TP2 Master2LSE SEE Ordonnancement

Jalil Boukhobza

Installation des sources du noyau linux

Nous allons commencer par télécharger le code source d'un noyau sur le site www.kernel.org. Toutes les versions y sont disponibles. Prenons un exemple : supposons que nous voulons compiler la version 2.6.x (x=17 dans ce TP) du noyau Linux. Nous devons télécharger le code source depuis cette adresse : http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.x.tar.bz2.
Vous pouvez aussi utiliser directement la commande **wget**:

```
wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.x.tar.bz2
```

Une fois le code source de la version du noyau requise téléchargé, il faut le décompresser avec **bunzip** et le désarchiver avec **untar** de la façon suivante (et dans le bon répertoire : voir le TP précédent):

```
tar xvjf linux-2.6.x.tar.bz2
```

L'option `x` désigne l'extraction de l'archive, « `v` » signifie en mode « verbeux », « `j` » spécifie que nous employons la commande **bunzip** avant de désarchiver et « `f` » indique le nom du fichier en entrée.
Le fichier sera désarchivé dans le répertoire `linux-2.6.x`.

Contenu du répertoire du noyau

Le code source du noyau contient un nombre donné de sous répertoires:

arch: le répertoire `arch` contient le code du noyau spécifique aux différentes architectures supportées. Il contient un répertoire par architecture supportée par exemple `i386 (ou x86)` et `alpha`.

include : Le répertoire `include` contient la plupart des fichiers que l'on doit inclure pour construire le noyau. Ce répertoire contient d'autres sous répertoires; un par architecture supportée. Pour changer l'architecture, on a besoin d'éditer le makefile du noyau et réexecuter le programme de configuration du noyau (ce sera fait ultérieurement).

init : Ce répertoire contient le code d'initialisation pour le noyau et peut être, de ce fait, un bon point de départ pour regarder ce que le noyau fait.

mm : Ce répertoire contient tout le code relatif à la gestion de la mémoire. Le code spécifique à une architecture donnée se trouve dans `arch/*/mm/`, par exemple `arch/i386 (ou x86 selon la version)/mm/fault.c`.

drivers : Tous les pilotes de périphérique du système se trouvent dans ce répertoire. Ils sont subdivisés en sous classes de pilotes de périphériques.

ipc : Ce répertoire contient le code relatif aux communications interprocessus (sémaphores, mémoires partagées, etc.).

modules : Ceci est le répertoire utilisé pour contenir les modules construits.

fs : contient tout le code du système de fichiers. Sous divisé en plusieurs sous répertoires, un par système de fichiers supporté, par exemple `cramfs, fat, ext3, NTFS et proc.`

kernel : le code principal du noyau se trouve ici. Encore une fois, le code du noyau spécifique à l'architecture se trouve dans `arch/*/kernel`.

net : le code relatif à tout ce qui est réseaux.

lib : Ce répertoire contient le code de librairie du noyau. Le code relatif à l'architecture se trouve dans `arch/*/lib/`.

scripts : Ce répertoire contient les scripts qui sont utilisés lorsque le noyau est configuré.

Exercices

Un compte rendu est demandé pour ce TP pour la prochaine séance de TP.

Les processus

1. La structure d'une tâche sous Linux se nomme `task_struct`, elle se trouve dans le répertoire `include/linux/sched.h`. Explorer cette structure et essayer de comprendre les champs utilisés. Citez l'ensemble des champs qui, selon vous, sont relatifs à l'ordonnancement.
2. L'appel aux primitives système `fork()`, `vfork()` et `clone()` se fait de la manière suivante :
   - L'utilisateur appelle l'une de ces primitives
   - La librairie libc génère une interruption logicielle (0x80) et bascule le système en mode noyau.
   - Le noyau exécute `system_call()` et enregistre les registres de la pile noyau (dans les 2 cadres mémoire préservées en RAM pour chaque processus, voir cours)
   - Le noyau invoque la fonction `sys_fork`/`sys_clone` ou `sys_vfork`
   - Le noyau quitte le handler en invoquant l'appel `ret_from_sys_cal()`.

   Les fonctions `sys_**` sont définies dans le fichier `arch/i386(ou x86)/kernel/process***.c`. Ces fonctions sont spécifiques à l'architecture. Que remarquez-vous par rapport à l'ensemble de ces fonctions ?

   Vous pouvez retrouver la fonction `do_fork()` dans `kernel/fork.c` (celle-ci est indépendante de l'architecture).

L'ordonnancement

Partie 1: lecture et compréhension du code

1. Regardez dans le fichier `kernel/sched.c`, comment est effectué le calcul du quantum de temps d'un processus en fonction de la priorité statique (voir le cours)
2. La fonction `schedule()` est le cœur de l'ordonnancement, cette dernière est appelée à plusieurs endroits du code du noyau. On peut y voir 3 phases différentes :
   1. Ce qui est effectué avant de faire le changement de processus
   2. Le changement de processus
   3. Ce qui est fait après le changement de processus.

   Nous nous intéresserons ici uniquement à la seconde partie (le changement de processus). Ce code commence au label `switch_tasks` : Donnez une explication (on ne cherche pas le détail ici) du code jusqu'à la `barrier()`.
   Aide :

- La variable `rq` est la `runqueue` du processeur actuel.
- La macro `prefetch` se trouve dans `include/asm-i386/processor.h`, et sert à signifier au processeur de précharger une partie du processus que l'on va activer.
- La macro `clear_tsk_need_resched()` se trouve dans `include/linux/sched.h`,
- La macro `rcu_qsctr_inc()` se trouve dans `include/linux/rcupdate.h`, (vous pouvez ignorer celle-ci).
- La fonction `update_cpu_clock()`, `sched_info_switch()`, `prepare_task_switch()` et `context_switch()` se trouvent dans le même fichier,
- la macro `likely(condition)` équivaut à un if qui indiquerai au compilateur que la condition a de forte chance d'être vrai. `Unlikely(condition)` fait l'inverse. Mais les deux effectuent juste un test de type `if(condition)`.

Partie 2: développement

1. Créez un programme dans lequel vous visualiserez selon deux méthodes différentes les propriétés de l'ordonnancement du processus :
   a. En invoquant les appels systèmes spécifiques à l'ordonnancement : voir le cours.
   b. En passant par le répertoire `/proc`. Pour accéder via un appel à une fonction au répertoire relatif au processus en cours d'exécution : `/proc/self`

2. Augmentez la priorité statique du processus et visualisez cette dernière.

3. Rendez le processus temps réel (FIFO) et affichez sa priorité (nécessite le mot de passe root).

4. Nous allons essayer, dans cette question, de mesurer le temps de gigue (jitter) des timers de Linux pour les tâches conventionnelles. On fera cela en plusieurs étapes:

   a. Créez, au sein du même processus, un timer avec une période d'une seconde en utilisant les fonctions : **timer_create()**, **timer_settime()**. Aussi vous utiliserez les signaux et handler de signaux afin d'exécuter les instructions (par exemple un printf) à chaque période.

La fonction **timer_create()** permet de créer un timer, elle a le prototype suivant:

```
int timer_create(       clockid_t clockid,
                        struct sigevent *evp,
                        timer_t *timerid);
```

Le premier argument indique l'horloge système sur laquelle va se greffer notre timer. Quatre possibilités se présentent: **CLOCK_REALTIME**, **CLOCK_MONOTONIC,** **CLOCK_PROCESS_CPUTIME_ID** , et **CLOCK_THREAD_CPUTIME_ID** (voir le man). On utilisera **CLOCK_REALTIME.**

La fonction **timer_create** initialise et remplit le pointeur passé en 3ème argument (timerid) qui représente l'identifiant du timer.
Le deuxième argument permet de définir l'événement lié à l'occurrence du timer (voir man):

```
union sigval {
        int   sival_int;
        void *sival_ptr;
      };
struct sigevent {
        int     sigev_notify;   /* Méthode de notification */
        int     sigev_signo;    /* Signal d'expiration de
```

```
                        la minuterie */
    union sigval sigev_value;    /* Valeur accompagnant le signal
                        ou étant fournie à la fonction
                        du thread */
    void     (*sigev_notify_function) (union sigval);
             /* Fonction utilisée pour la notification
               d'un thread (SIGEV_THREAD) */
    void      *sigev_notify_attributes;
             /* Paramètres pour la notification d'un thread
               (SIGEV_THREAD) */
    pid_t     sigev_notify_thread_id;
             /* Identifiant du thread auquel est envoyé
               un signal (SIGEV_THREAD_ID) */
};
```

Une fois le timer crée, on peut le configurer en indiquant 2 éléments: 1) le délai avant le premier déclenchement et 2) la période de déclenchement. Cela se fait avec la fonction **timer_settime():**

```
int timer_settime(      timer_t timerid, int flags,
                        const struct itimerspec *new_value,
                        struct itimerspec * old_value);
```

Le premier argument de la fonction est l'identifiant du timer obtenu avec la fonction **timer_create()**. Le second argument est un paramètre qui spécifie si la structure **itimerspec** contient une durée (par rapport à l'instant actuel de l'appel) ou une valeur absolue.
La structure **itimerspec *new_value** contiendra la nouvelle configuration du timer à positionner alors que l'ancienne sera sauvegardée dans old_value si besoin (ou sinon ce paramètre peut être positionné à NULL).

itimerspec est une structure définit comme suit:

```
struct itimerspec {
        struct timespec it_interval;  /* Intervalle pour les
                                         minuteries périodiques */
        struct timespec it_value;     /* Expiration initiale */
    };
struct timespec {
        time_t tv_sec;            /* Secondes */
        long   tv_nsec;           /* Nanosecondes */
    };
```

b. Faites évoluer le programme précédent pour mesurer le temps auquel s'exécute le handler du signal périodique (itérer sur 100 périodes de 100ms par exemple), puis calculer les différences entre le lancement des signaux contiguës (vous pouvez utiliser la fonction **clock_gettime()**).
c. Faites l'opération sur plusieurs périodes différentes et calculez les moyennes et écarts types des différences. Que constatez-vous ?
d. En parallèle avec votre programme, exécutez un autre programme sur un terminal séparé. Ce dernier va faire des boucles d'attentes actives puis passer en sommeil d'une manière aléatoire. Refaites les mesures et calculs précédents, que remarquez-vous ?
Commentaire: si vous exécuter votre programme en dehors de la machine virtuelle, il faudrait s'assurer que les 2 programmes s'exécutent sur le même processeur. Il faut donc fixer l'affinité des tâches. Vous pouvez le faire grâce à la commande "taskset". Autrement, la machine virtuelle telle que configurée ne s'exécute que sur un seul cœur (donc pas la peine d'utiliser cette commande).

5. A présent, on utilisera des processus temps réel, le processus perturbateur restera en temps partagé, refaites 4-c et 4-d, que remarque- vous ?