Master SII *Série 1 : Les processus(rappel)*

Création d'un processus sous Unix, réalisée par l'appel système fork :

• duplication des zones mémoires attribuées au père, le père et le fils ne partagent pas de mémoire,

• le fils hérite de l'environnement fichiers de son père, les fichiers déjà ouverts par le père sont vus par le fils. Ces fichiers sont partagés.

Juste après l'exécution de fork, la seule information qui diffère dans les espaces mémoire attribués au père et au fils est le contenu de la case contenant la valeur de retour de fork (ici f).

Cette case contient le pid du fils dans l'espace mémoire du père et contient zéro dans l'espace mémoire du fils

```
int main ()
{
	int f = fork ();
	if (f == -1){
		printf("Erreur : le processus ne peut etre cree\n")
		exit (1);
		}
	if (f == 0){
		printf("Coucou, ici processus fils\n")
		...;
	}

	if (f != 0){
		printf("Ici processus pere\n")
		...;
	}
}
```

Code de sortie d'un processus

Chaque processus a un code de sortie (**Exit Code**) : un nombre que le processus renvoie à son parent. Le code de sortie est l'argument passé à la fonction **exit** ou la valeur retournée depuis **main**. Par convention, le code de sortie est utilisé pour indiquer si le programme s'est exécuté correctement :

– Un code de sortie à **zéro** indique une **exécution correcte**,

– Un code **différent de zéro** indique qu'une **erreur** est survenue.

Synchronisation père-fils

Lorsqu'un processus fils se termine, il devient un processus **zombie**. Un processus zombie ne peut plus s'exécuter, mais consomme encore des ressources (son entrée dans la table des processus n'est pas enlevée). Il restera dans cet état jusqu'à ce que son processus père ait récupéré son code de sortie. Les appels systèmes **wait** et **waitpid** permettent au processus père de récupérer ce code. A ce moment le processus termine et disparaît complètement du système.

- L'appel système **wait()**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int* status);
```

L'appel système **wait** suspend l'exécution du processus appelant jusqu'à ce qu'un de ses fils se termine. Si un fils est déjà terminé, **wait** renvoie le PID du fils immédiatement sans bloquer. Il retourne l'identifiant du processus fils et son état de terminaison dans **status** (si **status** est différent de **NULL**). Si par contre le processus appelant ne possède **aucun fils**, **wait** retourne **-1**.

```
pid_t pid=fork();
switch(pid){
      case -1: /* Erreur de création */
               ...
      case 0: /* Programme du fils */
              printf("processus fils %d\n", getpid());
              exit(10);
      default: /* Programme du père */
              printf("processus père %d\n", getpid()) ;
              pid_t id = wait (&status);
              printf("fin processus fils %d\n", id);
              exit(0);
}
```

- L'appel système **waitpid()**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid , int* status , int options);
```

L'appel système **waitpid()** permet à un processus père d'attendre un fils particulier d'identité **pid**, de façon **bloquante** (**options=0**) ou **non bloquante** (**options=WNOHANG**).

Si l'appel réussit, il renvoie l'identifiant du processus fils et son état de terminaison dans **status** (si **status** n'est pas NULL). En cas d'erreur **-1** est renvoyé. Si l'option WNOHANG est utilisée et aucun fils n'a changé d'état, la valeur de retour est **0.**

```
pid_t id;
pid_t pid=fork();
switch(pid){
   case -1: /* Erreur de création */
              ...
   case 0: /* Programme du fils */
              ...;
              exit(10);
   default: /* Programme du père */
           ...
           While((id=waitpid(pid,&status,WNOHANG))==0)
             printf("processus fils non encore terminé\n");
           printf("fin processus fils %d\n", id) ;
            ...
}
```

L'appel **waitpid(-1, &status, 0)** est équivalent à l'appel **wait(&status)**

– **Code de sortie retourné** par wait() et waitpid()

Pour extraire le code de retour du processus fils de **status**, il faut d'abord tester que le processus c'est terminé normalement. Pour cela, on utilise la macro **WIFEXITED(status)**, qui renvoie **vrai** si le processus fils c'est terminé normalement.

Ensuite pour obtenir son code de retour, on utilise la macro **WEXITSTATUS(status)**.

```
pid_t id;
pid_t pid=fork();
switch(pid){
  case -1: /* Erreur de création */
              ...
  case 0: /* Programme du fils */
              ...;
              exit(10);
  default: /* Programme du père */
       ...
       while((id=waitpid(pid,&status,WNOHANG))==0)
          printf("processus fils non encore terminé\n");
       if(WIFEXITED(status))
       printf("fils %d terminé par exit(%d)\n",id,WEXITSTATUS(status));
       ...
}
```

Solution Exercice02:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<wait.h>
/*
// sans renvoyer la valeur de retou
void main()
{
int pid1, i,status;

pid1 = fork ( ) ;
if (pid1 == -1)
     {  printf ( " le fork ( ) a échoué \n " );

if (pid1 == 0)
     { //code correspondant à l'exécution du processus fils

      printf("FILS1_id= %d et le pid_pere est =%d\n", getpid(), getppid());
      exit(0);
     }
else
     {
 // code correspondant à l'exécution du processus père
       int pid2=fork();
       if (pid2 == -1)
     { // code si échec :
           printf ( " le fork ( ) a échoué \n " ) ;}
```

```
       if (pid2 == 0)
     { // code correspondant à l'exécution du processus fils

      printf("FILS2_id= %d et le pid_pere est =%d\n", getpid(), getppid());
         exit(0);
     }
       else
          {  printf("PERE_id  est  %d  et  les  pid  de  mes  fils  sont  %d
%d\n",getpid(), pid1,pid2);

           while (wait(NULL)!=-1) ;//tant que le père a encore des fils
           printf("Les processus fils se sont terminés\n");
          }
      }
}

*/
// les deux fils renvoient leurs valeurs de retour

void main()
{
int pid1, i,pid[2],status[2];

pid1 = fork ( ) ;
if (pid1 == -1)
     { // code si échec : printf ( ” le fork ( ) a échoué \n ” )
        }
if (pid1 == 0)
     { // code correspondant à l'exécution du processus fils
     int x=5;
     printf("FILS1_id= %d et le pid_pere est =%d\n", getpid(), getppid());
     exit(x);
     }
else
     {
 // code correspondant à l'exécution du processus père
  int pid2=fork();
  if (pid2 == -1)
     { // code si échec :
           printf ( " le fork ( ) a échoué \n " ) ;}
  if (pid2 == 0)
     { // code correspondant à l'exécution du processus fils
     int y=6;
     printf("FILS2_id= %d et le pid_pere est =%d\n", getpid(), getppid());
     exit(y);
     }
   else
     {printf("PERE_id est %d et les pid de mes fils sont %d %d\n",getpid(),
pid1,pid2);
      i=0;
      while ((pid[i]=wait(&status[i]))!=-1) { //tant que le père a encore
des fils
     if(WIFEXITED(status[i]))
     printf("Le processus fils ayant le PID %d s'est terminé avec le code
%d.\n", pid[i], WEXITSTATUS(status[i]));
     i++;
```

```
        }
      }
    }
}
```

execlp

NOM

execl, execlp, execle, execv, execvp, execvpe - Exécuter un fichier

int execlp(const char *file, const char *arg, ...);

La famille des fonctions **exec**() remplace l'image du processus en cours par une nouvelle image du processus.

Les arguments *const char *arg* ainsi que les points de suspension des fonctions execl(), execlp(), et execle() peuvent être vues comme *arg0*, *arg1*, ..., *argn*. Ensemble, ils décrivent une liste d'un ou plusieurs pointeurs sur des chaînes de caractères terminées par des caractères nuls, qui constituent les arguments disponibles pour le programme à exécuter. Par convention, le premier argument doit pointer sur le nom du fichier associé au programme à exécuter. La liste des arguments *doit* se terminer par un pointeur NULL

Sémantique particulière pour execlp() et execvp()

Les fonctions **execlp**() et **execvp**() dupliqueront les actions de l'interpréteur de commandes dans la recherche du fichier exécutable si le nom fourni ne contient pas de barre oblique « / ». Le chemin de recherche est spécifié dans la variable d'environnement **PATH**. Si cette variable n'est pas définie, le chemin par défaut sera « /bin:/usr/bin: ». De plus, certaines erreurs sont traitées de manière spécifique.

```
#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>
```

```
int main()
{execlp("ls","ls", NULL);
//execlp(chemin , nom, parametre1,...,NULL);
//system("ls");
printf("Bonjour");
return 0;
}
```