TP systèmes d'exploitation

1 Caractéristiques d'un processus

A. Écrivez une fonction getpinfo() qui affiche les caractéristiques du processus, en particulier : l'ID du processus et de son père, l'utilisateur et le groupe réels et effectifs, etc. Utilisez les fonctions `getpid(2)`, `getppid(2)`, `getuid(2)`, `geteuid(2)` etc. Vous pouvez aussi essayer `getcwd(3)`, `times(2)`, etc. On va utiliser cette fonction un peu partout dans cet exercice, écrivez sa définition dans un fichier getpinfo.c, et sa déclaration dans un fichier getpinfo.h.

   Écrivez aussi un petit programme appelé pinfo qui teste cette fonction.

B. Écrivez un programme qui donne lieu à quatre processus : le processus initial, un fils et deux petit-fils. Pour chaque processus, exécutez votre fonction getpinfo(). N'oubliez pas de sortir proprement de votre programme, en attendant la fin des processus fils créés avant de quitter.

C. Reprenez votre programme pinfo et positionnez le bit SUID sur l'exécutable.

   Puis écrivez un programme qui crée un processus fils, lequel exécute le programme pinfo de votre voisin. Appelez votre fonction getpinfo() d'une part dans le processus père, et d'autre part dans le processus fils avant l'appel à `exec..()`.

2 Word-count en parallèle

Il s'agit ici de compter les lignes, mots et caractères de plusieurs fichiers. Le programme doit :
- lancer un processus fils pour chacun des noms de fichiers passés en paramètre,
- chaque fils doit exécuter le programme wc sur le fichier dont il s'occupe,
- le père doit finalement afficher le nombre de fils qui ont échoué (par exemple parce que le fichier n'existe pas ou n'est pas lisible).

Important : évitez de lancer ce programme sur un grand nombre de fichiers !

Dans un deuxième temps, vous modifierez ce programme de façon à ce qu'il ne lance jamais plus de 10 processus simultanés. Vous pouvez tester ce nouveau programme, par exemple par : "wcp /usr/include/*" (il y a plus de 200 fichiers et répertoires dans /usr/include).

Enfin, posez vous la question de savoir ce qu'il manque à votre programme pour que le père puisse, à la fin de l'exécution de tous ses fils, afficher les totaux (comme le fait le programme wc).

3 Parcours de répertoire

Cet exercice est optionnel.

Écrivez un programme qui parcourt une hiérarchie de répertoires. L'unique paramètre du programme est un nom de répertoire. Chaque processus doit parcourir un répertoire et :
- lancer un processus fils pour chaque sous-répertoire,
- attendre la fin de ses fils,
- afficher le nombre de fichiers et le nombre de sous-répertoires du répertoire dont il a la charge.

Il s'agit donc d'un parcours postfixé d'une hiérarchie.

Notes :

Utilisez `opendir(3)`, `readdir(3)` et `closedir(3)` pour parcourir le contenu d'un répertoire ;

Utilisez `stat(2)` pour déterminer si une entrée du répertoire est un fichier ou un répertoire (vous pouvez ignorer les autres cas).

Si vous craignez de générer trop de processus, vous pouvez utiliser la commande "ulimit -u 10" pour limiter (ici à 10) le nombre de processus fils du shell courant.

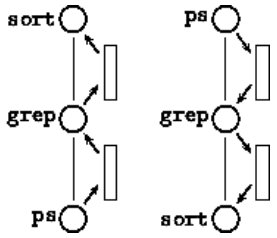
Question : est-il possible d'inverser l'ordre d'affichage (c'est-à-dire d'afficher les renseignements sur un répertoire avant les renseignements sur ses sous-répertoires) ?

4 Redirection des flux d'entrée et de sortie

1. Écrire un programme qui prend en paramètre deux noms de fichiers, et exécute `cat -n` avec une entrée standard redirigée sur le premier fichier et la sortie standard redirigée sur le second fichier. Votre programme, appelé par exemple par : `redir entree sortie`
   a exactement le même effet que la commande : `cat -n <entree >sortie`
2. Écrire ce même programme, sans faire appel à l'exécutable `cat`.

5 Pipelines : `ps|grep|sort`

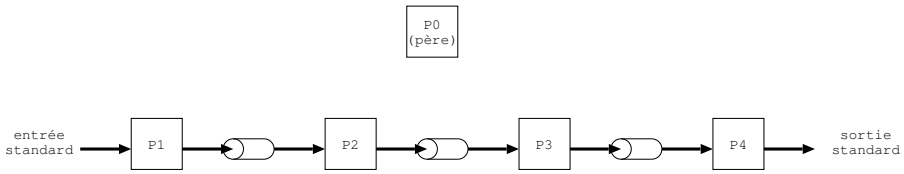
Écrire le programme correspondant à la commande : "`ps -fade | grep $LOGNAME | sort`", selon les deux schémas d'organisation père-fils des processus suivants :



6 Ncat

Écrire un programme qui prend en paramètre un entier n, puis qui crée n processus. L'entrée de chaque processus numéro i est redirigée vers la sortie du processus $i-1$, et la sortie de i redirigée vers l'entrée de $i+1$. L'entrée du premier processus est l'entrée standard, la sortie du dernier processus la sortie standard. Tout ce qui est envoyé à l'entrée du premier proccessus doit ressortir dans la sortie du dernier, donc ce programme doit simplement afficher ce qu'on tape.

Voici un exemple pour $n = 4$:



7 Pipes nommés

Cet exercice est optionnel.

Un tube nommé est un fichier spécial qui se comporte comme un pipe, mais qui est présent sur le filesystem. Un processus peut ouvrir un tube nommé soit en lecture soit en écriture, puis y lire ou y écrire. Un tube nommé est créé soit par la commande `mkfifo(1)` ou `mknod(1)`, soit par un programme utilisant l'appel système `mknod(2)`. L'ouverture, la lecture, l'écriture et la fermeture d'un tube nommé se font exactement comme pour un fichier normal.

Note : Si vous travaillez sur une machine Linux, votre répertoire est monté par NFS depuis une machine serveur. Il se trouve que NFS ne supporte pas bien les tubes nommés. Pour éviter les problèmes, il suffit de le créer sur un disque local, par exemple dans le répertoire /tmp.

Réalisez deux programmes, un *écrivain* lisant l'entrée standard et écrivant dans un tube nommé, et un *lecteur* lisant le contenu du tube nommé et écrivant sur sa sortie standard. Remarque : il existe une commande système qui permet de faire cela très facilement, inutile de la programmer en C.

Créez un tube nommé, et lancez un écrivain et un lecteur, chacun dans sa fenêtre. Vérifiez que vos programmes produisent le résultat attendu.

Lancez ensuite plusieurs écrivains et un seul lecteur, tous sur le même tube nommé. Observez ensuite ce qui se passe si vous avez plusieurs lecteurs sur le même tube nommé. Essayez de brancher un écrivain sur le tube nommé de votre voisin (qui doit, lui, lancer un lecteur).

Essayer d'imaginez comment vous pourriez mettre au point un système de communication multipoint (c'est-à-dire plusieurs utilisateurs pouvant chacun lire ce que tous écrivent, et écrire à tous) à l'aide de tubes nommés.

8 Word-count

On reprend ici l'exemple du word-count sur plusieurs fichiers, traité chacun par un processus différent. Il y a ici deux différences par rapport à l'exercice 2 : il faut écrire une fonction qui compte le nombre de caractères (et de lignes et de mots si vous voulez) présents dans un fichier ;

Le processus lance autant de *threads* qu'il reçoit de fichiers en paramètres (mais pas plus de 10 en même temps), et écrit le nombre de caractères du fichier dont il a la charge dans une variable globale. Mettez en œuvre un dispositif permettant au thread maître d'imprimer le total du nombre de caractères (et de lignes et de mots...) à la fin de son exécution.
Note : N'oubliez pas l'option de compilation `-D_REENTRANT`, et l'option de linkage `-lpthread` si nécessaire.

9 Producteur-consommateur

Écrivez cinq programmes :

1. un programme qui crée un segment de mémoire partagée (shm dans la suite) pouvant contenir un entier ;
2. un programme qui détruit le segment créé par le premier programme (ce deuxième programme n'est pas absolument nécessaire, mais il facilite la vie) ;
3. un programme (appelé producteur) qui incrémente l'entier contenu dans le shm toutes les N secondes (N est un paramètre du programme) ;
4. un programme (appelé consommateur) qui décrémente l'entier contenu dans le shm toutes les M secondes (M est aussi un paramètre) ;
5. un programme de visualisation qui affiche l'entier contenu dans le shm toutes les S secondes.

Ensuite, après avoir lancé le programme de création du shm, lancez plusieurs producteurs et plusieurs consommateurs, et examinez l'évolution du contenu du shm à l'aide du programme de visualisation. N'oubliez pas de détruire le shm à la fin de l'expérience.

Remarque. Ce système n'est pas correct en tant que tel : que se passe-t-il si plusieurs processus accèdent à l'entier en lecture puis en écriture en même temps ?

Peut-on éviter ce problème ?

10 Word-count

Reprenons l'exemple du word-count sur plusieurs fichiers, traité chacun par un thread différent. Il faut ici utiliser des processus, qui communiquent par l'intermédiaire d'un segment de mémoire partagée et de signaux.

Évitez de stopper et relancer les processus effectuant le travail, cette opération est trop coûteuse : une fois que les 10 processus sont créés, distribuez leur le travail à effectuer par l'intermédiaire du segment de mémoire partagée et des signaux (par exemple).

Mettez en œuvre un dispositif permettant au programme père d'imprimer le total du nombre de caractères (et de lignes et de mots...) à la fin de son exécution.

11 Tac

Écrivez un programme qui utilise un fichier projeté en mémoire pour effectuer une inversion par ligne, c'est-à-dire qu'il imprime les lignes du fichier dans l'ordre inverse de leur apparition (la commande `tac(1)` fait ça).

1. en parcourant le fichier octet par octet avec `lseek`.

2. en utilisant `mmap`.

Vous comparerez les durées d'exécution des deux versions.

12 Les cuisinières et le quatre-quart

Pour confectionner un quatre-quart, il faut : du sucre, du beurre, de la farine, des oeufs. Six cuisinières (6 threads) et un épicier (le thread master) collaborent de la manière suivante :

- chaque cuisinière possède deux ingrédients en quantité infinie, parmi les quatre ingrédients (c'est pour cela qu'il y en a 6) ;
- l'épicier apporte successivement deux ingrédients (choisis aléatoirement) en quantité suffisante pour la confection d'un gateau ;
- une des cuisinières doit prendre ces deux ingrédients, et avec les deux autres qu'elle possède elle confectionne un gateau (`sleep(rand()%5);` par exemple), puis elle envoie le gateau à l'épicier ;
- l'épicier attend qu'un gateau soit confectionné, avant d'apporter deux nouveaux ingrédient aux cuisinières.

Implémentez ce mécanisme, en évitant les boucles actives, les interblocages et les risques de conflit d'accès aux données partagées. Vous pouvez utiliser tous les mécanismes de synchronisation entre threads que vous connaissez.

Utilisez la fonction `time(2)` pour afficher la durée de confection de chaque gateau dans le thread master. Affichez également le status du thread master à chaque seconde à l'aide de la fonction `setitimer(2)` et des redirections de signaux.

13 PROJET :

(en cours)