# Processus Linux

## Principaux attributs

Chaque processus Linux est caractérisé par un numéro unique appelé PID (entier non signé de 32 bits) qui lui est attribué par le système au moment de sa création. Chaque nouveau processus reçoit comme valeur de PID, la dernière valeur attribuée lors de l'opération de création précédente augmentée de 1. Par ailleurs, afin de maintenir une certaine compatibilité avec les systèmes Unix traditionnels fonctionnant sur 16 bits, la plus grande valeur de PID admissible est 32 767. Le 32 768$^{e}$ processus créé reçoit le plus petit numéro de PID libéré par un processus mort entre-temps.

Le processus est également caractérisé par l'identifiant de l'utilisateur ayant provoqué sa création, l'UID. Par ailleurs, chaque processus Linux pouvant créer lui-même un autre processus, chaque processus est également caractérisé par l'identifiant du processus qui l'a créé (son père), appelé PPID.

Enfin, chaque processus Linux appartient à un groupe de processus, identifié par un numéro qui est le PID du processus créateur du groupe, encore appelé processus *leader*. L'identifiant du groupe, le GID, auquel le processus appartient fait partie de ses attributs.

UID et GID sont utilisés pour définir les droits d'accès du processus vis-à-vis des ressources de la machine.

Les primitives suivantes permettent à un processus respectivement de connaître la valeur de son PID, du PID de son père, l'identifiant de l'utilisateur qui l'a créé et le numéro du groupe auquel il appartient.

```
#include <unistd.h>
pid_t getpid(void);    retourne le PID du processus appelant.
pid_t getppid(void);   retourne le PPID du processus appelant.
```

## Création d'un processus Linux

Sous le système Linux, tout processus peut créer un nouveau processus qui est une exacte copie de lui-même. Le processus créateur est appelé « le père » et le processus créé est appelé « le fils ». Il découle de cette possibilité la formation d'une arborescence de processus, reliés entre eux par des liens de filiation.

La primitive fork() permet la création dynamique d'un nouveau processus qui s'exécute de manière concurrente avec le processus qui l'a créé. Tout processus Linux hormis le processus 0 est créé à l'aide de cette primitive. Le processus créateur (le père) par un appel à la primitive fork() crée un processus fils qui est une copie exacte de lui-même (code et données). Le prototype de la fonction est le suivant :

```
#include <unistd.h>
pid_t fork (void);
```

– le système retourne au processus père le PID du processus créé et au nouveau processus (le fils) la valeur 0.

Lors de cette création le processus fils hérite de tous les attributs de son père sauf :

– l'identificateur de son père ;
– son propre identificateur ;
– les temps d'exécution du nouveau processus sont nuls.

À l'issue de l'exécution de la primitive fork(), chaque processus, le père et le fils, reprend son exécution après le fork(). Le code et les données étant strictement identiques, il est nécessaire de disposer d'un mécanisme pour différencier le comportement des deux processus après le fork(). On utilise pour cela le code retour du fork qui est différent chez le fils (toujours 0) et le père (PID du fils créé).

Ainsi, dans le programme suivant, le processus n° 12 222 est le processus père. Le processus n° 12 224 est le processus fils créé par le processus n° 12 222 à l'issue de l'appel à la primitive fork(). Une fois la primitive fork() exécutée par le père, les deux processus (le père et le fils) reprennent leur exécution de manière concurrente. Le processus fils a pour retour de la primitive fork() la valeur 0. Il

va donc exécuter la partie de l'alternative pour laquelle (if pid == 0) est vrai. Le processus père au contraire a pour retour de la primitive fork() la valeur du PID du processus créé c'est-à-dire une valeur positive. Il va donc exécuter l'autre partie de l'alternative. Les traces générées par cette exécution sont par exemple[1]:

```
je suis le fils; mon pid est 12 224
je suis le père; mon pid est 12 222
pid de mon fils, 12 244
pid de mon père, 12 222
```

```
/*********************************************************************/
/*                  Création d'un processus fils                     */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main ()
{
pid_t ret;
ret = fork();
if (ret == 0)
{
      printf ("je suis le fils; mon pid est %d\n", getpid());
      printf ("pid de mon père, %d\n", getppid());
}
else
{
      printf ("je suis le père; mon pid est %d\n", getpid());
      printf ("pid de mon fils, %d\n", ret);
}
}
```

## Terminaison d'un processus et synchronisation avec son père

### Terminaison d'un processus

Un processus termine normalement son exécution en achevant l'exécution du code qui lui est associé. Cette terminaison s'effectue par le biais d'un appel à la primitive exit (status) où status est un code retour compris entre 0 et 255 qui est transmis au père par le processus défunt. Par convention, une valeur de retour égale à 0 caractérise une terminaison normale du processus et une valeur supérieure à 0 code une fin anormale. Le prototype de la fonction exit() est le suivant :

```
#include <stdlib.h>
void exit (int status);
```

Le processus peut également terminer son exécution de manière anormale, suite à une erreur ayant provoqué par exemple une trappe, et la terminaison du processus est alors forcée par le système d'exploitation. Dans ce cas, le système d'exploitation positionne un code d'erreur spécifique de l'erreur rencontrée qui est également retourné au processus père.

Lors de la terminaison d'un processus, le système désalloue les ressources encore possédées par le processus mais ne détruit pas le bloc de contrôle de celui-ci, passe l'état du processus à la valeur TASK_ZOMBIE puis avertit le processus père de la terminaison de son fils, en lui envoyant le signal SIGCHLD et en lui passant notamment la valeur status ou le code d'erreur positionné (fonction do_exit() du noyau).

[1] L'ordre des traces dépend de l'ordonnancement réalisé par le système et de l'ordre d'exécution des deux processus père et fils.

Un processus fils défunt reste zombie jusqu'à ce que son père ait pris connaissance de sa mort. Un processus fils orphelin, suite au décès de son père (le processus père s'est terminé avant son fils) est toujours adopté par le processus numéro 1 (Init) et non par son grand-père.

## Synchronisation avec le père

Un processus père peut se mettre en attente de la mort de l'un de ses fils, par le biais des primitives wait() et waitpid(). Lorsque le processus père prend connaissance de la mort de l'un de ses fils, il cumule les temps d'exécution en mode utilisateur et en mode superviseur de ce fils avec ceux des fils précédemment décédés, puis il détruit le bloc de contrôle du processus fils. Le prototype de la fonction wait() est le suivant :

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

L'exécution du processus père est suspendue jusqu'à ce qu'un processus fils se termine. Si un processus fils est déjà dans l'état zombie au moment de l'appel, la fonction retourne immédiatement le résultat, à savoir le PID du fils terminé et le code retour de celui-ci dans la variable status. Le prototype de la fonction waitpid() est le suivant :

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

Le principe de la primitive est le même que précédemment, mais elle permet en plus de spécifier dans la variable pid, l'identité du processus fils attendu. Si ce processus fils est déjà dans l'état zombie au moment de l'appel, la fonction retourne immédiatement le résultat, à savoir le PID du fils terminé et le code retour de celui-ci dans la variable status. Plus précisément, le contenu de la variable pid est interprété comme suit :

- si pid est une valeur strictement positive, alors le système suspend le processus père jusqu'à ce qu'un processus fils dont la valeur du PID est égal à pid se termine ;
- si pid est nul, alors le système suspend le processus père jusqu'à la mort de n'importe quel fils appartenant au même groupe que le père ;
- si pid est égal à – 1, alors le système suspend le processus père jusqu'à la mort de n'importe lequel de ses fils ;
- si pid est une valeur strictement inférieure à – 1 alors le système suspend le processus père jusqu'à la mort de n'importe lequel de ses fils, dont le numéro de groupe est égal à – pid.

L'interprétation du contenu de la variable status se fait à l'aide des macros WIFEEXITED, WEXITSTATUS, WIFSIGNALED, WIFSTOPPED définies dans le fichier <sys/wait.h>. Ainsi :

- WIFEEXISTED(status) est vrai si le processus fils s'est terminé par un appel à la primitive exit() ;
- WEXITSTATUS(status) permet de récupérer le code passé par le fils au moment de sa terminaison ;
- WIFSIGNALED(status) permet de savoir que le fils s'est terminé à cause d'un signal ;
- WIFSTOPPED(status) indique que le fils est stoppé temporairement.

Le champ options peut notamment prendre la valeur WNOHANG qui provoque un retour immédiat de la primitive si aucun fils ne s'est encore terminé.

‰ Un exemple

```
/*********************************************************************/
/*              Terminaison du processus fils                        */
/*            et affichage de sa valeur de retour                    */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
main ()
{
pid_t ret,fils_mort;
```

```
int status;
ret = fork();
if (ret == 0)
{
      printf(" je suis le fils; mon pid est %d\n", getpid());
      printf ("pid de mon père, %d\n", getppid());
      exit(0);}
else
{
      printf ("je suis le père; mon pid est %d\n", getpid());
      printf ("pid de mon fils, %d\n", ret);
      fils_mort = wait(&status);
      printf ("je suis le père; le pid de mon fils mort est %d\n", fils_mort);
      if (WIFEXITED(status))
          printf ("je suis le père; le code retour de mon fils est %d\n", WEXITSTATUS(status));
}
}
```

Les traces d'exécution de ce programme sont par exemple :

```
je suis le père; mon pid est 4243
je suis le fils; mon pid est 4244
pid de mon père, 4243
pid de mon fils, 4244
je suis le père; le pid de mon fils mort est 4244
je suis le père; le code retour de mon fils est 0
```

## Les primitives de recouvrement

‰ Interface d'un programme exécutable

La forme générale d'un programme principal en langage C est la suivante :

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

Cette interface permet au programme exécutable de récupérer un ensemble de paramètres dans un environnement particulier. Ainsi :

– argc est le nombre total de paramètres transmis au programme. C'est le nombre total de composants de la commande lancée ;
– argv est un tableau comprenant les différents paramètres passés au programme exécutable. Le premier élément de ce tableau argv[0] est toujours le nom de la commande elle-même ;
– arge est une liste de pointeurs permettant d'accéder à l'environnement d'exé-cution du processus. Chaque pointeur est une chaîne de caractères de la forme nom_variable = valeur. Nous revenons sur les variables d'environnement dans le paragraphe exercices de ce chapitre.

*Exemple*

On lance l'exécution d'un programme calcul avec trois paramètres :

```
sh > calcul 3 4
```

On a :

– argc = 3,
– argv[0] = "calcul", argv[1] = "3" et argv[2] = "4",
– arge contient la liste des variables d'environnement avec leur valeur telles qu'elles ont été positionnées pour l'interpréteur de commandes (shell sh), père du processus calcul.

## Les primitives exec

Les primitives de recouvrement constituent un ensemble de primitives (famille Exec) permettant à un processus de charger en mémoire, un nouveau code exécutable. L'exécution d'une des primitives de la famille Exec entraîne l'écrasement du code hérité au moment de la création (primitive fork()) par le

code exécutable spécifié en paramètre de la primitive. Des données peuvent être passées au nouveau code exécutable *via* les arguments de la primitive Exec qui les récupère dans le tableau argv[]. Il est également possible de modifier l'environnement que le processus a hérité de son père.

La famille Exec est constituée de 6 primitives dont les prototypes sont donnés ci-après :

- int execl (const char *ref, const char *arg, …, NULL): ref est le chemin d'un exécutable à partir du répertoire courant, const char *arg, …, NULL est la liste des arguments.
- int execlp (const char *ref, const char arg, …, NULL): ref est le chemin d'un exécutable à partir de la variable d'environnement PATH, const char *arg, …, NULL est la liste des arguments.
- int execle (const char *ref, const char *arg, …, const char *envp[]): ref est le chemin d'un exécutable à partir du répertoire courant, const char *arg, …, NULL est la liste des arguments, const char *envp[] est un tableau spécifiant les variables d'environnement sous la forme nom_var = valeur.
- int execv (const char *ref, const char *arg[]): ref est le chemin d'un exécutable à partir du répertoire courant, const char *arg [] est un tableau contenant les arguments.
- int execvp (const char *ref, const char *arg[]): ref est le chemin d'un exécutable à partir de la variable d'environnement PATH, const char *arg [] est un tableau contenant les arguments.
- int execve (const char *ref, const char *arg[], …, const char *envp[]): ref est le chemin d'un exécutable à partir du répertoire courant, const char *arg [] est un tableau contenant les arguments, const char *envp[] est un tableau spécifiant les variables d'environnement sous la forme nom_var = valeur.

Dans l'exemple suivant, le processus fils écrase le code hérité de son père par celui de la commande ls, avec comme paramètres les chaînes -l et /. La commande exécutée par le fils est donc ls –l / qui liste l'ensemble des fichiers du répertoire racine.

```
/*********************************************************************/
/*   Ecrasement du code hérité pour exécuter la commande ls –l /   */
/*********************************************************************/
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
pid_t pid;

pid = fork();
if (pid == -1)
     printf ("erreur création de processus");
else
if (pid == 0)
{
    printf ("je suis le fils, mon pid est %d\n", getpid());
    printf("Le pid de mon père est %d\n", getppid());
    execlp("ls","ls", "-l", "/", NULL);
 }
else
{
    printf ("je suis le père, mon pid est %d\n", getpid());
    printf("Le pid de mon fils est %d\n", pid);
    wait ();
}
```

Nous donnons ci-après un deuxième exemple pour lequel le processus fils créé effectue un appel à une primitive exec() en demandant le recouvrement de son code par celui d'un programme calcul, qui attend deux arguments entiers à additionner. On fera attention ici au fait que les arguments passés *via* les primitives exec() sont des chaînes de caractères. Aussi dans le processus fils, avant l'appel à la primitive exec(), il est nécessaire de convertir les entiers i et j en chaînes de caractères. Dans le

programme calcul.c, on notera que de manière inverse, les paramètres i et j récupérés dans les arguments argv[1] et argv[2] sont convertis de chaînes de caractères vers entiers.

```
/*********************************************************************/
/*            Ecrasement de code et conversion de types              */
/*********************************************************************/
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
pid_t pid;
int i, j;
char conv1[10], conv2[10];

pid = fork();
printf("Donnez les valeurs de i et j");
scanf ("%d, %d", i, j);
if (pid == 0)
{
  sprintf(conv1, "%d", i); /* conversion de entier vers caractères */
  sprintf(conv2, "%d", j); /* conversion de entier vers caractères */
  execl("/home/delacroix/calcul", "calcul", conv1, conv2, NULL);
 }
else
{
  wait ();
}

Calcul.c
main(argc,argv)
{
int somme;
if (argc <> 3) {printf("erreur"); exit();}
/* atoi: conversion caractère à entier */
somme = atoi(argv[1]) + atoi(argv[2]);
exit();
}
```