Programmation des Systèmes

Gestion des processus

Philippe MARQUET
et les équipes pédagogiques successives*

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 (Compréhension du clonage simple)
Observez le programme de la figure suivante.

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

   printf("A: ce message s'affiche combien de fois ?\n");
   fflush(stdout);

   pid=fork();
   if (pid==0) {
      /* qui suis-je, le pere ou le fils ? */
      printf("je suis le ...\n");
   } else {
      /* qui suis-je, le pere ou le fils ? */
      printf("je suis le ...\n");
   }

   printf("B: ce message s'affiche combien de fois ?\n");
   fflush(stdout);

   exit(EXIT_SUCCESS);
}
```

FIGURE 1 – Que fait ce programme ?

Question 1.1 Répondez aux questions posées dans le code. □

Question 1.2 Pouvez-vous déterminer l'ordre d'apparition des messages ? Si oui donnez-le, sinon pourquoi ? Pouvez-vous alors déterminer un ordre partiel d'apparition des messages ? □

Question 1.3 On ajoute dans le `else` et avant le `printf` l'instruction

```
wait(NULL);
```

Cela change-t-il quelque chose ? □

Question 1.4 Pour garantir un fonctionnement plus sûr de ce code, que manque-t-il ? □

Question 1.5 À quoi les instructions `fflush(stdout)` servent-elles ? □

Exercice 2 (Généalogie de processus)
On exécute le programme de la figure suivante.

```
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);
}
```

FIGURE 2 – Que fait ce programme ?

Question 2.1 Déroulez son exécution et indiquez ce qui est affiché. □

Question 2.2 Combien de processus sont-ils lancés en tout ? (il pourra être utile de représenter sous forme arborescente les processus) □

Question 2.3 Quels appels systèmes peut-on utiliser pour changer l'affichage et permettre un meilleur suivi des processus (savoir qui est fils de qui) ? □

Exercice 3 (Quatre fils)
Question 3.1 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. □

Question 3.2 Assurez que les processus fils affichent les entiers dans l'ordre croissant. □

Exercice 4 (Tri fourche)
Écrivez une fonction

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

Clonage interactif

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()`.

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 5 (Multi fourche)
La fonction `multif()` généralise la fonction `trif()` précédente.

```
typedef int (*func_t) (int);

int multif (func_t f[], int args[], int n);
```

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

Les arguments de `multif()` sont un tableau de telles fonctions, un tableau des arguments à passer à ces fonctions, et la taille `n` de ces tableaux. Chacune des fonctions est exécutée par

un processus différent. La fonction `f[i]()` sera appelée avec `arg[i]` comme argument. 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 6 (Pas de zombis)
Un processus ne désirant pas interagir avec son fils ne peut l'abandonner. À sa terminaison, le fils passerait à 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 6.1 Donnez le code d'une fonction

```
typedef void (*func_t) (void *);

void forkfork(func_t f, void *arg);
```

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

Question 6.2 Comment faire en sorte que le père récupère le PID de son petit-fils ? □

Question 6.3 Que penser, en terme de coût de copie mémoire, de cette technique du double fork ? □

Question 6.4 Donnez le corps d'un programme illustrant l'utilisation de `forkfork()`.

Ce programme devra mettre en évidence le fonctionnement de `forkfork()` en rendant « visibles » les trois processus, l'adoption par `init`, etc. □

2 Gestion de processus

Exercice 7 (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 100 millions, affiche un message ; compte à nouveau jusqu'à 100 millions ;
— attendre ses 10 fils et afficher leur ordre d'arrivée. □

Exercice 8 (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()`.

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 page suivante). Pour cela, il fera appel à la fonction `system()` qui permet d'invoquer une commande Shell (voir l'encart page suivante). Puis, jusqu'à ce qu'il n'ait plus de fils, il exécutera le code suivant :

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.

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

Question 8.1 (Préparez) Donnez, sous forme de pseudo-code, la structure de ce programme. Explicitez, a priori, son comportement. □

Question 8.2 (Exécutez) Utilisez la commande `kill` pour terminer les processus fils. Que se passe-t-il si vous tuez le processus père ? □

Question 8.3 (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()`.

— 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`. □

Arguments de la ligne de commande

Le programme suivant `argv.c` affiche les arguments qui lui sont passés sur la ligne 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 xclock  xterm
   argc = 4
argv[0] = ./argv
argv[1] = emacs
argv[2] = xclock
argv[3] = xterm
% ./argv  "emacs -u phm" "xclock -update 1" xterm
   argc = 4
argv[0] = ./argv
argv[1] = emacs -u phm
argv[2] = xclock -update 1
argv[3] = xterm
```

3 Mutation de processus

Exercice 9 (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 10 (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 11 (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

Bibliothèque de gestion de multiples commandes à arguments

La bibliothèque `makeargv` installée dans `/home/enseign/PDS/*/` fournit deux fonctions de création et destruction d'arguments :

```
extern int makeargv(const char *s, const char *delimiters, char ***argvp);
extern void freeargv(char **argv);
```

Vous vous inspirerez de l'exemple suivant `makeargv-main.c` pour utiliser cette bibliothèque :

```
#include "makeargv.h"

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

    for (i=1; i<argc; i++) { /* traiter argv[i] */
        char **cmdargv;
        char **arg;

        /* création du argv de l'argument i */
        status = makeargv(argv[i], " \t", &cmdargv);
        assert(status>0);

        /* test: affichage */
        fprintf(stderr, "[%s]\t%% ", cmdargv[0]);
        for (arg=cmdargv; *arg; arg++)
            fprintf(stderr, "%s ", *arg);
        fprintf(stderr, "\n");

        /* libération mémoire */
        freeargv(cmdargv);
    }

    exit(EXIT_SUCCESS);
}
```

Une exécution de ce programme d'exemple est la suivante :

```
% ./makeargv-main "cat -n" "grep -v foo" wc
[cat]   % cat -n
[grep]  % grep -v foo
[wc]    % wc
```

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 11.1 Implantez la commande `do`. □

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

```
do --and "emacs -u phm" "xclock -update 1" xterm
```

Question 11.2 Implantez cette extension de la commande `do`. Reportez-vous à l'encart page 6 explicitant la manière dont les arguments de la ligne de commande sont traités. Vous pourrez utiliser la bibliothèque `makeargv` décrite dans l'encart page précédente pour gérer les paramètres de la ligne de commande.

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 11.3 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 `-kill`, associée à `-cc`, indique de tuer (*kill*) les commandes non encore terminées alors que le résultat de `do` a pu être déterminé.

Question 11.4 Implantez cette dernière option `-kill` de la commande `do`. □