Programmation et gestion de processus

1 Travail préliminaire

Les fichiers nécessaires au déroulement du BE se trouvent sous Github. Pour les récupérer, exécutez la commande :

```
> git clone https://github.com/ermont/be2.git
```

Vous devriez obtenir un répertoire `be2` dont l'arborescence est la suivante :

```
be2
└── minishell
    ├── Makefile
    ├── minishell.c
    ├── readcmd.h
    └── SAVE
        └── libcmd.o
```

Le travail réalisé au cours du BE décrit dans la section 2 devra être remis sous Moodle et sera noté.

2 Minishell : un interpréteur de commandes simple

L'objectif du bureau d'étude est d'utiliser les différents appels système vus en cours pour réaliser un `minishell`. Les fichiers nécessaires sont disponibles dans le répertoire `minishell`.

Etape 1 (Testez le programme) Compilez le programme en tapant `make` dans le répertoire du répertoire `minishell` et lancez le en tapant `./minishell`. Quand une commande est tapée, le programme affiche (pour le moment) la commande et ses arguments. Pour sortir, tapez `exit`.

Etape 2 (Lancement d'une commande) Modifiez le code de manière à exécuter la commande saisie lorsqu'elle est différente de `exit` (`strcmp(cmd[0], "exit") != 0`). Comme vu en cours, `https://moodle.inp-toulouse.fr/pluginfile.php/55411/mod_resource/content/1/API_UNIX.pdf`, cela consiste à :

1. Création du processus fils :

```c
pid_t pid_fils;
pid_fils= fork();
if (pid_fils == -1) {
    // erreur de création du processus fils
    perror("Erreur fork\n");
    exit(EXIT_FAILURE);
} else if (pid_fils == 0) {
    // code du processus fils
    ...
} else {
    // code du processus père
    ...
}
```

Remarque : Comme vu au BE1, le père et le fils exécutent ici le même programme, leur comportement est différencié par la valeur de `pid_fils`.

2. Le processus fils exécute la commande tapée au clavier en utilisant la primitive `exec`. La commande à exécuter est contenue dans le tableau `cmd`. Par exemple, si l'utilisateur entre la commande `ls -l`, le tableau `cmd` contient alors :
   - `cmd[0] == "ls"`
   - `cmd[1] == "-l"`

   Il existe différentes versions de la commande `exec` :

```
int execl(char *chemin, char *arg0, char *arg1, ..., char *argn,
    NULL);
int execlp(char *chemin, char *arg0, char *arg1, ..., char *argn,
    NULL);
int execle(char *chemin, char *arg0, char *arg1, ..., char *argn,
    NULL, char *env[]);
int execv(char *chemin, char *argv[]);
int execvp(char *chemin, char *argv[]);
int execve(char *chemin, char *argv[], char *env[]);
```

   Les lettres suivant `exec` signifient :
   - l/v : liste/tableau(vecteur)
   - p : utilisation de la variable `PATH` pour la recherche de la commande à exécuter
   - e : passage de l'environnement

   Dans notre cas, la recherche du chemin de la commande à exécuter se fera à l'aide de la variable environnement `PATH` et les différents arguments de la commande sont contenus dans le tableau `cmd`. Aussi vous pouvez choisir la bonne version de la primitive `exec` à utiliser.

A ce stade, lorsque la commande est lancée, `minishell` se met immédiatement en attente d'une nouvelle commande, sans attendre la terminaison. On dit que la commande est lancée en « arrière plan » (background).

Etape 3 (Enchaînement séquentiel des commandes) L'exécution d'une commande est maintenant faite dans un processus fils. Enchaîner les commandes consiste à attendre la terminaison de la commande en cours avant d'en lancer une deuxième. Pour ce faire, le processus père exécute la commande `wait`, lui permettant attendre la terminaison de la commande en cours. L'enchaînement des commandes suit donc les étapes suivantes :

1. Création d'un processus fils ;
2. Le processus fils lance la commande à l'aide de la primitive `exec` ;
3. Le processus père attend la terminaison de la commande.

La 2ème commande tapée au clavier peut se lancer à son tour en suivant ces différentes étapes. Modifiez votre code afin qu'il attende la fin de la dernière commande lancée avant de passer à la lecture de la ligne suivante.

Exemple d'appel de la primitive `wait()` :

```
int status;
pid_t pidFils;
if ( (pidFils= wait(&status)) != -1 ) {
    if (WIFEXITED(status)) {
        printf("Le processus fils %d s'est terminé avec le code %i\n",
        pidFils, WEXITSTATUS(status));
    } else if (WIFSIGNALED(status) {
```

```
        printf("Le processus fils %d s'est terminé par le signal %i\n",
        pidFils, WTERMSIG(status));
    }
}
```

Il est possible de tester le bon fonctionnement de cet enchaînement en exécutant la commande `sleep 10` dans le `minishell`. Cette commande effectue une attente de 10s.

1. Quel est l'affichage du programme lorsque le processus se termine normalement (exécution de `exit`) ?
2. Quel est l'affichage du programme si on exécute dans un autre terminal la commande :

   ```
   kill -INT num_pid_fils
   ```

   où `num_pid_fils` est le pid du fils obtenu grâce à la commande `ps`.

Etape 4 (Lancement de commandes en tâche de fond) Lorsque l'utilisateur ajoute le caractère & après la commande, celle-ci s'exécutera en tâche de fond, c'est-à-dire le processus père n'attend pas sa terminaison :

```
> sleep 10 &
```

Dans le programme, `isBackgrounded()` nous indique que & a été positionné. La commande en avant-plan sera donc celle pour laquelle `isBackgrounded` est faux. Dans ce cas-là, le processus père doit attendre sa terminaison.

Exécutez la suite de commandes :

```
> sleep 10 &
> sleep 50
```

Dans `minishell.c`, le processus père exécute :

```
if (!isBackgrounded()) {
    wait(NULL);   // on ne souhaite pas récupérer
                  // le compte-rendu de terminaison du fils
}
```

Complétez votre code pour offrir cette possibilité.

Question. *Pourquoi l'affichage du caractère > s'effectue-t-il après 10s ? Si vous avez ce genre de comportement, modifiez le code pour l'éviter.*

Etape 5 (Attendre la terminaison du dernier fils lancé) Nous allons utiliser la fonction `waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

La primitive `waitpid` a un comportement similaire à celui de `wait`. Le premier argument `pid_t pid` peut prendre l'une des valeurs suivantes :

- -1 : n'importe quel processus fils ;
- > 0 : le numéro du processus fils attendu.

Comme pour `wait`, le code de retour est -1 en cas d'erreur ou le pid du processus terminé. Le status est le même que celui obtenu avec la primitive `wait`. Les options permettent de gérer les changements d'état des processus. La valeur est un ou logique entre zéro ou plusieurs parmi les constantes :

- `WNOHANG` : retour immédiat même si aucun changement d'état (dans ce cas le retour vaut 0) ;
- `WUNTRACED` : retour si un fils a été suspendu ;
- `WCONTINUED` : retour si un fils suspendu a été relancé par le signal `SIGCONT`.

Remarque `waitpid(-1, &status, 0)` est équivalent à `wait(&status)`.

Question. *Remplacez l'attente de processus fils réalisée avec `wait` par une attente du processus en avant-plan avec la primitive `waitpid` (si vous ne l'avez pas déjà fait ou déjà utilisé dans ce cas). Que remarquez-vous sur l'état des processus exécutant les commandes en arrière-plan lorsqu'elles se terminent (utilisez `ps -fg`) ?*

Etape 6 (Traitement du signal `SIGCHLD`) Lorsqu'un processus fils change d'état, le processus père reçoit un signal `SIGCHLD` de la part du système. Ce signal a pour but de débloquer le processus père en attente de terminaison du fils via la commande PID. Ajoutez un traitement à la réception du signal `SIGCHLD` qui indique qu'un processus fils vient de terminer. Pour cela, utilisez la primitive :

```
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction)
;
```

Pour rappel, les différents champs de `struct sigaction` sont :

- `void (*sa_handler)(int)` est le traitement associé au signal. La procédure de traitement (le handler) est donc de la forme : `void traitement(int sig)` où `sig` est le signal qui a provoqué le lancement du traitement.
- `sigset_t sa_mask` est le masque des signaux lorsque le signal est reçu. Initialisation de cet ensemble via la primitive `void sigemptyset(sigset_t *set)`.
- `int sa_flags` est un ou logique de 0 ou plusieurs options. Ici, nous utiliserons l'option `SA_RESTART` (au lieu de 0).

Dans le fichier `Makefile`, enlevez l'option de compilation `-std=c11` de la variable `CFLAGS`. Testez le programme en utilisant des processus en avant-plan et en arrière-plan (par exemple `sleep 10 &`). A ce stade, le `minishell` affiche un message lorsqu'un processus termine qu'il soit en avant-plan ou en arrière plan. Il est alors possible d'attendre la terminaison des fils à la fois pour les processus en avant-plan et en arrière-plan.

Etape 7 (Utilisation de `SIGCHLD` pour traiter la terminaison des processus fils) Le traitement du signal `SIGCHLD` doit être capable de récupérer le `pid` ainsi que le `status` du processus qui vient de terminer. Pour cela, il est possible d'utiliser `waitpid` avec les options `WNOHANG|WUNTRACED|WCONTINUED` qui rendent la commande non bloquante. Modifiez le programme pour traiter la terminaison de **tous** les processus par l'utilisation de la primitive `waitpid` et affichez son code de retour, qui correspond au `pid` du processus qui vient de terminer. Pour rappel, `waitpid` retourne -1 si aucun processus n'existe.

Etape 8 (Attendre un signal : `pause`) Puisque la terminaison des processus se fait dans le traitement du signal `SIGCHLD`, attendre la terminaison du fils en avant-plan revient à attendre le signal `SIGCHLD`. Modifiez le programme pour utiliser `void pause()` lorsque le processus est en avant-plan. Testez avec :

```
> sleep 10 &
> sleep 50
```

Que constatez-vous ?

Etape 9 (Suspension et reprise d'un processus en arrière-plan) Testez l'envoi des signaux `SIGSTOP` et `SIGCONT` vers un processus en arrière-plan. Dans quel état se trouve ce processus après lancement de chaque signal ?

Etape 10 (Affichage d'un message indiquant le signal reçu) Lorsqu'un processus fils change d'état, le processus père reçoit un signal `SIGCHLD`. Modifiez le code pour afficher un message lorsque le processus est terminé, suspendu ou repris. Pour identifier les différents cas à étudier, utilisez les macros fournies par l'API Unix : `WIFEXITED(status)`, `WIFSIGNALED(status` (que vous connaissez déjà), `WIFSTOPPED(status)` qui vaut vrai si le processus a été suspendu par le signal `SIGSTOP` et `SIGCONTINUED(status)` qui vaut vrai si le processus a été repris en utilisant le signal `SIGCONT`. Bien entendu, testez ces différents cas.

Etape 11 (Test de la frappe au clavier de `ctrl-C` et `ctrl-Z`) Lorsqu'on appuie sur `ctrl-C` (respectivement `ctrl-Z`), le signal `SIGINT` (respectivement `SIGTSTP`) est envoyé au processus en cours. La terminaison (respectivement la suspension) du processus `minishell` est alors demandée. Testez ce comportement.
Que se passe-t-il lorsque le `minishell` a lancé :

- une commande en avant plan, par exemple `sleep 50` ?
- une commande en arrière plan, par exemple `sleep 50 &` ?

Etape 12 (Gestion de la frappe au clavier de `ctrl-C` et `ctrl-Z`) Comme nous avons pu le remarquer, lorsqu'on frappe `ctrl-C` (ou `ctrl-Z`), les signaux sont transmis à la fois à `minishell` et à l'ensemble de ses fils. Nous allons alors traiter la non-terminaison ou la non-suspension du processus `minishell`.

Ignorer un signal revient à associer le traitement `SIG_IGN` à ce signal. Pour reprendre le traitement par défaut, il suffit d'associer le traitement `SIG_DFL`. Modifiez le code de manière à ignorer la réception des signaux `ctrl-C` et `ctrl-Z` dans le processus père. Testez ensuite la frappe de `ctrl-C` puis de `ctrl-Z`, avec un processus en arrière plan et un processus en avant plan.

Etape 13 (Détacher les processus fils en arrière plan) A ce stade, le processus `minishell` n'est plus sensible à la frappe sur les touches `ctrl-C` et `ctrl-Z` mais tous les processus fils reçoivent les signaux `SIGINT` et `SIGDFL`. Or, lors de la frappe, seul le processus en avant plan doit recevoir le signal. En réalité, les signaux sont transmis à tous les processus du même groupe que le processus `minishell`. La solution que nous proposons est de mettre les processus en arrière plan dans un autre groupe de processus. Nous utiliserons ici la primitive `int setpgrp()` qui associe une nouveau groupe au processus appelant. Testez une dernière fois, la frappe de `ctrl-C` puis de `ctrl-Z`, avec un processus en arrière plan et un processus en avant plan.

Etape 14 (Rendu) Archivez votre travail via la commande `make archive`. Le résultat est un fichier nommé `minishell-votreidentifiant.tar`. Chargez ce fichier dans la section rendu sous Moodle, dans la zone qui correspond à votre groupe de TD.