TP Système n°1 - Signaux et Processus Lourds

Table of Contents

1. Prise en main

Écrire un programme `sigusr1.py` qui :

- se protège contre le signal `SIGTERM` (on ne peut pas le tuer par un `kill <pid du processus>`).
- se protège contre le signal `SIGINT` (on ne peut pas le terminer par un Ctrl+C)
- attend un signal `SIGUSR1` pour se terminer en affichant un message à l'écran.

Exemple d'exécution dans un terminal :

SHELL

```
% python sigusr1.py
je boucle sans fin. Arrêtez-moi avec un kill -USR1 12681
^C (aucun effet...)
Dans un autre terminal :
% kill 12681 (aucun effet...)
% kill -USR1 12681 (le processus s'arrête et affiche "Ok, signal reçu, je m'arrête...")
```

2. Création de processus

Que produit le code suivant ? (essayez de déduire la réponse avant d'exécuter le code) :

PYTHON

```
import os

for i in range(4):
    if os.fork() != 0:
        print(f"PID = {os.getpid()}, i = {i}")
```

Donnez la relation entre le nombre de boucles et le nombre de lignes affichées.

Représentez les créations de processus par un arbre.

3. Addition et multiplication

Écrire le programme `addMult.py` qui demande à l'utilisateur deux entiers, crée deux processus fils, le premier additionne les deux entiers et affiche le résultat, le deuxième multiplie les deux entiers et affiche le résultat. Avant de quitter, le père informe l'utilisateur de la fin du calcul.

1. Comment le père communiquera les deux entiers à ces fils?
2. Peut-on garantir l'affichage de l'addition avant la multiplication ou inversement? Justifiez.
3. Que doit faire le père pour s'assurer qu'il informe l'utilisateur de la fin du calcul au bon moment?

4. ls et wc

Écrire le programme `lsWc.py` qui prend en paramètre le nom d'un fichier, et affiche le résultat de ls et wc sur ce fichier.

1. A quelle primitive doit faire appel votre programme pour exécuter ls et wc?
2. Quelle conséquence a cette primitive sur votre programme?
3. Sachant que votre programme devra afficher un message informant l'utilisateur de la fin du traitement, quelle solution pouvez-vous mettre en place?

5. Affichage paramètres en parallèle

Écrire la commande `affiche.py` qui prend un nombre quelconque de paramètres, crée autant de fils que de paramètres reçus et délègue à chaque fils l'affichage d'un paramètre.

- Peut-on garantir l'affichage des paramètres dans le même ordre de leur réception?

6. Un mini shell

Écrire un programme `minishell.py` qui répète les actions suivantes dans une boucle :

- Affiche le prompt `$` pour indiquer qu'il attend une commande shell sans paramètre (par exemple, `ls` , `ps` , `python` , `code` , etc).
- Lance cette commande et indique si elle a été correctement exécutée en affichant un message commençant par SUCCES ou ECHEC, selon le cas

Cette boucle se continue jusqu'à ce que l'utilisateur mette fin à la saisie en fermant l'entrée standard avec la combinaison de touches `Ctrl` + `D` (qui est détectée par l'exception `EOFError`) : le shell se termine alors en affichant le message "Bye..."

Comme amélioration, vous pouvez également faire en sorte que la boucle du shell se termine aussi si l'on tape la commande `exit` .

7. Compiler en parallèle

Écrire la commande `compile.py` qui prend un nombre quelconque de fichiers sources C en paramètre, plus un nom de fichier programme. Cette commande compile tous les fichiers source en parallèle et, s'il n'y a aucune erreur, fait une édition de liens pour produire le programme.

La commande python **compile.py toto.c titi.c tata.c tutu.c monprog** produit donc le programme exécutable `monprog` en compilant `toto.c` , `titi.c` , `tata.c` et `tutu.c` (à récupérer d'IRIS) en parallèle, puis fait l'édition de liens des fichiers objets obtenus pour produire le binaire exécutable `monprog` .

Ce programme doit gérer les éventuelles erreurs de compilation : chaque processus doit donc produire un fichier contenant les éventuelles erreurs. Si tout se passe bien, ces fichiers d'erreurs devront être automatiquement supprimés.

Évidemment, il suffit qu'une compilation échoue pour que l'édition de liens n'ait pas lieu. À la place, un message d'erreur informera l'utilisateur de cet échec et l'invitera à consulter les fichiers d'erreur.

Rappels

- Pour compiler un fichier .c afin d'obtenir un fichier objet :

```shell
% c99 -c fic.c (produit un fichier fic.o)
```

- Pour faire l'édition de liens afin d'obtenir un exécutable :

```shell
% c99 fic1.o fic2.o fic3.o -o executable
```

8. Compter en parallèle

La commande Unix `wc` permet de compter les lignes, mots et caractères d'un fichier texte :

```shell
$ wc /etc/passwd
21 59 1165 /etc/passwd
```

Écrire un programme `wcp` permettant de compter les lignes, mots et caractères de plusieurs fichiers. Le programme qu'il vous est demandé d'écrire doit :

1. lancer un processus fils pour chacun des noms de fichiers passés en paramètre ;
2. chaque fils doit exécuter le programme wc sur le fichier dont il s'occupe ;
3. 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).

Le programme doit évidemment vérifier que sa syntaxe d'appel est correcte.

Exemple de sortie :

```shell
$ python3 wcp.py /etc/passwd /etc/group /etc/service
21 59 1165 /etc/passwd
26 26 368 /etc/group
1 échec(s)

$ python3 wcp.py /etc/passwd /etc/group
26 26 368 /etc/group
21 59 1165 /etc/passwd
```

Last updated 2022-01-24 21:26:58 +0100