Programmation système : vie et mort des processus

Juliusz Chroboczek

9 Décembre 2009

Intuitivement, la création d'un processus s'accompagne de l'exécution d'un programme. Sous Unix[1] la création d'un processus et l'exécution d'un programme sont deux opérations distinctes. La création d'un processus exécutant un nouveau programme recquiert donc l'exécution de deux appels système : `fork` et `exec`.

1 Mort d'un processus

Normalement, un processus meurt lorsqu'il invoque l'appel système `_exit` :

```
void _exit(int status)
```

Lorsqu'un processus e□ectue un appel à `_exec`, il passe à l'état zombie, dans lequel il n'exécute plus de code. Le zombie disparaîtra dès que sont père fera un appel à `wait` (voir ci-dessous).

La foncontion `exit`, que vous avez l'habitude d'invoquer, est équivalente à `fflush(NULL)` suivi d'un appel à `_exit`.

Parenthèse : `_start`

Lorsqu'un processus est créé, la première fonction qui s'exécute est la fonction `_start` de la bibliothèque C. Cette fonction e□ectue les actions suivantes :

- elle calcule les arguments de ligne de commande et l'environnement, d'une façon dépendante du système ;
- elle stocke `environ` dans une variable globale ;
- elle invoque `main` en lui passant `argc` et `argv` ;
- si `main` retourne, elle invoque `exit`.

La fonction `_start` est la raison pour laquelle un retour normal de `main` (avec `return`) provoque une terminaison du programme.

1 Mais pas sous Windows ou MS-DOS.

2 Création de processus

2.1 L'appel système fork

Un processus est créé avec l'appel système `fork`

```
pid_t fork();
```

En cas de succès, un appel à `fork` a pour effet de dupliquer le processus courant. Un appel à `fork` retourne deux fois[2] : une fois dans le père, où il retourne le *pid* du fils nouvellement créé, et une fois dans le fils, où il retourne 0.

En cas d'échec, `fork` ne retourne qu'une fois, avec une valeur de retour valant -1.

2.2 L'appel système wait

L'appel système `wait` sert à attendre la mort d'un fils :

```
pid_t wait(int *status)
```

Cet appel à le comportement suivant :

- si le processus courant n'a aucun fils, il retourne -1 avec `errno` valant `ECHILD` ;
- si le processus courant a au moins un fils zombie, un zombie est détruit et son *pid* est retourné par `wait` ;
- si aucun des fils du processus courant n'est un zombie, `wait` bloque en attendant la mort d'un fils.

Si `status` n'est pas `NULL`, l'appel à `wait` y stocke la raison de la mort du fils. Cette valeur est opaque, mais peut être analysée à l'aide des macros suivantes :

- `WIFEXITED(status)` retourne vrai si le fils est mort de façon normale, i.e. du fait d'un appel à `_exit` ;
- `WEXISTSTATUS(status)`, retourne le paramètre de `_exit` utilisé par le fils ; cette macro n'est valide que lorsque `WIFEXITED(status)` est vrai.

L'appel système `waitpid`

L'appel système `waitpid` est une version étendue de `wait` :

```
pid_t waitpid(pid_t pid, int *status, int flags);
```

Le paramètre `pid` indique le processus à attendre ; lorsqu'il vaut -1, `waitpid` attend la mort de n'importe quel fils (comme `wait`). Le paramètre `flags` peut avoir les valeurs suivantes :

- 0 : dans ce cas `waitpid` attend la mort d'un fils ;
- `WNOHANG` : dans ce cas `waitpid` récupère un zombie s'il en existe un, mais retourne 0 si aucun des fils n'est mort.

L'appel système `waitpid` avec `flags` valant `WNOHANG` est un exemple de variante *non-bloquante* d'un appel système bloquant, ce que nous étudierons de façon plus détaillée dans deux cours.

2 Au contraire d'un appel à `_exit`, qui retourne 0 fois.

3 Exécution de programme

L'exécution d'un programme se fait à l'aide de l'appel système `execve` :

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

En cas de succès, `exitve` remplace le processus courant par un processus qui exécute le programme contenu dans le fichier `filename` avec les paramètres donnés dans `argv` et avec un environnement égal à celui contenu dans `envp`. Dans ce cas, `execve` ne retourne pas — le contexte dans lequel il a été appelé a été détruit (remplacé par un contexte du programme `filename`), il n'y a donc pas « où » retourner.

Fonctions utilitaires

L'appel système `execve` n'est pas toujours pratique à utiliser. La librairie standard contient un certain nombre de *wrappers*[3] autour d'`execve`.

Je mentionne particulièrement `execv`, qui duplique l'environnement du père, `execvp`, qui fait une recherche dans `PATH` lorsqu'il est passé un chemin relatif, et `execlp`, qui prend ses arguments en ligne terminés par un `NULL` (arguments *spread*) :

```
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
```

4 Exécution d'un programme dans un nouveau processus

Pour exécuter un programme dans un nouveau processus, il faut d'abord créer un nouveau processus (`fork`) puis exécuter le processus dans le père (`execve`). Il faut ensuite s'arranger pour exécuter `wait` dans le père.

Dans le cas d'une exécution *synchrone*, où le père ne s'exécute pas pendant l'exécution du fils, le schéma typique est le suivant :

```
pid = fork();
if(pid < 0) {
    /* Gestion des erreurs */
} else if(pid > 0) {
    execlp(...);
    /* Gestion des erreurs ? */
    exit(1);
}
pid = wait(NULL);
```

3 Vous voyez une bonne traduction ?

5 Parenthèse : le double fork

Lors d'une exécution asynchrone, où le père continue à s'exécuter durant l'exécution du fils, il faut s'arranger pour appeler `wait` afin de pouvoir éliminer les zombies. Une astuce souvent pratique consiste à déshériter le fils en appelant `fork` deux fois de suite, et en tuant le fils intermédiaire ; de ce fait, le fils devient un fils du processus `init` :

```
pid = fork();
if(pid < 0) {
    /* Gestion d'erreurs */
else if(pid == 0) {
    /* Fils intermédiare */
    pid = fork();
    if(pid < 0) {
        /* Gestion d'erreurs impossible */
    } else if(pid == 0) {
        /* Petit-fils */
        execlp(...);
        /* Gestion d'erreurs impossible */
        exit(1);
    }
    exit(0);
}
/* Père, attend la mort du fils intermédiare */
pid = wait(NULL);
```

Il existe une autre technique — hors programme pour ce cours — permettant d'obtenir le même résultat, qui consiste à ignorer le signal `SIGCHLD`.