IUP MIAGE
Système d'exploitation

Le système de gestion des processus sous Unix

Philippe MARQUET

1 Processus

Un processus est un programme s'exécutant.

Différentes ressources sont associées à un processus.

En particulier une zone mémoire est allouée au processus (figure ci-contre).

Un processus est identifié de manière non ambiguë par un entier (PID, *process id*).

Unix est un système multi-processus. Unix est un système temps partagé : on fait « avancer » les processus *prêts* concurremment ; on a une impression de parallélisme ; cette activité est nommée « ordonnancement des processus. »

Les processus sont organisés en un arbre de processus. Un processus particulier (`init`, de PID 1) est la racine de cet arbre.

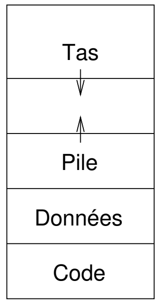

FIG. 1 – Organisation de la mémoire allouée à un processus

2 Manipulation de processus

Les opérations de base de gestion de processus sont :
- création de processus, `fork ();`
- identification d'un processus, `getpid ()` et `getppid ();`
- terminaison d'un processus, `exit ();`
- attente de la terminaison d'une processus fils : `wait ();`
- remplacement d'un processus, `execve ()` et dérivées.

3 Création de processus

L'appel système

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

pid_t fork (void) ;
```

crée un nouveau processus (processus fils). Celui ci est une copie exacte du processus appelant.

`fork ()` retourne
- 0 dans le processus fils ;
- le PID du fils dans le processus père (-1 en cas d'erreur).

L'utilisation de `fork ()` est illustré figure 2

Id: processus1.tex,v 1.15 2001/07/06 12:00:20 marquet Exp

```
% cat creer_procesus.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void) {
  pid_t pid ;

  pid = fork () ;

  if (pid == -1) {
    perror ("creation processus impossible") ;
    exit (1) ;
  }

  if (pid != 0)
    printf ("Je suis le pere, j'ai cree le processus %d\n", pid) ;
  else
    printf ("Je suis le processus fils\n") ;
}

% ./creer_processus
Je suis le pere, j'ai cree le processus 14602
Je suis le processus fils
```

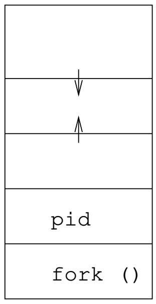

FIG. 2 – Utilisation de la primitive `fork ()` et état de la mémoire des processus suite à l'appel `fork()`

4 Terminaison du processus courant

Le processus courant se termine automatiquement lorsqu'il cesse d'exécuter la fonction `main ()`. Les primitives suivantes lui permettent d'arrêter explicitement son exécution :

```
#include <unistd.h>

void _exit (int status) ;
void exit (int status) ;
```

Ces deux primitives provoquent la terminaison du processus courant. Le paramètre `status` spécifie un code de retour, compris entre 0 et 255, à communiquer au processus père.

Par convention, en cas de terminaison normale, un processus doit retourner la valeur 0.

Avant de terminer l'exécution du processus, `exit ()` exécute les fonctions de « nettoyage » des librairies standard.

Exercice 1
Expliquer le comportement du programme donné à la figure 3.

Exercice 2
Donner un programme qui affiche les entiers de 0 à 3 par 4 processus différents. Ce programme doit se terminer après l'affichage des 4 entiers.

5 Identité d'un processus

Un processus est identifié par son PID. Il peut connaître le PID de son père.

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

int main (void)
{
  int i ;
  for (i=0 ; i < 4 ; i++) {
    int retour ;
    retour = fork () ;
    switch (retour) {
      case -1 : /* erreur */
        perror ("erreur fork\n") ;
        exit (1) ;
      case 0 : /* fils */
        printf ("fils : %d\n", i) ;
      default : /* pere */
        printf ("pere : \n") ;
    }
  }
}
```

FIG. 3 – Que fait ce programme ?

Les deux primitives

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

pid_t getpid(void);
pid_t getppid(void);
```

retournent respectivement le PID du processus et de son père.

Exemple

```
% cat pid.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int
main (void)
{
  pid_t pid_fils ;

  switch (pid_fils = fork ()) {
    case -1 :
      perror ("Fork ") ;
      exit (1) ;
    case 0 : /* le fils */
      printf ("Je suis le fils, PID %d, PPID %d\n",
              getpid (), getppid ()) ;
      exit (0) ;
```

```
    default :  /* le  pere */
      printf ("Je suis le pere, PID %d. Mon fils a le pid %d\n",
              getpid (), pid_fils);
      (void) wait (NULL) ;
      exit (0) ;
  }
}

% ./pid
Je suis le pere, PID 8078. Mon fils a le pid 8079
Je suis le fils, PID 8079, PPID 8078
```

6 Attente de terminaison d'un processus fils

La primitive

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

pid_t wait (int *status) ;
```

suspend l'exécution du processus appelant jusqu'à ce qu'un de ses processus fils se termine. Si un processus fils s'est déjà terminé, `wait ()` retourne le résultat immédiatement.

`wait ()` retourne le PID du processus fils si le retour est dû à la terminaison d'un processus fils ; -1 en cas d'erreur.

Si `status` n'est pas un pointeur nul, le status du processus fils (valeur retournée par `exit ()`) est mémorisé à l'emplacement pointé par `status`.

De manière plus précise :

- l'octet de poids faible est un code indiquant pourquoi le processus fils s'est arrêté ;
- si le processus fils a effectué un appel à `exit ()`, l'octet précédent contient le code de retour.

Ces informations peuvent être accédées facilement à l'aide des macros suivantes définies dans `sys/wait.h`. (Attention, ces macros utilisent le status, et non un pointeur sur ce status.) :

WIFEXITED (status) renvoie vrai si le statut provient d'un processus fils qui s'est terminé normalement ;

WEXITSTATUS (status) (si `WIFEXITED (status)` renvoie vrai) renvoie le code de retour du processus fils passé à `_exit ()` ou `exit ()` ou la valeur retournée par la fonction `main ()` ;

WIFSIGNALED (status) renvoie vrai si le statut provient d'un processus fils qui s'est terminé à cause de la réception d'un signal ;

WTERMSIG (status) (si `WIFSIGNALED (status)` renvoie vrai) renvoie la valeur du signal qui a provoqué la terminaison du processus fils.

D'autres macros sont disponibles ; leur documentation est accessible par la commande `man 2 wait`.

Exemple

```
% cat status_fils.c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
```

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

  pid = fork () ;

  switch (pid) {
    case -1 :
      perror ("fork") ;
      exit (1) ;
    case 0 : /* le fils */
      printf ("processus fils\n") ;
      exit (2) ;
    default : /* le pere */
      printf ("pere: a cree processus %d\n", pid) ;
      wait (&status) ;
      if (WIFEXITED (status))
        printf ("fils termine normalement: status = %d\n",
                WEXITSTATUS (status)) ;
      else
        printf ("fils termine anormalement\n") ;
  }
}
```

```
% ./status_fils
pere: a cree processus 907
processus fils
fils termine normalement: status = 2
```

Exercice 3
Donner un programme qui affiche, dans l'ordre, les entiers de 0 à 3 par 4 processus à partir de la structure suivante ;

```
for (i=0 ; i < 4 ; i++) {
  switch (fork ()) {
    case -1 :
        ...
    case 0 :
        ...
    default :
        ...
  }
```

Exercice 4 (Trifourche)
Écrire la fonction

```
  trifourche (void(*)(void), void(*)(void), void(*)(void)) ;
```

Le processus exécutant `trifourche (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 :

```
void fa (void)
{
```

```
  sleep (4) ;
  printf ("Fonction fa executee par le processus %d\n", getpid ()) ;
}

void fb (void)
{
  sleep (2) ;
  printf ("Fonction fb executee par le processus %d\n", getpid ()) ;
}

void fc (void)
{
  sleep (3) ;
  printf ("Fonction fc executee par le processus %d\n", getpid ()) ;
}

int main (void)
{
  trifourche (fa, fb, fc) ;
  printf ("Terminaison de main ()\n") ;
}
```

Exercice 5 (Multifourche)
La fonction `multifourche ()` généralise la fonction `trifourche ()` précédente.

```
typedef int (*intint ) (int) ;

void multifourche (int n, intint f[]) ;
```

Le type `intint` est défini comme « pointeur sur une fonction retournant d'entier à entier ». Les arguments de `multifourche ()` sont un tableau de telles fonctions et la taille `n` de ce tableau. Chacune des fonctions est exécutée par un processus différent qui affiche aussi le valeur retournée par la fonction.

Exercice 6
Soit la commande `sigma` à deux arguments. `sigma inf sup` affiche la somme des entiers de `inf` à `sup`.

En remarquant que l'on peut calculer indépendamment $\sum_{i=a,\frac{b-a}{2}} i$ et $\sum_{\frac{b-a}{2}+1,b} i$ pour calculer `sigma a b`, développer une version de la commande `sigma` utilisant des processus.

Exercice 7
Soit le programme suivant compilé en un exécutable nommé `partage` :

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

int main (void)
{
  char buf [6] ;
  int fd ;

  fd = open ("foo", O_RDWR, 420) ;

  switch (fork ()) {
```

```
    case -1 :
      perror ("fork") ;
      exit (1) ;
    case 0 :
      printf ("%d caracteres ecrits\n", write (fd, "12345", 5)) ;
      break ;
    default :
      printf ("%d caracteres lus\n", read (fd, buf, 5)) ;
      buf [5] = '\0' ;
      printf ("Ligne lue : %s\n", buf) ;
      close (fd) ;
  }
}
```

Donner le résultat de la séquence de commandes Unix suivante :

```
  % echo ABCDEFGH > foo
  % ./partage
  % cat foo
```

7 Changement de code d'un processus

La primitive

```
#include <unistd.h>
int execve (char *filename, char *argv[], char *envp[]) ;
```

transforme le processus appelant en un nouveau processus. Attention, il n'y a pas création d'un nouveau processus, mais recouvrement de l'ancien processus par le nouveau.

Le nouveau processus est construit à partir d'un fichier ordinaire exécutable dont le nom est pointé par `filename`.

Le paramètre `argv` pointe sur un tableau de pointeurs de chaînes de caractères terminé par un pointeur `NULL`. Ces chaînes constituent la liste d'arguments du nouveau processus. Par convention, `argv[0]` est toujours présent et doit être le nom du programme à exécuter.

Le paramètre `envp` est aussi un pointeur sur un tableau de pointeurs de chaînes de caractères terminé par un pointeur `NULL`. Elles identifient les variables d'environnement qui seront accessibles depuis le nouveau processus. Ces chaînes sont construites suivant le format :

<nom de la variable> = <valeur de la variable>

Les descripteurs ouverts dans le processus appelant restent ouverts dans le nouveau processus.

Quand le programme exécuté commence, il est appelé de la façon suivante : (les paramètres `argc`, `argv` et `envp` de `main ()` sont optionnels)

```
main (int argc, char *argv[], char *envp[])
```

`execve ()` retourne dans le processus appelant uniquement en cas d'erreur et renvoie alors -1.

Exemple

```
% cat exec1.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
```

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

char *path = "./exec2" ;
char *argval[] = {"exec2", "foo", "bar", NULL} ;

int main (int argc, char *argv[], char *envp[])
{
 pid_t pid ;

 switch (retour = fork()) {
   case -1 :
     perror ("fork") ;
     exit (1) ;
   case 0 :
     printf ("processus fils va executer exec\n") ;
     execve (path, argval, envp) ;
     perror ("execve") ;
     exit (1) ;
   default :
     printf ("pere: a cree processus %d\n", retour) ;
     wait(0) ;
     printf ("pere: a recu terminaison fils\n") ;
 }
}

% cat exec2.c
#include <stdio.h>

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

 printf ("execution de exec2\n") ;
 printf ("exec2: %d arguments\n", argc) ;
 for (i=0 ; i<argc ; i++)
   printf ("exec2 argument %d: %s\n", i, argv[i]) ;

 exit (0) ;
}

% ./exec1
processus fils va executer exec
pere: a cree processus 14625
execution de exec2
exec2: 3 arguments
exec2 argument 0: exec2
exec2 argument 0: foo
exec2 argument 0: bar
pere: a recu terminaison fils
```

D'autres primitives sont disponibles; elles sont écrites à l'aide de `execve ()`. Il 'agit de `execl ()`, `execlp ()`, `execle ()`, `execv ()`, et `execvp ()` dont les prototypes sont les suivants :

```
#include <unistd.h>

int execl (const char *filename, const char *arg, ...) ;
int execlp (const char *cmdname, const char *arg, ...) ;
int execle (const char *filename char *arg , ..., char* const envp[]) ;
int execv (const char *filename, char *const argv[]) ;
int execvp (const char *cmdname, char *const argv[]) ;
```

Elles permettent de spécifier les arguments du nouveau processus sous la forme d'une liste d'arguments (paramètres `arg`) et non d'un tableau de chaînes de caractères (paramètres `argv`), et/ou de rechercher le programme à exécuter parmi les chemins de recherches (variable d'environnement `$PATH`, paramètres `cmdname`) et non de spécifier complètement son chemin (paramètres `filename`), etc.

8 Exemple de fonctionnement

Nous illustrons l'utilisation des primitives précédentes sur le fonctionnement lors de l'exécution d'une commande (`commande`) par un shell Unix (`sh`).

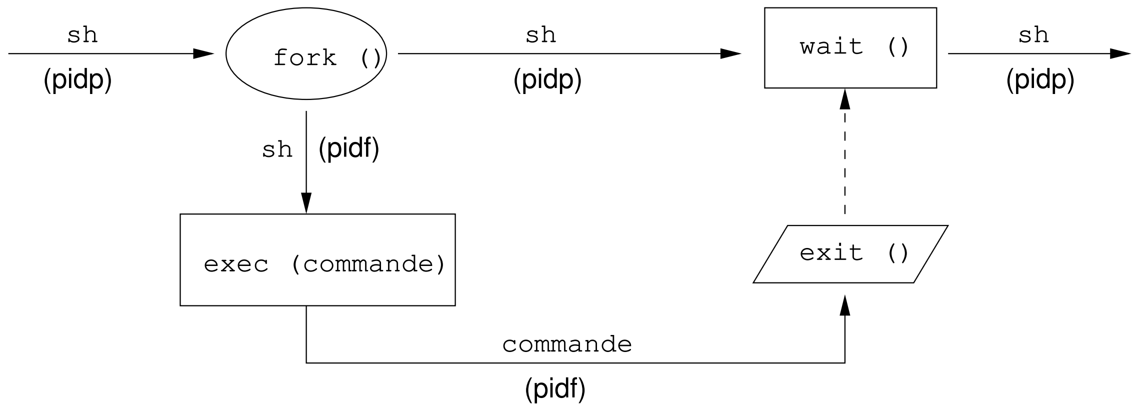

FIG. 4 – Exécution d'une commande par un shell Unix

9 Exercices

Exercice 8

Question 8.1 Au vu du résultat des commandes suivantes, dessiner l'abre des processus d'une commande de la forme `com1 ; com2 ; ... ; comn`.

```
% who ; ps -f www ; date ; cd ; ls -F
phm      tty1     Apr 28 22:26
lambert  tty2     Apr 24 13:34
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
phm        580   566  0 Apr28 pts/1    S      0:00 -csh
phm       1016   580  0 11:24 pts/1    R      0:00 ps -f www
Sat Apr 29 11:24:45 CEST 2000
INBOX                         ens/
auto/                         public_html/
bin/                          tex/
```

```
back/                          tmp/
dm/                            usr/
```

Question 8.2 Écrire un programme C correspondant la commande shell

```
who ; ps -f www ; date ; cd ; ls -F
```

On exécutera une suite d'`exec ()`. La commande `cd` étant une commande interne du shell, on ne peut la lancer directement par un `exec ()`.

Question 8.3 Étudier le résultat de la commande suivante (comparer avec la commande précédente)

```
% who & ps -f www & date & cd & ls -F &
phm      tty1     Apr 28 22:26
lambert  tty2     Apr 24 13:34
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
phm        580   566  0 Apr28 pts/1    S      0:00 -csh
phm       1213   580  0 11:27 pts/1    R      0:00 ps -f www
phm       1214   580  0 11:27 pts/1    S      0:00 date
phm       1216   580  0 11:28 pts/1    S      0:00 ls -F
Sat Apr 29 11:27:35 CEST 2000
ChangeLog                      processus1.log
Makefile                       processus1.make
Readme                         processus1.tex
processus1.aux                 source
processus1.dvi                 tubes.tex
```

et écrire un programme C correspondant aux commandes shell

```
% who & ps -f www & date & cd & ls -F
```

et

```
% who & ps -f www & date & cd & ls -F &
```

Exercice 9
Donner l'implantation sous forme d'un programme C des commandes shell suivantes :

```
tar xf foo.tar gee.txt && lpr gee.txt
```

et

```
tar xf foo.tar gee.txt || mail -s "il y a des erreurs" marquet@lifl.fr
```

Exercice 10
Au vu des résultats des commandes suivantes, dessiner l'arbre des processus d'une commande de la forme `com1 | com2 | ... | comn`.

```
% ps -f www | cat -n | tee foo | wc
      7      74     448
% cat foo
     1  UID        PID  PPID  C STIME TTY      STAT   TIME CMD
     2  phm        563   557  0 07:15 pts/1    S      0:00 -csh
     3  phm        672   563  0 08:29 pts/1    S      0:01 emacs
     4  phm       1169   563  0 11:26 pts/1    R      0:00 ps -f www
     5  phm       1170   563  0 11:26 pts/1    S      0:00 cat -n
     6  phm       1171   563  0 11:26 pts/1    S      0:00 tee foo
     7  phm       1172   563  0 11:26 pts/1    S      0:00 wc
```

Exercice 11
Le programme

```
#include <stdlib.h>

int main (void)
{
  system ("ps") ;
}
```

a été compilé en un exécutable nommé `monps` placé dans votre répertoire d'accueil. La commande shell

```
% monps
```

est tapé dès la connexion. Quelles sont les affirmations suivantes correctes :

1. Après affichage des processus en cours, le programme `monps` termine et le shell vous rend la main.
2. Après affichage des processus en cours, la session se termine et vous êtes déconnecté.
3. Il y a affichage de 2 processus.
4. Il y a affichage de 3 processus.
5. Il y a affichage de 4 processus.
6. Il y a affichage de 5 processus.

Exercice 12
Quelles différences y a-t-il entre les appels système `system ()` et `execve ()` ?

1. Il n'y a aucune différence.
2. `execve ()` permet d'exécuter le même programme plusieurs fois.
3. `execve ()` ne crée pas de nouveau processus.

Exercice 13
Lors de l'exécution d'un processus `shell` interactif de PID p, on lance la commande `ps &`.

1. Le processus p se transforme en un processus qui exécute `ps`.
2. Le processus p crée un fils f qui exécute `ps` et attend la terminaison de p avant de terminer lui-même.
3. Le processus p crée un fils f qui exécute `ps`, et p attend la terminaison de ce processus f avant de reprendre son exécution normale.
4. Le processus p crée un fils f qui exécute `ps`, et reprend son exécution sans attendre la terminaison de f.

Exercice 14
Vous connaissez déjà l'appel système `fork ()` pour créer un processus. Il en existe un autre, `vfork ()`, qui a un comportement un peu différent : il n'y a pas recopie des données et de la pile, le nouveau processus les partage avec son père. Le processus père est suspendu jusqu'à ce que le fils fasse un appel à `execve ()` ou termine.

Quels sont les avantages et les inconvénients de cet appel système ? Donnez un exemple de programme qui typiquement devrait se servir de `vfork ()`.