Notes sur la gestion des processus Unix

Licence 3^{ieme} année, Bruno Jacob

Octobre 2004

1 Introduction

Ces notes portent sur les primitives Unix System V pour créer et synchroniser des processus.

2 Primitives générales

2.1 Identification des processus

Les processus sont repérés par le système par un identificateur de processus (`pid`). Sous System V, c'est un entier long (4 octets). Les pids les plus généralement utilisés sont fournis par les fonctions suivantes :

```
#include <unistd.h>

pid_t getpid(void);  /* processus courant */
pid_t getppid(void); /* processus pere    */
```

Exemple

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

int
main()
{
        printf( "Processus %ld, de pere %ld, taille d'un pid = %d octets\n" ,
                getpid() , getppid() , sizeof(pid_t));

        return(0);
}
```

2.2 Mise en sommeil des processus

Il existe diverses solutions pour mettre en sommeil un processus. Les foncion suivantes permettent de faire cela facilement :

```
#include <unistd.h>

unsigned int sleep(unsigned int s);
unsigned int alarm(unsigned int s);
int pause(void);
```

sleep : la plus simple ; suspend le processus pendant `s` secondes

alarm : envoie le signal `SIGALRM` au processus au bout de `s` secondes, mais ne suspend pas le processus

pause : sommeil jusqu'à un signal

On peut également mettre à profit le cours sur ls signaux Unix en utilisant les signaux `SIGSTOP` et `SIGCONT`. Le code ci après donne un exemple d'utilisation de ces différentes méthodes.

Exemple

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

static
void hand_alrm( int sig )
{
      printf( "Signal ARLM recu\n");

}

int main( int nbarc , char * argv[] )
{
      printf( "Debut Processus %s , pid = %ld\n" , argv[0] , getpid() );
      signal( SIGALRM , hand_alrm );

      printf(" Sommeil avec sleep pendant 5 secondes\n");
      sleep(5);
      printf(" Envoie de SIGALRM dans 5 secondes\n");
      alarm(5) ;
      printf(" Sommeil avec pause jusqu'au signal SIGALRM\n");
      pause();
      printf(" Sommeil en s\'envoyant le signal SIGSTOP\n");
      kill( getpid() , SIGSTOP );
```

```
        printf( "Fin Processus %s , pid = %ld\n" , argv[0] , getpid() );
        exit(0);
}
```

3 Création de processus

Elle est réalisée par la primitive `fork`, qui permet de créer **dynamiquement** un nouveau processus (le *processus fils*) qui s'exécute en parallèle avec celui qui l'a créé (le *processus père*).

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

pid_t fork(void);
```

On peut visualiser l'exécution du code comme une *fourche* à 2 dents : jusqu'au `fork` le programme s'exécute linéairement (le manche), et ensuite dans 2 branches parallèles (les 2 dents de la fourche). Le processus fils est une *copie conforme* du processus père :

- même code
- même zone de donnée
- même environnement
- même priorité
- même descripteurs de fichiers courants
- même traitement des signaux

Le seul moyen de distinguer le processus fils du processus père est **la valeur de retour du fork** :

- $\rightarrow$ 0 dans le processus fils
- $\rightarrow$ pid du fils nouvellement créé dans le processus père

Si `fork` échoue, alors elle renvoie la valeur `-1`

Exemple

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

static
void hand_alrm( int sig )
{
        printf( "Signal ARLM recu\n");
```

```
}

int main( int nbarc , char * argv[] )
{
      printf( "Debut Processus %s , pid = %ld\n" , argv[0] , getpid() );
      signal( SIGALRM , hand_alrm );

      printf("Sommeil avec sleep pendant 5 secondes\n");
      sleep(5);
      printf("Envoie de SIGALRM dans 5 secondes\n");
      alarm(5) ;
      printf("Sommeil avec pause jusqu'au signal SIGALRM\n");
      pause();
      printf("Sommeil en s'envoyant le signal SIGSTOP\n");
      kill( getpid() , SIGSTOP );

      printf( "Fin Processus %s , pid = %ld\n" , argv[0] , getpid() );
      exit(0);
}
```

4 Synchronisation des processus

4.1 Attente de la terminaison d'un processus fils

Elle est réalisée par la primitive `wait` qui suspend l'exécution du processus appelant jusqu'à ce que l'un de ses processus fils se termine..

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

pid_t wait(int *cr);
```

- Si un processus fils s'est terminé avant l'exécution du `wait` alors le retour de la fonction est immédiat.
- Si il n'y a plus de processus fils alors `wait` renvoie `-1`
- Si `wait` est débloquée par un processus fils alors la fonction renvoie le pid de ce processus fils

Si `wait` renvoie une valeur $\neq 0$ alors on peut récupérer dans `cr` le code de retour (le *exit status*) du fils. Ceci permet de savoir si le fils s'est bien terminé ou non. Le code retour est un entier sur 16 bits. Cet entier est séparé en 2

- les 8 bits de poids forts
- les 8 bits de poids faibles

4.2 Terminaison normale d'un fils

Si un fils s'st terminé normalement, c'est à dire si il s'est terminé par `exit(valeur)`, alors les 8 bits de poids faibles de `valeur` sont affectés au bits bits de poids forts de `cr`. Les autres bits de `cr` sont à 0. La figure FIG. 1 illustre cette affectation.

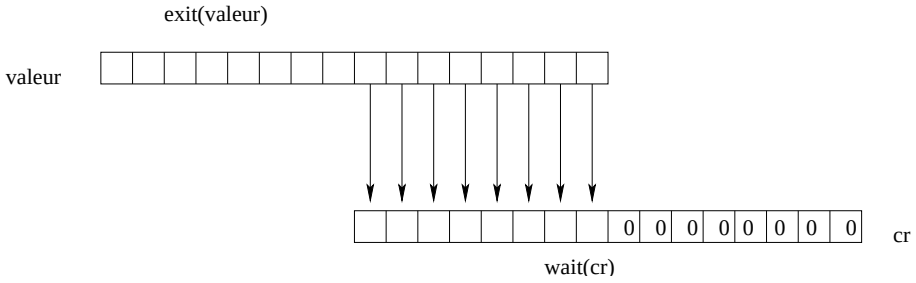

FIG. 1 – Affectation du code retour en cas de succès

4.3 Terminaison anormale d'un fils

En général, si un processus fils se termine anormalement, alors cela veut dire qu'il a reçu un signal. Le numéro de ce signal est affecté dans les 8 bits de poids faibles de `cr`.
Si le handler de ce signal crée une image mémoire (un fichier *core*) alors on ajoute la valeur décimale `128` au numéro de ce signal. La figure FIG. 2 illustre cette affectation.

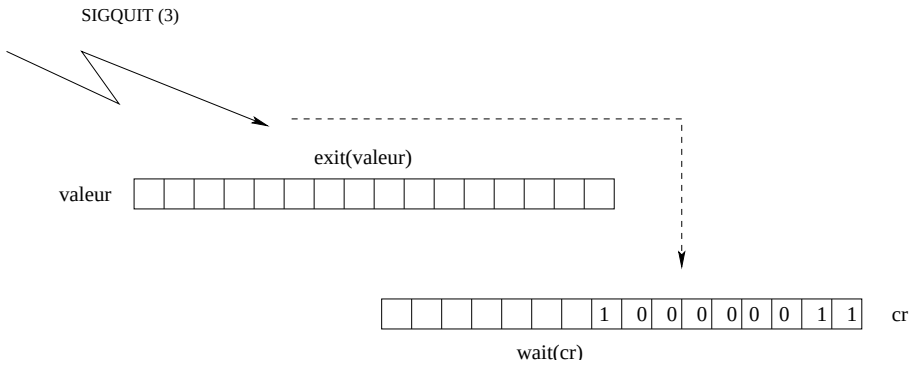

FIG. 2 – Affectation du code retour en cas d'échec

4.4 Exemple

```
#include <stdio.h>      /* fprintf */
#include <sys/types.h> /* fork, wait  */
#include <unistd.h>
#include <stdlib.h>     /* exit */
#include <sys/wait.h>  /* wait */

int
main()
{
  pid_t pid ;
  int cr ;

  switch(pid=fork())
  {
     case -1 : /* Erreur */
       fprintf( stderr , "Echec du fork\n");
       exit(-1);
       break ;
     case 0 : /* Processus fils */
       fprintf( stdout , "Fils : processus %ld\n" , getpid() );
       while(1) ;
       break ;
     default : /* Processus pere */
       fprintf( stdout , "Pere : processus %ld\n" , getpid() );
       pid=wait(&cr) ;
       printf(" Fin du processus fils %ld , code  retour = %d\n",
              pid , cr ) ;
       break ;
  }
     return(0);
}
```

4.4.1 Exemple avec *core*

On exécute ici le un handler par défaut du signal `SIGQUIT`, qui crée un *core*

```
$ ex_wait &
[5] 10420

$ Pere : processus 10420
  Fils : processus 10421

$ kill -3 10421
  Fin du processus fils 10421 , code  retour = 131
[5]    Done                          ex_wait
```

4.4.2 Exécution sans *core*

Ici on utilise le signal `SIGKILL` qui ne crée pas d'image mémoire *core*

```
$ ex_wait &
[5] 10524
$ Pere : processus 10524
  Fils : processus 10525

$ kill -9 10525
  Fin du processus fils 10525 , code  retour = 9
[5]    Done                          ex_wait
```

5 Recouvrement de processus

Le recouvrement (*overlay*) d'un processus représente comme un changement de vie d'un processus, dans le sens où le code qu'il est en train d'exécuter est changé pour un autre. Le recouvrement du code d'un processus par le code d'un autre programme (le programme lancé) est réalisé par la famille des commandes `exec`.

Retour de `exec`

- Pas de retour en cas de succès : une commande `exec` remplaçant le code existant par un autre, l'appel de `exec` est donc effacé
- -1 en cas d'échec.

La famille se compose des fonctions suivantes

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., const
char *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);

int execle(const char *path, const char *arg0, ..., const
char *argn, char * /*NULL*/, char *const envp[]);

int execve(const char *path, char *const argv[], char *const
envp[]);

int execlp(const char *file, const char *arg0, ..., const
char *argn, char * /*NULL*/);

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

La famille est divisée en 2 clans

les execl le nombre de paramètres du programme lancé est connus. Ils sont placés dans les `arg0 ...argn`

les execv le nombre de paramètres est inconnu *a priori*. C'est le tableau `argv` qui les contient (le dernier paramètre sera suivi de `NULL`)

A l'interieur de ces clans, une lettre précise le type d'appel :

(rien) (`execl, execv`) : la référence du programme lancé doit être explicite (nom complet depuis le répertoire courant d'où l'on exécute le `exec`)

e (`execle, execve`) : idem à `execl` et `execv` mais avec un tableau de variables d'environnement `envp` qui est transmis au programme lancé

p (`execlp, execvp`) : idem à `execl` et `execv` mais avec recherche du programme lancé dans les chemins connus par le système (contenus dans `$PATH`

Remarque sur les signaux dans l'utlisation d'un `exec`
- les signaux ignorés par le père restent ignorés
- pour les autres signaux le comportement par défaut est réadopté

5.1 Exemple

Voici un exemple des différents appels du clan `execv` :

```
#include <stdio.h>      /* fprintf */
#include <sys/types.h> /* fork, wait  */
#include <unistd.h>     /* exec */
#include <stdlib.h>     /* exit */
#include <sys/wait.h>  /* wait */
#include <strings.h>   /* strncmp , strcat */

int main( int nb_arg  , char * tab_arg[] , char * env[] )
{
     pid_t pid ;          /* pid du processus fils */
     int cr ;             /* Code Retour */
     int nb_env ;         /* Nombre de variable dans env */
     int type   ;         /* Type de exec a tester */

     /*----------*/

     if( nb_arg < 2 )
     {
          fprintf( stderr , "usage : %s <type exec> <fichier1> <fichier2 > ...\n" ,
                   tab_arg[0] );
          fprintf( stderr , " avec <type exec> = 0 ---> execv\n" );
          fprintf( stderr , "                  = 1 ---> execve\n" );
          fprintf( stderr , "                  = 2 ---> execvp\n" );
```

```
      exit(-1);
}
sscanf( tab_arg[1] , "%d" , &type ) ;

switch(pid=fork())
{
   case -1 : /* Erreur */
        fprintf( stderr , "Echec du fork\n");
        exit(-1);
        break ;
   case 0 : /* Processus fils */
        {
             char * params[256] ;
             int i ;

             fprintf( stdout , "Fils : processus %ld\n" , getpid() );

             /* Creation du tableau de parametres de exec */
             params[0] = "ls" ;
             for( i=1 ; tab_arg[i+1] != NULL ; i++)
                  params[i] = tab_arg[i+1] ;
             params[i] = NULL ;

             /* Aiguillage selon le type d'exec a tester */
             switch( type )
             {
                case 0 : /* test de execv */
                  {
                    printf("\n\ntest de EXECV\n\n");
                    if( execv( "/usr/bin/ls" , params ) == -1 )
                    {
                      perror( "Echec du lancement par execv" );
                      exit(-2);
                    }
                  }
                case 1 : /* test de execve */
                  {
                    printf("\n\ntest de EXECVE\n\n");

                    /* Creation d'une variable d'environnement COLUMNS
                     * COLUMNS==largeur de colonne
                     * cette variable est utlisee par "ls"
                     */
                    env[nb_env] = "COLUMNS=30" ;
                    env[nb_env+1] = 0 ;

                     if( execve("/usr/bin/ls" , params , env ) == -1 )
                     {
                          perror( "Echec du lancement par execve" );
```

```
                        exit(-3);
                    }
                }
                case 2 : /* test de execvp */
                {
                printf("\n\ntest de EXECVP\n\n");

                    if( execvp( "ls" , params ) == -1 )
                    {
                        perror( "Echec du lancement par execvp" );
                        exit(-4);
                    }
                }
                default : /* type inconnu */
                {
                  fprintf( stderr,"Type de exec inconnu = %d\n",type);
                  break ;
                }
            }
        }
    default : /* Processus pere */
      {
        fprintf( stdout , "Pere : processus %ld\n" , getpid() );
        pid=wait(&cr) ;
        printf(" Fin du processus fils %ld , code  retour = %d\n",
               pid , cr ) ;
        break ;
      }
  }
  exit(0);
}
```

5.2 Utilisation

En général, les appels à la famille `exec` sont utilisés après un `fork` pour créer dans l'un des 2 processus une nouvelle image mémoire différente. Cette création est en 2 temps

1. Création de la copie de l'image du programme courant
2. Recouvrement de cette copie