HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Reconfigurable & embedded Digital Systems

Prof. Yann Thoma

Laboratoire de Programmation Temps Réel

semestre automne 2020 - 2021

laboratoire 1 : Processus et threads Linux

Temps à disposition : 3 périodes

Objectifs pédagogiques

Durant ce laboratoire vous allez faire quelques mesures sur notre système.

Vous allez ensuite découvrir comment utiliser des processus et des threads sous Linux, comment contrôler leur exécution et quelques commandes spécifiques pour agir sur le comportement de l'ordonnanceur Linux.

Démarrage de l'ordinateur

Au démarrage, sélectionnez la version Ubuntu + Xenomai. Le nom d'utilisateur est "redsuser", et le mot de passe "reds". ⚠ Attention, lorsque vous vous déloguez les fichiers locaux sont effacés! N'oubliez pas de sauvegarder votre travail en lieu sûr.

1ères mesures

✠ 1. Écrivez un programme (cpu_loop.c) qui compte le nombre d'itérations d'une boucle exécutée durant `n_sec` secondes ; note : `n_sec` doit être définie par la directive `#define`

✓ 2. Le logiciel doit également imprimer en sortie le nombre d'itérations effectuées par seconde

3. Compilez avec la commande
   `gcc -o cpu_loop -Wall cpu_loop.c`

Le cœur du programme devrait ressembler à quelque chose comme

Exemple de boucle

```
start_time = time(NULL);

nb_iterations = 0;
while (time(NULL) < (start_time + n_sec)) {
    ++nb_iterations;
}
```

Vous aurez probablement besoin des headers suivants, qui sont déjà déclarés dans le fichier fourni :

Headers utiles

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
```

Si vous avez besoin d'information sur la fonction `time()`, exécutez `man 2 time`

4. Écrivez deux logiciels qui comptent respectivement le nombre d'opérations `*` et `/` sur une donnée de type `float`;
5. ✓ Le résultat est le nombre d'opérations par seconde (quelles operations ?);
6. ✓ Essayez différentes optimisations à l'aide des arguments -O2 et -O3 de `gcc` et commentez les résultats.

Multi-tâche : Commandes utiles

Linux est un système d'exploitation moderne et multi-utilisateurs. Il est multi-tâche et impose la séparation entre deux mondes : l'espace utilisateur (*user space*) et l'espace noyau (*kernel space*). Lors de la première manipulation vous avez travaillé dans l'espace utilisateur, en écrivant et exécutant de simples programmes. Cette manipulation va vous permettre de plonger dans l'interface des deux mondes. Premièrement, le noyau offre de nombreuses informations à propos de l'état du système dans des fichiers virtuels situé dans le dossier `/proc`. Par exemple, `/proc/cpuinfo` permet d'afficher des informations que le noyau connait à propos du ou des processeurs physiques. La commande `cat` permet d'afficher le contenu d'un fichier sur la sortie standard. Essayez donc la commande suivante :

```
$ cat /proc/cpuinfo
```

Cette commande devrait vous fournir des informations utiles à propos du processeur de la machine.

Si la machine est un ordinateur multi processeur ou multi-core, la commande `man taskset` vous apprendra comment forcer l'exécution d'un processus sur un coeur spécifique.

Une autre commande utile pour vérifier le statut d'un processus et plus généralement de la machine est `ps`. N'hésitez pas à utiliser la commande `man` pour en apprendre plus à son sujet. Exécutez :

```
$ ps maux
```

Cette commande vous permet d'observer le statut des processus en cours d'exécution : Que signifient
✓ les lettres et symboles de la colonne `STAT` ? Etablissez une petite table contenant les symboles, l'état du processus et une brève description de leur sens.

Une autre commande vous permet d'observer dynamiquement l'évolution du système : `top`.

Ordonnancement en temps partagé

Linux est un système à temps partagé (*shared time system*). Commençons par le démontrer [1]. Ces expériences seront plus pertinentes en n'exploitant qu'un seul CPU. La commande `taskset` vous permettra de forcer l'utilisation d'un seul coeur.

1. Pour ces expériences nous allons réutiliser le code du premier point (`cpu_loop.c`)

```
$ taskset -pc 0 $$
$ ./cpu_loop
```

Essayez maintenant d'exécuter

```
$ ./cpu_loop & ./cpu_loop
```

```
$ ./cpu_loop & ./cpu_loop & ./cpu_loop
```

```
$ ./cpu_loop & ./cpu_loop & ./cpu_loop & ./cpu_loop & ./cpu_loop
```

✓ Pouvez-vous expliquer le résultat de ces trois commandes ? Si vous avez des doutes, ajoutez `getpid()` à la sortie des `printf` de `cpu_loop.c`.

Si vous ouvrez 2 shells et utilisez la même commande `taskset` dans chacun, et exécutez ensuite la commande suivante sur le premier shell :

```
$ sleep 1; ./cpu_loop & ./cpu_loop
```

Et au même instant la commande suivante sur le second :

```
$ ./cpu_loop & ./cpu_loop
```

✓ Quel résultat obtenez-vous ? Pourquoi ?

Migration de tâches

Une fois en exécution, un processus est déplacé par le système d'exploitation sur les différents CPUs (tout en respectant l'affinité imposée par la commande `taskset`) pour mieux répartir la charge. La fonction `sched_getcpu()` est une extension spécifique à Linux qui permet de voir sur quel CPU la tâche qui a appelé la fonction est en exécution.

✠ Écrivez un programme (`get_cpu_number.c`) qui affiche à l'écran le numéro du CPU sur lequel il tourne au démarrage et chaque déplacement qu'il subit (avec une indication de l'heure à laquelle le déplacement se produit).

✓ Exécutez ensuite, à l'aide de la commande `taskset` précédemment vue, le logiciel `cpu_loop` sur le même CPU où `get_cpu_number` tourne. Qu'est-ce qu'il se passe ? Restreignez l'exécution de `get_cpu_number` sur 2-3 CPUs, ouvrez un nombre correspondant de terminaux et essayez de déloger `get_cpu_number` de son CPU. Utilisez la commande `htop` pour visualiser la charge du système et commentez par écrit.

Les priorités et niceness

Grâce à la commande `nice` il est possible de définir la niceness d'un processus. Lisez le manuel et lancez l'expérience précédente en exploitant `nice`. Voici un exemple :

```
$ nice -n 5 ./cpu_loop & ./cpu_loop
```

✓ Commentez les résultats.

Codage

La gestion des priorités et l'interfacage avec l'ordonnanceur sont également mises à disposition du programmeur. En utilisant la bibliothèque *PTHREAD*, complétez les fonctions du code suivant.
✓✠ Exécutez-le et commentez le code et les résultats. Le programme crée `n_threads` threads avec différentes priorités et, à la fin de l'exécution, affiche le nombre de cycles exécutés par chaque thread. Le coeur de la fonction `f_thread` est le même que celle de `cpu_loop.c`.

cpu_loop.c

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <inttypes.h>

#define EXECUTION_TIME    10 /* In seconds */

/* Barrier variable */
pthread_barrier_t barr;

void *f_thread(void *arg)
{
    /* ... */
}

int main(int argc, char **argv)
{
    /* ... */

    /* Parse input */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s NB_THREADS\n", argv[0]);
        return EXIT_FAILURE;
    }
    nb_threads = strtoimax(argv[1], (char **)NULL, 10);
    if (nb_threads <= 0) {
        fprintf(stderr, "NB_THREADS must be > 0 (actual: %d)\n", nb_threads);
        return EXIT_FAILURE;
    }

    /* ... */

    /* Get minimal and maximal priority values */
    min_prio = sched_get_priority_min(SCHED_FIFO);
    max_prio = sched_get_priority_max(SCHED_FIFO);
    max_prio -= min_prio;

    /* Initialize barrier */
    if (pthread_barrier_init(&barr, NULL, nb_threads)) {
        fprintf(stderr, "Could not initialize barrier!\n");
        return EXIT_FAILURE;
    }

    /* Set priorities and create threads */
    /* ... */

    /* Wait for the threads to complete and set the results */
    /* ... */

    for (i = 0; i < nb_threads; ++i) {
        fprintf(stdout, "[%02d] %ld (%2.0f%%)\n",
        prio_value[i], nb_iterations[i],
        100.0 * nb_iterations[i] / total_iterations);
    }

    return EXIT_SUCCESS;
}
```

N'hésitez pas à consultez l'aide de

Fonctions utiles

```c
int pthread_setschedprio(pthread_t thread, int prio);
pthread_t pthread_self(void);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

✓ Quelles sont les différences entre niceness et priorités ?

Travail à effectuer

1. Le laboratoire se fait de manière individuelle
2. Codez les programmes demandés (symbole ✠).
3. Notez les réponses aux questions indiquées par le symbole ✓. Nous en débattrons lors d'une séance ultérieure.
4. Rendez le tout sur cyberlearn, dans un fichier compressé nommé `rendu.tar.gz`
   — Ce fichier doit être généré en lançant le script `ptr_rendu.sh`
   — Le script vérifie qu'un fichier `rapport.pdf` est présent à son niveau, ainsi qu'un dossier `code` également présent au même niveau

Les connaissances acquises durant les laboratoires seront également testées lors des tests écrits ainsi que lors de l'examen final.