TD1 - Architecture d'un Système d'Exploitation

October 9, 2022

1 Fonctionnement de l'ordonnanceur

1.1

Nous avons besoin de l'ordonnanceur quand il faut gérer plusieurs processus en même temps, ce qui est toujours le cas lorsque l'on utilise un système d'exploitation.

1.2

On rentre dans le terminal:

```
cd debian_kernel/linux-4.9.30/kernel/sched ; grep -r sched_yield
```

On trouve alors dans le fichier core.c:

```
SYSCALL_DEFINE0(sched_yield)
{
        struct rq *rq = this_rq_lock();

        schedstat_inc(rq->yld_count);
        current->sched_class->yield_task(rq);

        /*
         * Since we are going to call schedule() anyway, there's
         * no need to preempt or enable interrupts:
         */
        __release(rq->lock);
        spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
        do_raw_spin_unlock(&rq->lock);
        sched_preempt_enable_no_resched();

        schedule();

        return 0;
}
```

Cette fonction permet la libération du CPU par l'appelant afin de permettre l'exécution d'autres threads. Elle arrête le thread code user, le relègue à la fin de la run queue puis appelle swapcontext puis retourne en user mode après éxécution.

1.3

On crée un fichier q3.c (man sched_yield pour trouver le header):

```
#include <stdlib.h>
#include <stdio.h>
#include <sched.h>

int main(){
```

```
        sched_yield();
        sched_yield();
        return 0;
}
```

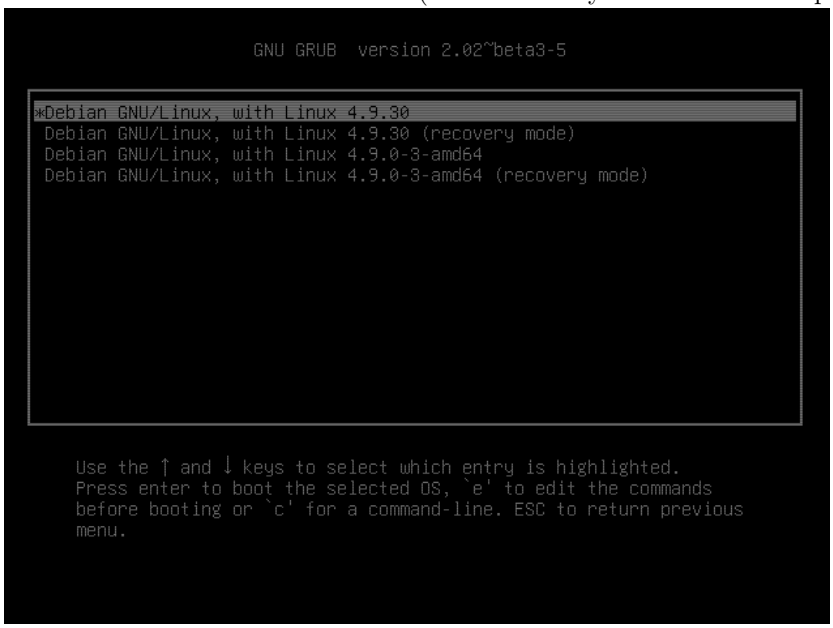
Que l'on compile avec

```
gcc -o q3 q3.c
```

Afin de constater les modifications liées à l'ajout des printk, on doit recompiler le noyau. Pour cela, on se place dans le répertoire linux-4.9.30:

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

Les derniers messages de compilation indiquent le choix du noyau (X=2) à dépaqueter. On redémarre ensuite la machine virtuelle (le nouveau noyau est sélectionné par défaut).



Puis on effectue dans le répertoire sched:

```
./q3 ; su
dmesg

...
[    9.662738] random: crng init done
[   30.502758] current->se.exec_start=30502277554
[   30.502760] current->se.exec_start=30502759746
[   30.502761] current->se.exec_start=30502759746
[   30.502762] current->se.exec_start=30502761738
```

On observe dans les derniers messages du noyau 4 threads correspondant à nos 2 appels de sched_yield .

1.4

Les fonctions s'enchaînent dans l'ordre suivant:

Etape	Mode	Fonction appelée
1	user	scheld_yield
2	kernel	contextswitch puis switch_to dans fair.c (pas de préemption)
3	kernel	schedule (pas de préemption)
4	kernel	swapcontext
5	user	scheld_yield
6	kernel	contextswitch puis switch_to dans fair.c (pas de préemption)
7	kernel	schedule (pas de préemption)
8	kernel	swapcontext
9	user	

1.5

L'ordonnanceur est le composant du noyau du système d'exploitation gérant l'allocation du processeur aux différents processus à exécuter et l'ordre d'éxécution de ces processus. Il assure qu'une certaine tâche sera terminée dans un délai donné.

1.6

Préemption : possibilité d'appropriation du processeur par un processus avant la fin du processus courant.

Il n'y en a pas en espace utilisateur car ilfaut être dans le noyau pour exécuter le code de l'ordonnancement. Cela peut se produire n'importe où dans code utilisateur via les interruptions.

1.7

next libère le verrou pour éxécuter le processus suivant

1.8

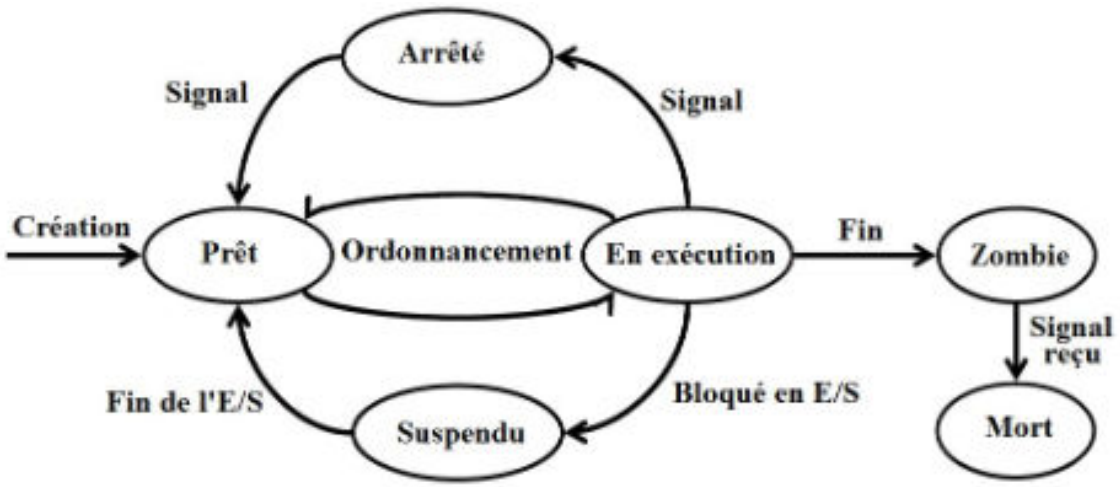
La fonction switch_to permet dans core.c de traiter un autre processus :

```
/* Here we just switch the register state and the stack. */
switch_to(prev, next, prev);
barrier();
```

1.9

L'ordonnanceur est basé sur des threads (cf Ordonnancement) et non une entité ordonnanceur, et détermine quel processus s'éxécute.

Il existe 3 types d'ordonnanceur: court, moyen et long terme (shed_(normal,batch,idle) sous Linux) qui attribuent des périodes d'éxécutions plus ou moins longues) .



De manière générale, un processus éxécuté en user mode fait un appel système via une interruption pour demander la libération du CPU (sched_yield) et appelle **swapcontext** lorsque le processus est prêt dans la run queue (file d'attente des processus).

Ensuite, l'ordonnanceur éxécute le processus en kernel mode pendant le temps CPU qui lui est attribué, et un signal est envoyé pour mettre à jour l'état du processus (en cours d'éxécution, endormi, mort, zombie).

S'il y a préemption (allocation des ressources CPU à un autre processus de priorité supérieure, sans monopolisation), le processus courant est suspendu. Enfin, l'ordonnanceur **cherche un autre processus** à éxécuter selon l'algorithme d'ordonnancement utilisé.

Bonus:

1.10

1.11

2 Abstraction dans l'ordonnanceur

2.1 12

On constate que les algorithmes d'ordonnacement manipulent des pointeurs dans des threads.

2.2 13

Algos non-préemptifs
* FIFO (traite les processus dans l'ordre d'arrivée, choisit le processus qui est depuis le plus longtemps dans la file jusqu'à fin)

* SJF (shortest job first)(choisit les processus ayant le plus court temps d'éxécution)

Algos préemptifs

* RR (Round Robin) (stratégie du "tourniquet" : recyclage des processus sur le proc tant que ceux ci ne sont pas terminés tranche temps fixe "quantum" pendant lequel un processus s'éxécute quand élu)

* SRT (Shortest Remaining Time) (choisit les processus ayant le plus court temps d'éxécution mais avec le quantum de temps; généralisation SJF)

* CFS (Completely Fair Scheduling) depuis linux 2.6

Les algorithmes préemptifs ci-dessus sont en général plus efficaces et donne une impression de système plus réactif.

3 Completely Fair Scheduling

3.1 14

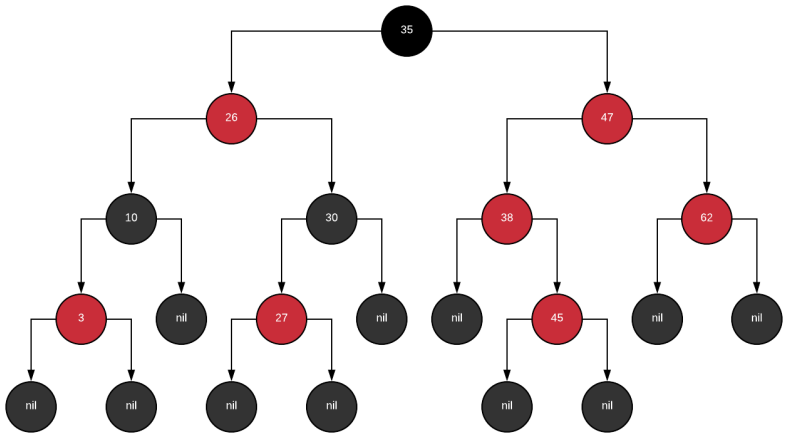
L'algorithme CFS est implémenté dans le fichier fair.c .

3.2 15

La structure est celle de l'arbre binaire rouge noir. C'est un arbre binaire de recherche dans lequel la racine est noire, chaque noeud possède 2 enfants, et chaque enfant d'un noeud rouge est noir.

```
#define NOIR 0
#define ROUGE 1
#define FEUILLE NULL

struct noeud {
  struct noeud *gauche; //Pointeur vers fils gauche
  struct noeud *droit; //Pointeur vers fils droit
  struct noeud *parent; //Pointeur vers pere
  int couleur; // ROUGE ou NOIR
  int cle; // Peut etre n'importe quel type
};
```



3.3 16

Les tâches sont classées par temps passé sur le CPU sur l'arborescence. CFS suit son exécution virtuelle avec une granularité de l'ordre de la nanoseconde. Les tâches avec moins de temps d'exécution virtuel doivent être planifiées plus tôt pour garantir l'équité.

Au fur et à mesure de l'algorithme, CFS choisira toujours le nœud le plus à gauche de l'arbre. Périodiquement, CFS ajustera le temps d'exécution virtuel de la tâche en cours d'exécution sur le processeur et le comparera au temps d'exécution virtuel de la tâche la plus à gauche de l'arborescence. S'il est plus petit, la tâche restera sur le CPU, sinon elle préemptera et sera placée dans l'arborescence. Le CFS sélectionnera le nœud le plus à gauche à exécuter.

CFS tient compte de la priorité du processus / nice que l'on peut modifier avec la commande top sous Linux.

References