Signaux

Thèmes

- Définition d'un traitant de signal, attente et envoi d'un signal

Ressources : pour ce TP, comme pour les suivants, vous pourrez vous appuyer sur

- Le polycopié intitulé « Systèmes d'exploitation : Unix », qui fournit une référence généralement suffisante sur la sémantique et la syntaxe d'appel des différentes primitives de l'API Unix. Chaque section du sujet de TP indique la (ou les) section(s) du polycopié correspondant au contenu présenté.
- Les pages du manuel en ligne (commande `man`), et plus particulièrement les sections 2 et 3.

1 Minishell : retour sur les 1ères étapes du projet

Etape 3. L'exécution des commandes étant faites dans un processus fils, enchaîner des commandes consistent tout simplement à créer les fils dans la boucle. Pour chaque fils créé, le père exécute la commande `wait` pour 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 commandes tapées au clavier peut se lancer à son tour en suivant ces différentes étapes.

Etape 4. 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, `commande->backgrounded != NULL` nous indique que & a été positionné. La commande en avant-plan sera donc celle pour laquelle `commande->backgrounded == NULL`. 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 (commande->backgrounded == NULL) {
    wait(NULL);  // on ne souhaite pas récupérer
                 // le compte-rendu de terminaison du fils
}
```

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.*

2 Minishell : contrôle de la terminaison des fils

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

Remarque 2. L'argument `pid_t pid` peut être < -1 ou $= 0$. Leur utilisation est hors du scope du TP. Nous ne les utiliserons pas.

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 ?*

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 ?

3 Minishell : envoi de signaux aux processus en arrière-plan

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 (Rendu) Comme au TP1, archivez votre travail via la commande `make archive`. Le résultat est un fichier nommé `minishell-votreidentifiant.tar`. Chargez ce fichier dans la section rendu, dans la zone qui correspond à votre groupe de TD.