TP2 OS avancés et Embarqué

Ordonnancement

Jalil Boukhobza (ENSTA-Bretagne)

Pour la question 4, une aide est proposée au travers de squelettes de programme à utiliser. L'archive est disponible sur moodle.

1. Créez un programme dans lequel vous visualiserez selon deux méthodes différentes les propriétés de l'ordonnancement du processus :
   a. En invoquant les appels spécifiques à l'ordonnancement (`getpriority()`).
   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 (setpriority()) du processus et visualisez cette dernière (nécessite le mot de passe root).

3. Rendez le processus temps réel (FIFO, sched_setscheduler()) et affichez sa priorité (nécessite le mot de passe root, faites un sudo en exécutant le programme).

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 d'un 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).

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 */
      };
```

Le squelette du code vous est fourni afin de vous aider pour la mise en œuvre, il s'agit du fichier **4-a_aide.c**

b. Faites évoluer le programme de la question précédente pour mesurer le temps auquel s'exécute le handler du signal périodique (on prendra une période d'une milliseconde), puis calculer les différences entre le lancement des signaux contiguës (vous pouvez utiliser la fonction **clock_gettime()** dans le handler pour ce faire). Vous pouvez partir du squelette fourni : **4_b_aide.c**
   Une fois le programme réalisé, vous pouvez sauvegarder les résultats dans un fichier texte :
   $ ./4_b 1000 > data-4-b.txt

c. Afin d'y voir plus clair, on peut analyser les données en calculant la moyenne et l'écart type et en cherchant le min et le max des mesures. Le programme qui permet de faire

cela se trouve dans **4_c_analysis.c** (n'oubliez pas de rajouter l'option `-lm` à la compilation) exécutez le et sauvegardez ce résultat.

$ ./4_c_analysis < data-4-b.txt

Que constatez-vous ?

d. On peut aussi tenter de créer un histogramme pour analyser les occurrences de chaque résultat. Nous utiliserons le programme **4_d_histo.c** pour cela. Exécutez-le sur vos données de mesures.

$ ./4_d_histo 100 500 15000 < data-4-b.txt > histo4-b.txt

Attention les arguments ici dépendent de la dispersion de vos mesures, à vous de les adapter (vous pouvez regarder le contenu du programme pour cela).

Une visualisation de l'histogramme peut aider ! On utilisera Gnuplot pour ce faire. Commençons par l'installer :

$ sudo apt-get install gnuplot

Puis, exécutez le script **4_d_histosh.sh**

$ ./4_d_histosh.sh histo4-b.txt "Histogramme sans perturbateur"

e. Nous allons tenter de perturber le fonctionnement du programme. 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 (vous pouvez vous baser sur **4_e_pert.c**). Refaites les mesures et calculs précédents (questions b, c, d), 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" (par exemple $ taskset –c 0 <nom_du_programme avec arguments>). 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-b, 4-c et 4-d, que remarque- vous ?

Pour aller plus loin **[Question au choix/les 2 questions pour les plus courageux]**

1. L'algorithme d'ordonnancement utilisé dans le noyau Linux depuis la version 2.6.23 est **le Complete Fair Scheduler** (CFS), illustrez son fonctionnement et comment cet algorithme gère

les valeurs de priorité (en quoi est il différent de la version précédente) ? vous pouvez vous baser sur [1][2][3].

2. On peut choisir l'algorithme d'ordonnancement SCHED_DEADLINE, pouvez vous expliquer son mode opératoire et donner quelques exemples de chronogrammes d'exécution de tâches [4] ?

[1] https://opensource.com/article/19/2/fair-scheduling-linux
[2] https://docs.kernel.org/scheduler/sched-design-CFS.html
[3] https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c
[4] https://docs.kernel.org/scheduler/sched-deadline.html