



Architecture d'un système d'exploitation

TD2

L'ORDONNANCEUR DU NOYAU LINUX

Etudiant : Romain PEREIRA

Encadrants : M. LOUSSERT
M. TABOADA

25/10/2018

TD1 : l'ordonnanceur du noyau Linux

Romain PEREIRA

25/10/2018

Table des matières

1	Préambule	1
2	Fonctionnement de l'ordonnanceur	2
3	Abstractions de l'ordonnanceur	6
4	Completely Fair Scheduling (CFS)	7
5	Références	8

1 Préambule

Le but de ce TP est de comprendre le fonctionnement d'un ordonnanceur, en étudiant l'ordonnanceur du noyau Linux.

2 Fonctionnement de l'ordonnanceur

Q.1 Nos ordinateurs de bureau dispose généralement d'un seul processeur (avec 1, 2, 4 ou 8 coeurs d'exécution). Le processeur est le seul point d'exécution possible pour les processus.

Lorsque plusieurs processus s'exécute sur le même processeur, il faut donc qu'il y ait un mécanisme de partage du processeur.

C'est la fonction principale de l'ordonnanceur. Il choisit dans quel ordre, et pendant combien de temps les processus peuvent être exécuté sur le processeur.

L'exécution de l'ordonnanceur peut avoir lieu pour répondre aux problèmes suivants :

- Lorsqu'un processus se clone (appel *fork()*) : lequel du père ou du fils a la priorité d'exécution ?
- Lorsque le processus courant s'arrête : quel processus doit prendre la main ensuite
- Lorsqu'un processus bloque sur une entrée/sortie, ou qu'il est en attente : doit t'il continuer de bloquer le processeur alors qu'il ne travaille pas ?
- Lorsqu'une entrée/sortie est interrompu : peut être que le processus a fini son job, ou bien c'est un signal indiquant qu'un processus précédemment en attente entrée/sortie peut maintenant effectué son travail.

Aussi, on distingue 2 grandes politiques décidant de l'exécution de l'ordonnanceur.

- **préemptif** : l'ordonnanceur peut stopper l'exécution d'un processus, et donner la main sur le CPU à un autre processus.
- **non-préemptif** : l'ordonnanceur ne tourne que si le processus utilisateur s'arrête, bloque, ou demande explicitement à changer de processus. Si le processus ne rends pas la main pendant 100 ans, l'ordonnanneur attendra impassiblement pendant 100 ans.

Q.2 La fonction *sched_yield* passe la main sur le processeur d'un processus à un autre (ou d'un processus vers lui même s'il reste le plus 'prioritaire').

S'il n'y a pas d'autre processus devant être exécuter, cette fonction s'arrête.

Q.3 Code modifié

```
[...]
printf("current=%p\n", current);
schedule();
printf("current=%p\n", current);
[...]
```

Compilation de l'image linux modifié et déploiement

```
cd ~/debian_kernel/linux-4.9.30/
make bindeb-pkg
dpkg -i ../linux-image-4.9.30_4.9.30-2_amd64.deb
reboot
```

Programme utilisateur 'main.c'

```
int main() {
    sched_yield();
    return 0;
}
```

Compile et lance le programme utilisateur

```
gcc main.c
./a.out
dmesg
```

Sortie dmesg

```
[ 170.029953] current=ffff9337bbe47140
[ 170.029956] current=ffff9337bbe47140
```

=> Le pointeur 'current' reste inchangé Ceci est dû au fait que peu de processus tourne sur la machine virtuelle. Le processus prioritaire reste le même entre 2 appels de *schedule()*

En lançant d'autres processus sur la machine, la valeur du pointeur aurait changé

Q.4 On génère d'abord les 'ctags' pour naviguer plus facilement dans le code source.

```
cd ~/debian_kernel/linux-4.9.30/
ctags -R .
```

Lecture de 'core.c'

```
ligne  profondeur/schedule()  execution
4898  0  schedule();

3454  1  sched_submit_work(task);
3438  2  ...

//debut  boucle : desactive le pr emption
3456  1  preempt_disable();

3457  1  __schedule(false);
3342  2  cpu_rq(); // recuperes la 'run queue' du CPU
3343  2  prev = rq->curr; // enregistre le processus courant dans la run queue
3391  2  next = pick_next_task(rq, prev, cookie); // recuperes la task suivante

// r active la pr emption
3457  1  sched_preempt_enable_no_resched();

3459  1  need_resched();
// tant que 'need_resched()' renvoie 'vrai',
// on boucle sur partir de preempt_disable()
```

Q.5 La fonction principal de l'ordonnanceur est 'schedule()'

```
// choisit le prochain processus      tre execut
// avant l'appel de cette fonction, un processus tourne
schedule();
// apres l'appel, un autre processus peut tourner
```

Q.6 La préemption est un mecanisme d'ordonnancement.

Il donne à chaque processus à un temps d'exécution maximal. L'ordonnanceur est executé à intervalle régulier, et détermine le prochain prochain processus qui doit être executé sur le prochain 'créneau'.

La préemption est desactivé dans l'ordonnanceur, car le code de l'ordonnanceur à la priorité d'exécution sur le CPU. Il ne doit pas être interrompu dans son execution.

De plus, la desactivation de la préemption permet d'éviter de compatibiliser le temps passé par le processus dans le code de l'ordonnanceur.

Q.7 La 'run-queue' est une file de priorité. Elle stocke les processus par ordre de priorité d'exécution. A chaque processeur est associé une run-queue.

La run-queue est verouillé au début de la fonction *schedule* (alors que le processus 'prev' vient de finir son créneau d'exécution)

Ensuite, le prochain processus devant être executé est élu par l'ordonnanceur : c'est le processus 'next'

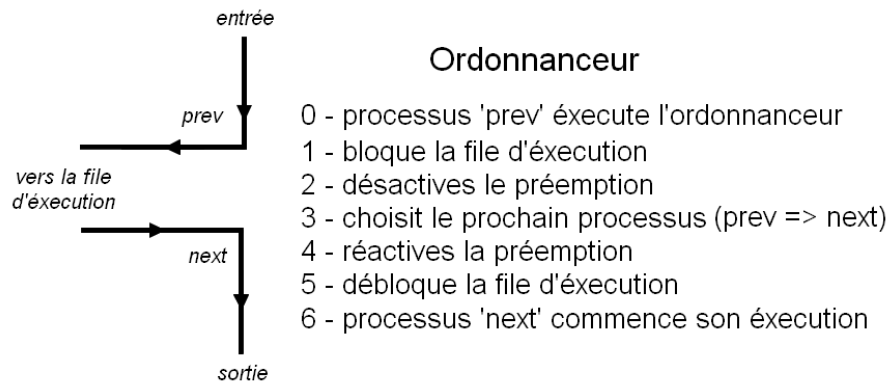
Une fois l'élection terminé, le verrou de la run-queue est retiré et le processus 'next' commence son execution.

Ce verrou sur la run-queue permet donc de ne pas modifier la priorité des processus pendant l'éléc-tion. Une fois l'élection terminé, le processus 'next' libère ce verrou pour que la priorité des processus puissent être mis à jour par la suite.

Q.8 ligne 3258 core.c

```
struct task_struct *pick_next_task (struct rq *rq,  
                                     struct task_struct *prev,  
                                     struct pin_cookie cookie);
```

Q.9 Description brève du fonctionnement de l'ordonnanceur



Q.10 La fonction `context_switch` se trouve à la ligne 2862 du fichier 'core.c'. Elle est déclarée en `__always_inline`, ce qui signifie que le code de cette fonction doit être dupliqué par le compilateur, et ré-insérer dans le binaire exécutable à chaque appel vers cette fonction (voir [3](#))

Cette fonction rétablit l'état du processus sur le processeur, état dans lequel il avait été stoppé lors de son dernier créneau d'exécution (ou dans un nouvel état si c'est un nouveau processus).

3 Abstractions de l'ordonnanceur

Q.12 L'usage de pointeur de fonctions permet de relier l'ordonnanceur à des algorithmes d'ordonnement.

La structure *struct sched_class* définit toutes les fonctions qu'un algorithme d'ordonnement doit implémenter.

Il suffit d'initialiser cette structure avec des implémentations des fonctions de l'algorithme, et l'ordonnanceur l'exécutera

Dans le fichier *core.c*, c'est le CFS (Completely Fair Scheduling) qui est utilisé

```
/*
9023 * All the scheduling class methods:
9024 */
9025 const struct sched_class fair_sched_class = {
9026     .next          = &idle_sched_class,
9027     .enqueue_task   = enqueue_task_fair,
9028     .dequeue_task    = dequeue_task_fair,
9029     .yield_task      = yield_task_fair,
9030     .yield_to_task   = yield_to_task_fair,
9031     [...]

```

Q.13 Voici quelques algorithmes d'ordonnement du noyau Linux.

- **SCHED_NORMAL : Completely Fair Scheduling (CFS)** est l'algorithme utilisé par défaut sur de très nombreuses distributions Linux. Il est détaillé en [4](#)
- **SCHED_DEADLINE : Deadline Scheduler**
- **SCHED_RR : Round-Robin Scheduling**
- **SCHED_FIFO : First in, first out** : c'est l'algorithme le plus simple envisageable. Les processus demandent du temps d'exécution sur le CPU. Ces demandes sont placées dans une file FIFO (les processus sont exécutés dans l'ordre dans laquelle ils ont demandé le CPU : 1er arrivé, 1er servi).
- **Brain Fuck Scheduler (BFS)** [4](#) est utilisé par défaut dans quelques distributions. Il a été proposé comme une alternative plus légère au CFS (avec nombre plus faible d'heuristiques et de paramètres à ajuster).

robably the simplest of all scheduling algorithms ever devised is nonpreemptive first-come, first-served. With this algorithm, processes are assigned the CPU in the order they request it. Basically, there is a single queue of ready processes. When the first job enters the system from the outside in the morning, it is started immediately and allowed to run as long as it wants to. It is not interrupted because it has run too long. As other jobs come in, they are put onto the end of the queue. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue, behind all waiting processes. SEC. 2.4 157 SCHEDULING The great strength of this algorithm is that it is easy to understand and equally easy to program. It is also fair in the same sense that allocating scarce concert tickets or brand-new iPhones to people who are willing to stand on line starting at 2 A . M . is fair. With this algorithm, a single linked list keeps track of all ready processes. Picking a process to run just requires removing one from the front of the queue. Adding a new job or unblocked process just requires attaching it to the end of the queue. What could be simpler to understand and implement? Unfortunately, first-come, first-served also has a powerful disadvantage. Suppose there is one compute-bound process that runs

4 Completely Fair Scheduling (CFS)

Q.14 Il est implémenté dans le fichier `kernel/sched/fair.c`

Q.15 Cet algorithme se base sur les 'red-black tree' (arbre binaire bicolore [2](#))

Q.16 Cette algorithme est décrit avec détail dans l'ouvrage [1](#) , page 749.

L'algorithme CFS stocke les processus dans une file de priorité ayant la structure d'un red-black tree (appelé 'run-queue').

Ils sont positionnés dans l'arbre en fonction du temps qu'ils ont passé à s'exécuter sur le CPU (appelé 'vruntime', précision en nanosecondes).

Les sommets avec le plus petit 'vruntime' sont placés 'à gauche' dans l'arbre, et ceux avec plus de 'vruntime', 'à droite'.

étape 1 - Le CFS sort de l'arbre le processus qui a le plus petit 'vruntime' (celui le plus 'à gauche' dans l'arbre). Ce processus est mis en exécution sur le CPU.

étape 2 Périodiquement, le CFS incrémente le 'vruntime' du processus en cours, en fonction du temps qu'il a déjà passé à s'exécuter.

étape 3.1 Si le 'vruntime' du processus en cours est plus petit que le minimum de l'arbre, son execution continu.

étape 3.2 Sinon, il est inséré dans l'arbre à l'endroit approprié, puis retour à l'**étape 1**.

Pour prendre en compte la priorité des processus, le CFS incrémente plus ou moins vite le 'vruntime' des processus (au cours de l'**étape 1**)

Pour les processus avec une priorité plus basse, le temps s'écoule plus vite. Cela permet de passer plus rapidement à l'**étape 3.2** pour ces processus (c'est à dire de stopper son exécution, et passer au processus suivant).

L'insertion d'un processus dans l'arbre se fait en $O(\log(n))$, avec 'n' le nombre de processus actif dans le système. Dans les systèmes actuelles, on ne dépasse généralement pas 100 processus actifs ($\log(100) \simeq 6.7$).

Aussi, cette file d'exécution en 'red-black tree' ne contient que les processus pouvant être exécuté.

Les processus bloquant, en attente d'un signal (type I/O), sont placés dans une file d'attente concurrente ('wait-queue'). Lorsqu'un évènement les débloquent et qu'ils peuvent être exécuté, ils sont alors placés dans la 'run-queue'.

5 Références

- [1] Modern Operating Systems - Andrew S. Tanenbaum, Herbert Bos (page 149-166 et 749)
<https://github.com/concerttttt/books/>
- [2] Arbre bicolore - Wikipédia
https://fr.wikipedia.org/wiki/Arbre_bicolore
- [3] Inlining in Linux - Linux Kernel Documentation
<https://www.kernel.org/doc/local/inline.html>
- [4] Brain_Fuck_Scheduler - Wikipédia
https://en.wikipedia.org/wiki/Brain_Fuck_Scheduler