DUI EIL 2018-2019
TP Systèmes

Ce document est un sujet de TP qui :
- contient plusieurs parties indépendantes qui peuvent être considérées dans le désordre
- liste des exercices/activités plus ou moins poussés. Nous avons essayé d'indiquer la difficulté avec des couleurs

initial | intermédiaire | avancé

- n'est absolument pas calibré pour une séance de 1h30
- peut être approfondi par un travail personnel à la maison ou en distanciel

Pour tous les exercices, vous devez être connectés sur une machine sous Linux.
A l'UFR IM2AG, vous avez à disposition également la machine **`mandelbrot`**.
Une fois connectés, ouvrir (au moins) un terminal.

Il est possible de se connecter à distance sur `mandelbrot` (depuis votre poste chez vous ou depuis votre établissement). Nous demander pour plus de précisions.

Nom complet de la machine : `mandelbrot.e.ujf-grenoble.fr`
Pour se connecter et ouvrir un terminal : `ssh login@ mandelbrot.e.ujf-grenoble.fr`

I. Processus

1. Observation de processus : manipulations de base

1. La commande `ps` liste les processus en exécution sur le système. Cette commande vient avec beaucoup d'options d'affichage que l'on peut consulter avec `man ps`. Remarquer le processus du `shell` qui est là pour interpréter les commandes tapées par l'utilisateur.
2. Lancer un éditeur de texte en tâche de fond `gedit &`. Regarder la liste de processus pour identifier le nouveau processus lancé correspondant. Pur voir tous les processus d'un utilisateur, vous pouvez utiliser `ps -u <nom de login>`
3. Lancer le programme `hello.py`. Pouvez-vous l'observer avec la commande `ps` ? Pour le voir dans la liste, il faudrait que son temps d'exécution soit suffisamment long pour vous laisser le temps de taper `ps`! Pour remédier à cela, vous pouvez rajouter une attente artificielle en utilisant la fonction `time.sleep(nb_de_secondes)`.
4. Observer l'arborescence des processus avec `ps j` ou `ps f` et voir qui est le père du processus `gedit`.
5. Lister tous les processus avec `ps -A`
6. Avec `ps aux` vous pouvez voir l'utilisateur, le nom complet de l'exécutable et des informations sur l'utilisation des ressources système et sur les états des processus. Typiquement, vous pouvez observer les colonnes suivantes¶

Colonne	Information
CPU	Utilisation du CPU
MEM	Utilisation mémoire
COMMAND	La commande qui a lancé le processus
PID	Identifiant du processus
PPID	Identifiant du processus père
S ou STAT	Etat du processus
TT ou TTY	le terminal qui lui est associé
UID	propriétaire

7. Dans quel état est le processus `gedit` ?
8. Vous pouvez terminer un processus avec la commande `kill <numéro de processus>`. Si cela ne marche pas, forcer l'arrêt avec `kill -9 <numéro de processus>`
9. Relancer le processus `gedit` mais cette fois-ci sans le & pour le lancer en premier plan (vous n'avez plus la main sur le shell). Taper `Ctrl+Z` pour le suspendre. Vérifier son état avec `ps u`. Vous pouvez maintenant tuer le processus (`kill` numero processus), le relancer en premier plan (`fg` num de job où le numéro de job peut être trouvé avec la commande `jobs`) ou en arrière plan (`bg`)

2. Observation de processus : pour les curieux

1. La commande `top` ou `htop` donne la consommation de ressources des différents processus en temps réel. En faisant `man top` vous pouvez trouver la signification des différentes colonnes, ainsi que les commandes pour changer l'affichage.
2. Sous Unix, toutes les informations sur les processus peuvent être trouvées dans le répertoire `/proc`. Si vous faites `ps` et repérez le PID d'un de vos processus, vous serez en mesure de trouver un sous-répertoire de `/proc` ayant ce numéro pour nom. Le contenu de ce répertoire donne les nombreuses informations sur ce processus.
   Pour plus d'informations, vous pouvez utiliser `man proc`.

3. Création de processus de manière programmatique (`fork()`)

L'appel système `fork()` crée un nouveau processus par clonage du processus appelant : le nouveau processus créé est appelé *fils*, tandis que le processus appelant est appelé *père*. En vertu du clonage effectué, à l'issue du `fork()`, les deux processus exécutent le même programme mais se distinguent par la valeur de retour du `fork()` :
— dans le père : le numéro (pid) du processus fils est retourné
— dans le fils : la valeur 0
Bien entendu, les processus se distinguent également au niveau du système par des numéros (PID) différents. En cas d'échec (table des processus pleine), aucun processus n'est créé par `fork()`, et la valeur −1 est renvoyée au processus appelant.

1. Exécuter et observer les sorties des programmes Python fournis `1_fork_illustration.py`, `2_fork_illustration.py` et `3_fork_illustration.py`.

   Veillez à ce que vos fichiers aient les droits d'exécution (`chmod u+x nom_fichier`)
2. Ecrire un programme où le processus initial (main) crée trois fils et le deuxième fils crée deux autres fils (des petits-fils du main) à son tour.

3. Ecrire un programme qui effectue la création de processus en arbre i.e le processus main doit créer N processus fils, N étant donné en ligne de commande.
4. Ecrire un programme qui effectue la création de processus en chaine i.e le processus main doit créer 1 processus fils, qui lui même doit créer 1 fils etc. La longueur de la chaine doit être N+1, N étant donné en ligne de commande.

4. Communication entre processus : pour les très curieux

Cette partie n'a pas été discutée en cours et sera à détailler (nous demander...).

Le processus peuvent communiquer à travers différents mécanismes, dont :

- les `signaux`

  Un signal est un événement logiciel (généré/géré par le système d'exploitation) pour notifier qqchose à un processus. La liste des signaux peut être vue avec `man signal`. Entre autres, elle comprend :

  SIGINT : signal pour interrompre (terminer) un processus (Ctrl-C)

  SIGCHLD : processus reçu par le père quand le fils change d'état

  Voir en Python le module `signal`

- les `tubes` (pipes)

  En Python, module `os. (pipe())`

  Les tubes peuvent être expérimentées en ligne de commande, p.ex `ls -l | wc -l` mais peuvent aussi être programmés.

- la communication par messages au-dessus du réseau (typiquement via les `sockets`)
  Cette partie sera étudiée dans la partie "Réseaux".

II. Threads

5. Programmation et exécution de programmes multi-threadés

1. Considérer et exécuter les programmes fournis `0_threads.py, 1_threads.py et join_threads.py`. Observer l'entrelacement/l'ordre d'exécution des threads.
2. Modifier le programme join_threads.py pour que le graphe d'attentes soit
   `main -> thread1 -> thread3 -> thread2 -> thread4`
   Attention : Python donne automatiquement des noms aux threads créés (Thread-1, Thread-2, etc. dans l'ordre de création). Il s'agit donc de faire attendre par main le 1er thread créé, qui lui-même devrait attendre le 3ème thread créé, etc.

3. Ecrire un programme qui crée des threads pour représenter des voitures. Le programme doit demander à l'utilisateur de rentrer plusieurs marques de voiture et d'indiquer le nombre de voitures pour chaque marque. Ensuite, il doit lancer autant de threads que de voitures. Chaque thread-voiture devrait simuler l'activité d'une voiture pendant une journée : sortir du garage, rouler, attendre, essence, rentrer au garage. A chaque changement de phase, le thread doit écrire un message. La durée d'une phase sera à simuler avec des attentes (time.sleep).

4. Ecrire un programme (sans synchronisation) pour le problème des lecteurs-rédacteurs. Il s'agit d'accès concurrents à une ressource partagée par deux types d'entités : les lecteurs et les rédacteurs. Les lecteurs accèdent à la ressource sans la modifier. Les rédacteurs, eux modifient la ressource. Le programme devrait lancer des threads rédacteurs et des threads lecteurs et effectuer des affichages correspondant aux moments où les threads accèdent à la ressource partagée.

   Vous pouvez considérer que la ressource partagée est une variable entière. Le fonctionnement des lecteurs et des rédacteurs devrait ressembler à :

```
LECTEUR
print("j'accède à la ressource")
//lecture de la valeur
```

```
REDACTEUR
print("j'accède...")
//modification
```

5. Synchronisation entre threads

1. Considérer et exécuter le programme fourni `pb_threads.py.`
2. Modifier le programme `pb_threads.py` afin d'utiliser des valeurs entières. Le problème est-il toujours présent?
3. Exécuter le programme `lock_threads.py.` Le problème de synchronisation est-il résolu?
4. Considérer le programme `deadlock.py` et comprendre la situation d'interblocage
5. Modifier le programme `deadlock.py` afin de créer un graphe d'attentes à l'aide de trois verrous.

6. Mettre en évidence des problèmes de synchronisation pour le problème de lecteurs-rédacteurs.

Les verrous, dont nous avons brièvement parlé en cours sont une solution de synchronisation très simple. Dans beaucoup de situations, les threads ne doivent se bloquer que si certaines conditions sont remplies. Pour cela, il existe d'autres structures de synchronisation dont les *variables conditionnelles* (appelées également *conditions*). En Python, l'interface est la suivante :

- `condition = threading.Condition()` création d'une condition
- `condition.acquire(), condition.release()` derrière chaque condition, il y a un verrou. Manipuler la ressources partagée nécessite donc de prendre/relâcher ce verrou ce qui se fait à l'aide de ces deux fonctions.
- `condition.wait()` : cette fonction est intéressante : elle s'utilise dans la construction `tant que la condition n'est pas satisfaite wait()` i.e tant que la chose que le thread attend n'est pas survenue, il va se mettre en attente (va s'endormir)
- `condition.notify() :` c'est la fonction qui permet de réveiller un thread.

7. Regarder, comprendre et exécuter le programme `parking.py` qui simule l'activité d'un parking.
8. Ecrire une solution de synchronisation pour le problème des lecteurs-rédacteurs.