Systèmes d'exploitation **GLO-2001**
Philippe Giguère H16

Travail Pratique N^{o} 1
Utilisation pratique de Linux et des threads POSIX

Le rapport

Le rapport consiste en un compte rendu des manipulations et des exercices demandés. Le travail est divisé en plusieurs sections composées parfois de plusieurs points. Pour les questions ayant rapport à des commandes sur Linux, la réponse doit comprendre la commande telle que vous l'avez entrée (tapée) suivie de la réponse du système. Si la réponse excède 10 ou 15 lignes, il faut sélectionner seulement les lignes les plus pertinentes et les inclure dans le rapport, évitant ainsi au correcteur à avoir à digérer des informations superfétatoires. Évitez de faire des captures d'écrans en format image, car elles génèrent des fichiers de plusieurs Mo. Sélectionnez plutôt le texte et copiez-le dans le document. Il faut également inclure les réponses aux questions et vos explications ou commentaires lorsqu'ils sont demandés.

Vous devez utiliser obligatoirement le fichier "TP1-rapport-glo2001-h16.rtf " pour le rapport, afin d'uniformiser la présentation pour le correcteur. À la fin, sauvez dans le format **PDF**. Lors de votre remise sur PIXEL, incluez ce rapport ainsi que les codes sources, zippés dans un seul fichier. Notez que le code doit compiler sur la machine virtuelle fournie, puisque cette machine est celle du correcteur. Des erreurs de compilations entraîneront des pénalités pouvant aller jusqu'à 50 %.

Ce travail est en équipe de 1 à 2. **Écrivez vos noms** au début du document, et cochez votre programme d'étude.

Encore une fois, n'oubliez pas de joindre tous les codes sources pour les exercices de programmation.

Pour la machine virtuelle, utilisez le compte **`etu1`** avec le mot de passe suivant : « **`etudiant`** » (sans accents)

Notes

❶ Insérez dans le rapport la commande telle qu'entrée dans le shell et son résultat.

❷ Si la réponse du système est longue, sélectionnez les lignes les plus pertinentes, par exemple, quelques lignes du début et/ou de la fin.

❸ Insérez la réponse et/ou une explication dans le rapport. Le texte doit est complet et auto-suffisant: pas de "oui" ou "non" seul. Il faut reprendre la question. Généralement, les réponses peuvent être brèves, une ou deux phrases.

1. Répertoires et fichiers (18 pts)

1.1 Examiner un répertoire

Placez-vous dans le répertoire de tête avec la commande **`cd /`** et listez son contenu avec **`ls -l`** (notez que le paramètre est la lettre "L" minuscule et non "un"). Insérez la commande et quelques lignes de son résultat dans le rapport. Le rapport doit contenir la commande telle que vous l'avez tapée et quelques lignes du résultat. Expliquez ce que représente chacune des colonnes, en vous servant de l'information obtenue avec la commande **`info ls`** . Servez-vous des flèches vers le haut ou vers le bas pour trouver la section sur l'option **`-l`**.

Remarquez en particulier les répertoires "usr", "dev" et "tmp".

1.2 Protection

Pouvez-vous ajouter une entrée (fichier ou répertoire) dans le répertoire "/home"? Pourquoi? Insérez la commande utilisée et son résultat dans le rapport. Expliquez ce résultat. Pouvez-vous ajouter une entrée dans le répertoire "/tmp"? Pourquoi? Insérez la commande utilisée et son résultat dans le rapport. Expliquez ce résultat.

On peut utiliser la commande > echo "un texte" > new-file.txt pour créer un fichier "new-file.txt". Une autre façon de créer un fichier (vide) est la commande **`touch new-file.txt`** .

1.3 Modification des droits d'accès

Comment modifiez-vous l'accès à vos fichiers de façon à être le seul à pouvoir les lire, écrire et exécuter? (consulter la commande **`chmod`**). Donnez un exemple de la commande qu'il faut utiliser. Pouvez-vous réaliser ceci sur le fichier "/usr/local"? sur le fichier "/home/etu1"? Pourquoi? Insérez les commandes dans le rapport et expliquez les résultats.

1.4 Répertoires "bin"

Dans Unix, on regroupe souvent des programmes exécutables dans des répertoires que l'on nomme "bin". Les programmes systèmes sont en particulier regroupés dans les répertoires "/bin" et "/usr/bin. Examinez le contenu de ces répertoires (commande **`ls`**). Le rapport doit contenir la commande utilisée et quelques lignes de la sortie (environ 5 lignes). Pouvez-vous trouver des fichiers correspondant aux commandes "ls", "mkdir", "cat", "g++" et "gedit"? Dites dans quel répertoire se trouve chacun de ces programmes.

Essayez la commande "find *dir* -name *filename* -print" en remplaçant *dir* et *filename* par le répertoire de départ et le nom du fichier recherché. Quel est la somme de l'espace disque occupé par les programmes ls, mkdir, mv et rm?

1.5 Répertoire "dev"

Les systèmes d'exploitations modernes ont recourt à l'abstraction des ressources via les fichiers. Les périphériques n'y font pas exception, et sont représentés par des fichiers spéciaux dans Unix/Linux. Cette abstraction permet de réutiliser toutes les fonctions de lecture `read()` et d'écriture `write()` pour accéder aux périphériques, ainsi que de bien contrôler qui a accès à ces périphériques. Le répertoire "/dev" contient ces fichiers spéciaux. Ce répertoire "/dev" est un pseudo-répertoire qui est mis à jour dynamiquement par le noyau. Examinez le contenu de ce répertoire. Seulement quelques fichiers du répertoire correspondent à des périphériques actuellement disponibles sur le système. Identifiez 4 périphériques matériels communs qui sont présents dans ce répertoire.

1.6 Répertoire "include"

Le langage C est utilisé pour le développement d'Unix et est donc plus particulièrement intégré au système. Le répertoire "/usr/include" contient les fichiers d'entête (avec une extension de .h) des fonctions en C. Examinez le contenu de ce répertoire. Pouvez-vous trouver le fichier "stdio.h"? Que contient-il?

1.7 Répertoire "proc"

Le répertoire "/proc" contient l'information sur les processus en cours et sur le système. Les informations sur le système sont placées dans des fichiers à l'intérieur de ce répertoire "/proc". Trouvez le nom de modèle et la vitesse du CPU en utilisant la commande **`cat /proc/cpuinfo`**. Combien de processeurs avez-vous dans la machine virtuelle? (indice, chaque processeur est identifié par une entrée "processor").

Chaque processus, par contre, possède son propre répertoire. Le nom du répertoire correspond au numéro du processus. Avec la commande **`ps`**, trouvez le numéro (PID) correspondant à votre shell (c'est le processus *bash*). Déplacez-vous à l'intérieur de ce répertoire. Faites **`ls`** pour voir l'ensemble des fichiers disponibles. Avec **`cat status`**, examinez l'état du processus, incluant le nom. En particulier, observer comment l'entrée "`voluntary_ctxt_switches`" incrémente entre chaque appel de **`cat status`**. Reproduisez les résultats dans votre rapport.

1.8 Fichier "math.h"

Trouvez où est le fichier "math.h" et donnez la valeur de la constante mathématique $\pi/2$ qu'il contient, pour la précision en 128 bits.

1.9 Fichiers cachés

Les fichiers dont le nom commence par un point (".") ne sont pas affichés par défaut par la commande "ls". Beaucoup de fichiers système sont ainsi cachés à l'utilisateur. Allez dans votre répertoire principal (**`cd ~`**) et tapez la commande **`ls -al`** pour obtenir une liste de ces fichiers. Insérez dans le rapport la commande et quelques lignes de son résultat. Avez-vous un répertoire nommé ".config", ".gnome2" ou ".mozilla"?

2. Compilation et exécution (30 pts)

2.1 Création d'un fichier source

Pour éditer un texte, vous pouvez utiliser une fenêtre "`gedit`". Par contre, sur un terminal sans système graphique, vous devriez utiliser les éditeurs plus classiques d'Unix, tels que "`nano`" ou "`pico`" ou "`vi`". Utilisez un éditeur de votre choix pour créer le fichier "`qui-suis-je.c`" qui contiendra le programme en langage C suivant :

```
# include  <unistd.h>
# include <stdio.h>
int main() {
   printf(" Bonjour! Le numero de l'utilisateur est %d\n",getuid());
   printf(" numero du processus est %d\n",getpid());
   printf(" numero du processus parent est %d\n",getppid());
   printf(" -    nom    ??votre nom??\n");
   return 0;
}
```

Il faut insérer votre nom à la place de "**??votre nom??**". Insérez dans le rapport le contenu de ce fichier.

2.2 Compilation

Compilez le programme "`qui-suis-je.c`" et obtenez un fichier exécutable appelé "`qui-suis-je`". Il y a plusieurs compilateurs disponibles. "`g++`" est un compilateur C++ et "`gcc`" est un compilateur C. Ici, il faut utiliser le compilateur "`gcc`", car nous avons un programme en C. Tous ces compilateurs ont des options communes. Il faut en particulier connaître les options "`-c`" et "`-o`". Nous voulons en sortie un fichier exécutable nommé "`qui-suis-je`" sans extension. La sortie par défaut est "`a.out`". Il faut utiliser l'option "`-o`" pour changer le défaut. Insérez dans le rapport la ligne de commande que vous avez utilisée et son résultat.

2.3 Exécution d'un programme

Exécutez le programme "`qui-suis-je`". Pour exécuter un programme dans le répertoire courant (par exemple, le programme "`qui-suis-je`"), il faut le faire précéder de "`./`" (par exemple **`./qui-suis-je`**), car souvent le répertoire courant n'est pas dans le path (variable shell PATH). Cette pratique est plus sécuritaire et permet d'éviter de mauvaises surprises. Insérez la commande et son résultat dans le rapport.

Quel est le numéro du processus parent? Quel processus (le nom du programme) est donc le processus parent de "qui-suis-je"?

2.4 Appels systèmes fait par un programme : `strace`

Il est intéressant de pouvoir regarder tous les appels systèmes fait par un programme. L'utilitaire "`strace`" permet de faire la trace de tous ces appels système. Faites la capture des appels systèmes du programme qui-suis-je du numéro précédent, en tapant la commande suivante : `strace ./qui-suis-je` .

Quel est le premier appel système fait, et quel est son rôle?
Quel est le deuxième appel système, et quel est son rôle?

Combien d'appels systèmes distincts sont faits par ce programme? (Par exemple, si *write* apparait deux fois, comptabilisez-le qu'une seule fois).

Par déduction, quel appel système est invoqué par les *printf*?

2.5 Utilisation pratique de `strace`

Parfois nous sommes confrontés à des problèmes difficiles à débugger, car nous n'avons pas accès au code source d'une application. Ainsi, il se peut qu'un programme plante ou retourne sans donner suffisamment d'information (code d'erreur) pour nous permettre d'investiguer la chose.

Dans le répertoire qui vous est fourni pour le tp1, vous avez un exécutable `ProgrammeMystere` compilé pour votre machine virtuelle. Ce programme termine sans vraiment vous donner de l'information sur la cause de la terminaison. Déterminez ce qui a causé probablement le programme à terminer sans rien faire, en regardant les messages fournis par `strace`.

2.6 Inspection d'un *core dump*

ATTENTION! Bien comprendre comment exploiter les *core dumps* à l'aide d'un débuggeur comme gbd vous fera sauver énormément de temps pour les TP2 et TP3! C'est aussi une habileté très pratique pour votre future carrière! Portez donc une attention toute particulière à cet exercice.

Lorsqu'un processus plante, le système d'exploitation offre la possibilité de copier l'image en mémoire du processus dans un fichier appelé *core dump*, pour fin d'autopsie. De cette manière, il est possible de mieux comprendre ce qui a causé une erreur fatale comme un *Segmentation Fault*. Par défaut, cette option n'est pas activée dans la machine virtuelle Linux distribuée dans le cours. Pour demander la création de fichiers *core dump*, il faut modifier un des paramètres du shell. Utilisez la commande **`ulimit -c unlimited`** dans la session shell ouverte afin d'autoriser la création de fichiers *core dump* de taille illimitée pour cette même session.

Ensuite, le débogueur `gdb` a besoin d'information supplémentaire pour pouvoir analyser les fichiers *core*. Ces informations permettant de lier du code machine à des noms de variable/ligne de code source sont générées par le compilateur, lorsque configuré avec l'option « **`-g`** ».

Écrivez un programme appelé `plante.c` qui effectue un accès mémoire interdit, ce qui générera un *Segmentation Fault*. Vous pouvez générer une telle faute en écrivant à l'aide d'un pointeur initialisé à NULL, par exemple. Avec le débuggeur `gdb`, montrez que le programme a bel et bien planté sur cette ligne. La commande pour inspecter le *core dump* est **`gdb nom_executable core`** . Assurez-vous que l'exécutable ***`nom_executable`*** a été compilé avec l'option **`-g`**. Incluez le listing du code source dans votre rapport, et la sortie d'écran, et les explications demandées.

3. Fonction fork (12 pts)

Écrivez un programme qui génère deux processus, à l'aide de l'appel fonction **`fork()`**. Le processus parent devra écrire à l'écran « Coucou! Je suis le parent, et mon fils a le numéro de processus xxx », où xxx est le numéro du processus fils. Il devra par la suite attendre la fin du processus enfant avant de quitter. Le processus enfant écrira à l'écran « Bonjour! Je suis le fils xxx ». Le processus fils dormira par la suite pendant deux minutes (en utilisant la fonction **`sleep`**), avant de terminer.

À l'aide de la commande **`ps`** et d'une autre fenêtre shell, récupérez ces mêmes numéros de processus pendant que le processus enfant dort. Reproduisez la sortie d'écran de cette commande **`ps`**.

4. Exercice de programmation par threads (40 pts)

L'idée de cet exercice est de vous faire diviser une tâche entre 4 threads pour accélérer, en théorie, le traitement sur un système possédants plusieurs cœurs. La tâche à accomplir sera de trouver quel mot du dictionnaire français (inclus dans le tp sous le nom de fichier **`mots.txt`**) aura servi à générer des signatures cryptographiques, à l'aide d'une méthode force brute.

Une signature cryptographique (qui est une fonction de hachage) prend une entrée de taille quelconque, et calcule une empreinte (signature) de taille fixe. Un élément important de cette fonction de hachage est qu'il est facile de calculer une signature pour une entrée, mais qu'il est mathématiquement très difficile de concevoir une entrée qui générera une signature donnée. Ainsi, cette signature permet de vérifier qu'un document numérique n'a pas été modifié. Pour plus d'information, consulter le site web suivant : http://fr.wikipedia.org/wiki/Fonction_de_hachage

Dans ce tp, la signature utilisée est SHA-256 (Secure Hash Algorithm, 256 bits). Pour vous aider, vous pouvez vérifier la signature SHA-256 pour un mot en particulier à partir du site suivant :
http://www.movable-type.co.uk/scripts/sha256.html

La méthode de force brute consiste à prendre à tour de rôle chacun des mots du dictionnaire, de calculer sa signature SHA-256, et de la comparer à la signature recherchée. Si les deux sont identiques, vous avez trouvé le mot qui a servi à générer la signature. Un exemple de cette approche, en monothread, est donné en attaché, sous le nom **`ChercherSHA256.c`**, avec son fichier Makefile et le fichier **`sha256.c`** qui contient l'algorithme de signature. Pour construire l'exécutable de cet exemple, faites

```
make all
```

dans une fenêtre shell. Pour effacer tous les fichiers binaires, vous pouvez faire

```
make clean
```

Le fichier script **`test.sh`** montre l'utilisation de cette fonction pour « décoder » les signatures. Pour extraire les fichiers, faites

```
tar –xvf code.tar
```

Votre tâche consiste à modifier le programme pour faire la recherche en parallèle, à l'aide de 4 threads. Votre programme divisera le dictionnaire en 4 parties à peu près égales qui seront cherchées indépendamment par les 4 threads. « Diviser » signifie au fond que chaque thread ouvrira le fichier (donc possédera son propre descripteur de fichier pour **`mots.txt`**), mais que le premier thread fera sa recherche séquentielle dans les octets 0 à $s/4 - 1$ du dictionnaire, le deuxième dans les octets $s/4$ à $2s/4 - 1$, etc… où s est la taille du fichier **`mots.txt`**. Utilisez la fonction **`fseek()`** pour déplacer l'offset de lecture pour la première lecture effectuée par un thread (si on utilise **`open()`** pour ouvrir un fichier, il faut utiliser **`lseek( )`**). Pour trouver la taille d'un fichier, voir l'exemple ici-bas :

```
#include <sys/stat.h>
struct stat st;
stat(filename, &st);
size = st.st_size;
```

La fonction main devra attendre que les 4 threads terminent avant de retourner la réponse, avec la fonction **`pthread_join()`**. Aussi, pour vous simplifier la vie, mettez une variable ***globale*** servant de drapeaux pour indiquer si un match a été trouvé : si un thread trouve le match, il place ce drapeau à 0. À chaque début d'itération (boucle **`while`** dans le code donné), un thread vérifiera que ce drapeau n'est pas à 0, auquel cas le thread quittera. Ceci permettra à tous les threads de quitter rapidement lorsqu'un match a été trouvé.

Pour passer les informations à chacun des threads, il vous faut définir un tableau de structure et l'initialiser avec les informations nécessaires pour que chaque thread effectue son travail. L'information pour un thread lui est alors passée lors de sa création avec **`pthread_create()`** via un pointeur sur son entrée respective (par exemple `&Arguments[iThread]`) **IMPORTANT!** Il n'est pas permis de passer cette information via le *nesting* de fonction, c'est-à-dire en créant une fonction à l'intérieure de la fonction main. Un exemple de ce type de *nesting* interdit pour ce tp (mais intéressant à regarder!) est

```
float E(float x)
{
    float F(float y)
    {
        return x + y;
    }
    return F(3) + F(4);
}
```

Cette manière n'est pas supportée par tous les compilateurs, et peut induire des bugs subtils.

N'oubliez pas de compiler le programme en ajoutant la librairie **`pthread`**. Pour ce faire, vous devez modifier le fichier **`Makefile`** pour ajouter l'option **`-lpthread`** après **`-lrt`**. Ceci indique au linker que la librairie pthread doit être ajoutée. Incluez le listing de votre programme dans votre rapport, et les sorties d'écran de votre programme parallélisé qui exécute le script **`test.sh`**. Aussi, assurez-vous que le code est zippé avec votre rapport, pour que le correcteur puisse compiler votre code.