TP1 : Introduction

1 Consommation de C.P.U.

Nous allons commencer par étudier le comportement d'un processus consommateur de CPU. Pour ce faire, préparez le petit script *shell* ci-dessous (une fichier texte rendu exécutable et interprété par le *Bourne Shell*). Par la suite nous l'appellerons « `job.sh` » :

```
#!/bin/bash

for ((I=0; I<90000; I++))
do J=I
done
exit 0
```

N'oubliez pas de rendre ce fichier exécutable avec la commande

```
chmod u+x job.sh
```

Vous pouvez maintenant évaluer le temps d'exécution de ce processus avec la commande `time` :

```
time job.sh
```

Réglez le nombre de boucles de manière à ce que ce processus occupe la CPU pendant 2 secondes. Par défaut nous utilisons la version de `time` directement intégré au shell courant (souvent le *bash*), mais cette version est limitée. Pour utiliser la commande `time` nous devons spécifier le chemin complet :

```
/usr/bin/time job.sh
```

Les résultats sont sensiblement différents.

Interruptions. Cherchez dans le manuel (`man time`) comment récupérer le nombre d'interruptions de fin de tranche de temps subies par le processus. En déduire la taille de la tranche de temps.

Utiliser les processeurs. Utilisez la commande `top` pour surveiller le taux d'utilisation des processeurs de votre machine. Une fois `top` lancé, tapez « 1 » pour consulter le taux d'utilisation de chaque processeur. Faites en sorte (dans plusieurs terminaux) d'exécuter plusieurs boucles infinies afin d'utiliser tous les processeurs :

```
while true; do true; done
```

Lancez de nouveau `job.sh` et comptez le nombre d'interruptions. Que pouvons-nous en déduire ?

Jouer avec la priorité. La commande `nice` permet de diminuer la priorité d'une processus (faites `man nice`). Appliquez `nice` à la commande `time`. Quel est l'effet sur la tranche de temps ?

2 Entrées/sorties et CPU

2.1 Simuler des E/S

Nous allons maintenant ajouter à notre travail deux étapes d'entrée sorties, l'une avant le calcul CPU et l'autre après :

```
#!/bin/bash

sleep 2s
for ((I=0; I<90000; I++))
do J=I
done
sleep 2s
exit 0
```

Pour l'instant, nous allons simuler ces E/S par de simples attentes qui ne consomment pas de CPU. Utilisez de nouveau la commande `time` pour évaluer la nouvelle version de votre script et calculez ensuite le taux d'utilisation de la CPU.

2.2 Multi-processus

Nous allons maintenant simuler la présence de plusieurs travaux en préparant un nouveau script d'enchaînement que nous appellerons « `job10.sh` »

```
#!/bin/bash

for ((I=0; I<10; I++)); do
./job.sh
done
exit 0
```

Quel est sa durée d'exécution et le taux d'utilisation de la CPU ?

A ce stade nous n'utilisons pas le parallélisme potentiel entre exécution et entrées/sorties. Pour ce faire, nous allons modifier ce dernier script afin de lancer les dix processus en parallèle :

```
#!/bin/bash

for ((I=0; I<10; I++)); do
./job.sh &
done
exit 0
```

La commande `time` ne permet plus de mesurer le temps pris par les processus. Pour régler ce problème il suffit d'ajouter à la fin du script `job.sh` l'affichage de l'heure (en secondes) avec la commande `date +%s`. Le dernier affichage nous donnera la date de la fin de la dernière exécution. Dans ce cadre, calculez le nouveau taux d'utilisation de la CPU.

Une autre solution consiste à demander au script `job10.sh` d'attendre la fin de ses dix fils. Pour ce faire, vous devez utiliser la commande `wait`. Dans ce cas, la commande `time` appliquée à `job10.sh` va calculer la somme des temps CPU de tous les fils.

2.3 Limiter le nombre de périphériques

Actuellement, nous simulons des E/S par une simple attente. Nous pouvons donc avoir plusieurs processus qui réalisent leur E/S en même temps ce qui n'est pas très réaliste.

Si nous considérons que les lectures (première E/S) se déroulent sur un disque et les impressions (deuxième E/S) sur un autre, nous ne devrions pas avoir deux lectures en même temps (idem pour les impressions).

Pour intégrer cette contrainte, nous allons vérifier, avant chaque lecture, que nous sommes le seul processus. Pour ce faire nous allons essayer de créer un répertoire. Si nous sommes le premier la création **réussira**, dans le cas contraire elle **échouera**. Nous pourrions utiliser la fonction *bash* ci-dessous pour attendre qu'un périphérique (dont le nom est passé en paramètre) soit libre :

```
function prendre_peripherique()
{
   local nom="$1"
   while ! mkdir "$nom" 2>/dev/null; do
      sleep 0.01s
   done
}
```

Utilisez cette fonction pour vous assurer qu'un seul processus à la fois utilise le disque d'entrée. Faites de même pour le disque de sortie. Calculez le nouveau temps de réponse et le nouveau taux d'utilisation de la CPU.

3 Les appels au système

Nous allons utiliser dans cet exercice, la commande `strace`. Elle permet d'imprimer sur la sortie standard d'erreur les appels au système d'exploitation effectués par un processus.

Commencez par écrire un programme « vide » en langage C

```
int main(void) {
    return 0;
}
```

et compilez-le en utilisant la commande ci-dessous. Lancez le ensuite en utilisant la commande `strace` (voir le manuel en ligne de cette commande). Faites des essais en ajoutant des arguments à votre commande, quelles sont les changements sur le résultat de la commande `strace` ?

```
cc -static votre_fichier.c -o votre_fichier
```

Si vous supprimez l'option `-static` dans la compilation de votre commande, quelles sont les modifications dans les résultats de la commande `strace` ? Comment expliquez-vous ces nouveaux résultats ?

4 Tracer les entrées/sorties

Commencez par écrire un programme qui va lire l'entrée standard caractère après caractère pour ensuite écrire sur la sortie standard :

```
#include <stdio.h>

int main () {
    int c;

    while ((c = getchar()) != EOF) {
        putchar(c);
    }
    return (0);
}
```

Pour clairement faire la différence entre les sorties de ce programme et celles de `strace`, nous allons utiliser une nouvelle console (commande `konsole&`). Récupérez, grâce à la commande `tty`, le pseudo terminal associé à cette nouvelle console. Vous pouvez maintenant rediriger les sorties de `strace` comme indiqué ci-dessous :

```
strace votre-programme 2> pseudo-terminal
```

4.1 Taille des tampons utilisateur

Bien que dans votre programme il y ai deux appels par caractère, dans la pratique, il y a deux appels au système par ligne lue. Ce comportement est du à la présence de tampon mémoire (*buffer*) qui stocke la ligne. Quelle est la taille de ce tampon ?

Avec la fonction `setvbuf` (`man setvbuf`) faites varier la taille du tampon de sortie (`stdout`) de zéro à plusieurs kilo-octets et observez le résultat sur les traces d'appel au systèmes.

4.2 Utilité des tampons d'entrée/sortie

Avec la commande `dd` (voir exemple ci-dessous) vous pouvez facilement créer des fichiers de donnée remplis de zéro (caractère de code ascii zéro). Faites en sorte que votre programme lise ce fichier de données. Faites varier la taille du buffer d'entrée et évaluez le temps pris par votre processus. Vous pouvez également remplacer les appels à `getchar` par des appels à `fread`.

```
dd if=/dev/zero of=/tmp/bidon bs=1M count=10
```

4.3 Débit des entrées/sorties

1) Essayez d'estimer le débit des disques en effectuant plusieurs opérations de lecture puis d'écriture. **Attention** : n'utilisez par votre répertoire personnel (accessible par le réseau) mais plutôt le répertoire `/tmp` pour être sur de travailler sur le disque local.

Pour être sur que les opérations d'écriture soient physiquement réalisées, utilisez la commande `sync` (elle vide les tampons en écriture du système).

2) Faites la même chose sur une clef USB (1,5 Mo/s en USB 1 et 60 Mo/s en USB 2.0).