

1 Objectifs:

Le **shell** est chargé de faire l'intermédiaire entre le système d'exploitation et l'utilisateur grâce aux lignes de commandes saisies par ce dernier. Son rôle consiste ainsi à lire la ligne de commande, interpréter sa signification, exécuter la commande, puis retourner le résultat sur les sorties.

Le shell est ainsi un fichier exécutable chargé d'interpréter les commandes, de les transmettre au système et de retourner le résultat.

Une ligne de commande est une chaîne de caractère constituée d'une commande, (correspondant soit à un fichier exécutable du système, soit à une fonction interne du shell) et d'arguments optionnels :

```
ls -al /home/pi/
```

ls est la commande **-al** et **/home/if/** représentent les arguments.

Le but de ce TP est de programmer un **shell minimal en C**.

2 Exécution de programmes au premier plan :

Notre shell sera une simple boucle d'interaction qui affiche une invite de commande **SNIR shell >**, lit une ligne entrée au clavier (**fgets**), l'exécute, et recommence. Le shell quitte quand l'utilisateur tape **Contrôle+D** à la place d'une commande.

Travail demandé :

- 1) Dans un premier temps, on suppose que la commande est simplement un nom de programme à exécuter dans un processus séparé (**fork()** et **execpl()**). Le *shell* doit attendre que le programme se termine (**waitpid()**) avant de rendre la main à l'utilisateur. Le minishell précise comment le programme se termine : soit
Terminaison normale écrit en vert si le status est 0
Terminaison normale écrit en jaune si le status est différent de 0
Terminaison par signal écrit en rouge avec son identifiant



```
pi@raspberrypi3: ~/IPC/8_TP_MINI_SHELL
pi@raspberrypi3:~/IPC/8_TP_MINI_SHELL $ ./minishell1
SNIR shell1 > ls
a.out      minishell1.c  minishell13  minishell4.c
division.c minishell12   minishell3.c TP sujet mini shell_v1.odt
minishell1 minishell12.c minishell4   TP sujet mini shell_v1.pdf
Terminaison normale, status 0
SNIR shell1 > ./a.out
test division par zéro

Terminaison par signal 8
SNIR shell1 > █
```

Attention: `waitpid()` peut échouer avec l'erreur **EINTR**, auquel cas il faut renouveler l'appel à `waitpid()`.

Attention: le **nom du programme apparaît deux fois** dans `execlp()` (une fois en tant que programme à exécuter et une fois en tant qu'argument 0 du programme à exécuter). De plus la liste d'arguments doit obligatoirement se terminer par **NULL**.

Vous effectuerez les tests unitaires suivants :

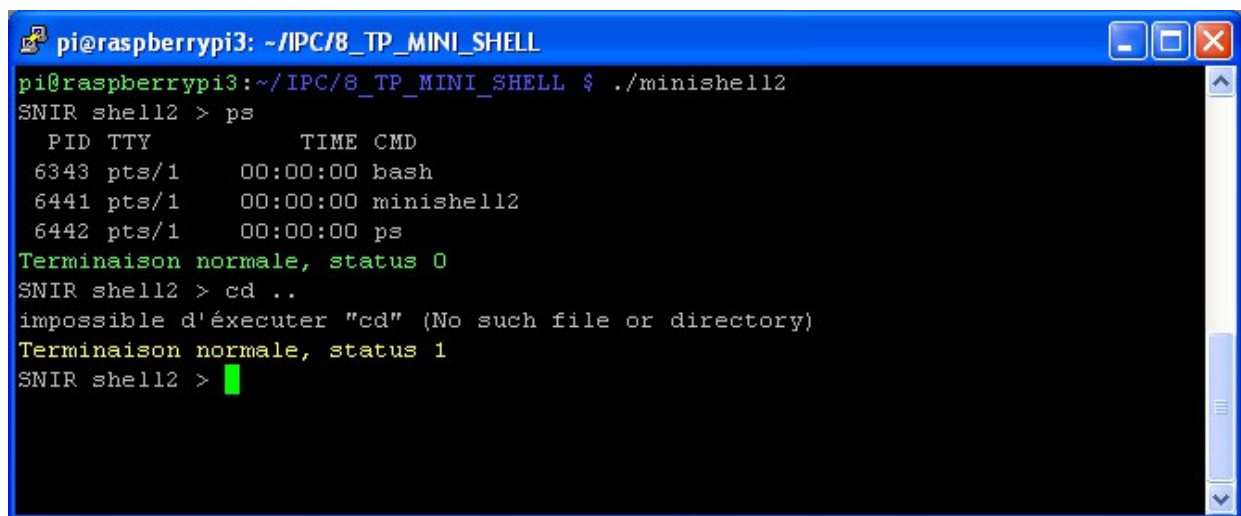
- en exécutant la commande `ls` ;
- en exécutant un programme qui effectue une division par zéro ;
- en exécutant un programme qui effectue une erreur de segmentation.

2) On suppose maintenant que la ligne de commande peut contenir un nombre arbitraire d'arguments, séparés par un ou plusieurs espaces.

Afin de ne pas passer trop de temps sur le problème du découpage d'une ligne en mots, on vous propose une solution toute faite (voir en annexe à la fin du TP).

Modifier votre programme pour découper la ligne de commande et passer le tableau des arguments à `execlp()`.

3) Vous allez à partir de maintenant effectuer à partir de votre shell2 les tests suivants :



```
pi@raspberrypi3: ~/IPC/8_TP_MINI_SHELL
pi@raspberrypi3:~/IPC/8_TP_MINI_SHELL $ ./minishell12
SNIR shell12 > ps
  PID TTY          TIME CMD
 6343 pts/1        00:00:00 bash
 6441 pts/1        00:00:00 minishell12
 6442 pts/1        00:00:00 ps
Terminaison normale, status 0
SNIR shell12 > cd ..
impossible d'exécuter "cd" (No such file or directory)
Terminaison normale, status 1
SNIR shell12 > █
```

Que se passe t'il si on exécute la commande `cd` . Pour vous aider à comprendre cette situation vous pouvez à partir du shell bash exécuter les commandes **type ps** puis **type cd**.

Modifier votre programme pour que la commande **cd** soit exécutée. Vous ajouterez une commande **exit** pour quitter proprement votre mini shell.

4) Que se passe-t'il si on interrompt la commande **cat** avec **contrôle C** ?



```
pi@raspberrypi3: ~/IPC/8_TP_MINI_SHELL
pi@raspberrypi3:~/IPC/8_TP_MINI_SHELL $ ./minishell3
SNIR shell3 > cat
toto^C
pi@raspberrypi3:~/IPC/8_TP_MINI_SHELL $
```

Modifiez votre *shell* pour que, si une commande est en cours d'exécution, un appui sur Contrôle+C interrompe la commande et redonne la main à l'utilisateur sans toutefois quitter le mini *shell*. On utilisera pour cela **sigaction()**.



```
pi@raspberrypi3: ~/IPC/8_TP_MINI_SHELL
pi@raspberrypi3:~/IPC/8_TP_MINI_SHELL $ ./minishell4
SNIR shell4 > cat
toto fait des bêtises^C
Terminaison par signal 2
SNIR shell4 >
```

5) Ajoutez à votre *shell* la redirection de la sortie standard. Si le dernier argument commence par un caractère **>**, on suppose que le reste du mot est le nom du fichier où rediriger la sortie de la commande à exécuter (voir **dup2()**).

Que se passe-t-il si la commande écrit sur la sortie d'erreur?

3 Exécution de programmes en tâche de fond

On ajoute maintenant l'exécution de programmes en tâche de fond. Quand un tel programme est lancé, le *shell* n'attend pas qu'il se termine mais redonne tout de suite la main à l'utilisateur. Attention toutefois, le *shell* devra tout de même détecter la terminaison (asynchrone) des programmes en tâche de fond afin de

- éviter les processus *zombies*,
- prévenir l'utilisateur de la bonne ou mauvaise terminaison de la commande.

Il faudra pour cela gérer le signal **SIGCHLD** grâce à un *handler* (voir **sigaction(2)**). Notez vous pouvez ne recevoir qu'un seul signal **SIGCHLD** pour indiquer la

terminaison de plusieurs fils. Attention également aux fonctions (e.g., **fgets**) qui peuvent être interrompues par l'exécution asynchrone du *handler*.

- 6) Modifiez votre *shell* pour que toutes les commandes soient lancées en tâche de fond. Que se passe-t-il si une commande en tâche de fond essaye de lire l'entrée standard (e.g., **cat**)?
- 7) Modifiez votre *shell* pour qu'il gère deux types de commandes: une commande terminée par **&** sera lancée en tâche de fond tandis qu'une commande non terminée par **&** sera lancée au premier plan. À un moment donné, il y a au plus une commande au premier plan et un nombre arbitraire de commandes en tâche de fond. (Attention aux interactions entre le *handler* du signal **SIGCHLD** et la commande **wait(2)**...)

4 Options complémentaires

On propose maintenant plusieurs extensions du mini shell. Celles-ci sont assez indépendantes. Vous pouvez en faire quelques unes (en TP si vous avez le temps, ou à la maison pour vous entraîner un peu) ou proposer les vôtres.

- 8) **Édition conviviale** : On souhaite rendre l'édition de la ligne de commande un peu plus conviviale: support des flèches pour la navigation et la correction, accès à l'historique des commandes.

C'est difficile à faire à la main! Heureusement, il existe la bibliothèque **readline** de GNU pour faire cela. Vous devez tout d'abord installer la bibliothèque.

sudo apt-get install libreadline-dev

Pour compiler ne pas oublier de linker avec la biblio readline.

gcc minishell8.c -o minishell8 -lreadline

Voir en annexe un exemple d'utilisation de readline

9) Variables d'environnement

Annexe découpe ligne:

On propose une fonction **decoupe** permettant de découper une ligne en mots:

L'entrée de **decoupe** est la chaîne à découper, contenue dans la variable globale **ligne**.

La sortie de **decoupe** sera une liste de pointeurs vers le début de chaque mot. La liste est stockée dans le tableau global **elems** de **MAXELEMS** pointeurs et est terminée par le pointeur **NULL**.

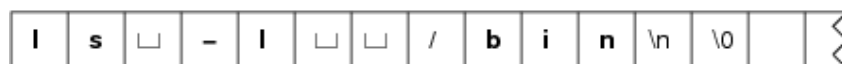
```
char ligne[4096];
#include <ctype.h>
#define MAXELEMS 32
char* elems[MAXELEMS];

void decoupe()
{
    char* debut = ligne;
    int i;
    for (i=0; i<MAXELEMS-1; i++) {
        /* saute les espaces */
        while (*debut && isspace(*debut)) debut++;
        /* fin de ligne ? */
        if (!*debut) break;
        /* on se souvient du début de ce mot */
        elems[i] = debut;
        /* cherche la fin du mot */
        while (*debut && !isspace(*debut)) debut++; /* saute le mot */
        /* termine le mot par un \0 et passe au suivant */
        if (*debut) { *debut = 0; debut++; }
    }
    elems[i] = NULL;
}
```

Lors de son parcours de **ligne**, la fonction **decoupe** transforme les espaces et retours à la ligne (voir **isspace()**) en fin de mots en caractère **\0** et fait pointer **elems** à l'intérieur de **ligne**. L'avantage de cette méthode est qu'il n'est pas nécessaire d'allouer de la mémoire supplémentaire pour stocker les mots isolés de **ligne**.

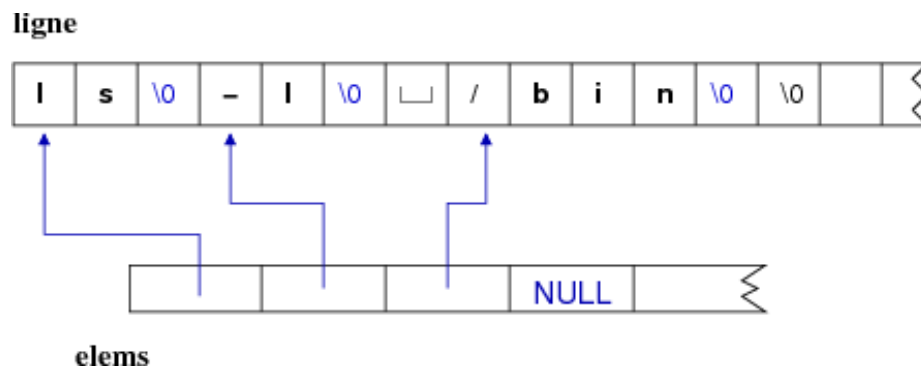
Supposons que, avant l'appel à **decoupe**, **ligne** contienne la ligne **ls -l /bin**: (avec deux espaces en -l et /bin)

ligne



elems

Au retour de **decoupe**, on aura alors:



Ainsi,

elems[0] pointe sur la chaîne **ls**,

elems[1] pointe sur la chaîne **-l** et

elems[2] pointe sur la chaîne **/bin**.

Toutes ces chaînes sont bien terminées par le caractère **\0**.