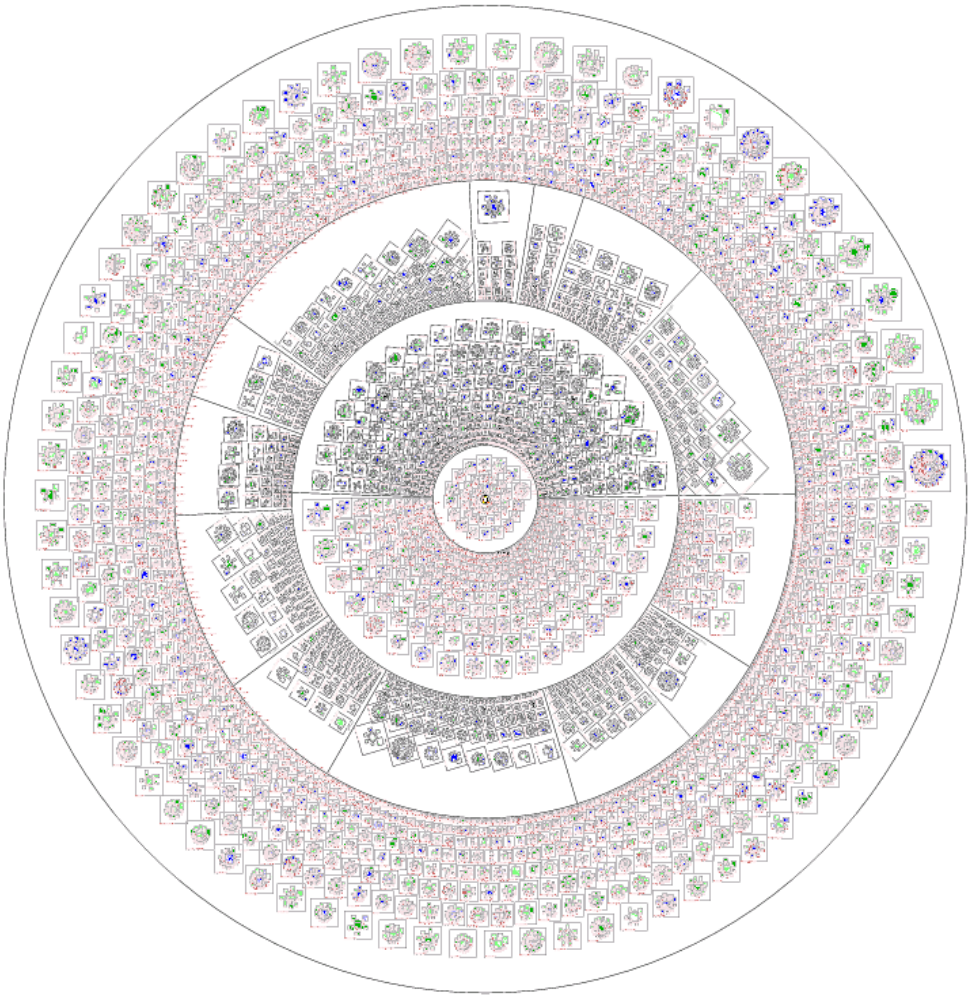
Université de Poitiers
Antenne de Niort

Master Sécurité des
Systèmes d'Informations

Processus et Ordonnancement



dans le noyau Linux

Julien VEHENT

M1SSISI – 2005/2006

Sommaire

1. Introduction

Processus et ordonnancement sont la clé de voûte des systèmes à temps partagés. L'illusion de l'exécution simultannée est obtenue grâce, d'une part, à une représentation efficace des programmes en mémoire et, d'autre part, à des algorithmes d'ordonnancement puissants. Le noyau Linux, principalement développé et maintenu par Linus Thorvald, possède ces deux caractéristiques, faisant de lui un des plus performants systèmes à temps partagé à ce jour.

Nous commencerons par une description de la représentation des processus en mémoire et dans le pseudo système de fichiers « /*proc* ». Nous verrons comment Linux gére les états des processus et comment il stocke les informations de chacun.

Ensuite, nous nous interesserons à la politique d'ordonnancement de Linux (*le « scheduler »*). Nous étudierons en détail l'algorithme utilisé dans la version 2.4 du noyau ainsi que les appels systèmes qui permettent à l'ordonnanceur de communiquer avec les différentes entités du système d'exploitation.
L'objectif de ce chapitre est de montrer comment Linux choisi un processus plutôt qu'un autre au cours de l'exécution du système.

La version 2.6 du noyau a apporté des améliorations significatives à l'algorithme d'ordonnancement, particuliérement en ce qui concerne le choix du niveau de préemptivitée. Ces aspects seront abordés dans le troisième chapitre.

2. Gestion des processus

Linux kernel 2.4.29 {ftp.kernel.org}, architecture 32bits

Linux représente chaque processus dans une structure de données nommée « *task_struct* ». Le nombre maximum de processus qu'un système Linux peut gérer dépend de la quantité de mémoire dont il dispose. Ce paramètre est géré par la fonction « *fork_init()* » dans *kernel/fork.c :*

```
void __init fork_init(unsigned long mempages)
{
      /*
       * The default maximum number of threads is set to a safe
       * value: the thread structures can take up at most half
       * of memory.
       */
      max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 8;
```

Ce qui, sur une architecture 32bits, correspond généralement a la fonction « num_physpages/4 » soit, sur une machine équipée de 512Mo de mémoire vive, 32k processus.

On peut consulter cette valeur via le fichier *threads-max* :

```
$ cat /proc/sys/kernel/threads-max
16383
```

Valeur que l'on peut changer à la volée, sans interrompre le système:

```
# echo 100000 > /proc/sys/kernel/threads-max
$ cat /proc/sys/kernel/threads-max
100000
```

Dans un système Linux, aucun processus n'est indépendant. Tous processus, à l'exception de *init*, a un pére. Un nouveau processus n'est pas créé, il est cloné de son pére. Libre à lui par la suite de continuer son exécution avec les données dont il a hérité (*« fork »*) ou d'exécuter un nouveau programme (*« execve »*). Les liens parentaux entre les processus sont facilement visualisables:

```
$ pstree
init─┬─apache2─────9*[apache2]
     ├─atd
     ├─bash
     ├─cron
     ├─events/0─┬─aio/0
     │          ├─kacpid
     │          ├─kblockd/0
     │          ├─khelper
     │          ├─2*[pdflush]
     │          ├─xfsdatad/0
     │          └─xfslogd/0
     ├─5*[getty]
     ├─inetd
     ├─khubd
     ├─2*[kjournald]
     ├─klogd
     ├─kseriod
     ├─ksoftirqd/0
     ├─kswapd0
     ├─master─┬─pickup
     │        └─qmgr
     ├─md0_raid1
     ├─md1_raid1
     ├─md2_raid1
     ├─md3_raid1
     ├─md4_raid1
     ├─mdadm
     ├─mysqld_safe─────mysqld_safe─┬─logger
     │                             └─mysqld
     ├─named
     ├─screen─────bash─────aircrack
     ├─sshd─────sshd─────sshd─────bash─────pstree
     ├─syslogd
     ├─vsftpd
     ├─xfsbufd
     └─3*[xfssyncd]
```

On vois bien les liens entre processus péres et fils. Notez que certains processus sont précédés d'un coefficient. cela signifie que le processus pére a créé autant de *« fork »* de même nom que l'indique le coefficient.

L'ensemble des processus représentés dans les structures « *task_struct* » sont liés de deux façons:

- dans une table de hachage, les processus sont hashé par leur valeur de PID.
- dans une liste circulaire doublement chaînée par deux pointeurs: `p->next_task` `p->prev_task`

Au cours de son exécution, un processus change d'état. Il démarre avec une valeur inférieure à zéro, puis il atteint zéro et est placé dans la *« runqueue »* (*file d'exécution*).

Il peut ainsi se trouver dans différents états décris ci-aprés.

```
volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */

#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
```

- `TASK_RUNNING`: le processus « *doit normalement être* » soit sur la file d'attente d'exécution soit en exécution. Comme cette action est préemptible, il est possible que le processus soit marqué présent dans la *runqueue* alors qu'il n'y est pas encore.
- `TASK_INTERRUPTIBLE`: c'est un des deux état d'attente. Le processus peut être réveillé n'importe quand par le scheduler, ou bien se réveiller tout seul à la fin d'un timer.
- `TASK_UNINTERRUPTIBLE`: c'est le deuxième état d'attente. Le processus ne se réveillera que dans un seul cas: à la fin de son timer. Cet état est rarement utilisé mais peut être utile pour éviter qu'un processus ne se réveille et ne corrompe des données partagées.
- `TASK_ZOMBIE`: c'est un état particulier. Le processus a terminé son exécution mais le pére n'a pas exécuté l'appel « `wait()` » sur son état. Ces processus deviennent alors des fils adoptifs du processus *init* qui exécute un « `wait()` » périodiquement pour les éliminer.
- `TASK_STOPPED`: le processus est stoppé , généralement par un signal système.

Le pseudo système de fichiers */proc* est monté au démarrage du système d'exploitation. Il contient, entre autres, les informations relatives aux processus. Chaque PID de processus existant constitue une entrée de répertoire dans */proc*. Dans le répertoire d'un processus, on trouve un représentation complète du mot mémoire découpée en plusieurs fichiers. Voici un exemple avec le processus *init [PID=1]*:

```
root@zerhuel:/proc/1# ls
attr  auxv  cmdline  cwd  environ  exe  fd  maps  mem  mounts  oom_adj
oom_score  root  seccomp  smaps  stat  statm  status  task  wchan
```

Les informations vitales à l'exécution du processus sont stockées dans ces fichiers.

Par exemple:

- `environ`: contient les variable de l'environnement du processus

- `maps`: les zones mémoires, avec leurs droits d'accès, utilisées par le processus.

```
root@zerhuel:/proc/1# cat maps
08048000-0804f000 r-xp 00000000 03:01 110187     /sbin/init
0804f000-08050000 rwxp 00007000 03:01 110187     /sbin/init
08050000-08071000 rwxp 08050000 00:00 0          [heap]
b7dce000-b7dcf000 rwxp b7dce000 00:00 0
b7dcf000-b7ef8000 r-xp 00000000 03:01 2885       /lib/tls/i686/cmov/libc-2.3.6.so
b7ef8000-b7efb000 rwxp 00129000 03:01 2885       /lib/tls/i686/cmov/libc-2.3.6.so
```

- `status`: donne des informations sur le status actuel du processus

```
root@zerhuel:/proc/1# cat status
Name:   init
State:  S (sleeping) <=========== le processus est actuellement endormi
SleepAVG: 88%  <================= il a passé 88% de son temps d'exécution a dormir
[...]
Pid:   1 <======================== PID du processus
PPid:  0 <======================== PID du pére, init est un cas particulier orphelin
[...]
Uid:    0       0       0       0 <= utilisateur qui lance init, typiquement root
Gid:    0       0       0       0 <= idem pour le groupe
[........]
sortie tronquée pour plus de lisibilitée
```

On peut ainsi, grâce à */proc*, visualiser facilement l'ensemble des processus existants, les ressources qu'ils utilisent, le temps qu'ils passent en exécution, etc...

Linux est très performant dans la gestion des processus. Un programme de calculs mathématiques peut s'exécuter plusieurs jours d'affilés sans perturber les fonctions serveurs Internet d'une machine. Ceci est rendu possible grâce à une utilisation efficace de la mémoire mais aussi, et surtout, grâce à une priorisation des tâches particuliérement bien implémentée. C'est cet aspect que nous allons maintenant aborder.

3. Ordonnancement dans le noyau 2.4

3.1 Politique

L'algorithme d'ordonnancement d'un système Linux doit répondre à plusieurs objectifs contradictoires:

- Faible temps de réponse des processus
- Bon fonctionnement des processus en tâches de fond
- Eviter absolument la famine
- etc ...

L'ensemble des régles qui détermine quand et comment choisir un nouveau processus à exécuter est appelée *scheduling policy (politique d'ordonnancement)*.

Les processus peuvent être classés selon deux méthodes:

La premiére, la plus classique, consiste à faire la distinction entre les processus qui demandent plutôt des opérations d'entrées/sorties (*I/O-bound*) et ceux qui sont plutôt orientés vers le calcul (*CPU-bound*).

La seconde méthode consiste à classer les processus selon leurs finalités. On en distingue trois classes:

- **Processus interactifs**

Ces processus sont en interaction constante avec l'utilisateur, et, fatalement, passent beaucoup de temps à attendre les entrées claviers/souris. Quand une entrée est reçue, le processus doit être reveillé rapidement sinon l'utilisateur pense que le système ne répond pas. Un bon délai de réponse se situe entre 50 et 150 ms.

- **Processus batch**

Ils ne nécessitent pas d'interaction avec l'utilisateur. Ces processus sont souvent lancés en tâches de fond.

- **Processus temps réel**

Ils ont de très fortes contraintes d'ordonnancement. Ces processus ne doivent jamais être bloqués par un processus de priorité inférieure, doivent avoir un temps de réponse très cours et, le plus important, ce temps de réponse ne doit pas varier. Ces processus sont généralements dédiés au contrôle de robots, à la collecte de données depuis des sondes ou encore à des applications de vidéo-surveillance.

Ces deux classifications sont indépendantes. Un processus temps réel peut faire beaucoup d'entrées/sorties ou, à l'inverse, utiliser massivement le CPU pour des calculs. Le *scheduler (ordonnanceur)* de Linux est capable de distinguer facilement les processus temps réel grâce à leurs priorités. Mais il ne sais pas faire la différence entre un processus batch et un processus interactif. Donc, afin de garantir un bon temps de réponse aux applications interactive, le *scheduler* favorise implicitement les processus de type *I/O-bounds* par rapport aux processus *CPU-bounds*.

Linux est un système à temps partagé. Bien que le processeur ne puisse exécuter qu'une tâche à la fois, il faut que l'utilisateur ait l'impression que tous les processus sont exécutés en même temps. Le *scheduler* utilise trois techniques pour réaliser cela:

- **quantum**: un intervalle de temps au bout duquel le processus en exécution s'arrête et le *scheduler* regarde à qui il va maintenant donner le processeur.
- **préemptivitée** : le *scheduler* peut interrompre l'exécution d'un processus non-noyau pour donner le processeur à un autre processus, et ce même au milieu d'un quantum.
- **priorité** : chaque processus possède une priorité comprise entre -20 et +19. Plus la priorité est basse plus le *scheduler* privilégiera ce processus.

3.1.1 Appels systèmes

Pour interagir avec le *scheduler*, Linux dispose d'un certain nombre d'appels système mis à la disposition de l'utilisateur.

Appel système	Description
`nice( )`	change la priorité[1] d'un processus
`getpriority( )`	renvoi la priorité max d'un groupe de processus
`setpriority( )`	définit la priorité d'un groupe de processus
`sched_getscheduler( )`	renvoi la politique d'ordonnancement[2] d'un processus
`sched_setscheduler( )`	définit la politique d'ordonnancement et la priorité d'un processus
`sched_getparam( )`	renvoi la priorité d'un processus
`sched_setparam( )`	définit la priorité d'un processus
`sched_yield( )`	abandon volontaire du processeur sans blocage
`sched_get_priority_min( )`	renvoi la priorité min. pour une politique
`sched_get_priority_max( )`	renvoi la priorité max. pour une politique
`sched_rr_get_interval( )`	renvoi le quantum de temps pour une politique *Round Robin (tourniquet)*

[1] *la plage de priorité est: [-20,19] pour* ***root*** *et [0,19] pour un utilisateur. Plus le chiffre est faible, plus la priorité est élevée.*

[2] *les trois valeurs possibles sont:* *SCHED_FIFO => First In First Out*
SCHED_RR => Round Robin
SCHED_OTHER => type non définit

3.1.2 Préemptivitée

Linux permet aux processus communs d'être préemptés. Quand un processus entre dans l'état `TASK_RUNNING`, le *scheduler* vérifie que sa priorité n'est pas supérieure à celle du processus qui utilise actuellement le processeur. Si c'est le cas, l'exécution du processus courant est interrompu, une commutation de contexte à lieu et le processus qui a la prioritée la plus élevée prend le processeur. Le processus préempté n'est pas suspendu pour autant, il repasse dans l'état `TASK_RUNNING` et sera réinvoqué plus tard. Un processus est également préempté quand il atteint la fin de son quantum.
Pour invoquer le *scheduler*, le bit « `need_resched` » du processus courant (*en exécution*) est passé à 1.

Si les processus communs peuvent être préemptés, ce n'est pas le cas des processus qui s'exécutent en mode noyau (*kernel mode*). L'appel système « *fork()* » est un exemple de ce type de processus. Pendant un appel à « *fork()* », le noyau ne permettra aucune préemption avant la fin du quantum ou la fin du processus.
Il arrive que des systèmes temps réel implémente des noyaux préemtifs, Linux 2.4 (*et précédents*) à fait ce choix pour simplifier l'architecture de l'ordonnanceur.

3.1.3 Quantum

Définir la durée d'un quantum de temps est critique pour les performances du système. Un quantum de temps trop faible risque de monopoliser les ressources pour l'ordonnancement. A l'inverse, un quantum trop long dégrade la maniabilitée du système.

La durée d'un quantum est donc un choix difficile. Linux utilise des tranches de temps d'environ 50ms, mais la « *formule* » généralement utilisée est « *Choose a duration as long as possible, while keeping good system response time* ».

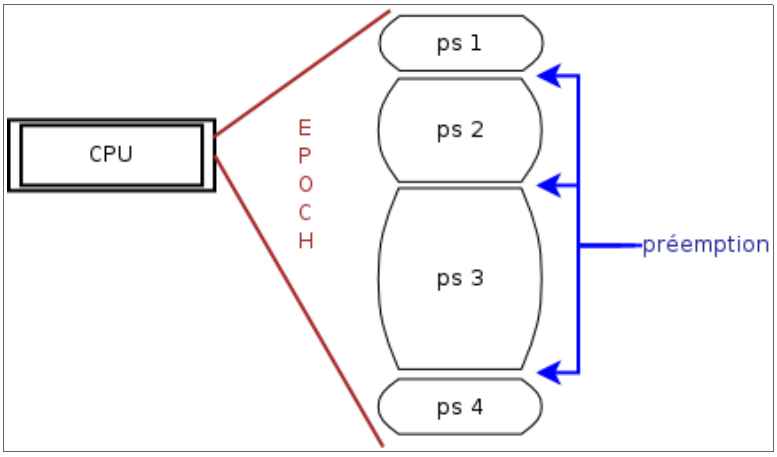
3.2 Algorithmique

3.2.1 Généralités

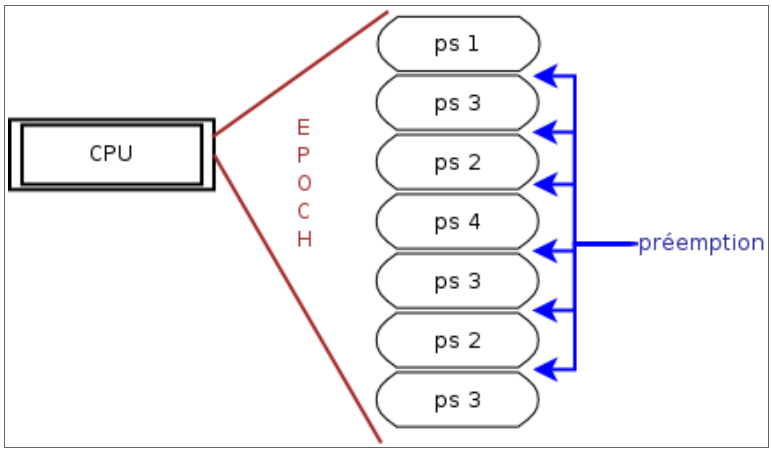
Epoque

Linux divise l'utilisation du processeur en époques *(epochs)*. Avant et aprés les époques, le scheduler calcule les quantums qu'il attribue à chacun des processus de la *runqueue*. Ainsi, au sein d'une époque, un processus va s'exécuter jusqu'à ce qu'il épuise les quantum de temps qui lui ont été accordé (*il peut le faire en une fois ou en plusieurs, s'il est préempté avant la fin du quantum*).

Le schéma suivant représente l'exécution des processus au sein d'une époque. Chaque processus posséde un nombre de quantums qui lui est propre. Si aucun processus ne s'ajoute à la *runqueue*, l'époque s'exécute de la façon suivante.



Le cas de figure suivant est également possible:



Dans ce cas, les processus ne consomment pas leurs quantums en une fois. Toutefois, ils effectuent tous leurs quantums complétement et sont donc rappelés par le *scheduler* réguliérement.

Une époque se termine quand tous les processus ont consommés leurs quantums. Le *scheduler* recalcule alors tous les quantums et une nouvelle époque commence.

Quantum

La valeur des quantums attribués à un processus est calculée en fonction de l'architecture. HZ est la fréquence du *kernel timer interrupt (représente l'interval de temps entre deux interruptions générées par le noyau*). Pour une architecture i386, il est définit à 100. Pour une architecture PowerPC 64 bits (*par exemple*), il sera définit à 1000 (*voir les fichiers include/{architecture}/param.h dans les sources).*

```
/*
 * Scheduling quanta
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif
```

Plus HZ est élevé, plus le nombre de *ticks* attribués à un processus sera élevé. Un *tick* est une parcelle de temps de l'époque. Sa valeur est définit par la fonction:

« `tick= (1000000+(HZ/2)) / HZ` » soit 10,0005 ms sur une architecture i386.

La valeur de *« x »* de la fonction « *`tick_scale()`* » dépend de la fonction suivante: `(TICK_SCALE(20-(nice))+1)`. *nice* étant compris entre -20 et +19, on en déduit une valeur de x variant entre 9 et 0 (*arch. I386*).

Priorités

Pour choisir quel processus va être exécuté, le *scheduler* doit prendre en compte les priorités de chacun des processus. Il en existe trois types:

- **Statique**: le scheduler devrait autoriser le processus à s'exécuter autant de temps qu'il est spécifié dans ce type de priorité.
- **Dynamique**: le temps restant dans l'*epoch* moins le temps que le processus à déjà passé en exécution.
- **Temps réel**: processus qui passe avant tous les autres.

Structure de données

Le *scheduler* a besoin d'informations pour travailler, voici les champs qu'il utilise dans chaque processus:

- `need_resched [int]` : un flag positionné quand un processus a terminé son quantum et attend l'intervention du *scheduler*.
- `policy [unsigned long]` : la classe d'ordonnancement. FIFO, Round Robin ou Other.
- `rt_priority [unsigned long]` : priorité statique d'un processus temps réel. Rarement utilisée.
- `priority [long]`: le quantum de temps de base (*ou priorité de base*) du processus
- `counter [long]`: le nombre de *ticks* qu'il reste au processus pour cette époque. Typiquement, lorsqu'une nouvelle époque commence, cette variable représente les quantums de temps attribués au processus. Lors d'un *fork*, cette variable est divisée en 2: une moitié reste au pére, une autre moitié va au fils. Cette technique permet d'éviter qu'un utilisateur s'accorde tout le temps CPU en utilisant des *fork* à l'infini.

Il est à noter que `priority` et `counter` jouent des rôles différents selon la `policy` choisie. Pour `SCHED_OTHER`, ils sont tout deux utilisés pour implémenter le partage du temps et pour calculer la priorité dynamique du processus. Dans le cas de `SHED_RR`, ils ne servent qu'au partage du temps. Enfin avec `SCHED_FIFO`, ils ne sont pas utilisés.

- `nice [long]`: valeur de nice du processus.
- `processor [int]`: ID du processeur assigné au processus (*utile en architecture SMP*).
- `list_head run_list [struct]`: la tête de la *runqueue*.

3.2.2 schedule()

C'est la fonction qui implémente le *scheduler*. Son rôle est de prendre un processus dans la *runqueue* et de lui assigner le processeur. Elle est invoquée directement ou via le flag `need_resched` par de nombreuses routines du noyau.

Invocation directe

Le *scheduler* est invoqué directement quand le processus `current` va être bloqué à cause d'une ressource non disponible dont il a besoin. Dans ce cas, la routine qui veut le bloquer va procéder comme suis:

1. Insérer `current` dans la file *wait*.
2. Changer l'état de `current` vers `TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE`.
3. Invoquer « `schedule()` ».
4. Regarder si la ressource est disponible, si elle ne l'est pas, retourner au point 2.
5. Une fois que la ressource est disponible, remettre `current` dans la *runqueue*.

La routine du noyau vérifie réguliérement si la ressource que le processus demande est disponible. Si elle ne l'est pas, il laisse le CPU aux autres processus que « `schedule()` » va envoyer en exécution.

flag need_resched

Nous avons déjà vu que ce flag peut être mis à 1 afin d'invoquer le *scheduler*. « `schedule()` » est appelée via le flag `need_resched` dans les cas suivants:

- `current` a utilisé tout son quantum de temps CPU.

- Un processus arrive dans la *runqueue* et sa priorité est plus grande que celle de `current`. C'est la fonction « `reschedule_idle()` » (*appelée par « `wake_up_process()` »*) qui effectue cette opération:

```
if (goodness(current, p)>goodness(current, current))
      current->need_resched = 1;
```

La fonction « `goodness()` » sera décrite plus loin.

- Les appels systèmes `sched_setscheduler( )` et `sched_yield( )` sont utilisés.

Déroulement de la fonction schedule

Voyons plus en détail ce que fait la fonction « `schedule()` ».

La valeur de `current` est mise dans la variable locale `prev`. Ensuite, on regarde si `prev` est un processus utilisant l'ordonnancement Round Robin et si il n'aurait pas déjà épuisé tout ses quantums. Si c'est le cas, il est placé à la fin de la runqueue.

```
if (unlikely(prev->policy == SCHED_RR))
      if (!prev->counter) {
            prev->counter = NICE_TO_TICKS(prev->nice);
            move_last_runqueue(prev);
      }
```

Ensuite, on examine l'état de `prev`. Si il n'a pas de signal bloquant et si son état est interruptible, il est réveillé. Cela signifie qu'il est placé dans la *runqueue* et qu'il aura donc une chance d'être exécuté.

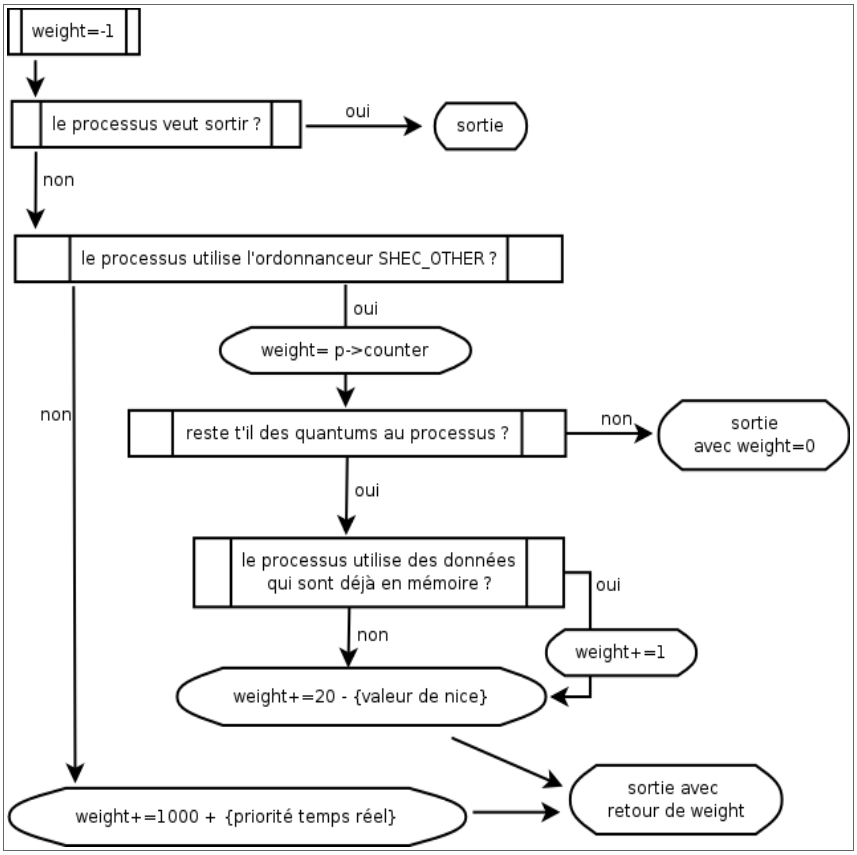
```
switch (prev->state) {
      case TASK_INTERRUPTIBLE:
            if (signal_pending(prev)) {
                  prev->state = TASK_RUNNING;
                  break;
            }
```

Maintenant, la fonction doit choisir quel est le processus qui va être exécuté pendant le prochain quantum de temps. Pour cela, elle scan la *runqueue* en commençant par le *swapper*. L'objectif est, en parcourant la liste, d'initialiser une variable `next` au premier processus lançable et de calculer sa qualité (*goodness*).

Le coeur de la fonction « `schedule()` » est l'identification du meilleur candidat à l'exécution parmis tous les processus présents dans la *runqueue*. C'est le rôle de la sous-fonction « `goodness()` ». Elle reçoit en paramétre `prev`, le processus actuellement exécuté, et `p`, le processus a évaluer. Elle retourne `weight` qui est un entier évaluant la qualité du processus `p`.

```
/*fonction goodness() */

static inline int goodness(struct task_struct * p, int this_cpu, struct
mm_struct *this_mm)
{     int weight; weight = -1;
```

```
        /*si le processus a demandé à sortir, il est rejeté */
        if (p->policy & SCHED_YIELD)
                goto out;

        /* Processus non temps réel      */
        if (p->policy == SCHED_OTHER) {

                /* si le quantum de temps de ce processus est dépassé
                 * on ne regarde rien d'autre, il est rejeté */
                weight = p->counter;
                if (!weight)
                        goto out;

                /* pour les autres, on continu l'évaluation et
                 * on donne un léger avantage aux processus qui vont
                 * utiliser des données déjà en mémoire */
                if (p->mm == this_mm || !p->mm) weight += 1;

                /* et enfin on prend en compte la valeur de nice */
                weight += 20 - p->nice;
                goto out;
        }

        /* Les processus temps réel ont un poids très élevé */
        weight = 1000 + p->rt_priority;
out:
        return weight;
}
```



En récupérant *weight*, on sais quel processus doit être exécuté juste aprés. Evidemment, les

processus temps réels sont largements privilégiés par ce système:

1. « `schedule()` » initialise une variable `c = -1000` et aprés chaque appel de « `goodness()` », stocke `weight` dans `c` si `weight` lui est supérieur (*si le processus qui vient d'etre testé a une priorité plus grande que tout ceux testés jusque là*).
2. `next` prend la valeur du processus en question, puis la boucle recommence jusqu'à ce que tous les processus de la *runqueue* ait été évalués.

Ce code a les particularités suivantes:

- Le premier processus ayant la plus grande qualité est choisi pour l'exécution. Si le processus qui vient de terminer est dans cette liste, il est préféré aux autres et il n'y a donc pas de commutation de contexte.
- Si le processus qui vient de finir est celui ayant la plus grande qualité, il continu son exécution.
- Si, à la fin de la boucle, la valeur de `c` est 0, cela signifie que tous les processus de la *runqueue* ont épuisés leurs quantums de temps. Dans ce cas, une nouvelle époque commence et la fonction « `schedule()` » assigne à tous les processus existants (*pas seulement ceux dans l'état* *`TASK_RUNNING`*) un nouveau quantum de temps. Sa durée est la somme de la valeur du champ `priority` et la moitié du champ `counter`.

  ```
  for_each_task(p){
        p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
  ```

  Cette technique permet d'augmenter la priorité des processus suspendus ou stoppés. C'est l'implémentation sous Linux du vieillissement de priorités.

Explications:

Admettons que P, un processus quelconque, entre en exécution avec une valeur de nice de -20. Il aura donc droit à 9 quantums d'exécution pour l'époque qui démarre.
P s'exécute et demande l'accès à une ressource non disponible pendant son premier quantum. Il doit attendre que cette ressource R se libére et sors donc de l'état TASK_RUNNING mais conserve, dans son champ counter, les 8 quantums d'exécution qu'il posséde encore.
Si l'époque se termine sans que P n'ai pu avoir accès à la ressource R, il n'aura pas utilisé la totalité de ses quantums. Mais, le scheduler, quand il recalcule la valeur du champ counter de chaque processus, redonnera à P la moitié des quantums qu'il n'a pas utilisé soit 4 quantums (en plus des 9 qui lui sont dûs par époque, soit un total de 13 quantums).
Comme le champ counter est un des paramètres pris en compte dans la fonction « goodness() », P aura une meilleure probabilité d'exécution pendant la prochaine époque.

4. Kernel 2.6: évolutions

Le noyau de Linux est en mouvement permanent. Chaque semaine, une nouvelle version est diffusée incluant son lot de corrections de bugs et de nouveaux drivers. Mais c'est via les changements majeurs de versions, comme le passage du 2.4 au 2.6, que sont implémentées les modifications les plus importantes.

Parmis les améliorations du noyau 2.6, il y en a une qui nous interesse tout particuliérement: c'est le choix, avant compilation, du *scheduler* que l'on souhaite utiliser. Ils sont au nombre de trois, présentés ci-dessous:

No Forced Preemption (*Server*)

C'est le modèle traditionnel, celui que nous venons d'étudier. Il offre de faibles temps de latences la pluspart du temps, mais ce n'est pas garanti et occasionnellement quelques délais plus long peuvent apparaître.

Preempt Voluntary (*Desktop*)

Ce scheduler réduit les temps de latences en ajoutant plus de « *points de préemptions explicites* » au code du noyau. Ces nouveaux points de préemptions sont sélectionnés pour réduire la durée maximale d'un réordonnancement, permettant une plus grande réactivitée des applications, au prix d'une légère diminution du débit en sortie.

Ceci améne une meilleure réaction aux évènements interactifs en permettant à un processus de forte priorité de se préempter lui même même s'il s'agit d'une processus noyau exécutant un appel système. L'objectif final étant de rendre le système plus « *lisse* » dans son exécution, même lorsqu'il est fortement chargé.

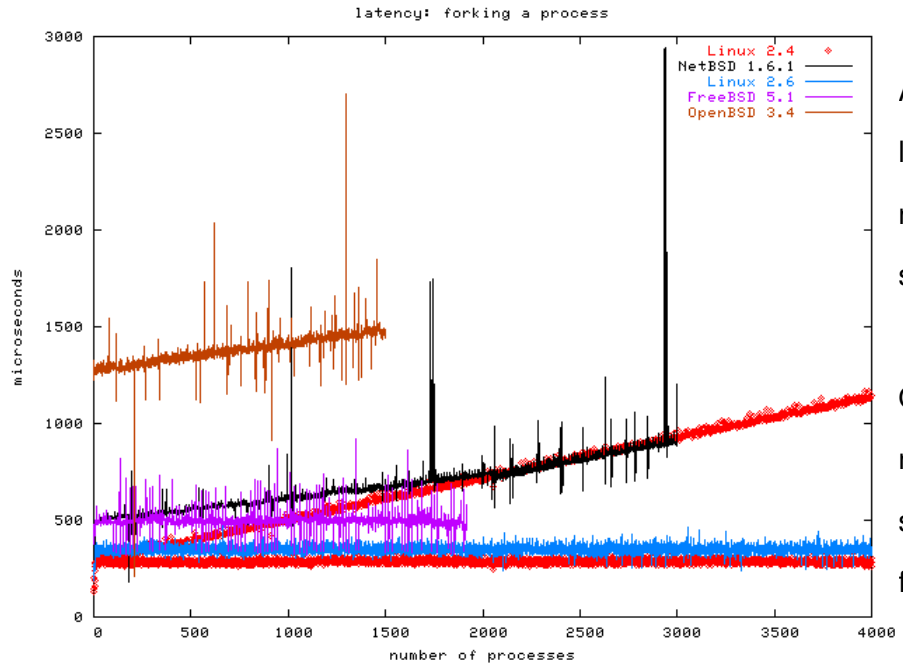
Preemptible Kernel (*Low-Latency Desktop*)

Avec ce *scheduler*, tous les processus du noyau (*qui ne sont pas en section critique*) sont préemptibles. Cela permet au système d'être très réactif aux interactions en autorisant un processus à forte priorité d'être préempté, même s'il s'agit d'un processus noyau exécutant un appel système, et ceux sans avoir à atteindre un point de préemption définit (*à la différence du scheduler précédent*). A la réduction du débit de sortie il faut ajouter un coût d'avantage élevé pour l'exécution du code du noyau.

Benchmark

Ces nouveaux *scheduler* font de Linux un système stable et solide en toutes circonstances, par exemple pour un serveur, ou à l'inverse plus instable mais extrêmement réactif, par exemple pour une station multimédias.

On peut se faire une idée de la performance du *scheduler* « *Preemptive Kernel* » au travers d'un test simple: le *fork* massif d'un processus. Ce n'est pas un benchmark à prendre au pied de la lettre mais il donne une idée des performances de ce nouveau *scheduler*:



A noter que, de l'aveu de l'auteur du benchmark, les résultats du noyau Linux 2.4 sont quelques peu erronés...

On remarque toutefois une nette supériorité de Linux 2.6 sur ses collègues de la famille BSD.

5. Conclusion

Le système Linux, fort de presque 15 ans de vie, a acquis à juste titre ses lettres de noblesses. Les techniques d'ordonnancement de Linux 2.4 sont écrites très simplement et le code, bien que complexe à lire, n'est pas si difficile à comprendre. C'est probablement grâce à cela que le système Linux évolue si vite.

Un autre point fort du *scheduler* Linux est qu'il est constamment remis en question. Le code est souvent modifié, critiqué, corrigé et amélioré pour lui permettre d'exploiter au mieux la partie *hardware* de la machine. Parmis les quelques versions du noyau que j'ai pu parcourir, on y vois une évolution nette de la taille du fichier `sched.c`. Passant de quelques 1400 lignes, dans la version finale du noyau 2.4, à prêt de 6000 pour les versions actuelles du noyau 2.6 (*toujours en développement*).

Les notions de temps sont infinitésimales en comparaison de la perception de l'utilisateur, pourtant la moindre faute d'implémentation a des conséquences telles qu'un OS peut très vite devenir inutilisables. Cette étude m'a permis de comprendre comment un système gére, dans les couches les plus basses, ces aspects invisibles de l'utilisateur.

Bibliographie:

Understanding the Linux Kernel de D.P. Bovet & M. Cesati, Ed. Oreilly

Linux Kernel 2.4 Internals de T. Aivazian,
http://www.moses.uklinux.net/patches/lki.html

Real-Time and Performance Improvements in the 2.6 Linux Kernel de W. von Hagen,
http://www.linuxjournal.com/node/8041

et les principaux sites: http://www.kernel.org; http://www.lkml.org; http://cs.kaist.ac.kr/