Mines de Paris – Option 3A – MSI
Georges-André Silber

TP UNIX – Les processus

Les prototypes des appels système sont donnés pour C sous Linux, il peut y avoir de légères différences avec d'autres types de systèmes Unix comme macOS ou FreeBSD. Dans tous les cas, se référer à la commande `man` pour chaque commande.

---

fork(2)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

La primitive `fork()` permet de créer un nouveau processus. Lorsqu'un processus (appelé le processus parent) exécute la primitive `fork()`, le système crée un nouveau processus (appelé processus enfant) en effectuant une duplication intégrale du contexte du processus père, y compris le compteur ordinal, mais en lui attribuant un nouveau `pid`. En particulier, le processus fils hérite de tous les fichiers ouverts par le processus père. Après `fork()`, les deux processus exécutent en pseudo parallèle l'instruction qui suit l'appel à `fork()`. La différentiation des deux processus se fait par le code retour de la primitive `fork()` qui est 0 dans le processus fils, le `pid` du fils dans le processus père et -1 en cas d'erreur (le processus fils n'a pas été créé). La structure d'un programme utilisant la primitive `fork()` sera la suivante :

```
pid_t pid;

pid = fork();
if (pid == -1) {
    perror("ERREUR fork");
    exit(-1);
}
if (pid == 0) {
    /* processus fils */
    exit(0);
}
/* processus pere */
exit(0);
```

Si le processus parent ne fait pas de `wait()` sur le processus enfant et que le processus enfant n'est pas terminé au moment où le processus parent se termine, le processus enfant devient *orphelin* et est adopté par le processus de pid 1. Si le parent ne fait pas de `wait()` sur son enfant, le processus enfant devient un *zombie* et reste dans la table des processus. Il ne consomme cependant pas de ressources et sera enlevé après un certain temps.

Il faut utiliser `_exit()` au lieu d'`exit()` si `execve()` n'est pas appelée, car `exit()` va vider et fermer les canaux d'entrée/sortie standards, corrompant ainsi les structures de donnée d'entrée/sortie du parent. Voir `exit(2)`.

getpid(2)

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

La primitive `getpid()` renvoie le numéro du processus. La primitive `getppid()` renvoie le numéro du processus parent.

wait(2)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

La fonction `wait()` suspend l'exécution d'un thread jusqu'à ce que l'information sur l'un de ses enfants qui a terminé soit disponible, où jusqu'à ce qu'un signal soit reçu par le thread. Si l'information sur l'un de ces parents est disponible, `wait()` renvoie le pid du processus qui a terminé et si `stat_loc` est différent de 0, stocke des informations dans celui-ci. Ces informations sont utilisables par l'intermédiaire des macros décrites dans la page du manuel `wstat(5)` et sont mises à jour par la primitive `exit(2)`. Les différentes fins des processus enfants sont empilés et plusieurs appels à `wait()` permettent de récupérer successivement les pids de tous les processus enfants ayant terminé leur exécution. Si il n'y a plus de processus enfant, `wait()` retourne -1 et `errno` est positionné à `ECHILD`. Si la primitive `wait()` est interrompue par un signal, elle retourne -1 et `errno` est positionné à `EINTR`. Voici un exemple d'utilisation de `wait()` :

```
pid_t pid;

pid = fork();
if (pid == -1) {
    perror("ERREUR fork");
    exit(-1);
}
if (pid == 0)
    /* processus fils */
else
    while (pid != wait(0));
```

exec(2)

```
#include <unistd.h>
int execl(const char *path, const  char  *arg0,  ...,
          const char *argn, char * /*NULL*/);
int execv(const char *path, char *const argv[]);
int execle(const char *path, char *const arg0[], ...,
          const char *argn, char * /*NULL*/, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char  *arg0,  ...,
           const char *argn, char * /*NULL*/);
int execvp(const char *file, char *const argv[]);
```

Cette primitive permet d'activer l'exécution d'un nouveau programme, au début de celui-ci, pour le compte du processus qui appelle `exec()`. Le code du nouveau programme remplace le code actuel et le segment de données remplace le segment de données actuel. Il n'y a pas de création de nouveau processus. En particulier, les fichiers ouverts restent ouverts pour le nouveau programme (mêmes descripteurs). Le programme qui sera exécuté se trouve dans le fichier dont le chemin d'accès absolu est `path`. L'exécution de ce nouveau programme se fera avec les arguments `arg0`, ..., `argn` pour la

forme `execl` ou avec les arguments dont les adresses sont dans le tableau de pointeurs `argv` pour la forme `execv`. Les formes `execlp` et `execvp` admettent un chemin d'accès `file` relatif aux différents répertoire de la variable `PATH`. La forme `execve` permet de passer en paramètre les variables d'environnement.

---

exit(2)

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

La primitive `exit()` met fin au processus qui l'a appelée. Elle commence par exécuter toutes les fonctions enregistrées par `atexit(3c)` dans l'ordre inverse de leur définition, puis elle vide tous les flots en sortie, les ferme tous et appelle la primitive `_exit()`.

Les 8 bits de poids faible de la variable `status` sont transmis dans les positions binaires 8 à 15 de la variable pointée par `stat_loc` (paramètre d'appel du `wait()` dans le processus parent).

La primitive `exit()` ne rend jamais la main à la fonction appelante. Elle est exécutée automatiquement après la fonction `main()`, avec la valeur de retour de celle-ci comme paramètre.

Exercice 1. À l'aide des appels système présentés ici, écrire un mini-shell en C qui permet à l'utilisateur de choisir parmi 4 commandes a, b, c et d exécutant par exemple :

— a : `date '+%Hh%M'`
— b : `uname -a`
— c : `top`
— d : arrêt

Exercice 2. Réécrire le mini-shell en Rust.