



Licence d'informatique  
Module de Programmation des systèmes

## Gestion des processus

Philippe MARQUET

Février 2005

Révision de mars 2006

Ce document est le support de travaux dirigés et de travaux pratiques relatifs aux notions de processus, exécutions concurrentes, et signaux. On étudie la manipulation de ces notions par les primitives POSIX et les fonctions de la librairie standard.

### 1 Clonage de processus

#### Exercice 1 (Généalogie de processus)

Expliquez le comportement du programme de la figure suivante. □

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int
main (void)
{
    int i;
    pid_t pid;

    for (i=0; i<3; i++) {
        pid=fork();
        if (pid == -1) { /* erreur */
            perror("erreur fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) { /* fils */
            fprintf(stderr, "fils : %d\n", i);
        } else { /* pere */
            fprintf(stderr, "pere : %d\n", i);
        }
    }
    exit(EXIT_SUCCESS);
}
```

FIG. 1 – Que fait ce programme ?

#### Exercice 2 (Quatre fils)

**Question 2.1** Donnez un programme qui affiche les entiers de 0 à 3 par 4 processus différents. L'exécution de ce programme se termine après l'affichage des 4 entiers. □

### **Clonage interactif**

(sur une idée de..., merci !)

Le nombre de processus pouvant être lancés par un utilisateur, mais aussi par le système sont limités.

Si un programme (erroné) crée des processus en boucle, on risque de ne plus pouvoir allouer de processus, même pour tuer ledit programme.

On peut se garder de telles erreurs en TP en utilisant la fonction suivante

```
pid_t
ifork()
{
    fprintf(stderr, "fork() %d ? (^C to abort) ", getpid());
    fflush(stderr);
    getchar();
    return fork();
}
```

La saisie d'un retour chariot suffit à accepter le `fork()`.

**Question 2.2** Assurez que les processus fils affichent les entiers dans l'ordre croissant. ☐

### **Exercice 3 (Tri fourche)**

Écrivez une fonction

```
void trif (void(*f1)(void), void(*f2)(void), void(*f3)(void));
```

Le processus exécutant `trif(f1, f2, f3)` engendre des processus exécutant respectivement les fonctions `f1()`, `f2()`, et `f3()`, et attend la fin des processus engendrés pour terminer la fonction.

Cette fonction est par exemple utilisée dans le contexte suivant :

```
static void
f(int seconds, const char *fname)
{
    sleep(seconds) ;
    fprintf(stderr, "Fonction %s() executee par le processus %d\n",
            fname, getpid()) ;
}

static void fa(void) { f(4, "fa"); }
static void fb(void) { f(2, "fb"); }
static void fc(void) { f(3, "fc"); }

int
main(void)
{
    trif(fa, fb, fc);
    fprintf(stderr, "terminaison de main()\n");

    exit(EXIT_SUCCESS);
}
```

☐

### **Exercice 4 (Multi fourche)**

La fonction `multif()` généralise la fonction `trif()` précédente.

```
typedef int (*func) (int);

int multif (func f[], int n);
```

Le type `func` est défini comme « pointeur sur une fonction à un paramètre entier retournant un entier ».

Les arguments de `multif()` sont un tableau de telles fonctions et la taille `n` de ce tableau. Chacune des fonctions est exécutée par un processus différent. Ce processus se termine en retournant comme statut la valeur de la fonction.

La fonction `multif()` se termine elle-même en retournant la conjonction des valeurs retournées par les processus fils : elle ne retourne une valeur vraie que si chacun des processus fils a retourné une valeur vraie. □

### Exercice 5 (Pas de zombies)

Un processus en désirant pas interagir avec son fils ne peut l'abandonner. À sa terminaison, le fils passerait en l'état zombi.

Cependant, considérant qu'un processus orphelin est adopté par le processus `init` de PID 1, on peut utiliser la technique dite du double fork :

- le père engendre un processus fils dont le rôle est d'engendrer à son tour un processus petit-fils qui réalisera le travail ; ce processus fils termine aussitôt ;
- à sa terminaison le petit-fils, orphelin sera adopté par `init` ;
- le père attend la terminaison de son fils qui doit survenir rapidement.

**Question 5.1** Donnez le code d'une fonction

```
typedef void (*func) (void *);  
  
void forkfork(func f, void *arg);
```

qui utilise la technique du double fork pour faire exécuter le fonction `f()` par un processus petit fils. □

**Question 5.2** Comment faire en sorte que le père récupère le PID de son petit-fils? □

**Question 5.3** Que pensez, en terme de coût de copie mémoire, de cette technique du double fork? □

**Question 5.4** Donnez le corps d'un programme illustrant l'utilisation de notre fonction `forkfork()` □

## 2 Gestion de processus

### Exercice 6 (À vos marques, prêts...)

Afin de mettre en évidence le partage du temps processeur entre les processus prêts, construisez un programme figurant une course entre processus ; ce programme doit

- créer dix fils ;
- chacun des fils compte jusqu'à 5 millions, affiche un message; compte à nouveau jusqu'à 5 millions ;
- attendre après ses 10 fils et afficher leur ordre d'arrivée. □

### Exercice 7 (Observation de processus)

Il s'agit d'écrire un programme à plusieurs processus qui effectue la tâche suivante. Le processus père crée `N` processus fils. Chaque processus fils réalise dans une boucle infinie les instructions suivantes :

- afficher son `pid` pour préciser qu'il est toujours vivant ;
- s'endormir pour 5 secondes grâce à la fonction de la bibliothèque standard `sleep()` ;

### Commandes de gestion des processus

Les deux commandes `ps` et `kill` sont utilisées pour la gestion des processus.

**La commande `ps`** affiche des informations sur les processus actifs. Sans options, elle affiche les processus du même terminal que le processus `ps`. La sortie contient uniquement le `pid` du processus, le nom du terminal, le temps cumulé d'exécution et le nom de la commande.

Les options les plus courantes sont :

- a** liste tous les processus sauf ceux non associés à un terminal ;
- e** liste tous les processus du système ;
- f** donne plus d'information pour chaque processus ;
- u uidlist** liste les processus des utilisateurs donnés dans `uidlist`.

**La commande `top`** affiche les mêmes informations que `ps` en rafraîchissant périodiquement l'affichage.

**La commande `kill`** permet d'envoyer un signal à un processus. Deux syntaxes sont disponibles :

**`kill -s nomdesignal pid...`** envoie le signal dont le nom est `nomdesignal` aux processus dont on donne le `pid` ;

**`kill -numerodesignal pid...`** envoie le signal de numéro `numerodesignal` aux processus dont on donne le `pid`.

Un signal peut donc être nommé symboliquement (`kill -l` donne la liste des noms) ou par son numéro. Nous utiliserons ici le signal `KILL` de numéro 9 qui permet de tuer un processus.

### Fonction `system()`

La fonction de la bibliothèque standard

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

```
int system (const char *command);
```

exécute la commande `command` et retourne après sa terminaison. La valeur de retour de la commande sous la forme habituellement retournée par `wait()` est retournée, la valeur -1 en cas d'erreur.

Le processus père, après avoir créé tous ses fils, affiche les processus actifs du terminal et ce à l'aide de la commande `ps` (voir l'encart). Pour cela, il fera appel à la fonction `system()` qui permet d'invoquer une commande Shell (voir l'encart). Puis, jusqu'à ce qu'il n'ait plus de fils, il exécutera le code suivant :

- attendre la terminaison d'un fils ;
- afficher le `pid` du fils terminé.

**Question 7.1 (Préparez)** Donnez, sous forme de pseudo-code, la structure de ce programme. Explicitiez, a priori, son comportement. □

**Question 7.2 (Exécutez)** Utilisez la commande `kill` pour terminer les processus fils. Que se passe-t-il si vous tuez le processus père ? □

**Question 7.3 (Observez)** Modifiez le programme de telle façon que le processus père attende, par exemple sur une saisie clavier, avant de commencer à attendre la terminaison de ses processus fils à l'aide de `wait()`.

### Arguments de la ligne de commande

Le programme suivant `argv.c` affiche les arguments qui lui sont passés sur la liste de commande

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

int
main (int argc, char *argv[])
{
    int i;

    fprintf(stderr, "    argc = %d\n", argc);

    for (i=0; i<argc ; i++) {
        fprintf(stderr, "argv[%d] = %s\n", i, argv[i]);
    }

    exit(EXIT_SUCCESS);
}
```

Ces quelques exemples d'exécution explicitent la manière dont les arguments de la ligne de commande sont gérés :

```
% ./argv emacs mozilla xterm
    argc = 4
argv[0] = ./argv
argv[1] = emacs
argv[2] = mozilla
argv[3] = xterm
% ./argv "emacs -u phm" "mozilla -mail" xterm
    argc = 4
argv[0] = ./argv
argv[1] = emacs -u phm
argv[2] = mozilla -mail
argv[3] = xterm
```

- Tuez un de ses processus fils et examinez l'état des processus à l'aide de la commande `ps-a`. Expliquez le résultat.
- Faites continuer le processus père (saisie clavier) afin qu'il attende la terminaison de son fils et examinez l'état des processus à l'aide de la commande `ps-a`. ☐

## 3 Mutation de processus

### Exercice 8 (condition)

Il s'agit d'implanter une commande `on` qui exécute une seconde commande `cmd` si et seulement si l'exécution d'une première commande `pred` se termine sur un succès :

```
on pred do cmd
```

On utilise cette commande comme sur l'exemple suivant :

```
on test -r spool/file do lpr spool/file
```

qui ne fait appel à la commande d'impression `lpr` que si le fichier existe (commande `test`). ☐

### Exercice 9 (Redirection)

Il s'agit d'implanter une commande

```
to file cmd...
```

qui exécute les commandes `cmd...` les unes après les autres en redirigeant leur sortie standard vers le fichier `file` qui est éventuellement tronqué au début de l'exécution de `to`. □

### Exercice 10 (Exécutions de commandes)

On désire implanter un programme `do` qui exécute indépendamment et simultanément une série de commandes Shell données sur la ligne de commande. L'exécution du programme se termine quand l'ensemble des commandes a terminé. Le programme retourne alors un statut formé de la conjonction (et, défaut) ou de la disjonction (ou) des statuts retournés par les commandes selon la valeur de l'option. La syntaxe est la suivante :

```
do [-and|-or] command...
```

On utilise cette commande `do` suivant l'exemple suivant :

```
do -and emacs mozilla xterm
```

**Question 10.1** Implantez la commande `do`. □

On désire que les commandes lancées par la `do` puissent accepter des paramètres et ainsi utiliser la commande de la manière suivante :

```
do -and "emacs -u phm" "mozilla -mail" xterm
```

**Question 10.2** Implantez cette extension de la commande `do`. Reportez vous à l'encart page précédente explicitant la manière dont les arguments de la ligne de commande sont traités. □

Une utilisation possible de la commande `do` est de ne s'intéresser qu'aux valeurs de retour des commandes lancées. Dans ce cas, il est possible de conclure sur la valeur de retour de la commande `do` dès qu'une des commandes retourne un succès (pour l'option `-or`, ou retourne un échec pour l'option `-and`) : on parle de conjonction ou disjonction coupe-circuit. Ce fonctionnement est activée par l'option `-cc` de la commande `do`.

**Question 10.3** Implantez cette option `-cc` de la commande `do`. □

Considérant l'option `-cc`, une fois le résultat acquis, il n'est pas utile de laisser poursuivre les exécutions des commandes dont le résultat ne sera pas exploité. L'option `-k`, associée à `-cc`, indique de tuer (*kill*) les commandes non encore terminées alors que le résultat de `do` a pu être déterminé.

**Question 10.4** Implantez cette dernière option `-k` de la commande `do`. □