Introduction

Le but de ce TP est d'utiliser les facilités offertes par l'interpréteur Shell et les commandes accessibles à partir du Shell à travers des programmes écrits en langage C.

Un programme C interagit avec le Shell par 2 moyens, interaction avec la ligne de commande et interaction pendant le traitement propre.

1. Un programme C récupère les informations tapées sur la ligne commande du Shell par l'intermédiaire des variables prédéfinies `argc` et `argv`. Ces variables doivent être déclarées en argument de la fonction `main()` qui sera le point d'entrée initial du programme C. L'analyse syntaxique de ligne de commande étant alors à la charge du programme C.
2. Au niveau de l'intéraction des données entre le programme C et le Shell il existe 2 fonctions de la bibliothèque C standard qui permettent d'interagir avec les commandes du Shell, à savoir `system()` et `popen()`.

---

argc, argv

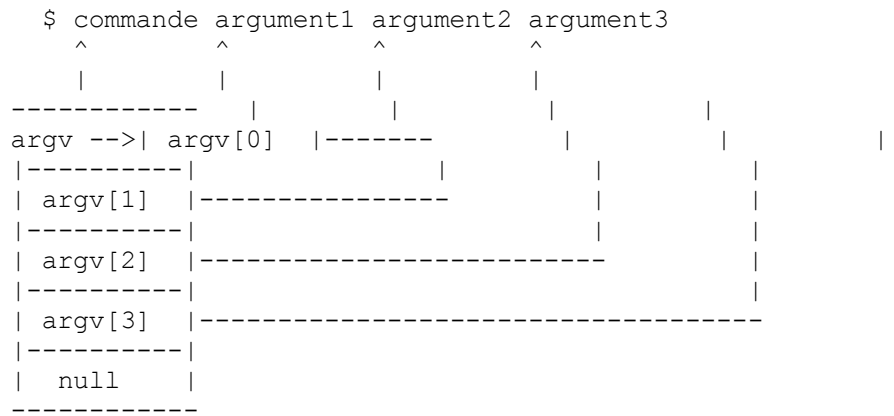
Les 2 arguments `argc` et `argv` sont transmis du Shell à la fonction `main()` du programme utilisateur. La syntaxe de la fonction `main()` est la suivante:

```
main(int argc, char ** argv)
```

où

- `argc` est un entier qui est égal au nombre de chaînes de caractères, séparées par un ou plusieurs blanc (espace ou tabulation), dans la ligne de commande. Le nom de la commande est donc inclu dans le compte de la variable `argc`. Une commande shell sans argument aura donc un `argc = 1`.
- `argv` est un pointeur sur un tableau de pointeurs sur des chaînes de caractères, celles-ci correspondant à celles tapées sur le clavier. Chaque pointeur du tableau pointe sur le début des chaînes respectives. Le tableau est terminé par un pointeur nul indiquant qu'il n'y a plus d'arguments.

Le schéma ci-dessous illustre le tableau `argv`:



---

fork

Nom

fork - Créer un processus fils (child).

Synopsis

#include <unistd.h>

pid_t fork(void);

Description

fork crée un processus fils qui diffère du processus parent uniquement par ses valeurs PID et PPID et par le fait que toutes les statistiques d'utilisation des ressources sont remises à zéro. Les verrouillages de fichiers, et les signaux en attente ne sont pas hérités.

Sous Linux, **fork** est implementé en utilisant une méthode de copie à l'écriture. Ceci consiste à ne faire la véritable duplication d'une page mémoire que lorsqu'un processus en modifie une instance. Tant qu'aucun des deux processus n'écrit dans une page donnée, celle-ci n'est pas vraiment dupliquée. Ainsi les seules pénalisations induites par fork sont le temps et la mémoire nécessaires à la copie de la table des pages du parent ainsi que la création d'une structure de tâche pour le fils.

Valeur Renvoyée

En cas de succès, le PID du fils est renvoyé au processus parent, et 0 est renvoyé au processus fils. En cas d'échec -1 est renvoyé dans le contexte du parent, aucun processus fils n'est créé, et *errno* contient le code d'erreur.

Erreurs

ENOMEM
: Impossible d'allouer assez de mémoire pour copier la table des pages du père et d'allouer une structure de tâche pour le fils.

EAGAIN
: Impossible de trouver un emplacement vide dans la table des processus.

Exemple

```
pid_t   pid;
pid = fork ();
if (pid > 0) {
     /* Processus père      */
} else if (pid == 0) {
     /* Processus fils      */
} else {
     /* Traitement d'erreur */
}
```

FILE * fopen(const char *nomFichier, char *mode);

Cette fonction permet d'ouvrir un fichier dans le but d'une manipulation de son contenu. Cette manipulation pourra soit être une consultation, une construction, ou bien une mise à jour. Le dernier paramètre permet justement de choisir le type de la manipulation.

Paramètres

- **nomFichier** : ce paramètre permet de spécifier le nom du fichier à ouvrir. Attention, certain système de fichiers font la différence au niveau des minuscules/majuscules : regardez bien à deux fois le nom du fichier renseigné.
- **mode** : cette chaîne de caractères permet de préciser dans quel but ce fichier est ouvert. En effet, on peut ouvrir un fichier en lecture "r", en écriture "w", pour écrire, mais à la fin du fichier "a", ...

Valeur de retour

Cette fonction vous renvoie, normalement, un pointeur sur une structure de type FILE. Cependant, si la fonction échoue (ouverture en lecture sur un fichier inexistant, par exemple) le pointeur est alors nul. Notez une chose importante : la fonction se charge d'allouer l'espace nécessaire pour contenir la structure. La fonction fclose qui permet de fermer le fichier, se charge bien entendu de désallouer la mémoire.

Fonctions connexes

fclose, fread, fwrite, feof, fprintf, fscanf

Exemple

```
#include <stdio.h>
#include <stdlib.h>

void main(void) {
  FILE *file = fopen("destination.txt","w");

  if (file == NULL) {
    fprintf(stderr,"Erreur dans l'ouverture du
fichier");
    exit(-1);
  }

  fprintf(file,"Mon texte");

  fclose(file);
```

wait, waitpid - Attendre la fin d'un processus.

SYNOPSIS

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

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

DESCRIPTION

La fonction **wait** suspend l'exécution du processus courant jusqu'à ce qu'un enfant se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si un processus fils s'est déjà terminé au moment de l'appel (il est devenu "zom bie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.

La fonction **waitpid** suspend l'exécution du processus courant jusqu'à ce que le processus fils numéro *pid* se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si le fils mentionné par *pid* s'est déjà terminé au moment de l'appel (il est devenu "zombie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.

La valeur de *pid* peut également être l'une des suivantes :

< -1 attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID *pid*.

-1 attendre la fin de n'importe quel fils. C'est le même comportement que **wait**.

0 attendre la fin de n'importe quel processus fils du même groupe que l'appelant.

> 0 attendre la fin du processus numéro *pid*.

La valeur de l'argument option *options* est un *OU* binaire entre les constantes suivantes :

WNOHANG ne pas bloquer si aucun fils ne s'est terminé.

WUNTRACED
recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Si *status* est non **NULL**, **wait** et **waitpid** y stockent l'information sur la terminaison du fils.

Cette information peut être analysée avec les macros suiv antes, qui réclament en argument le buffer *status* (un **int**,
non nul si le fils s'est terminé normalement

WEXITSTATUS(*status***)**
donne le code de retour tel qu'il a été mentionné dans l'appel **exit()** ou dans le **return** de la rou

tine **main**. Cette macro ne peut être évaluée que si **WIFEXITED** est non nul.

WIFSIGNALED(*status***)**
indique que le fils s'est terminé à cause d'un signal non intercepté.

WTERMSIG(*status***)**
donne le nombre de signaux qui ont causé la fin du fils. Cette macro ne peut être évaluée que si **WIFSIGNALED** est non nul.

WIFSTOPPED(*status***)**
indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option **WUNTRACED**.

WSTOPSIG(*status***)**
donne le nombre de signaux qui ont causé l'arrêt du fils. Cette macro ne peut être évaluée que si **WIFSTOPPED** est non nul.

VALEUR RENVOYÉE

En cas de réussite, le PID du fils qui s'est terminé est renvoyé, en cas d'echec -1 est renvoyé et *errno* contient le code d'erreur.

ERREURS

ECHILD Le processus indiqué par *pid* n'existe pas, ou n'est pas un fils du processus appelant. (Ceci peut arriver pour son propre fils si l'action de SIGCHLD est placé sur SIG_IGN, voir également le passage de la section NOTES concernant les threads).

EINVAL L'argument *options* est invalide.

ERESTARTSYS
WNOHANG n'est pas indiqué, et un signal à inter cepter ou **SIGCHLD** a été reçu. Cette erreur est renvoyée par l'appel système. La routine de bib liothèque d'interface n'est pas autorisée à ren voyer **ERESTARTSYS**, mais renverra **EINTR**.

NOTES

Les spécifications Single Unix décrivent un attribut SA_NOCLDWAIT (absent sous Linux) permettant (lorsqu'il est fixée à SIG_IGN (ce qui toutefois n'est pas autorisé par POSIX). Un appel à *wait()* ou *waitpid()* bloquera jusqu'à ce qu'un fils se termine, puis échouera avec *errno* contenant ECHILD.

Dans le noyau Linux, un thread ordonnancé par le noyau n'est pas différent d'un simple processus. En fait, un wait,

waitpid - Attendre la fin d'un processus.

SYNOPSIS

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

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

DESCRIPTION

La fonction **wait** suspend l'exécution du processus courant jusqu'à ce qu'un enfant se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si un processus fils s'est déjà terminé au moment de l'appel (il est devenu "zom bie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.

La fonction **waitpid** suspend l'exécution du processus courant jusqu'à ce que le processus fils numéro *pid* se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si le fils mentionné par *pid* s'est déjà terminé au moment de l'appel (il est devenu "zombie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.

La valeur de *pid* peut également être l'une des suivantes :

< -1 attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID *pid*.

-1 attendre la fin de n'importe quel fils. C'est le même comportement que **wait**.

0 attendre la fin de n'importe quel processus fils du même groupe que l'appelant.

> 0 attendre la fin du processus numéro *pid*.

La valeur de l'argument option *options* est un *OU* binaire entre les constantes suivantes :

WNOHANG ne pas bloquer si aucun fils ne s'est terminé.

WUNTRACED
recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Si *status* est non **NULL**, **wait** et **waitpid** y stockent l'information sur la terminaison du fils.

Cette information peut être analysée avec les macros suiv antes, qui réclament en argument le buffer *status* (un **int**,
non nul si le fils s'est terminé normalement

WEXITSTATUS(*status***)**
donne le code de retour tel qu'il a été mentionné dans l'appel **exit()** ou dans le **return** de la rou

tine **main.** Cette macro ne peut être évaluée que si **WIFEXITED** est non nul.

WIFSIGNALED(*status***)**
indique que le fils s'est terminé à cause d'un signal non intercepté.

WTERMSIG(*status***)**
donne le nombre de signaux qui ont causé la fin du fils. Cette macro ne peut être évaluée que si **WIFSIGNALED** est non nul.

WIFSTOPPED(*status***)**
indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option **WUNTRACED.**

WSTOPSIG(*status***)**
donne le nombre de signaux qui ont causé l'arrêt du fils. Cette macro ne peut être évaluée que si **WIFSTOPPED** est non nul.

VALEUR RENVOYÉE

En cas de réussite, le PID du fils qui s'est terminé est renvoyé, en cas d'echec -1 est renvoyé et *errno* contient le code d'erreur.

ERREURS

ECHILD Le processus indiqué par *pid* n'existe pas, ou
n'est pas un fils du processus appe wait, waitpid - Attendre la fin d'un processus.

SYNOPSIS

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

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

DESCRIPTION

La fonction **wait** suspend l'exécution du processus courant jusqu'à ce qu'un enfant se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si un processus fils s'est déjà terminé au moment de l'appel (il est devenu "zom bie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.

La fonction **waitpid** suspend l'exécution du processus courant jusqu'à ce que le processus fils numéro *pid* se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si le fils mentionné par *pid* s'est déjà terminé au moment

de l'appel (il est devenu "zombie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.

La valeur de *pid* peut également être l'une des suivantes :

< -1 attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID *pid*.

-1 attendre la fin de n'importe quel fils. C'est le même comportement que **wait**.

0 attendre la fin de n'importe quel processus fils du même groupe que l'appelant.

> 0 attendre la fin du processus numéro *pid*.

La valeur de l'argument option *options* est un *OU* binaire entre les constantes suivantes :

WNOHANG ne pas bloquer si aucun fils ne s'est terminé.

WUNTRACED
recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Si *status* est non **NULL**, **wait** et **waitpid** y stockent l'information sur la terminaison du fils.

Cette information peut être analysée avec les macros suiv antes, qui réclament en argument le buffer *status* (un **int**,
non nul si le fils s'est terminé normalement

WEXITSTATUS(*status***)**
donne le code de retour tel qu'il a été mentionné dans l'appel **exit()** ou dans le **return** de la rou tine **main**. Cette macro ne peut être évaluée que si **WIFEXITED** est non nul.

WIFSIGNALED(*status***)**
indique que le fils s'est terminé à cause d'un signal non intercepté.

WTERMSIG(*status***)**
donne le nombre de signaux qui ont causé la fin du fils. Cette macro ne peut être évaluée que si **WIFSIGNALED** est non nul.

WIFSTOPPED(*status***)**
indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option **WUNTRACED**.

WSTOPSIG(*status***)**
donne le nombre de signaux qui ont causé l'arrêt du fils. Cette macro ne peut être évaluée que si **WIFSTOPPED** est non nul.

VALEUR RENVOYÉE

En cas de réussite, le PID du fils qui s'est terminé est renvoyé, en cas d'echec -1 est renvoyé et *errno* contient

le code d'erreur.

ERREURS

ECHILD Le processus indiqué par *pid* n'existe pas, ou n'est pas un fils du processus appelant. (Ceci peut arriver pour son propre fils si l'action de SIGCHLD est placé sur SIG_IGN, voir également le passage de la section NOTES concernant les threads).

EINVAL L'argument *options* est invalide.

ERESTARTSYS
WNOHANG n'est pas indiqué, et un signal à inter cepter ou **SIGCHLD** a été reçu. Cette erreur est renvoyée par l'appel système. La routine de bib liothèque d'interface n'est pas autorisée à ren voyer **ERESTARTSYS**, mais renverra **EINTR**.

NOTES

Les spécifications Single Unix décrivent un attribut SA_NOCLDWAIT (absent sous Linux) permettant (lorsqu'il est fixée à SIG_IGN (ce qui toutefois n'est pas autorisé par POSIX). Un appel à *wait()* ou *waitpid()* bloquera jusqu'à ce qu'un fils se termine, puis échouera avec *errno* contenant ECHILD.

Dans le noyau Linux, un thread ordonnancé par le noyau n'est pas différent d'un simple processus. En fait, un thread est juste un processus qui est créé à l'aide de la routine - spécifique Linux - **clone(2)**. Les routines porta bles, comme **pthread_create(3)** sont implémentées en appelant **clone(2)**. Ainsi, si deux threads A et B sont frères, alors le thread A ne peut pas se mettre en attente sur un processus créé par le thread B ou ses descendants, car un oncle ne peut pas attendre un neveu. Sur d'autre systèmes Unix, où de multiples threads sont implémentés au sein d'un unique processus, le thread A peut naturellement attendre la fin de n'importe quel processus lancé par un thread frère B. Pour qu'un programme qui se base sur ce comportement fonctionne sous Linux, il faudra en modifier le code.

lant. (Ceci
peut arriver pour son propre fils si l'action de SIGCHLD est placé sur SIG_IGN, voir également le passage de la section NOTES concernant les threads).

EINVAL L'argument *options* est invalide.

ERESTARTSYS
WNOHANG n'est pas indiqué, et un signal à inter cepter ou **SIGCHLD** a été reçu. Cette erreur est

renvoyée par l'appel système. La routine de bib liothèque d'interface n'est pas autorisée à ren voyer **ERESTARTSYS**, mais renverra **EINTR**.

NOTES

Les spécifications Single Unix décrivent un attribut SA_NOCLDWAIT (absent sous Linux) permettant (lorsqu'il est fixée à SIG_IGN (ce qui toutefois n'est pas autorisé par POSIX). Un appel à *wait()* ou *waitpid()* bloquera jusqu'à ce qu'un fils se termine, puis échouera avec *errno* contenant ECHILD.

Dans le noyau Linux, un thread ordonnancé par le noyau n'est pas différent d'un simple processus. En fait, un thread est juste un processus qui est créé à l'aide de la routine - spécifique Linux - **clone(2)**. Les routines porta bles, comme **pthread_create(3)** sont implémentées en appelant **clone(2)**. Ainsi, si deux threads A et B sont frères, alors le thread A ne peut pas se mettre en attente sur un processus créé par le thread B ou ses descendants, car un oncle ne peut pas attendre un neveu. Sur d'autre systèmes Unix, où de multiples threads sont implémentés au sein d'un unique processus, le thread A peut naturellement attendre la fin de n'importe quel processus lancé par un thread frère B. Pour qu'un programme qui se base sur ce comportement fonctionne sous Linux, il faudra en modifier le code.

thread est juste un processus qui est créé à l'aide de la routine - spécifique Linux - **clone(2)**. Les routines porta bles, comme **pthread_create(3)** sont implémentées en appelant **clone(2)**. Ainsi, si deux threads A et B sont frères, alors le thread A ne peut pas se mettre en attente sur un processus créé par le thread B ou ses descendants, car un oncle ne peut pas attendre un neveu. Sur d'autre systèmes Unix, où de multiples threads sont implémentés au sein d'un unique processus, le thread A peut naturellement attendre la fin de n'importe quel processus lancé par un thread frère B. Pour qu'un programme qui se base sur ce comportement fonctionne sous Linux, il faudra en modifier le code.