Travail pratique 1

Utilisation de la ligne de commande Linux et des threads POSIX

Instructions

- Ce travail doit être fait en équipe.
- Tous les fichiers sources remis doivent compiler sur la machine virtuelle du cours. Les programmes qui ne compilent pas seront fortement pénalisés.
- La correction est faite automatiquement avec Draveur, elle sera faite à la fin de la période de dépôt
- Aucun retard ne sera toléré
- Pour accéder à la machine virtuelle, utilisez l'utilisateur « etud1 » et le mot de passe « etudiant ».
- Par convention, on préfixe toutes les commandes de ce document par le symbole $. Cela signifie que si, par exemple, on vous demandait d'exécuter la commande `$ echo toto`, vous devriez taper les mots `echo` et `toto` dans le terminal, puis appuyer sur Entrée.
- Vous devez remettre un zip contenant votre code source et votre rapport TXT
- Pour

1. Ligne de commande Linux (30 pts)

Cette première partie est un exercice qui vous incite à vous renseigner sur des fonctions fréquemment utilisées quand on utilise Linux via une ligne de commande. Les réponses aux questions n'ont pas forcément été fournies en classe, mais elles sont largement disponibles sur le web.

Pour répondre aux questions de cette section, donnez des commandes qui fonctionnent, peu importe le répertoire courant. Par exemple, si on vous demande de lister le contenu du dossier `/x/y/z`, vous devriez répondre `$ ls /x/y/z` au lieu de seulement `$ ls`.

1.1 Commandes fréquentes (5 pts)

Associez les commandes de la colonne de gauche avec leur utilisation fréquente dans la colonne de droite. Les commandes dans ce tableau seront réutilisées tout au long de ce TP.

> **Astuce:** Les commandes `$ man <nom de la commande>` et `$ info <nom de la commande>` sont parfois utile pour apprendre la fonction d'une commande. Souvent, une recherche web vous apportera des réponses plus faciles à comprendre.

Commande	Utilisation fréquente
1. `cd`	a. Créer un lien symbolique.

2.	pwd	b.	Redémarrer l'exécution d'un processus suspendu en arrière-plan (*background*).
3.	ps	c.	Émettre un message dans la console.
4.	`echo`	d.	Lister le contenu d'un dossier.
5.	`touch`	e.	Afficher le contenu d'un fichier.
6.	less	f.	Exporter une variable d'environnement pour le reste de la session.
7.	`chown`	g.	Changer les permissions d'un fichier.
8.	`ls`	h.	Démarrer le débogueur gdb.
9.	`ln -s`	i.	Changer le répertoire courant.
10.	`sudo`	j.	Créer un fichier vide.
11.	`find`	k.	Lister le contenu des sous-dossiers du répertoire courant.
12.	`grep`	l.	Lister tous les fichiers dans un dossier, incluant les fichiers cachés.
13.	`wc`	m.	Compter les mots ou les lignes dans un fichier. (*W*ord *C*ount).
14.	`chmod`	n.	Exécuter une commande en tant que super utilisateur.
15.	`yes`	o.	Lister les processus en train de s'exécuter.
16.	`ls -a`	p.	Filtrer la sortie d'une commande.
17.	`time`	q.	Lancer un programme et intercepte tous ses appels système. Imprime les appels système à l'écran.
18.	`env`	r.	Faire la liste des variables d'environnement disponibles déjà exportées.
19.	`cat`	s.	Redémarrer l'exécution d'un processus suspendu en avant-plan (*foreground*).
20.	`strace`	t.	Calculer le temps d'exécution d'un programme.
21.	`kill`	u.	Tuer un processus.
22.	`bg`	v.	Paginer la sortie d'une commande quand elle est trop grande.
23.	`fg`	w.	Émettre une chaîne de caractères à répétition dans la console.
24.	`gdb`	x.	Imprimer le dossier courant (*P*rint *W*orking *D*irectory).
25.	`export`	y.	Changer le propriétaire (owner) d'un fichier.

1.2 Commande ls (3 pts)

Voici la sortie de la commande `$ ls` sur une machine Linux.

```
[aleuf@glo-2001-vm tp1]$ ls -l
total 36
drwxrwxr-x  4 aleuf    def-glo   4096 Jan 12 10:39 dlandry
drwxr-sr-- 10 vader    def-sith  4096 Sep 20 15:45 executor
drwxr-s---  6 dlandry  def-philg 4096 Jul 24 16:16 fixed_velodyne
drwx--S---  2 jftrem   def-philg 4096 Jan 19 10:05 jftrem
drwx--S---  2 magicarp def-philg 4096 Jan 17 11:05 magicarp
drwxr-sr--  8 dlandry  def-philg 4096 Aug  9 15:00 oru
drwx--S---  2 anakin   def-jedi  4096 Aug 17 20:46 phibabin
drwx--S---  2 qui-gon  def-dooku 4096 Jan 15 11:05 glo-2001
```

1. Combien les lignes de la sortie représentent-elles des fichiers standards?
2. Combien les lignes de la sortie représentent-elles des dossiers?
3. Quel utilisateur est propriétaire d'executor?
4. Quel groupe est associé à qui-gon?

1.3 Permissions (6 pts)

1. Sur la machine virtuelle, vous est-il possible de créer un fichier directement dans le répertoire `/home` sans utiliser la commande `sudo`?
2. Vous est-il possible de créer un fichier dans le répertoire `/home/etud1` sans utiliser la commande `sudo`?
3. Supposons que vous créez un fichier `toto.txt` avec la commande `$ touch toto.txt` sur la machine virtuelle. Par défaut, est-ce que les utilisateurs suivants pourront écrire dans ce fichier?(Vrai ou faux)
   a) Un utilisateur quelconque.
   b) Un membre du groupe etud1.
   c) L'utilisateur etud1 (vous-même).
4. Supposons que les membres du groupe etud1 ont les droits d'écriture sur le fichier `toto.txt`. Quelle commande pourriez-vous utiliser pour leur retirer ce droit?

1.4 Flux de sortie (2 pts)

Les opérateurs > et >> permettent de rediriger la sortie de commandes vers des fichiers. Par exemple la commande `$ echo Bonjour > hi.txt` crée le fichier `hi.txt` et y insère le texte « Bonjour ».

1. Utilisez les opérateurs de redirection pour créer une commande qui insère la liste des processus courants dans le fichier `/home/etu1/processes.txt`. Ajoutez à la fin le temps de votre commande avec la commande `$ time`. Inscrivez vous deux commandes au rapport
2. Avez-vous utilisé > ou >> dans votre commande d'ajout précédente?

1.5 Opérateur de chaînage (Pipe) (4 pts)

L'opérateur de chaînage (`|`) permet d'envoyer la sortie d'une première commande en entrée d'une deuxième commande. Grâce à cet opérateur, on peut exécuter des opérations plus complexes en une seule ligne. Par exemple, la commande `$ ls | grep toto` liste les fichiers et dossiers du répertoire courant et filtre cette liste pour garder seulement les lignes qui contiennent le mot « toto ».

1. Donnez une commande qui affiche le contenu du fichier `/proc/cpuinfo` dans la console.
2. Utilisez la commande trouvée précédemment, la commande grep et l'opérateur de chaînage pour créer une nouvelle commande qui filtre le contenu du fichier `/proc/cpuinfo` et fournit seulement la ligne contenant le nom du processeur, inscrivez la commande dans le rapport.
3. Utilisez la commande trouvée en 1., la commande `grep` et l'opérateur de chaînage pour créer une nouvelle commande qui filtre le contenu du fichier `/usr/include/math.h` et affiche seulement les lignes qui contiennent le mot `M_E`. Inscrivez la commande et sa sortie au rapport. `M_E`, communément appelée *e* ou le nombre d'Euler, apparaît dans plusieurs contextes en mathématiques.
4. Utilisez la commande `ls`, la commande `wc` et l'opérateur de chaînage pour créer une nouvelle commande qui compte le nombre de fichiers et de dossiers contenus dans le répertoire `/usr/bin/` *de la machine virtuelle*. Inscrivez la commande et le nombre de fichiers contenus dans `/usr/bin/` au rapport.

1.6 Gestion des tâches (jobs) (8 pts)

1. Quelle est l'utilité du fichier spécial `/dev/null`?
   a. Elle représente un fichier qui n'existe, mais n'aura pas de contenu
   b. Elle représente le vide dans un développeur
   c. Le pire ennemi de tout développeur
   d. Permets d'enlever la sortie d'une commande quand ça ne nous sert à rien
   e. Garde en mémoire toutes les informations à une seule place si on lui pousse du data
   f. Garde que nos références de null pointer exception (quand on accède à un objet null)

   Indiquez toutes les lettres qui s'appliquent selon le format suivant : e,x,e,m,p,l,e

2. Faite la commande `$ yes > /dev/null`. En l'absence d'intervention de l'utilisateur, combien de temps prendra-t-elle à s'exécuter, environ?

   Fait amusant : La commande yes est capable d'émettre la lettre y dans la sortie standard à une vitesse de 10 Go/s. Plus d'informations ici.

3. En quatre ou cinq mots, quel est l'effet de la combinaison de touches Ctrl-c sur un processus qui est en train de s'exécuter?

4. En quatre ou cinq mots, quel est l'effet de la combinaison de touches Ctrl-z sur un processus qui est en train de s'exécuter?

5. Voici la sortie de la commande `$ jobs` sur une machine. Donnez une courte commande qui tue le processus qui est en train d'exécuter la commande `sleep`.

```
[aleuf@glo-2001-vm tp1]$ jobs
[1]-  Running                 yes > /dev/null &
[2]+  Running                 sleep 100000 &
```

6. Voici la sortie de la commande jobs sur une machine. Donnez une courte commande qui redémarre l'exécution de la commande stoppée et qui la laisse en *arrière-plan*.

```
[aleuf@glo-2001-vm tp1jobs
[1]+  Stopped                 yes > /dev/null
```

7. Étant donné la même sortie de jobs qu'à la question précédente, donnez une courte commande qui redémarre l'exécution de la commande stoppée et qui la ramène en *avant-plan*.

8. Lancez la commande `$ yes > /dev/null`. Ouvrez un second terminal.

   La *job* du processus que vous venez de lancer apparaît dans la liste des *jobs* du second terminal, vrai ou faux

1.7 Variables d'environnement (2 pts)

1. Utilisez l'opérateur de chaînage, les commandes grep et env en combinaison pour créer une commande qui imprime la valeur des variables qui contiennent PATH à l'écran. Inscrivez la commande créée.

   Astuce : On peut aussi utiliser le symbole `$` pour accéder aux valeurs des variables d'environnement. Par exemple, la commande `$ echo $HOME` imprime la localisation du dossier personnel de l'utilisateur courant à l'écran.

2. Utilisez la commande export pour créer une nouvelle commande qui ajoute le dossier `/x/y/z` au PATH. Inscrivez la commande créée au rapport.

Note Comme la variable PATH indique dans quels répertoires chercher des exécutables, il n'y a pas de problème majeur à y ajouter un dossier inexistant.

2. Compilation et exécution (14 pts)

Cette section vous guide dans la compilation d'un programme simple en C sur Linux. Vous bâtirez des programmes C de plus en plus complexes tout au long de la session.

Note : Un éditeur de texte simple nommé *Text edit* est déjà installé sur la machine virtuelle.

2.1 Création d'un fichier source

Créez un fichier `qui-suis-je.c` quelque part dans la machine virtuelle. Éditez le fichier de sorte qu'il contienne le code source suivant.

```
# include <unistd.h>
# include <stdio.h>

int main() {
      printf("Utilisateur: %d\n", getuid());
      printf("Processus: %d\n", getpid());
      printf("Processus parent: %d\n", getppid());
      printf("- Nom: ??votre nom d'équipe??\n");

      return 0;
}
```

`Ajoutez ce fichier à votre archive de remise sous src/`

2.2 Compilation (4 pts)

Le compilateur `gcc` est disponible dans notre ligne de commande. Il nous permet de compiler du code source C vers un exécutable.

1. En 4 ou 5 mots, quelle est la fonction de l'option -c du compilateur `gcc`?
2. Utilisez la commande gcc pour compiler le fichier `qui-suis-je.c` vers un exécutable qui nommé `qui-suis-je`. Inscrivez la commande au rapport.
3. Inscrivez la sortie de l'exécutable `qui-suis-je` au rapport. Vous pouvez exécuter un fichier compilé dans le répertoire courant avec la commande `$ ./nom-du-fichier-compile`.

Note : En général, on ne peut pas simplement appeler la commande `$ qui-suis-je` après la compilation. La raison est que le dossier dans lequel l'exécutable se trouve n'est pas présent dans la variable d'environnement PATH.

2.3 Appels système (2 pts)

Utilisez la commande `$ strace <nom-de-l'executable>` pour faire la liste des appels systèmes provoqués par le programme `qui-suis-je`.

1. Quel est le premier appel système fait par qui-suis-je, et quel est son rôle?
2. Par déduction, quel appel système est invoqué par `printf`?

2.4 Débogage (5 pts)

Ce travail pratique, ainsi que les prochains de cette session, vous invitera à programmer en C. La programmation dans ce langage est parfois difficile et il est important de savoir déboguer ses programmes pour analyser les erreurs qu'on a pu commettre. Cette section vous donne l'occasion d'apprendre à vous servir du débogueur `gdb` sur vos propres programmes.

> **Astuce** Pour répondre aux questions sur `gdb`, vous pouvez démarrer gdb puis utiliser sa documentation en tapant `help`. Parfois les tutoriels en ligne sont plus faciles à comprendre.
>
> **Note** `gdb` est un programme en ligne de commande interactive. Une fois qu'on démarre `gdb`, on peut exécuter des commandes `gdb`, mais les commandes standard du terminal ne fonctionnent plus. Pour quitter `gdb` et revenir à la console standard, tapez la commande `quit`.

1. Donnez une commande qui utilise `gcc` pour compiler le fichier `qui-suis-je.c` en *debug mode*. L'exécutable généré doit s'appeler `qui-suis-je-debug`.
2. Donnez une commande qui démarre le débogueur `gdb` sur l'exécutable `qui-suis-je-debug`.
3. Donnez une commande `gdb` qui ajoute un *breakpoint* dans le fichier `qui-suis-je.c` à la ligne 6.
4. Donnez une commande `gdb` qui démarre l'exécution du programme (après avoir ajouté le *breakpoint*).
5. Donnez une commande `gdb` qui affiche la pile d'exécution courante. Pendant que votre session de débogage est arrêtée à un breakpoint, exécutez cette commande. Inscrivez la commande et sa sortie au rapport.

2.5 Core dump (3 pts)

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. La commande `$ ulimit -c unlimited` demande à la session courante de créer un *core dump* en cas d'erreur d'exécution d'un programme.

1. Créez un fichier `plante.c`. À l'intérieur, écrivez un programme qui plante systématiquement à cause d'un accès mémoire interdit. Compilez ce programme vers un exécutable nommé `plante`, en *debug mode*. Fournissez le fichier `plante.c` avec votre remise dans le dossier src/
2. Donnez une commande qui démarre le débogueur `gdb` avec l'exécutable `plante` et la *core dump* que vous venez de créer.
3. Une fois le débogueur lancé de cette façon, exécutez la commande `frame`. Inscrivez la sortie de cette commande au rapport.

3. La fonction `fork` (20 pts)

Écrivez un programme `skywalker.c` qui génère deux processus, à l'aide d'un appel à la fonction `fork`. Le processus parent devra écrire à l'écran « Non! Je suis ton père. Ton numéro de processus est xxxx » où xxxx est le numéro du processus fils. Il devra ensuite attendre la fin du processus fils avant de quitter.

Le processus enfant écrira à l'écran « Nooon! Ce n'est pas vrai! Je suis le processus xxxx ». L'enfant dormira ensuite 2 minutes 11 secondes avant de tuer le processus parent. Le processus parent ne peut mourir par lui-même, il est trop puissant pour cela.

Compilez et exécutez ce programme. Pendant qu'il dort, ouvrez une autre fenêtre shell et exécutez `$ ps -ef`, afin de récupérer les numéros des processus en train de s'exécuter pour tester votre programme.

Fournissez un fichier `skywalker.c` contenant le programme demandé avec votre remise.

4. Programmation avec *threads* (36 pts)

Cet exercice vous donne l'occasion d'apprendre à séparer une tâche en plusieurs *threads* pour en accélérer l'exécution sur des systèmes multicœurs. La tâche à accomplir est de multiplier deux matrices, ***G*** et ***H***. Le produit des deux matrices sera la matrice résultante ***R***. ***G*** sera de traille *N*×*M*, ***H*** de taille *M*×*P*. Le programme lancera *N*×*P* threads, et chacun des threads fera une partie de la multiplication matricielle.

Chaque thread devra recevoir deux vecteurs: la i^{ème} rangée de ***G*** et la j^{ème} colonne de ***H***. Les rangées de ***G*** et les colonnes de ***H*** sont en fait des vecteurs de même dimension. Chaque thread aura à calculer le produit scalaire de ces deux vecteurs, et ranger le résultat dans la matrice résultante ***R***.

À titre d'exemple, si nous avions les matrices

$$G = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \mathrm{H} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

en entrée, le produit de ces deux matrices serait

$$GH = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} = R.$$

Le premier thread serait alors responsable de calculer *a*, le deuxième *b*, et ainsi de suite.

Pour simplifier le problème, les tailles *M*, *N* et *P* sont définies statiquement au début du programme. Du code pour créer des matrices avec les bonnes dimensions est fourni dans le fichier `threads.c`. Il ne vous reste qu'à multiplier les matrices!

Toute l'information nécessaire sera passée lors de la création du *thread* avec une `struct` du type `ParametresThread`, qui a la définition suivante.

```
typedef struct {
      int iThread;
      int Longueur;
      double Vecteur1[M];
      double Vecteur2[M];
      double *pResultat;
} ParametresThread;
```

Dans `ParametreThread`, `iThread` est l'indice du *thread*, `Longueur` et la taille de `Vecteur1` et `Vecteur2`, et `pResultat` sont l'emplacement mémoire où ranger le résultat du produit scalaire. Lorsqu'un *thread* termine, il écrit son résultat dans `pResultat` et quitte avec `pthread_exit(NULL)`.

Vous devez créer une copie de cette structure pour chaque *thread*, par exemple avec un tableau de type `ParametresThread`. Vous passerez ensuite l'adresse d'une des entrées du tableau en argument lorsque vous créerez le *thread*:

```
status = pthread_create(&threads[i], NULL, ProduitScalaire, (void *) &mesParametres[i]);
```

La fonction `ProduitScalaire` est celle exécutée par le *thread*. Vous devez coder cette fonction pour exécuter le calcul.

N'oubliez pas de *linker* la librairie `pthread` quand vous compilez votre programme, par exemple avec la commande `$ gcc MonProgramme.c -o MonProgramme -lpthread`.

Fournissez dans votre remise un fichier `threads.c` modifié de sorte qu'il fasse le travail demandé. Dans votre rapport, répondez aux questions suivantes.

1. Configurez votre machine virtuelle afin qu'elle ne dispose que d'un de calcul. Combien de temps votre programme prend-il pour s'exécuter (en secondes)?
2. Configurez votre machine virtuelle afin qu'elle dispose de plusieurs cœurs de calcul. Combien de temps votre programme prend-il à s'exécuter? Ce résultat correspond-il à vos attentes? Pourquoi?

   Attention! Sur certains ordinateurs, il est impossible de changer le nombre de cœurs de calcul dont dispose la machine virtuelle. Prenez soin de l'indiquer dans votre rapport si cela affecte votre analyse.

3. Changez les constantes `M` et `P` du programme de sorte que les matrices à multiplier soient de beaucoup plus grandes tailles. Refaites l'expérience avec un et plusieurs cœurs de calcul. Inscrivez le temps d'exécution du programme au rapport pour chaque cas. Ces résultats correspondent-ils mieux à vos attentes?
4. Suite à ces observations, diriez-vous qu'il vaut toujours la peine d'accélérer du calcul avec des threads?

Attention! Prenez soin de ramener `M` et `P` à leur valeur initiale avant de remettre le fichier `threads.c` . cela va faciliter le travail de correction. Vous pourriez être pénalisés autrement.

5. *Checklist* de remise

Votre archive de remise devrait contenir les éléments suivants.

- Un fichier nommé `rapport.txt` contenant vos réponses. Le rapport doit être basé sur le gabarit de rapport fourni avec l'énoncé.
- Un fichier nommé `plante.c` contenant le programme demandé à la section 2.5.
- Un fichier nommé `skywalker.c` contenant le programme demandé à la section 3.
- Un fichier nommé `threads.c` contenant le programme demandé à la section 4. Ce fichier doit être basé sur le fichier `threads.c` qui était fourni avec l'énoncé.
- Aucun autre fichier.