Système d'exploitation – TD1
Ordonnancement des processus

Conservez toutes les versions des différents programmes que vous allez écrire ainsi que leur sortie, elles resserviront par la suite.

1 Observation des processus et des processeurs

Dans cette partie, vous allez apprendre quelques bases sur `/proc` et sur l'utilisation de la commande `top`.

1.a) Commencez par regarder le fichier `/proc/cpuinfo` pour voir le nombre de processeurs (en espérant en trouver plusieurs). Quel genre de fichiers trouve-t-on dans `/proc` de manière générale ? Expliquez `/proc/<pid>/stat` et `/proc/self`.

Lancez la commande `top` et laissez son terminal ouvert dans un coin (il servira pendant tout le TD et même plus).

1.b) A quoi correspondent les colonnes de `top` ? et les lignes du haut ?

1.c) Appuyez sur `u` puis entrez votre login pour filtrer les processus affichés. Appuyez sur `f` puis sélectionnez une colonne de tri et tapez `s`, puis `Echap` pour sortir. Appuyez sur `1` pour afficher l'utilisation des différents processeurs. Appuyez sur `f`, descendez sur la ligne `P` puis appuyez sur `Espace` pour ajouter une colonne montrant le processeur sur lequel s'exécute votre commande (avec une vieille version de `top`, faire `f` puis `j` puis `Entrée`). Sur quels processeurs les processus actuels s'exécutent-ils et pourquoi ?

1.d) Changez la granularité temporelle du rafraîchissement de l'affichage en appuyant sur `s` puis en entrant un nombre (flottant) de secondes. Que se passe-t-il si on diminue beaucoup cette granularité et pourquoi ?

2 Répartition des processus et du temps processeur

Pour tout le reste des TD de système, on compilera avec `-Wall`. Il vaut mieux voir un warning et l'ignorer que ne pas le voir du tout.

2.a) Écrivez un programme qui fait une attente active infinie. Qu'observe-t-on dans `top` quand on lance ce programme ? Si on en lance plusieurs en même temps, comment se répartissent-ils sur les différents processeurs ? Pourquoi la consommation processeur baisse-t-elle si on affiche un message à chaque iteration ? Pourquoi une redirection dans `/dev/null` corrige-t-elle le problème ?

Pour un meilleur affichage dans top, Vous pouvez ajouter un filtre en tapant `o` puis `COM.=a.out` pour ne garder que le programme *a.out* (ou `COMMAND=a.out`, selon le nom affiché pour la colonne dans top). Vous pouvez ajouter d'autres filtres à votre convenance, tels que `%CPU>5.0`. Pour réinitialiser les filtres, tapez =.

2.b) Utilisez le programme `taskset` (ou `numactl` ou `hwloc-bind` s'il est disponible) pour verrouiller un processus sur un seul processeur. Par exemple

```
taskset -c 0,2 <programme> # lancer une commande sur les processeurs 0 ou 2
```

```
taskset -p -c 0,2 7456     # déplacer le processus courant de pid 7456
taskset -p 03 7456         # lister les processeurs par un masque hexadécimal
```

Lancez plusieurs fois votre programme sur le même processeur en même temps et vérifiez dans `top` qu'ils ne se répartissent plus sur les différents processeurs disponibles. Comment le processeur cible gère-t-il ces différents processus qu'il doit exécuter ?

2.c) Lancez deux fois votre processus sur le même processeur avec des priorités différentes en utilisant les commandes `nice` ou `renice`. Comment la répartition du temps processeur varie-t-elle avec la priorité ?

2.d) Ecrivez un programme qui dort très longtemps dans l'appel système `sleep`. Regardez si le système répartit également ces tâches dormantes entre les différents processeurs, et expliquez.

3 Mesure des changements de contexte

On va maintenant utiliser la fonction C `gettimeofday` pour mesurer des durées. Pour éviter les débordements, on pourra par exemple calculer une durée entre deux invocations avec

```
#include <sys/time.h>
unsigned long microseconds;
struct timeval tv1, tv2;
gettimeofday(&tv1, NULL);
/* chose à mesurer */
gettimeofday(&tv2, NULL);
microseconds = (tv2.tv_sec-tv1.tv_sec)*1000000
              +(tv2.tv_usec-tv1.tv_usec);
```

3.a) Confirmez que votre code fonctionne correctement en mesurant la durée d'un appel-système `sleep`.

3.b) Mesurez le coût de l'appel `gettimeofday()` lui-même. Mesurez le coût d'un `getpid()` (probablement le plus simple des appels-système Linux). Comparez les résultats avec l'utilisation du code suivant pour forcer des vrais appels-système :

```
#include <unistd.h>
#include <asm/unistd.h>
syscall(__NR_getpid);
syscall(__NR_gettimeofday, &tv, NULL);
```

3.c) Ecrivez un programme qui fait une boucle de 100 000 appels-système `sched_yield` (pour passer la main sans cesse à un autre processus), affiche la durée totale de la boucle puis recommence. Lancer deux instances de ce programme en même temps sur le même processeur. Que signifie la durée affichée ?

3.d) Lancer maintenant une seule instance de ce programme à la fois. Que signifie sa sortie ? Lancer maintenant trois (puis beaucoup plus) instances du programme *sur le même processeur*, tracez rapidement la courbe du temps affiché en fonction du nombre de processus, et expliquez la. Que pensez-vous du coût du changement de contexte et comment va-t-il jouer sur les performances selon la durée des timeslices ?

4 Mesure des timeslices

4.a) Ecrivez un programme qui fait des `gettimeofday` et affiche un message lorsque la durée entre les deux derniers appels est *inhabituellement* longue. Le message devra contenir cette durée et le `pid`. Réglez le seuil pour détecter des désordonnancements significatifs. Lancez deux fois ce programme sur le même

processeur et vérifier que l'alternance d'exécution des 2 programmes est bien affichée, mais pas le bruit autour.

4.b) Ecrivez maintenant un programme qui affiche en boucle combien de temps il a été exécuté puis combien de temps il a été désordonnancé. On utilisera les variations de la durée d'un `gettimeofday` pour ce faire. Expliquez le comportement observé quand on lance un seul puis deux instances de ce programme sur le même processeur.

4.c) Tracez un graphe montrant l'évolution de la timeslice selon la priorité.

5 Temps utilisateur, système et réel

5.a) Utilisez l'outil `/usr/bin/time` (légèrement mieux que la commande `time` de votre shell) pour savoir où votre programme faisant des `sched_yield` passe du temps. Expliquez les différents temps affichés (on pourra comparer avec l'appel-système `gettimeofday()` ou `getpid()`, réel ou virtuel). Relancez l'expérience avec 2 puis 3 instances simultanément sur le même coeur, et expliquez les nouveaux temps observés.

5.b) Reprenez maintenant votre programme qui fait des `sleep` et expliquez les temps que `/usr/bin/time` lui mesure.

5.c) Ecrivez un programme qui fait des entrées-sorties logiques (par exemple des petites lectures et écritures dans un tube) et expliquez les temps observés par `/usr/bin/time`. Faites varier la taille des écritures et lectures, tracez l'évolution de la proportion de temps système/utilisateur, et expliquez. Que se passe-t-il avec des très grandes écritures ? En déduire la taille du tampon des tubes Linux.

5.d) On va ici lire un fichier présent sur le disque (ou sur le réseau) mais pas encore en mémoire. On pourra utiliser un gros fichier local pas encore lu depuis le démarrage de la machine (films, musique, dans `ls -lSr /usr/lib`, ...). Si on est root, on pourra aussi vider le cache de fichiers avec

```
sudo bash -c "echo 1 >/proc/sys/vm/drop_caches"
```

Si on a NFS, on pourra créer un gros fichier sur une autre machine avec

```
dd if=/dev/zero of=fichier count=100000 bs=1000
```

Bref, une fois choisi le fichier non-présent en mémoire, lisez le avec

```
/usr/bin/time cat fichier > /dev/null
```

Expliquez les temps observés lors du premier lancement. Rappelez ce qu'est `/dev/null` et pourquoi on a redirigé la sortie dedans. Relancez plusieurs fois cette commande et expliquez ce que vous observez. Comment le comportement change-t-il si on écrit dans un vrai fichier ?

6 CPU Burner

Avec tout ce qui a été fait précédemment, écrivez un *CPU Burner*, c'est-à-dire un programme qui gaspille une fraction fixe (par exemple 40%) d'un processeur donné, quelle que soit l'activité de ce processeur (utilisé à 100% ou non, par un seul processus ou plusieurs, ...). On montrera le bon fonctionnement avec `top` sur différents exemples et on expliquera les limites du programme.