fixes, ne peuvent pas être changées par le noyau
- sans réquisition : un processus s'exécute jusqu'à sa fin ou se bloquer

Temps-partagé (TimeSharing) :

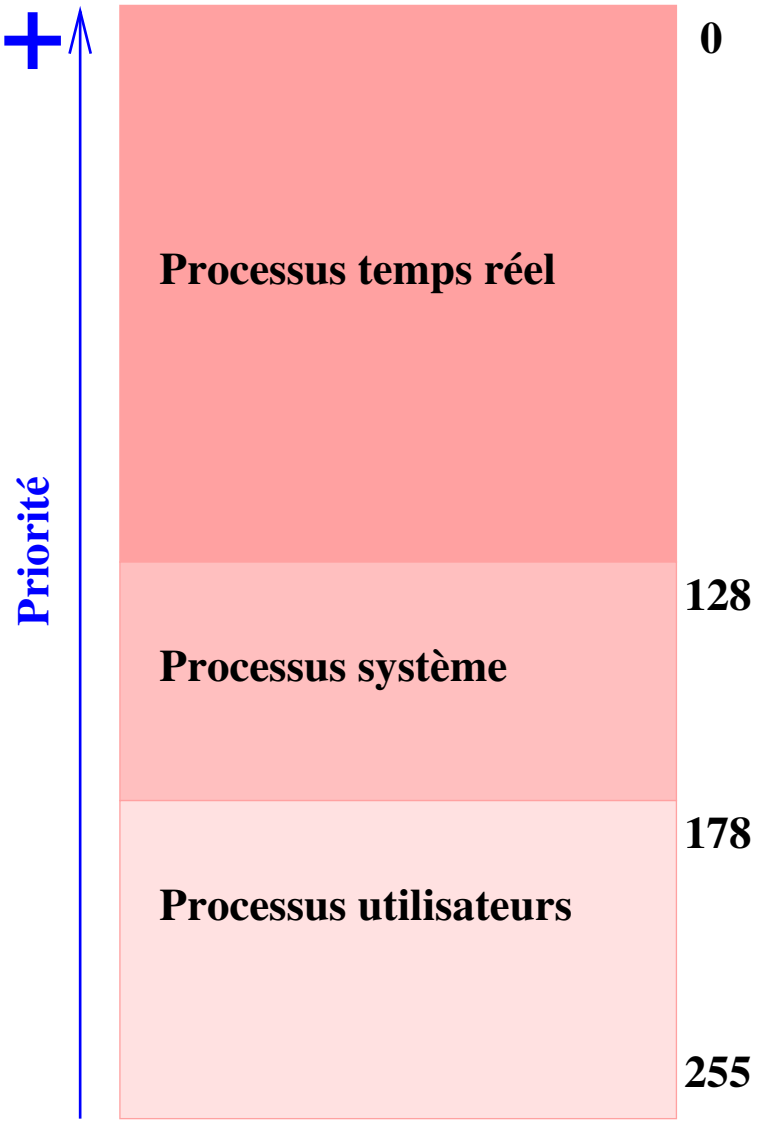
- RR
- la valeur de la priorité augmente (priorité diminue) avec l'utilisation de l'UC et diminue en attendant
- avec réquisition

Ordonnancement HP-UX

L'ordonnanceur temps partagé

Le noyau différencie en termes de priorité, les processus utilisateurs des processus système. Ces derniers ont une priorité supérieure.

- en mode utilisateur, un processus peut être réquisitionné, arrêté ou même transféré en mémoire secondaire
- en mode noyau, un processus s'exécute jusqu'à se bloquer, une interruption ou se terminer



Ordonnancement 4.4BSD (1/4)

- Un processus a 2 priorités :
  - mode utilisateur p_usrpri $\in$ [PUSER,127] où PUSER=50 et correspond à la priorité attribuée au processus utilisateur le plus prioritaire.
  - mode noyau p_priority $\in$ [0,PUSER[donnant plus de chance à un processus en mode noyau afin qu'il libère dès que possible les ressources système qu'il détient.
- un quantum = 0.1s (valeur empirique)
- la priorité d'un processus est ajustée dynamiquement :

$$p_{usrpri} = PUSER + \frac{p_{cpu}}{4} + 2p_{nice} \quad (1)$$

  - p_{nice} permet à l'utilisateur de moduler sa priorité,
  - p_{cpu} incrémentée toutes les 10 ms et donne une estimation de la consommation UC du processus actif.

$\Rightarrow$ La priorité d'un processus diminue avec sa consommation UC.

Ordonnancement 4.4BSD (2/4)

- toutes les secondes p_{cpu} est réajustée selon la formule :

$$p_{cpu} = \frac{2\ load}{2\ load + 1}\ p_{cpu} + p_{nice} \tag{2}$$

où *load* est une estimation de la charge du système et correspond à la longueur de la file d'attente des processus prêts.

- Lors de la réactivation d'un processus utilisateur en attente (Asleep), l'ordonnanceur réajuste p_{cpu} :

$$p_{cpu} = p_{cpu} \left(\frac{2\ load}{2\ load + 1} \right)^{p_{slptime}} \tag{3}$$

où $p_{slptime}$ comptabilise le temps d'attente du processus

$\Rightarrow$ sert à estomper le passé lointain.

Ordonnancement 4.4BSD (3/4)

Exemple

On considère un seul processus actif et qui consomme toute l'UC. Ce processus consomme T_i ticks à la fréquence de l'horloge pendant la durée i.
$load = 1$
Toutes les secondes, le filtre est appliqué avec la formule suivante
$p_{cpu} = 0.66\ p_{cpu}$:

$$p_{cpu} = 0.66\ T_0$$

$$p_{cpu} = 0.66\ T_1 + 0.44\ T_0$$

$$p_{cpu} = 0.66\ T_2 + 0.44\ T_1 + 0.3\ T_0$$

$$p_{cpu} = 0.66\ T_3 + ... + 0.20\ T_0$$

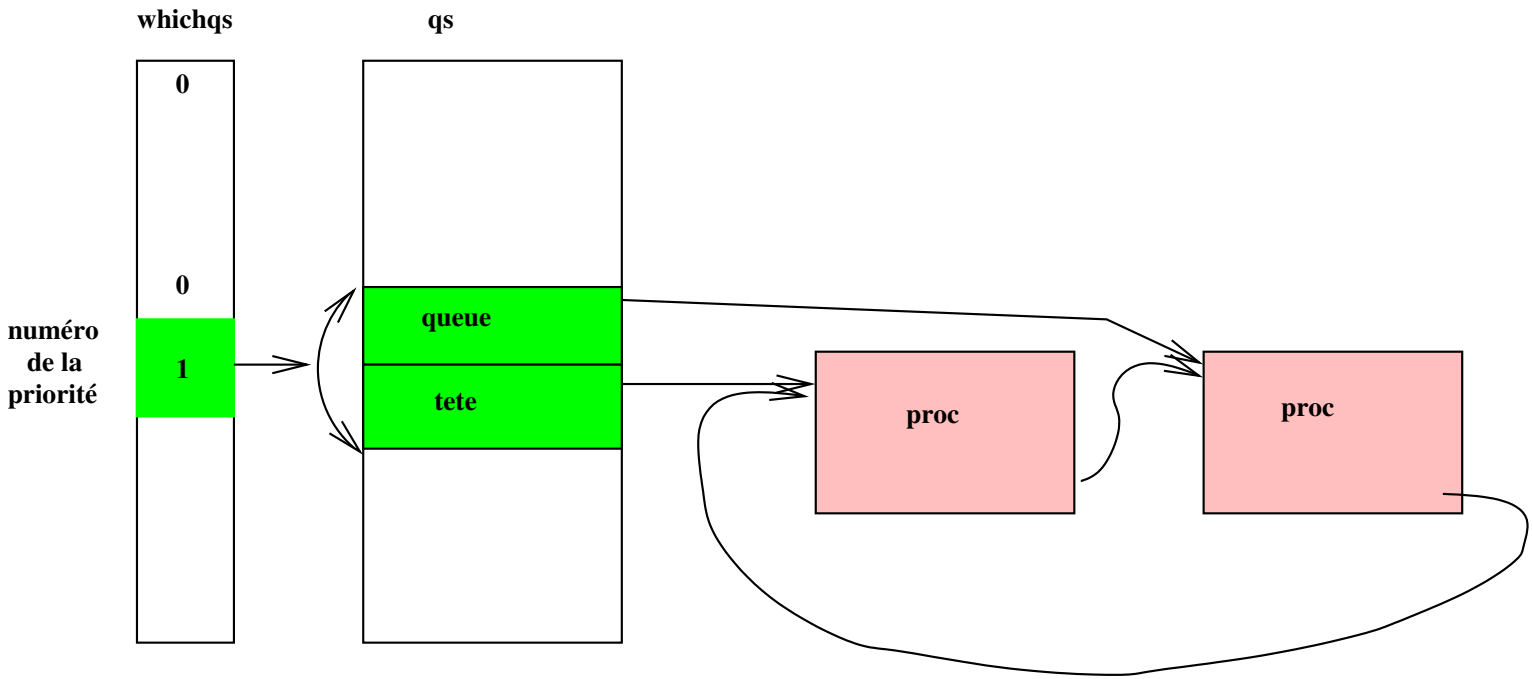
$$p_{cpu} = 0.66\ T_4 + ... + 0.13\ T_0$$

On remarque que l'effet de T_0 s'estompe avec le temps.

Ordonnancement 4.4BSD (4/4)

La runqueue

- L'ensemble des processus dans l'état Runnable constitue la file des processus prêts : la runqueue
- L'implantation du scheduling est réalisée par une liste chaînée des processus associée à chaque groupe de priorités flottantes.
- qs la table des têtes et queues de listes des files
- whichqs une table associée à qs pour indiquer l'occupation de chaque file.



Linux 2.6 : Classes d'ordonnancement

- Processus temps réel (priorité temps réel : 0-99)
  - SCHED_FIFO (FIFO real-time process) : si aucun autre processus n'est plus prioritaire, un processus continue à s'exécuter.
  - SCHED_RR (Round-Robin real-time process) : permet un équité parmi ceux ayant la même priorité.
- Processus conventionnels (priorité conventionnelle : 100-139)
  - SCHED_NORMAL (conventional, time shared process)
  - SCHED_BATCH (traitements par lots : FIFO)
- Processus idle SCHED_IDLE ne s'exécute que si aucun processus n'est prêt des autres classes.

La commande chrt permet de classer un processus dans l'une des 3 classes.

Linux 2.6 : Listes de descripteurs de processus

- liste de tous les processus,
- liste des processus prêts, une liste par niveau de priorité. Usage de la structure prio_array_t :
  - int nr_active : nombre des descripteurs de la liste,
  - unsigned long[5] : bitmap, si un flag à 1 alors la liste correspondante est non vide,
  - struct list_head[140] queue, les têtes des 140 listes de priorité.
- liste des processus en attente, une liste par évènement
  - un flag est utilisé, s'il est à 1, réveiller un seul processus sinon réveiller tous.

Linux 2.6 - Principe de l'ordonnanceur

- Deux domaines séparés de priorités statiques :
  - priorité conventionnelle : 100-139 correspondant au nice de -20 à 19. La valeur du nice peut être changée avec l'appel système nice() ou setpiority()
  - priorité temps réel : 0-99
- Ordonnancement à priorité dynamique : chaque processus a une priorité initiale qui peut diminuer (si tributaire UC) et augmenter (si tributaire E/S)
- Utilisation d'un quantum (timeslice) variable qui peut être consommé en plusieurs fois.
- Recalcul des priorités dynamiques lorsque tous les processus consomment la totalité des timeslices.
- Ordonnancement avec réquisition (preemptive scheduling) :
  - arrivée d'un nouveau processus avec une plus grande priorité
  - timeslice devient nul

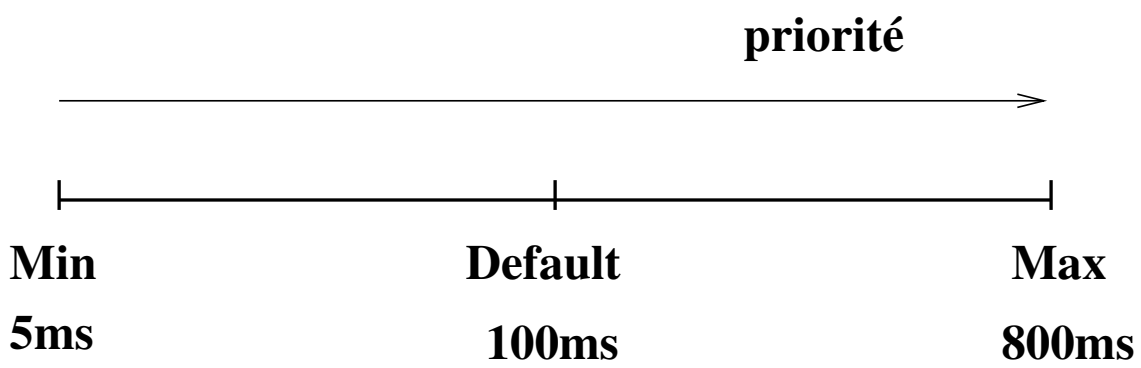
Linux 2.6 : cas des processus conventionnels

Calcul du timeslice

$$timeslice = \begin{cases} (140 - staticP) \times 20 & \text{if } staticP < 120 \\ (140 - staticP) \times 5 & \text{sinon} \end{cases}$$

où *staticP* est la priorité statique du processus.

Priorité statique	Nice value	timeslice (ms)
100	-20	800
110	-10	600
120	0	100
130	+10	50
139	+19	5



Linux 2.6 : cas des processus conventionnels

Priorité dynamique

- le nombre auquel se réfère le scheduler actuellement pour élire le prochain processus à exécuter :

$$dynamicP = max(100, min(staticP - bonus + 5, 139))$$

- $bonus \in [0..10]$: bonus < 5 correspond à une pénalité
- le bonus dépend de l'historique du processus ("average sleep time")
- un processus est considéré comme interactif si

$$dynamicP \leq 3 \times staticP/4 + 28$$

ou

$$bonus - 5 \geq staticP/4 - 28 = interactiveDelta$$

Avg sleep time	bonus
0-100 ms	0
100-200 ms	1
200-300 ms	2
300-400 ms	3
400-500 ms	4
500-600 ms	5
600-700 ms	6
700-800 ms	7
800-900 ms	8
900-1000 ms	9
1 seconde	10

Linux 2.6 : cas des processus conventionnels

Pour éviter la famine et optimiser le recalcul des timeslices :

- processus actifs, qui n'ont pas fini leurs timeslices
- processus expirés, ceux déjà servis

Remarque 1 : les processus temps réels sont toujours placés dans la liste des processus actifs.

Remarque 2 : le scheduler 2.6 trouve le prochain processus à exécuter en un temps constant ($O(1)$) contrairement au 2.4.

Rappel

liste des processus prêts, une liste par niveau de priorité. Usage de la structure prio_array_t :

- int nr_active : nombre des descripteurs de la liste,
- unsigned long[5] : bitmap, si un flag à 1 alors la liste correspondante est non vide,
- struct list_head[140] queue, les têtes des 140 listes de priorité.