Labo 4

Objectifs : d'autres appels systèmes liés à la gestion des processus

Introduction

Maintenant que la notion de processus ainsi que la gestion de base de ceux-ci est maîtrisée, nous pouvons aborder d'autres appels systèmes intéressants. Il est en effet possible d'attendre un processus en particulier, de modifier le comportement d'un processus…

Attendre un processus en particulier

L'appel système permettant de préciser le processus à attendre est `waitpid`.

waitpid	
Librairie	#include <sys/wait.h> #include <sys/types.h>
Prototype	`pid_t waitpid (pid_t pid, int *status, int options);`
Commentaires	Suspend l'exécution du processus appelant jusqu'à ce que le fils spécifié par son *pid* ait changé d'état. Le premier argument, `pid`, permet de déterminer le processus fils dont on désire attendre le changement d'état : • si `pid` est strictement positif, la fonction attend la fin du processus dont le *pid* correspond à cette valeur. • si `pid` vaut 0 (ou WAIT_MYPGRP), on attend la fin de n'importe quel processus fils appartenant au même groupe que le processus appelant. • si `pid` vaut -1 (ou WAIT_ANY), on attend la fin de n'importe quel fils, comme `wait`. • si `pid` est strictement inférieur à -1, on attend la fin de n'importe quel processus fils appartenant au groupe de processus dont le numéro est *-pid*. Par défaut, il n'attend que les fils terminés, mais ce comportement est modifiable avec l'argument `options`. La valeur de `options` est une des constantes suivantes ou une combinaison de celles-ci via un OU binaire (`|`) : `WNOHANG` ne pas bloquer si aucun fils ne s'est terminé. Si aucun fils spécifié par *pid* n'a encore changé d'état, 0 est renvoyé. En cas d'échec -1 est renvoyé `WUNTRACED` recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue Si `status` n'est pas `NULL`, l'identifiant du processus fils et son état de terminaison est renvoyé via `status`.

Le programme ci-dessous montre un exemple d'utilisation de cet appel système. En effet, le père crée un fils qui fait un `sleep` d'une durée reçue en paramètre via le père… Le père attend **ce** fils en affichant un message d'attente (espacé d'une seconde) jusqu'à ce que le fils se termine.

Pour rappel, on peut passer des paramètres à un programme en utilisant la signature complète de la fonction `main` : `int main (int argc, char *argv[])`. Le premier argument, `argc`, est le nombre d'arguments que reçoit la fonction `main`. Le second, `argv`, est un tableau de pointeur dont chaque élément pointe sur une chaine de caractère. La première chaine contient le nom du processus et les autres pointent sur des chaines de caractères correspondant à chacun des paramètres.

Pour passer le paramètre, on utilise la ligne de commande en procédant comme suit

```
$ ./nomExécutable arg1 arg2…
```

Voici le programme en question :

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

// Dans certain cas, les variables globales sont utiles !
int sleepTime;
int status;
int myPid;

void fils (char * nomPgm) {
   myPid = getpid();
   printf("PID %d - Debut du fils (programme = %s) [PPID = %d]\n",
            myPid, nomPgm, getppid());
   sleep(sleepTime);
   printf("PID %d - Fin du fils\n", myPid);
   exit(EXIT_SUCCESS);
}

int main (int ac, char **av) {
   int pid;

   myPid = getpid();
   printf("PID %d - Debut du Pere\n", myPid);

   // N'oubliez pas de vérifier que l'utilisateur a bien passé un argument !
   if (ac < 2) {
      printf("Usage : ./ex sleepTime [sleepTime est un entier entre 1 et 9 inclus]\n");
      exit (EXIT_FAILURE);
   }
   // Tous les paramètres sont passés sous la forme de chaînes de caractères.
   // Lorsque le programme doit travailler avec des entiers, il faut d'abord les
   // transformer en variable de type int via la fonction atoi
   sleepTime = atoi(av[1]);
   if (sleepTime < 1 || sleepTime  > 9) {
      printf("Temps d'attente non compris entre 1 et 9\n");
      exit (EXIT_FAILURE);
   }

   pid = fork();
   if (pid < 0) {
      perror("Erreur lors du fork du fils");
      exit(EXIT_FAILURE);
   }
   if (pid == 0) {
      fils(av[0]);
   }
   /* Suite le pere */
   while (waitpid(pid, &status, WNOHANG) == 0) {
      printf("PID %d - Le processus fils n'est pas encore termine...\n", myPid);
      sleep(1);
   }
   printf("PID %d - Le fils [PID %d] s'est termine avec le statut %04.4X\n",
            myPid, pid, status);

   printf("PID %d - Fin du Pere\n", myPid);
   exit(EXIT_SUCCESS);
}
```

L'exécution de ce programme avec un `sleepTime` de 4 et sans interruption donne la sortie présentée en figure 1.

```
piroc@svr24:~$ ./expl41 4
PID 28876 - Debut du Pere
PID 28876 - Le processus fils n'est pas encore termine...
PID 28877 - Debut du fils (programme = ./expl41) [PPID = 28876]
PID 28876 - Le processus fils n'est pas encore termine...
PID 28876 - Le processus fils n'est pas encore termine...
PID 28876 - Le processus fils n'est pas encore termine...
PID 28876 - Le processus fils n'est pas encore termine...
PID 28877 - Fin du fils
PID 28876 - Le fils [PID 28877] s'est termine avec le statut 0000
PID 28876 - Fin du Pere
```

Figure 1 - Appel système `waitpid`

Si on assigne un sleepTime de 9 et que, en cours d'exécution, on tue le processus fils via la commande `kill` *pid*, on obtient la sortie visible à la figure 2.

```
piroc@svr24:~$ ./expl41 9
PID 28834 - Debut du Pere
PID 28834 - Le processus fils n'est pas encore termine...
PID 28835 - Debut du fils (programme = ./expl41) [PPID = 28834]
PID 28834 - Le processus fils n'est pas encore termine...
^Z
[1]+  Stoppé                 ./expl41 9
piroc@svr24:~$ kill 28835
piroc@svr24:~$ fg
./expl41 9
PID 28834 - Le fils [PID 28835] s'est termine avec le statut 000F
PID 28834 - Fin du Pere
```

Figure 2 - Appel système `waitpid` avec un kill...

Lancer un nouveau programme

Un processus fils créé peut remplacer son code de programme par un autre programme. La famille d'appels système exec permet à un processus de charger et d'exécuter un nouveau programme en lui transmettant les arguments nécessaires.

Après l'exécution d'un appel système de la famille exec, l'image mémoire du processus est écrasée par la nouvelle image mémoire d'un programme exécutable, mais le *pid* ne change pas.

execv	
Librairie	#include < unistd.h>
Prototype	`int execv (const char *application, char *const argv[]);`
Commentaires	Cette fonction active l'exécutable `application` à la place du processus courant. Les paramètres éventuels sont transmis sous forme d'un tableau de pointeurs sur des chaines de caractères, le dernier étant NULL.

Si le programme (exécutable) précédent est nommé ex, et que celui-ci se trouve dans le répertoire courant, l'appel à execv permet d'exécuter ce programme comme étant un processus indépendant.

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

int main (int argc, char *argv[]) {
   char * args[] = {"ex", "3", NULL};

   execv("ex", args);
   // On ne doit pas passer ici puisque normalement l'execv remplace le programme
   perror("execv") ;
   exit(EXIT_FAILURE);
}
```

L'exécution de ce programme donne la sortie propose en Figure 3.

```
piroc@svr24:~$ gcc expl2.c -o testExecv
piroc@svr24:~$ ./testExecv
PID 28907 - Debut du Pere
PID 28907 - Le processus fils n'est pas encore termine...
PID 28908 - Debut du fils (programme = ex) [PPID = 28907]
PID 28907 - Le processus fils n'est pas encore termine...
PID 28907 - Le processus fils n'est pas encore termine...
PID 28907 - Le processus fils n'est pas encore termine...
PID 28908 - Fin du fils
PID 28907 - Le fils [PID 28908] s'est termine avec le statut 0000
PID 28907 - Fin du Pere
```

Figure 3 - Appel système execv

Exemple de résolution d'un exercice

Avant de regarder la solution, essayez de la construire par vous-même !

Énoncé

Ecrire un programme qui reçoit un nombre compris entre 1 et 9 (inclus) en paramètre. Ce nombre est le nombre d'appels récursifs du processus. D'où, si le nombre est inférieur ou égal à 0, le processus se termine normalement (`EXIT_SUCCES`) ; si le nombre est positif, le processus lance un nouveau programme (dont le code est identique) en lui passant comme nouveau paramètre le nombre d'appels diminué de 1.

Fonctions qui pourraient être utiles…

Ces fonctions sont de la même famille que le `printf` vu au bloc 1.

printf, fprintf	
Librairie	#include <stdio.h>
Prototype	`int printf(const char *format, ...);` `int fprintf(FILE *stream, const char *format, ...);`
Commentaires	Permet d'écrire sur le flux `stream` en respectant le `format` précisé. En réalité, la fonction `printf` est très proche de celle-ci au détail près que, dans le cas du `printf`, le flux est imposé : il s'agit de `stdout`.

sprintf	
Librairie	#include <stdio.h>
Prototype	`int sprintf(char *buffer, const char *format, ...);`
Commentaires	Permet de convertir des données en mémoire par transformation en chaine de caractère. En réalité, cette fonction est très proche de `printf` au détail près qu'elle écrit dans un `buffer` plutôt que dans `stdout`.

Quelques pistes…

Dans cet exemple, la valeur de argc doit être 2 : le premier élément du tableau pointe sur une chaine de caractère contenant le nom du programme et le second pointe sur une chaine de caractère contenant le nombre d'appels récursifs à effectuer.

La première chose à faire est donc de s'assurer que `argc` contient 2. Ensuite il s'agit de vérifier si le deuxième élément de `argv` est un "entier compris entre 1 et 9 inclus" ! Comme il s'agit d'une chaine de caractères, il faudra d'abord transformer cette chaine en entier grâce à la fonction `atoi`. Pour chacun de ces tests, en cas d'erreur, on envoie un message sur `stderr` via `fprintf`.

Ces vérifications faites, il faut tester si on a atteint le *cas de base* de la récursivité, c'est-à-dire un nombre d'appels égal à 0 ! Il reste ensuite à traiter le *cas de propagation*, c'est-à-dire préparer l'utilisation de l'appel système `execv` pour lancer un nouveau processus. `execv` nécessite deux paramètres, le *pathname* de l'exécutable qui va prendre la place du processus courant et les éventuels paramètres sous la forme d'un tableau de pointeur sur des chaines de caractères, ici le nombre d'appels. Pour rappel, le dernier élément du tableau sera NULL.

Une solution…

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

int main (int argc, char *argv[]) {
   int mypid;
   int nbAppels;
   char sNbAppels[8];
   char execPath[64];
   // Déclaration du tableau args qui va être utilise pour execv
   char *args[] = {argv[0], sNbAppels, NULL };

   mypid = getpid();
   printf ("PID %d - Debut de %s\n", mypid, argv[0]);

   if (argc != 2) {
      fprintf(stderr, "Nombre de parametre incorrect ! "
                      "Usage : %s nbAppels (0 < nbAppels < 10)\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   nbAppels = atoi(argv[1]);
   if (nbAppels > 10) {
      fprintf(stderr, "Valeur de nbAppels (%s) incorrect ! "
                      "Usage : %s nbAppels (0 < nbAppels < 10)\n", argv[1], argv[0]);
      exit(EXIT_FAILURE);
   }
   // Tester s'il y a encore de appels à effectuer...
   if (nbAppels > 0) {
      fprintf(stdout, "Appel numero %d\n", nbAppels);
      //compléter le pathname du programme
      sprintf(execPath, "%s", argv[0]);
      // Modifier le nombre d'appels
      nbAppels --;
      // Transformer le nombre d'appels en chaine de caractères
      sprintf(sNbAppels, "%d", nbAppels);
      // Lancer le nouveau programme
      execv(execPath, args);
      // On ne doit pas passer ici puisque normalement l'execv remplace le programme
      perror("Probleme avec l'execv");
      exit(EXIT_FAILURE);
   }
   printf ("PID %d - Fin de %s\n", mypid, argv[0]);
   exit(EXIT_SUCCESS);
}
```

L'exécution du programme produit la sortie présentée en Figure 4.

```
piroc@svr24:~$ ./ex
PID 3356 - Debut de ./ex
Nombre de parametre incorrect ! Usage : ./ex nbAppels (0 < nbAppels < 10)
piroc@svr24:~$ ./ex 11
PID 3374 - Debut de ./ex
Valeur de nbAppels (11) incorrect ! Usage : ./ex nbAppels (0 < nbAppels < 10)
piroc@svr24:~$ ./ex 4
PID 3381 - Debut de ./ex
Appel numero 4
PID 3381 - Debut de ./ex
Appel numero 3
PID 3381 - Debut de ./ex
Appel numero 2
PID 3381 - Debut de ./ex
Appel numero 1
PID 3381 - Debut de ./ex
PID 3381 - Fin de ./ex
```

Figure 4 - Exécution de execv...

Rediriger la sortie standard STDOUT vers un fichier ouvert...

Le programme proposé ci-après implémente l'équivalent de `system("ls > ls.res")`. Bien sûr ce programme ne fait pas appel à la fonction `system`. Le but est d'utiliser d'autres fonctions dont `execv`.

Il faut donc créer un processus fils dont le code sera remplacé par celui réalisant la commande `ls`. L'exécutable de cette commande est nommé `ls` et se trouve dans le répertoire `/bin`.

Il faut également faire en sorte que le descripteur de fichier[1] (*file descriptor*) associé au standard de sortie `stdout` (1 selon le standard POSIX) devienne le descripteur du fichier `ls.res`. Pour ce faire, il est nécessaire d'utiliser les appels système open, `close` et dup.

En voici les descriptions :

open	
Librairie	#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>
Prototype	`int open(const char *pathname, int flags);` `int open(const char *pathname, int flags, mode_t mode);`
Commentaires	Essaye d'ouvrir un fichier et retourne un descripteur de fichier (petit entier non négatif à utiliser avec read, write...). `flags` est l'un des éléments `O_RDONLY`, `O_WRONLY` ou `O_RDWR` qui réclament respectivement l'ouverture du fichier en lecture seule, écriture seule, ou lecture/écriture. `flags` peut aussi être un OU binaire (|) entre plusieurs des éléments suivants : `O_CREA`, `O_EXCL`, `O_APPEND`… `mode` indique les permissions à utiliser si un nouveau fichier est créé. Les constantes symboliques suivantes sont disponibles pour `mode` : `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRWXU`, `S_IRGRP`…

[1] Un descripteur de fichier est un entier qui fait référence à une instance donnée d'un fichier ouvert au sein d'un processus.

close	
Librairie	#include <unistd.h>
Prototype	`int close(int fd);`
Commentaires	Ferme le descripteur fd, de manière à ce qu'il ne référence plus aucun fichier, et puisse être réutilisé. Si `fd` est la dernière copie d'un descripteur de fichier donné, les ressources qui lui sont associées sont libérées. Renvoie 0 s'il réussit, -1 sinon, auquel cas *errno* contient le code d'erreur.

dup	
Librairie	#include <unistd.h>
Prototype	`int dup(int oldfd);` `int dup2(int oldfd, int newfd);`
Commentaires	Créent une copie du descripteur de fichier `oldfd`. L'ancien et le nouveau descripteur peuvent être utilisés de manière interchangeable. Ils partagent les verrous, les pointeurs de position et les drapeaux. Par exemple si le pointeur de position est modifié en utilisant `lseek` sur l'un des descripteurs, la position est également changée pour l'autre. Les deux descripteurs ne partagent toutefois pas le drapeau `Close-on-exec`. dup utilise le plus petit numéro inutilisé pour le nouveau descripteur. `dup2` transforme `newfd` en une copie de `oldfd`, fermant auparavant `newfd` si besoin est.

REMARQUE

Si le open pose problème, lancer la commande suivante afin de voir si on a les permissions sur le fichier `ls.res`.

```
$ ls -l
```

Si les droits pour le fichier ne sont pas `-rw-XX-XX-`, donner les droits en lecture et écriture à l'utilisateur via la commande suivante :

```
$ chmod u+w ls.res
```

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

int myPid;

void fils(void) {
   int res;
   // Déclaration du tableau args qui va etre utilise pour execv
   // Le premier argument est le nom de l'exécutable : ls
   char *args[] = {"ls", NULL };
   int fd;
   // Ouvrir le fichier
   fd = open("ls.res", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
   if (fd < 0) {
      perror("open");
      exit (EXIT_FAILURE);
   }
   // Fermer stdout
   close(1);
   // Associer le fichier ouvert avec le premier fd disponible (copie)
   // Comme on vient de libérer le stdout, c'est lui qui est attribué
   res = dup(fd);
   if (res != 1) {
      // stdin (fd = 0) était aussi fermé...
      fprintf(stderr,"dup n'a pas mis le fd dans l'entree 1\n");
      exit (EXIT_FAILURE);
   }
   // Fermer le fd de l'ouverture originale
   close(fd);
   // Lancer la commande ls avec le nouveau fichier comme sortie
   execv("/bin/ls", args);
   // On ne doit pas passer ici puisque normalement l'execv remplace le programme
   perror("execv");
   exit(EXIT_FAILURE);
}

int main (void) {
   int pid;
   int finishedPid;
   int status;

   myPid = getpid();
   printf("Debut du Pere (PID = %d)\n", myPid);
   if ((pid = fork()) < 0) {
      perror("Erreur lors de la creation (fork) du Fils");
      exit(EXIT_FAILURE);
   }
   if (pid == 0) {
      fils();
   }
   // Suite du père.
   printf("Le Pere (PID = %d) attend son fils (PID = %d)\n", myPid, pid);

   finishedPid = wait(&status);

   printf("Le Fils (PID = %d) s'est termine avec le statut %04.4X\n",
            finishedPid, status);
   printf("Fin du Pere (PID = %d)\n", myPid);
   exit(EXIT_SUCCESS);
}
```

À vous de trouver !

1. À la figure 4, expliquez pourquoi on ne retrouve qu'une seule fois le message de fin de processus ?

2. Sur base de l'exemple précédent, réaliser le programme qui implémente l'équivalent de `system("cat < fichier.dat")`.