DU ISN 2019-2020
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 | avancé | intermédiaire

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`

Ce TP traite des points suivants :
- Manipulation et observation de processus en ligne de commande
- Création programmatique de processus
- Communication entre processus

1. Observation et manipulations de base de processus

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 longue pour vous laisser le temps de taper `ps`! Pour remédier à cela, vous pouvez rajouter une attente artificielle dans le programme 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`. En d'autres termes, toutes les informations dont dispose le système d'exploitation sont reflétés sous la forme d'un répertoire spécial. 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`.

   Par exemple, sur mandelbrot le début de l'aide de man proc commence comme ceci :

   ```
   Le système de fichiers proc est un pseudosystème de fichiers qui
   fournit une interface avec les structures de données du noyau.
   Il est généralement  monté  sur  /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`
   `3_fork_illustration.py, multiple_process_fork.py`

   Ce sont les exemples vus en cours.
   Veillez à ce que vos fichiers aient les droits d'exécution (`chmod u+x nom_fichier`)

   Observer l'ordre d'exécution des processus : que pouvez-vous constater en ce qui concerne l'ordonnancement?
   Sauriez-vous retrouver/lire les arborescences (les relations père/fils) de processus?
   Pour ce dernier point, vous pouvez encore ralentir artificiellement les processus avec des appels à la fonction `sleep`.

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.

   Un exemple de récupération et de manipulation des paramètres passés en ligne de commande en Python est donné dans **`python_args.py`**

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. Exécution d'un programme au sein d'un processus (**`execve()`**)

L'appel système **`execve()`** ne crée pas de nouveau processus mais change le contenu/la nature du processus existant. C'est une fonction qui, si elle réussit, ne retourne pas. Elle est utilisée typiquement dans le fonctionnement des interprètes de commande (**`shell`**). En effet, le shell lit ce que l'utilisateur tape, si ce n'est pas une commande qu'il connaît (implémente, on appelle ces commandes des commandes intégrées), il crée un nouveau processus avec **`fork()`** et fait exécuter à ce processus l'exécutable correspondant en utilisant **`execve`**.

1. Regarder le code source et exécuter le programme **`my_ls.py`**, qui illustre l'utilisation de cette fonction.

2. Implémenter un pseudo-shell, c'est à dire un programme qui demande à l'utilisateur de choisir parmi quelques commandes standard et les exécute pour lui.

   Vous devriez suivre le schéma standard qui consiste à créer un processus fils avec **`fork`**() et le faire exécuter la commande en utilisant une des fonctions de la famille **`exec`** (pour **`exec`** voir https://docs.python.org/3/library/os.html). Le processus principal devrait attendre (**`waitpid`**) la terminaison du processus fils avant de retourner vers l'utilisateur (i.e. tout est exécuté en premier plan)

5. Communication entre processus : pour les très curieux

Cette partie est en rouge tout simplement parce qu'elle n'a pas été traitée en cours.

Le processus sont par défaut isolés et ne partagent pas de mémoire. Leur communication et coordination est donc difficile. Néanmoins, il existe plusieurs mécanismes pour faire communiquer les processus. Une partie des solutions est basée sur de la mémoire partagée gérée par le SE. Il s'agit des signaux, de la mémoire partagée et des tubes. Les processus peuvent également communiquer en utilisant des messages au-dessus du réseau (typiquement via les `sockets`). Dans ce cas là, la communication est gérée comme si les processus s'exécutaient sur des machines différentes i.e. sans mémoire partagée. *Les* `sockets` *seront étudiés dans la partie "Réseaux".*

1. Les **`signaux`**

Un signal est un événement logiciel (généré/géré par le système d'exploitation) pour notifier quelque chose 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 : signal reçu par le père quand le fils change d'état

SIGTSTP : signal pour suspendre l'exécution d'un processus (`Ctrl-Z`)

Les processus fournissent des traitants de signaux qui sont des fonctions qui s'exécutent à la réception d'un signal. Il existe des traitants par défaut et pour beaucoup de signaux ces traitants peuvent être redéfinis.

En Python, il existe le module `signal.`

1. Regarder le code source et exécuter le programme `signal1.py`, qui illustre lle fonctionnement des signaux. Une fois le programme lancé, vous pouvez essayer de taper Ctrl-C, Ctrl-Z ou alors attendre.
2. Regarder le code source et exécuter le programme `signal2.py` où le processus père capture un signal lors de la terminaison du fils.
3. Regarder le code source et exécuter le programme `signal3.py` où le processus père envoie un signal au fils. Vous pourriez également envoyer des signaux aux processus en utilisant
   `kill -<nom ou numéro de signal> <pid>`
   par exemple, dans le cas de ce programme, pour envoyer le signal `SIGUSR1`
   `kill -USR1 10234`

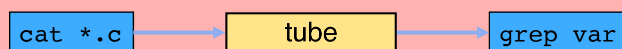
2. Les `tubes`

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

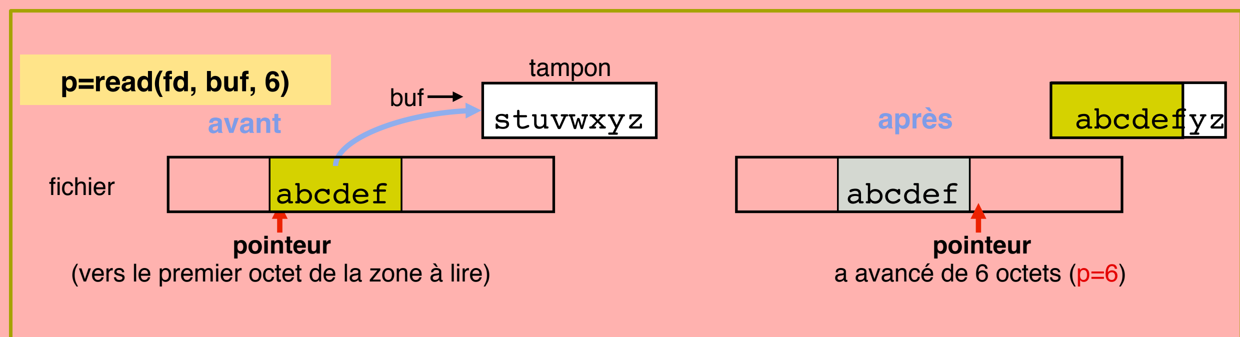
Un tube est un fichier anonyme qui sert de tampon entre deux processus fonctionnant en producteur-consommateur. La commande

```
cat *.c | grep var
```

a) crée un tube et deux processus : p1 qui exécute cat *.c, p2 qui exécute grep var
b) connecte la sortie de p1 à l'entrée du tube et l'entrée de p2 à la sortie du tube

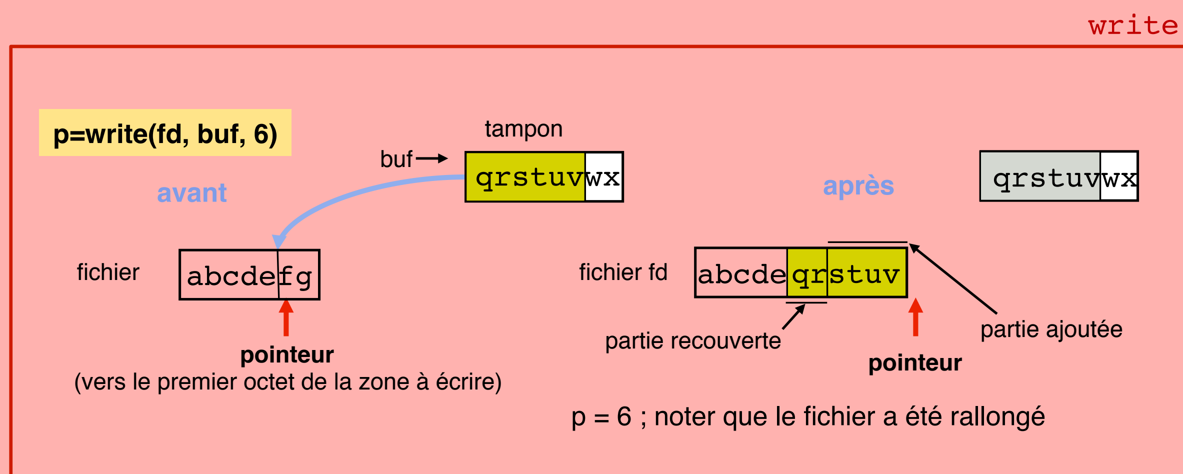


Les tubes sont des fichiers spéciaux. Ils sont manipulés à travers la même interface :
read & write.



`read`

`write`



La primitive pipe() crée un tube, dont l'entrée et la sortie sont associées à des descripteurs de fichier choisis par le système.

Si OK, elle crée un tube, et renvoie le descripteur pour écrire dans le tube, ainsi que le descripteur pour y lire.

Si elle échoue, elle renvoie -1

Un père peut communiquer avec un fils à travers un tube.

1. Regarder le code source et exécuter le programme `pipe_example.py`, qui illustre le fonctionnement des tubes.

I. ANNEXE : Ligne de commande et commandes de base Linux

1. Pour démarrer

Lancer un terminal. Dans ce terminal, vous devez voir un prompt qui vous invite à taper des commandes.

2. Où est-vous ? Chemins dans l'arborescence de fichiers.

En lançant un terminal, si vous ne faites rien de spécial, le répertoire dans lequel vous vous trouvez (répertoire courant) correspond à votre "maison" (typiquement quelque chose comme `/home/users/nom-de-login`).

Pour vérifier où est votre "maison", vous pouvez vérifier la valeur de la variable d'environnement `HOME` avec
```
echo $HOME
```

Pour voir où vous vous trouvez dans l'arborescence de fichier, taper
pwd

Cette commande donnera l'emplacement du répertoire courant à partir de la racine de l'arborescence (la racine est marquée /). Ce chemin est appelé *chemin absolu*. Un *chemin relatif* est un chemin qui considère un emplacement à partir du répertoire courant.

Le répertoire courant est désigné par `.`
Le répertoire père est désigné par `..`
Le répertoire de votre "maison" est désigné par ~

3. Changer de répertoire courant (cd)

Pour changer de répertoire courant, la commande à utiliser est cd (change directory)
Essayer par exemple (ce sont des commandes qui utilisent des chemins relatifs)
```
cd .
cd ..
cd ~
```

Pour tester les chemins absolus
```
cd /usr/bin
cd le_chemin_absolu_que vous_avez_vu_avec_pwd_au_debut
```

4. Lister le contenu d'un répertoire (ls)

Tester dans le répertoire courant.
```
ls
ls .
```

```
ls -l
ls -la
```

Pour voir toutes les options de ls, faire `man ls`.

5. Créer un répertoire (mkdir)

Tester (dans votre répertoire racine (= maison))

```
mkdir DU_SR
ls
mkdir DU_SR/TP1
ls DU_SR
cd DU_SR/TP1
ls -la
```

6. Copier un fichier (cp)

Créer le fichier dans `~/DU SR/TP1/fichier.txt` (avec votre éditeur préféré).
Copier le fichier avec (si vous êtes dans `~/DU SR/TP1`)

```
cp fichier.txt copie.txt
```

Vérifier le contenu du répertoire `~/DU SR/TP1` (en ligne de commande) Vérifier le contenu de copie.txt (en l'ouvrant avec votre éditeur ou tout simplement avec cat copie.txt.

7. Deplacer un fichier (mv)

Essayer

```
mv copie.txt ..
```

Regarder le contenu du répertoire courant et du répertoire père.

8. Effacer un fichier (rm)

Essayer et vérifier les effets de la commande suivante

```
rm ~/DU SR/copie.txt
```

9. Effacer un répertoire (rmdir)

`rmdir` ne marche que si le répertoire est vide. Pour effacer un répertoire non vide, utiliser `rm -rf`.
Effacer le répertoire DU SR.