Programmation Unix 2 – cours n°6

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Janvier 2017

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Boîte à outils Unix en C

Le noyau du système est écrit presque entièrement en C
(le reste est écrit en assembleur : drivers, etc)

Unix fournit une API complète en C pour manipuler le système :

- une grosse partie est normalisée POSIX ;
- l'autre partie est spécifique au système.

Nous allons étudier des fonctions système que `bash` utilise, et comment un processus interagit avec le système.

Plan du cours n°6

1. Création des processus
2. Terminaison des processus
3. Synchronisation
4. Recouvrement de processus

1 - Création des processus

Filiation des processus, évoquée au cours n°4 :

- Chaque processus
  - a un parent, et possède 0, 1 ou + fils
  - est identifié par son PID
  - connaît le PPID de son parent
- Le processus racine est `init`, de PID 1
  - `init` adopte automatiquement les orphelins (processus dont le père est terminé)

Obtenir le PID :

```
#include <unistd.h>      // Fonctions Unix standard
pid_t getpid  (void);    // pid_t = type entier
pid_t getppid (void);
```

Mécanisme de création

Mécanisme unique de création de processus :

Un processus (le père) demande, en appelant `fork()`, la création dynamique d'un nouveau processus (le fils).

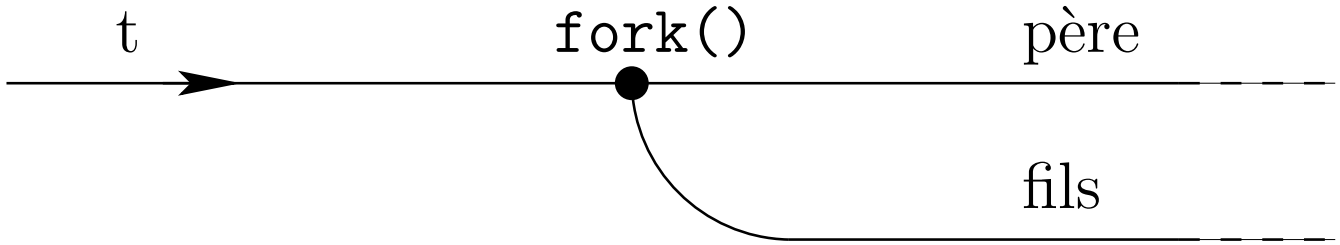
Le fils s'exécute ensuite de façon concurrente avec le père.

Tous les processus Unix sont créés de cette façon
(excepté le processus originel de PID 0).

Spécification :

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

Création par duplication



Avant `fork` :
un seul processus, qui va faire l'appel système.

Après `fork` :
2 processus reviennent de l'appel, et continuent le même programme.

Le fils est un clone du père : presque tout ce qui concerne le père est dupliqué (par *copy on write*), sauf : `man fork`

- le PID et PPID
- les temps d'exécutions
- alarmes et signaux en attente

→ Pas de partage des données : chaque processus travaille sur ses propres données, n'a pas d'accès aux autres.

Premier exemple

```
#include <stdio.h>          // printf
#include <unistd.h>         // fork, getpid

int main () {
  printf ("Je suis le futur père %d\n", (int) getpid());
  fork();
  printf ("Je suis %d\n", (int) getpid());
  return 0;
}
```

Exécution :
```
Je suis le futur père 436
Je suis 440
Je suis 436
```

Les 2 processus font bien le 2^e `printf`.
L'ordre d'affichage est aléatoire : dépend de l'ordonnanceur.

Comportement différent

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

int main () {
  pid_t p = getpid();
  printf ("Je suis le futur père %d\n", (int) p);
  fork();
  if (getpid() == p)
       printf ("Je suis le père %d mais ignore le "
               "PID de mon fils\n", (int) p);
  else printf ("Je suis %d fils de %d\n",
               (int) getpid(), (int) p);
  return 0;
}
```

Exécution :

```
Je suis le futur père 444
Je suis le père 444 mais ignore le PID de mon fils
Je suis 448 fils de 444
```

Utilisation du retour de `fork`

`fork()` renvoie une valeur différente selon le processus :

`-1`	Échec création processus (seul le père revient)
`0`	On est dans le fils
`>0`	On est dans le père ; PID du fils

En cas d'erreur système, on affiche la cause avec :

```
#include <stdio.h>
void perror(const char *s);
```

Exemple à retenir

```
#include <stdio.h>      // printf, perror
#include <stdlib.h>     // exit
#include <sys/types.h>  // pid_t
#include <unistd.h>     // fork, getpid, getppid

int main () {
  pid_t p = fork();
  if (p < 0) { perror("fork"); exit (1); }

  if (p == 0) {  /* Fils */
    printf ("Je suis %d fils de %d\n",
            (int) getpid(), (int) getppid());
    exit (0);
  } /* Fin fils */

  /* Suite du père */
  printf ("je suis %d père de %d\n", (int) getpid(), (int) p);
  exit (0);
}
```

Utilisation par `bash`

Chaque sous-shell est créé avec `fork()`.

Par exemple dans :

```
( commandes )
```

```
commande | commande
```

La commande est ensuite chargée avec `exec`, voir plus loin.

2 - Terminaison des processus

La terminaison d'un processus survient

- à réception d'un signal Unix ;
- à la demande du programme lui-même.

À tout endroit du programme :

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

→ Terminaison immédiate avec code de terminaison `status`

Seul l'octet de poids faible est significatif : 0 succès, $\neq 0$ échec.

Constantes `stdlib.h` : `EXIT_SUCCESS`, `EXIT_FAILURE`

Retour de `main`

Appel de `main` par le système :

```
_start() {

  int status = main (...);

  exit (status);
}
```

Dans `main` :

`return status;` est équivalent à `exit (status);`

Différence : dans un appel récursif de `main`, on revient d'un niveau.

Fonctions d'avant-terminaison

= fonctions de nettoyage appelées par le système juste avant la terminaison.

Enregistrer une fonction :

```
#include <stdlib.h>
int atexit ( void (*fonction)(void) );
```

On peut enregistrer plusieurs fonctions, elles seront appelées dans l'ordre inverse.

Nombre max : `ATEXIT_MAX` dans `<limits.h>`

Exemple

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

void au_revoir (void) { printf ("Au revoir\n"); }
void bye_bye   (void) { printf ("Bye bye\n"); }

int main () {
  atexit (au_revoir);
  atexit (bye_bye);
  printf ("Avant exit ...\n");
  exit (0);
  printf ("Après exit\n");
}
```

Exécution :

```
Avant exit ...
Bye bye
Au revoir
```

Durant la terminaison

Lorsqu'on appelle `exit()` :

- les fonctions d'avant-terminaison sont appelées ;
- les buffers sont flushés ;
- les fichiers ouverts sont fermés ;
- la mémoire est rendue ;
- le processus passe à l'état de Zombie.

Appel plus bas niveau : `_exit()` :

- les fonctions d'avant-terminaison ne sont pas appelées ;
- flush des buffers : système-dépendant ;
- (le reste est idem).

Utilisation par `bash`

La commande `exit n` provoque l'appel de `exit(n)`

Si on est dans un sous-shell, provoque sa terminaison :

```
( exit )
```

sinon provoque la terminaison du script (ou du terminal en mode interactif).

3 - Synchronisation

Un processus terminé passe à l'état de Zombie :

- tout ce qui le concerne est déchargé de la mémoire ;
- ne subsiste qu'une entrée dans la table des processus (état `DEFUNCTED`) avec son exit status.

Utilité du mécanisme :

permettre au père de connaître le exit status de son fils, même si le fils est mort depuis un moment.

Inconvénient :

Il faut nettoyer la table des processus des zombies, sinon vite saturée.

Attente de terminaison

Mécanisme unique pour relever le exit status et nettoyer la table :

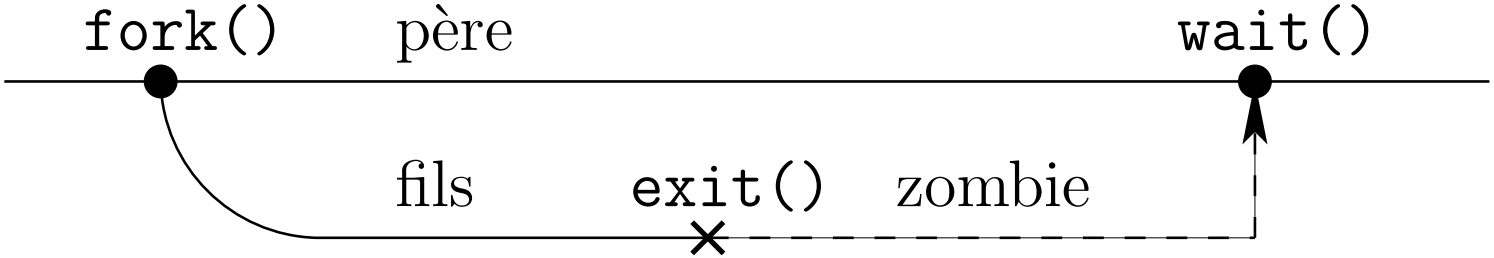
```
#include <sys/types.h>
#include <sys/wait.h>

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

`wait` attend la fin de n'importe quel fils et renvoie son PID ;
`waitpid` attend la fin du fils ayant le PID `pid`

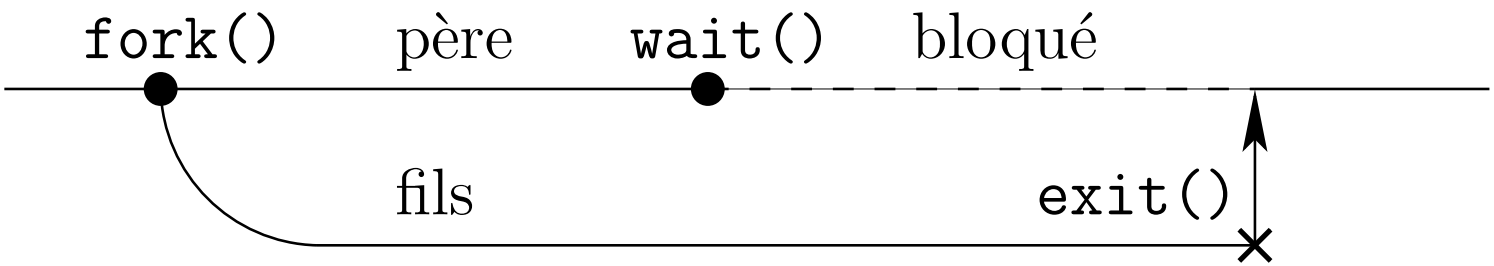
Appel de `wait`

A. Si le processus appelant ne possède aucun fils : retour immédiat, renvoie `-1`.

B. Si le processus appelant possède des fils zombies : retour immédiat, renvoie l'un des PID ; ce zombie est supprimé de la table des processus.



C. Si le processus appelant possède des fils mais aucun zombie : bloqué jusqu'à terminaison de l'un des fils, puis **B**.



Récupération du status

 `wait();` → `SIGSEGV`

`wait(NULL);` → attente du zombie sans récupération

Pour récupérer le status :

```
int status, stat_infos;
wait (&stat_infos);
status = WEXITSTATUS(stat_infos);
```

`stat_infos` contient le `exit status` du fils (décalé de 8 bits) plus d'autres informations sur la terminaison (`man wait`).

Attente d'un fils particulier

`waitpid (pid, &stat_infos, options);`

permet d'attendre la fin du fils `pid > 0`, ou de n'importe quel fils (`pid = -1`).

`options = 0` : attente bloquante

`options = WNOHANG` : retour immédiat

Remarque : à chaque terminaison de processus, un signal Unix `SIGCHLD` est envoyé au père $\rightarrow$ S'il le capte, il peut lever le zombie avec `waitpid (-1, NULL, WNOHANG);`

Mais que fait `init` ?

Lorsque le père A d'un fils B se termine, B est adopté par `init`

- Pour B, `getppid() = 1`
- Quand B se terminera, `init` recevra un signal `SIGCHLD`, puis supprimera le zombie B.

Utilisation par `bash`

Lorsque `bash` exécute des commandes ou sous-shells

- séquentiellement : il attend la fin de chacun avec `wait`.
- en tâche de fond (`&`), il est informé de leur terminaison par un signal `SIGCHLD`.

Après toute commande ou sous-shell, il récupère le status dans `$?`

En mode interactif, il affiche le prompt après le retour de `wait`.

4 - Recouvrement de processus

Recouvrement = chargement d'un nouveau programme binaire ou script en mémoire.

- l'espace d'adressage du processus est entièrement écrasé : perte irréversible de l'ancien programme et des données ;
- le nouveau programme démarre au début (`_start` puis `main`) ;
- pas de création de nouveau processus : on conserve le PID, le PPID, la TD (redirections) ;
- retour d'un recouvrement uniquement s'il n'a pu être fait.

Recouvrement en bash

```
#! /bin/bash
exec ls -l /home
echo "ls: échec recouvrement" >&2
exit 1
```

Le script `bash` se recouvre avec la commande `ls` et les arguments `ls -l /home`.

$\rightarrow$ `ls` aura le même PID, PPID et redirections que le script original.

$\rightarrow$ Lorsque `ls` sera terminée, pas de retour dans le script (qui n'existe plus), c'est la fin du processus.

Recouvrement en C

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

`execv*` : arguments par tableau `argv` terminé par `NULL`

`execl*` : arguments en liste `arg1, arg2, .., NULL`

`...` = fonction variadique : `man stdarg`

Recouvrement en précisant le chemin

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

Le programme est cherché dans le chemin `path` :

```
char *myargs[] = { "ls", "-l", "/home", NULL};
execv ("/bin/ls", myargs);
perror ("exec ls");
exit (1);
```

ou de manière équivalente :

```
execl ("/bin/ls", "ls", "-l", "/home", NULL);
perror ("exec ls");
exit (1);
```

Recouvrement avec le PATH

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

Le programme `file` est cherché dans `$PATH` :

```
char *myargs[] = { "ls", "-l", "/home", NULL};
execvp ("ls", myargs);
perror ("exec ls");
exit (1);
```

ou de manière équivalente :

```
execlp ("ls", "ls", "-l", "/home", NULL);
perror ("exec ls");
exit (1);
```

Utilisation par `bash`

Chaque fois que `bash` lance une commande :

- il crée un fils avec `fork()` ;
- le fils se recouvre avec `execvp()` par la commande et les arguments ;
- au niveau du père :
  - Si la commande est lancée en tâche de fond :
    `$?=0` et passe à la suite.
  - si commande séquentielle :
    attend la fin avec `wait()`, récupère `$?` et passe à la suite.

Méthode alternative

Pour lancer une commande dans un programme C, au lieu de faire `fork` / `exec`, on peut appeler

```
#include <stdlib.h>
int system(const char *command);
```

Crée un fils et un petit-fils :

- Le fils se recouvre par le shell `/bin/sh` , qui va découper les arguments ;
- le petit-fils se recouvre avec la commande et les arguments.

Renvoie le `status` de la commande terminée.

Deux processus, coût de l'interprétation par `sh`, impossibilité de choisir le shell : à éviter !