Processus

Thèmes abordés

- Primitives de gestion des processus : création, terminaison, recouvrement.
- Boucle de base d'un interpréteur de commandes.

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.
- Les pages du manuel en ligne (commande `man`), et plus particulièrement les sections 2 et 3.

1 Gestion des processus

On considère le programme suivant :

```
int main(int argc, char *argv[]) {
    int tempsPere, tempsFils;
    pid_t pid_fork;

    tempsPere= 120;
    tempsFils= 60;

    pid_fork= fork();
    // bonne pratique : tester systématiquement le retour des
   primitives
    if (pid_fork == -1) {
        printf("Erreur fork\n");
        exit(1);
        /* par convention, renvoyer une valeur > 0 en cas d'erreur,
         * différente pour chaque cause d'erreur, ici 1 = erreur fork
         */
    }
    if (pid_fork == 0) {    /* fils */
        printf("fils: processus %d, de père %d et code du fork %d\n",
                    getpid(), getppid(), pid_fork);
        sleep(tempsFils);
        printf("fin du fils\n");
        exit(EXIT_SUCCESS);
        /*  bonne pratique :
          terminer les processus par un exit explicite */
    }
    else {    /* père */
        printf("père: processus %d, de père %d et code du fork %d\n",
                    getpid(), getppid(), pid_fork);
        sleep(tempsPere);
        printf("fin du père\n");
    }
    return EXIT_SUCCESS; /* -> exit(EXIT_SUCCESS); pour le père */
}
```

1.1 Exécution et état des processus

1. Téléchargez-le depuis Moodle : `https://moodle-n7.inp-toulouse.fr/pluginfile.php/149314/mod_resource/content/1/exo1.c`.
2. Compilez le programme en utilisant `gcc` et exécutez le.
3. Utilisez la commande `ps -fjn` dans un autre terminal et indiquez l'état des processus (`S` : `Sleeping`, en attente ; `R` : `Running`, actif ; `T` : `sTopped`, terminé ; `Z` : `Zombie`, ...) :
   - après le lancement du programme ;
   - après la fin du processus fils.
4. Modifiez le code en échangeant les valeurs de `tempsFils` et `tempsPere` et exécutez le programme. En utilisant la commande `ps -fj`, que constatez-vous lorsque le processus père a terminé ?

1.2 Héritage des données

Modifiez le programme précédent avec :

- `variable= 10;` au début du code du fils ;
- `variable= 100;` au début du code du père.

Affichez `variable` à la fin du code du fils et à la fin du code du père. Que constatez-vous ?

1.3 Attente de la terminaison du fils

Remplacez la ligne `sleep(tempsPere)` pour que le processus attende la terminaison du fils.

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));
    }
}
```

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`.

2 Projet Minishell

Cette partie met en place le début du projet `minishell`. Le sujet du projet et les fichiers nécessaires sont disponibles sous Moodle `https://moodle-n7.inp-toulouse.fr/course/view.php?id=606` dans la section Projet.

Rendu à la fin de la séance

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.

Etape 1 (Testez le programme) Compilez le programme en utilisant la commande `make` 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 dans un processus fils en utilisant la primitive `exec` vue en TD.

A ce stade, lorsque la commande est lancée, `minishell` se met immédiatement en attente d'une nouvelle commande, sans attendre la terminaison.

Etape 3 (Enchaînement séquentiel des commandes) L'exécution des commandes étant faite dans un processus fils, enchaîner des commandes consiste 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 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.

Etape 4 (Lancement de commandes en tâche de fond) Le comportement du code initial (celui écrit en réponse à l'étape 2) correspond cependant à une possibilité utile offerte par les shells, à savoir le lancement de commandes en tâche de fond, spécifié par un & en fin de ligne :

```
> sleep 10 &
```

La commande `sleep` s'exécutera en tâche de fond.

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`. Modifiez le comportement du père pour réaliser cette fonctionnalité.

Exécutez la suite de commandes :

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

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.*