TP 8

Processus et mini-shell

Durée prévue : 2 séances (3h)

L'archive du TP : Téléchargez l'archive `mini-shell.tar.gz` à partir du site de l'UE. Désarchivez-la *dans votre répertoire de travail* pour l'UE (avec `tar xvzf mini-shell.tar.gz` par exemple)

xclock : Dans les exercices de ce TP, on utilise la commande `xclock -update 1` : elle affiche dans une fenêtre graphique une horloge avec une trotteuse (voir Figure 8.1 ce qui est pratique pour constater si le processus correspondant est en cours d'exécution (la trotteuse progresse) ou bien stoppé (la trotteuse est arrêtée). Cette commande peut ne pas être disponible sur votre machine de TP. Voici quelques solutions :

- Sur le machine des salles de TP, vous pouvez utiliser le script `clock.py` de l'archive : vous remplacez la commande `xclock -update 1` par `./clock.py` depuis votre répertoire contenant les fichiers de l'archive du TP.
- Si vous êtes sur votre ordinateur sous Linux, vous pouvez l'installer ! Sous Debian/-Mint/Ubuntu le package contenant cette commande s'appelle `x11-apps`, donc `sudo apt install x11-apps` doit faire l'affaire. Vous pouvez tenter l'installation aussi si vous utilisez WSL.
- En dernier recours, vous pouvez utiliser une autre commande qui lance une fenêtre graphique, par exemple `gedit` : dans ce cas, vous constaterez que le processus est stoppé quand vous ne pourrez plus entrer de texte dans `gedit`.

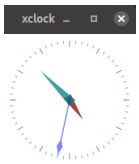

FIGURE 8.1 – xclock, un utilitaire so 20ième siècle !

8.1 Processus, signaux

EXERCICE 1 ▶ **Signaux en ligne de commande**

Les signaux sont une manière simple de communiquer avec les processus, sans passer par les entrée et sortie standards. On expérimente ici simplement pour pratiquer les bases.

On a déjà utilisé les signaux lors d'exercices précédents, nous allons ici pratiquer un peu les signaux à la ligne de commande. Un signal est un code entier `sig`, que l'on peut envoyer à un processus identifié par son `pid` : la commande `kill -sig pid` envoie le signal `sig` au processus `pid`. En pratique, on n'utilise pas les codes entiers, mais des noms plus faciles à retenir :

— `TERM` pour demander aimablement au processus de se terminer : certains processus peuvent l'ignorer.
— `KILL` pour demander énergiquement au processus de se terminer : aucun processus ne peut l'ignorer.
— `STOP` pour demander au processus de s'arrêter.
— `CONT` pour demander au processus de reprendre son exécution.

Par exemple, pour envoyer le signal `CONT` au processus 69007, la commande à utiliser est `kill -CONT 69007`. On ne va travailler qu'avec ces quatre signaux. En fait, il en existe une (très) longue liste : tapez `kill -l` pour afficher toute la liste.

Ouvrez deux terminaux. Dans l'un, lancer `xclock -update 1` (sans éperluette - `&`). Dans l'autre terminal, à l'aide de la commande `ps -e | grep xclock` (il y a aussi plus direct, mais pas forcément facile d'y penser : `pgrep xclock`), déterminez le `pid` du processus `xclock` que vous venez de lancer, et notez-le.

Depuis le terminal qui n'est pas monopolisé par `xclock`, effectuez les actions suivantes.

1) Envoyez le signal `STOP` à `xclock` : qu'observez-vous ?

.....

2) Envoyez le signal `CONT` à `xclock` : qu'observez-vous ?

.....

3) Envoyez le signal `TERM` à `xclock` : qu'observez-vous ?

.....

4) Faites les mêmes manipulations avec `gedit` : lorsque vous envoyez `STOP` à `gedit`, que constatez-vous ?

.....

.....

5) A votre avis, quand dans un terminal vous tapez `Ctrl+C` pour fermer un processus, que se passe-t-il ?

.....

8.2 Mini-shell

L'objectif de votre travail pour ce TP est de :

— comprendre l'algo du shell (pseudocode).

— comprendre comment on lance une commande et pourquoi il faut créer un nouveau processus pour cela.

— comprendre la différence de traitement entre arrière plan et premier plan.

— savoir utiliser les primitives `fork()`, `waitpid()`, et la famille des primitives `exec()`.

— comprendre comment gérer le signal `SIGCHLD`.

EXERCICE 2 ▶ **Le mini-shell**

Dans le répertoire produit par l'archive, vous devez trouver le fichier `shell.cpp` dans lequel vous allez travailler, ainsi qu'un `Makefile` pour générer l'exécutable nommé `shell`. Le shell ("le programme du terminal") est un programme qui lit une commande sur l'entrée standard et l'exécute. L'objectif est de reproduire cette fonctionnalité. *Grosso modo*, vous pouvez imaginer pour l'instant que le fonctionnement d'un shell suit l'algorithme suivant :

```
fini ← False
tant que non fini faire
    Afficher une invite
    Lire une commande
    Analyser la ligne de commande
    si la commande est quit alors
        fini ← True
    sinon
        si la commande se termine par '&' alors
            Exécuter la commande à l'arrière plan
        sinon
            Exécuter la commande au premier plan
        fin si
    fin si
fin tant que
```

L'une des premières difficultés est d'analyser la ligne de commande. Dans le programme fourni, cette difficulté est escamotée : le code effectue une analyse qui décompose la ligne de commande dans un format qui sera utilisable par l'appel système `execvp`, et sort de la boucle si la commande est « `quit` ».

Il vous reste donc à gérer correctement l'exécution des commandes. **En l'état, le programme qui vous est fourni se contente d'afficher « `L'exécution d'une commande n'est pas encore implantée.` » à chaque fois que vous entrez une commande à l'invite. Mais vous pouvez quand même taper `quit` pour quitter.**

Vous allez devoir modifier le source `shell.cpp`, surtout dans la partie « `Traitement de la ligne de commande` », en suivant les « `TODO` ». Suivez les consignes de l'énoncé, et aidez-vous aussi des commentaires présents dans le code.

1) Compilez, testez le programme. *Pour l'instant, la commande de compilation vous donne deux* warning *de type "unused variables", c'est normal!*

2) Modifiez le programme façon à ce que votre shell permette d'exécuter une commande simple en premier plan. Cela se passe au niveau du commentaire `TODO 1` (le commentaire `TODO 2` est là aussi, mais c'est pour la question suivante). Vous utiliserez pour cela l'appel système `execvp()`, avec pour paramètres ceux fournis par `splitToChar`. Assurez-vous que :

— la commande lancée par `execvp` soit bien un fils du shell (utiliser `fork()`),

— le shell attende bien que son fils se termine (utiliser un `waitpid()` bloquant),

— le shell continue d'exister après la fin de la commande,

— si la commande lancée est erronée, le shell le signale et retourne dans un état convenable.

Les messages de débogage dans le code fourni commencent tous par `##` : il vous est recommandé de faire la même chose dans tout le TP, de façon à bien distinguer les affichages produits par le shell de ceux produits par les commandes que vous lancez. Voici un exemple d'exécution du shell terminé :

```
Entrez une commande > ls
## Lecture de la commande "ls"
## Lancement de ls
clock       clock.py  Makefile.etu    prev   shell.cpp
clock.cpp   Makefile  mini-shell.tar.gz  shell
```

3) Modifiez votre shell pour pouvoir exécuter les commandes au premier plan ou en tâche de fond (en fonction de la variable booléenne `attend`) :

```
Entrez une commande > xclock -update 1 &
## Lecture de la commande "xclock -update 1 &"
## Ligne de commandes lancée en tâche de fond.
Entrez une commande > ## Lancement de xclock -update 1
```

Pour l'instant, contentez-vous d'utiliser un `waitpid()` bloquant de façon à ce que le shell attende la fin de son fils si la commande est lancée au premier plan ; lorsqu'elle est lancée à l'arrière plan, le shell n'attend pas pour revenir à l'invite de commandes. Les modifications doivent avoir lieu au niveau du commentaire `TODO 2`.

4) Testez votre shell dans deux situations :

— lancez `xclock -update 1` au premier plan ; dans un autre terminal, vérifiez que cette commande est bien fille de votre shell (`ps -lu $UID` ; la variable `UID` est votre identifiant sur le système, et `-u $UID` ne liste donc que vos processus ; `-l` c'est pour avoir aussi l'état des processus) ; fermez `xclock` ; dans l'autre terminal, vérifiez que le processus correspondant a bien disparu du système (encore `ps -lu $UID`).

— lancez `xclock -update 1` à l'arrière plan ; vérifiez que cette commande est bien fille de votre shell ; fermez `xclock` : quel est l'état du processus `xclock` (toujours `ps -lu $UID`) ?

5) Pour gérer (un peu) plus proprement la fin des tâches lancées à l'arrière plan, une solution est de mettre en place dans le shell un gestionnaire pour le signal `SIGCHLD`. En effet, tout processus dont l'un des fils se termine reçoit le signal `SIGCHLD`, et peut donc prendre en compte sa terminaison (avec `waitpid()`) pour qu'il ne reste pas dans l'état zombie. Terminez l'installation du gestionnaire `traitement_signal_fils()` dans le `main()`, au niveau du commentaire `TODO 3`. Recompilez votre shell, et vérifiez que votre gestionnaire de signal à bien été mis en place (pour tester, mettez un processus en arrière plan et le fermer-le.)

6) Complétez le gestionnaire de signal `traitement_signal_fils()`, de façon à ce que, lorsqu'un des fils du processus courant se termine, le gestionnaire prenne en compte la fin de ce processus avec `waitpid()` et affiche son PID. De plus, si le PID du processus qui vient de se terminer est le même que celui qui se trouve dans la variable globale `pid_attendu`, alors le gestionnaire mettra cette variable à 0 (on utilisera plus tard cette variable).

7) Testez votre shell comme précédemment, en lançant puis en fermant un processus, d'abord en tâche de fond, puis au premier plan : quel problème se produit quand un fils lancé au premier plan se termine ?

8) Jusqu'à présent, le shell attend la fin d'un processus lancé à l'avant plan avec un `waitpid()` dans le programme principal. Vous devez modifiez le programme de façon à ce que seul le gestionnaire du signal `SIGCHLD` intervienne pour prendre en compte la fin d'un processus, même lancé au premier plan. Vous utiliserez pour cela l'appel système `pause()`, qui permet de suspendre le processus appelant en l'attente d'un signal.

Désormais, à chaque réception du signal `SIGCHLD`, le shell exécutera le gestionnaire de signal, et sortira de l'appel bloquant à `pause()` : tant que le PID du processus qui vient de se terminer n'est pas celui du processus qui était lancé à l'avant plan, il doit se remettre en attente avec `pause()`. Pour implanter cela, vous utiliserez pour cela la variable globale `pid_attendu` (en lui affectant le PID du processus à attendre ; rappelons qu'elle sera repassée à 0 par le gestionnaire pour `SIGCHLD`).

9) Vérifiez que désormais votre shell prend bien en compte la terminaison de tous ses fils, qu'ils aient été lancés à l'avant ou à l'arrière plan. Notamment, vérifiez bien que si vous lancez plusieurs processus à l'arrière plan et un processus à l'avant plan, vous n'observez aucun zombie après qu'ils se soient terminés : déboguez si tel n'est pas le cas !

10) Maintenant, on veut ajouter au shell la possibilité d'exécuter deux commandes liées par un tuyau (rappelons que l'on dit *pipe* en australien), par exemple `ls -l | less`. Pour cela, vous allez compléter la partie « cas où il y a exactement un '|' sur la ligne de commandes« du code. Dans ce cas, la ligne de commandes est de la forme `com1 | com2` : le shell doit d'abord créer un tuyau `p` avec l'appel système `pipe()`, puis deux fils :

— dans le premier fils (pour `com1`), `STDOUT_FILENO` est remplacé par une copie du descripteur de fichier `p[1]` (descripteur pour l'écriture dans le tuyau),

— dans le deuxième fils (pour `com2`, `STDIN_FILENO` est remplacé par une copie du descripteur de fichier `p[0]` (descripteur pour la lecture dans le tuyau).

Ainsi, tout ce que le premier fils écrit sur sa sortie standard (`STDOUT_FILENO`) est en fait écrit dans le tuyau (`p[1]`), et tout ce que le second fils lit sur son entrée standard (`STDIN_FILENO`) est lu depuis le tuyau (`p[0]`).

Pour qu'un descripteur de fichier `newfd` soit remplacé par une copie d'un descripteur de fichier `oldfd`, il faut utiliser l'appel système suivant :

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Dans chaque fils, l'extrémité du tuyau qui n'est pas utilisée doit être fermée avec `close`. Comme précédemment, `execvp` est utilisé pour lancer chaque commande.

Tout cela doit se passer au niveau du `TODO 5`. Il faudra vous assurer que les deux commandes se terminent proprement (pas de processus zombie), qu'elles soient lancées à l'avant ou à l'arrière plan.

11) *Pour aller plus loin...* Ajoutez la possibilité d'exécuter une commande avec redirection de la sortie standard vers un fichier (par exemple, `ls -l > sortie.txt`), puis la possibilité de gérer les autres redirections (>>, <), ou encore d'exécuter des commandes avec plusieurs tuyaux (par exemple, `ls -l | grep toto | wc -l`).