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

Le but de ce TP est de comprendre le fonctionnement de l'ordonnancement des processus sur les processeurs.

Pour cela, on étudie l'ordonnancement sous le 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'execution). Le processeur est le seul point d'execution possible pour les processus.

Lorsque plusieurs processus s'execute sur le même processeur, il faut donc qu'il y ait un méchanisme 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 executé sur le processeur.

L'execution 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'éxecution ?
- 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 précèdemment en attente entrée/sortie peut maintenant effectué son travail.

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

- **préemptif** : l'ordonnanceur peut stopper l'éxecution 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'ordonanneur 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 executé, cette fonction s'arrête.

Q.3 Code modifié

```
[...]
printk("current=%p\n", current);
schedule();
printk("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 preemption
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 le processus suivant

// reactive la preemption
3457   1  sched_preempt_enable_no_resched();

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

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

```
// choisit le prochain processus a executer
// 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'execution 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'execution 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'éxecution. A chaque processeur est associé une run-queue.

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

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

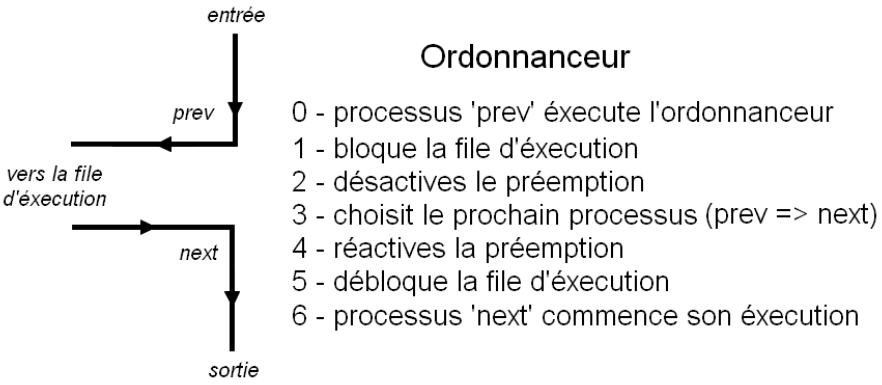
Une fois l'election terminé, le verrou de la run-queue est retiré et le processus 'next' commence son execution.

Ce verrou sur la run-queue permet donc de ne pas modifier la priorité des processus pendant l'éléction. Une fois l'election 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 declaré en *__always_inline*, ce qui signifie que le code de cette fonction est duppliqué par le compilateur, et ré-insérer dans le binaire executable à chaque point appel de cette fonction (voir 3)

Cette fonction rétablie l'état du processus sur le processeur, état dans lequel il avait été stoppé lors de sa dernière d'execution.

Le code source a été parcouru en utilisant 6.

Du code écrit en assembleur executé par cette fonction a été trouvé dans :

- **arch/include/asm/mmu_context.h**, dans la fonction *arch_start_context_switch*
- **arch/*/include/asm/atomic.h**, dans les fonctions *atomic_inc*. Ces fonctions servent à incrémenter une valeur de manière atomique (avec un verrou), selon l'architecture du processeur (les architectures sont listées ici https://elixir.bootlin.com/linux/latest/source/arch)
- **arch/*/include/switch_to.h**, déclares le prototype de **la fonction de changement de contexte** (du changement d'état du processeur : registres et mémoire).

En remontant dans l'historique Git du dépot de Linus Torvald, je suis tombé sur ce commit :

https://github.com/torvalds/linux/blob/f05e798ad4c09255f590f5b2c00a7ca6c172f983/arch/x86/include/asm/switch_to.

On y voit une implémentation en assembleur du changement de contexte, pour l'architecture Intel x86.

Q.11 Cette ordonnanceur est bien adapté au temps partagé (plus de processus que de coeurs), car le méchanisme de changement de contexte est au coeur de l'ordonnanceur.

L'ordonnanceur a été historiquement construit de sorte à partager 1 processeur entre plusieurs processus. Le nombre de processus est variable, et ne peut pas être déterminé en amont.

Avec l'augmentation du nombre de coeurs, l'ordonnanceur a été ajusté, mais la logique historique reste la même.

Dans un cadre massivement parallèle (type **HPC**), le nombre de processus peut être déterminé.

Lors de l'éxecution d'un programme sur un HPC, on peut imaginer que l'utilisateur choisisse en amont le nombre de processus qu'il souhaite utilisé.

Ainsi, on peut donc créer un ordonnanceur moins généraliste, mais plus performant en réduisant l'overhead dû au calcul de priorité et au changement de contexte.

Par exemple, on peut imaginer avoir 1 seul processus par coeur de calcul.

Ou encore, avoir un nombre fixe de processus par coeur, et les faire executer à intervalle régulier ('par créneau')

On peut aussi imaginer ajouter des registres aux processeurs uniquement dédié au stockage. Ces registres pourraient servir à stocker le contexte d'un processus. A chaque processus est associé un lot de registre, on supprime ainsi toute l'étape de re-copiage des registres dans le changement de contexte.

Finalement, ***non****, cette ordonnanceur n'est pas adapté à un contexte HPC. Dans un contexte HPC, on cherche à tout prix la performance. On peut réduire la place de l'ordonnanceur par le déterminisme du nombre de processus, et ainsi optimiser cette partie du système*

3 Abstractions de l'ordonnanceur

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

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

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

Dans le fichire 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 algorithme d'ordonnancement du noyau Linux.

- **SCHED_FIFO : First in, first out** : c'est l'algorithme le plus simple envisageable. Les processus demandent du temps d'éxecution sur le CPU. Ces demandes sont placées dans une file FIFO (les processus sont éxecutés dans l'ordre dans lequelle ils ont demandé le CPU : 1er arrivé, 1er servi).
- **SCHED_DEADLINE : Deadline Scheduler** (5) : chaque processus déclare lui même le temps d'éxecution dont il a besoin pour éxecuter une tâche, et l'ordonnanceur les place dans la run-queue. Si un processus tente de tourner plus longtemps que prévu, il est stoppé et reprendra où il s'est arrêté à la prochaine éxecution.
- **SCHED_RR : Round-Robin Scheduling**. Cet algorithme défini un quantum de temps. Les processus sont ordonnancés par quantum de temps. Le choix de la taille du quantum est primordiale pour avoir de bonnes performances. Si le changement de contexte (de processus)

prends Xms, et que le quantum fait Yms, le processeur passe $\frac{X}{X+Y}$ de son temps à changer de contexte. Pour $X = 1ms$, et $Y = 3ms$, le processeur passe $\frac{1}{4}$ de son temps à changer de contexte. Il faut donc trouver un quantum suffisement grand par rapport au temps pris par le changement de contexte, mais suffisement petit (pour éviter que des tâches courtes ne retiennent le processeur plus longtemps qu'elles n'en ont vraiment besoin).

— **SCHED_NORMAL : Completely Fair Scheduling (CFS)** est l'algorithme utilisé par défaut sur de très nombreuses distributions Linux. Il est detaillé en 4

— **Brain Fuck Scheduler (BFS)** 4 est utilisé par défaut dans quelques distributions. Il a été proposé comme une alternative plus légère au CFS (avec un nombre plus faible d'heuristiques et de paramètres à ajuster).

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 Cet 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'éxecuter 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 éxecution 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'éxecuter.

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 approrié, 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 éxecution, 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'éxecution en 'red-black tree' ne contient que les processus pouvant être éxecuté.

Les processus bloquant, en attente d'un signal (type I/O), sont placés dans une autre file d'attente, concurrante car des évènements peuvent surgir à tout moment, appelé 'wait-queue'. Lorsqu'un évènement débloque un processus, il est placé dans la 'run-queue'.

5 Références

[1] Modern Operating Systems - Andrew S. Tanenbaum, Herbert Bos (page 149-166 et 749)
https://github.com/concertttttt/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

[5] SCHED_DEADLINE - Wikipédia
https://en.wikipedia.org/wiki/SCHED_DEADLINE

[6] Parcourir code source Linux - elixir.bootlin.com
https://elixir.bootlin.com/linux/latest/source