Chapitre 03 (CM/TP) : Mise en Oeuvre de Processus (UNIX/Linux)

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Diplôme Universitaire de Technique (DUT, 2^e année)
Diplôme Supérieure de Technologie (DST, 2^e année)
Option : Informatique

ECOLE SUPERIEURE POLYTECHNIQUE
www.esp.sn

Plan du Chapitre

Sommaire

Définition

- Processus = Instance d'un programme en cours d'exécution
  - plusieurs exécutions de programmes
  - plusieurs exécutions d'un même programme
  - plusieurs exécutions « simultanées » de programmes différents
  - plusieurs exécutions « simultanées » du même programme
- Ressources nécessaires à un processus :
  1. Ressources matérielles : processeur, périphériques, etc.
  2. Ressources logicielles :
     - code
     - contexte d'exécution : compteur ordinal, fichiers ouverts
     - mémoire
     - etc.
- Mode d'exécution
  1. utilisateur
  2. noyau (ou système ou superviseur)

Identifiant de processus : PID

- Chaque processus est identifié de façon unique par un numéro : son PID (Process IDentification).
- Le processus de PID=0 est crée au démarrage de la machine :
  - a un rôle spécial pour le système (surtout pour la gestion de la mémoire)
- Le processus zéro crée, grâce à un appel de fork, le processus **init** dont le **PID** est égal à **1**
- Le processus de PID=1 est l'ancêtre de tous les autres processus (le processus 0 ne réalisant plus de fork())
- Les processus sont organisés en un arbre de processus :
  - Un processus particulier (init, de pid 1) est la racine de cet arbre

Informations sur les processus

- Commandes shell :
  1. **ps** : liste les processus en cours d'exécution
  2. **pstree** : la même sous la forme d'un arbre
- Représentation sous forme d'un système de fichier (Linux)
  - /proc (e.g., information sur les ressources, cpu, etc.)

Attributs d'un processus

- Identification :
  - numéro du processus (process id) : pid_t getpid(void);
  - numéro du processus père : pid_t getppid(void);
- Propriétaire réel :
  - Utilisateur qui a lancé le processus et son groupe
    - uid_t getuid(void);
    - gid_t getgid(void);
- Propriétaire effectif
  - Détermine les droits du processus
    - uid_t geteuid(void);
    - gid_t getegid(void);
  - Le propriétaire effectif peut être modifié
    - int setuid(uid_t uid);
    - int setgid(gid_t gid);

Sommaire

Création de processus avec fork()

- Sous UNIX la création de processus est réalisée, en langage C, par l'appel système : pid_t fork(void);

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

- Cette création de processus se fait par clonage du processus père
- Tous les processus sauf le processus d'identification 0 (le main), sont créés par un appel système **fork**.
- Le processus qui appelle le fork est appelé **processus père**
- Le nouveau processus crée par un fork est appelé **processus fils**
- Tout processus a un seul processus père.
- Tout processus peut avoir zéro ou plusieurs processus fils

Exercice d'application : création de processus par fork()

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    printf("avant fork\n");
    fork();
    printf("apres fork\n");
    return 0;
}
```

Étude des valeurs de retour de fork()

- La primitive fork() crée un nouveau processus (appelé fils)
  - qui est une copie exacte du processus appelant (processus père)
- La différence est faite par la valeur de retour de fork(), qui est :
  1. égale à zéro chez le processus fils,
  2. égale au pid du processus fils chez le processus père
  3. égale à -1 en cas d'erreur

Différenciation du processus père et du processus fils

☛ Le processus est une copie du processus père à l'exception de :

- la valeur de retour de fork
- son identité **pid** et de celle de son père **ppid**

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4
5  int main() {
6
7      pid_t id;
8
9      id = fork();
10
11     printf("id = %d, pid = %d, ppid = %d\n",id, getpid(), getppid());
12
13     return 0;
14 }
```

Exercice d'application : utilisation typique de fork()

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t status;
    status = fork();
    switch (status) {
        case -1 :
            perror("Creation de processus");
            return -1;
        case 0 : /*Code du fils*/
            printf("[%d] Je viens de naitre\n", getpid());
            printf("[%d] Mon pere %d\n", getpid(), getppid());
            break;
        default : /* Code du pere*/
            printf("[%d] J\'ai engendre\n", getpid());
            printf("[%d] Mon fils est %d\n", getpid(), status);
    }
    printf("[%d] Je termine\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Duplication de la mémoire du processus père

☛ Comme la mémoire est copiée : les données sont copiées

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int glob = 1;
int main() {
    int loc = 1;
    switch (fork()) {
        case -1 :
            perror("Creation de processus");
            return -1;
        case 0 :
            glob++; loc++;
            printf("Fils : (%d, %d)\n", glob, loc);
            break;
        default :
            sleep(1);
            printf("Pere : (%d, %d)\n", glob, loc);
    }

    printf("[%d] Je termine\n", getpid());
    return 0;
}
```

Duplication des buffers d'écriture

☛ Comme la mémoire est copiée : les buffers d'écriture de la bibliothèque standard d'entrées/sorties sont dupliqués

```
1
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  int main() {
6      printf("avant ");
7      fork();
8      printf("apres\n");
9      return 0;
10 }
```

☛ Il faut vider les buffers avant fork (par un appel à fflush)

Attributs non copiés ?

- Numéro de processus
- Numéro de processus du père
- Temps d'exécution
- Priorité du processus
- Verrous sur les fichiers

Sommaire

Différents types de terminaison de processus

- **Terminaison normale**, avec un code de retour :
  1. fin de la fonction main()
     - return 0;
     - Un processus courant se termine automatiquement lorsqu'il cesse d'exécuter la fonction main()
  2. primitive système exit() :
     - exit(1);
- **Terminaison anormale**, par un signal :
  1. envoyé par l'utilisateur :
     - touches <Ctrl-C> ou commande **kill**
  2. envoyé par un autre processus :
     - primitive kill()
  3. envoyé par le noyau :
     - en cas d'erreur d'exécution

Actions réalisées à la terminaison d'un processus

☛ A la terminaison d'un processus :

1. ses ressources sont libérés
2. ses fichiers ouverts sont fermés
3. ses enfants sont adoptés par init
4. son père reçoit un signal SIGCHLD
5. son état d'exécution devient « terminé » : **zombie**
6. son entrée dans la table de processus n'est pas libérée
   - Le PID et le PCB d'un processus terminé ne sont pas libérés tant que son père ne ramasse ses cendres

Terminaison du processus courant

- ☛ Terminaison explicite d'un processus :
  - Appel système : *_exit*
  - Appel de la fonction de bibliothèque : **exit()**
    - Ces deux primitives provoquent la terminaison du processus courant, i.e. termine le processus appelant
  - Le paramètre status spécifie un code de retour :
    - compris entre 0 et 255, à communiquer au processus père
  - En cas 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.
    - ☛ ferme les descripteurs de fichiers ouverts
    - ☛ un signal SIGCHLD est envoyé au processus père.
    - ☛ le PPID des processus fils du processus sortant devient 1 (init)

Attente de la terminaison d'un fils

- L'instruction *pid_t* wait(int * status);
  - Attend la fin d'un processus fils (on ne choisit pas lequel).
  - Retourne immédiatement si un fils est mort avant l'appel
- Renvoie le PID du fils terminé dans la valeur de retour
  - pid_t wait(int *status);
  - retourne le PID de fils ou -1 en cas d'erreur (n'a pas de fils)
  - bloquant si aucun fils n'a terminé
  - *status renseigne sur la terminaison du fils
- On peut appeler wait(NULL) si on n'en a pas besoin
- Libère l'entrée du défunt dans la table de processus.

1^e Exemple d'attente de terminaison d'un processus fils

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t id = 0;
    printf("Processus Pere [%d]\n", getpid());
    if (fork() == 0) {
        printf("Processus Enfant [%d] : mon pere est %d\n", getpid(), getppid());
        exit(0);
    }
    id = wait(NULL);
    printf("Processus pere [%d] : mon Enfant [%d] est mort\n", getpid(), id);
    return 0;
}
```

Étude des macros de la fonction wait()

- La primitive **wait** ci-après :

```
#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*

Étude des macros de la fonction wait()

Le compte-rendu de wait()

- Les macros définies dans sys/wait.h :

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

pid_t wait (int *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() ;

Étude des macros de la fonction wait()

Le compte-rendu de wait()

- Les macros définies dans sys/wait.h :

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

pid_t wait (int *status) ;
```

- **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.

2nd Exemple d'attente de terminaison d'un processus fils : utilisation des macros

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main (void)
{
    pid_t pid ;
    int status ;
    pid = fork () ;
    switch (pid) {
        case -1 :
            perror ("Error dans l'appel fork") ;
            exit (1) ;
        case 0 : /* le fils */
            printf ("Processus fils [%d]: mon pere est [%d]\n", getpid(), getppid()) ;
            exit (2) ;
        default : /* le pere */
            printf ("Pere [%d] : a cree processus [%d]\n",getpid(), pid) ;
            wait (&status) ;
            if (WIFEXITED (status))
                printf ("Le fils termine normalement: status = %d\n",
                        WEXITSTATUS (status)) ;
            else
                printf ("fils termine anormalement\n") ;
    }
}
```

Sommaire

Processus orphelins

- La terminaison d'un processus parent ne termine pas ses processus fils
  - les processus fils sont orphelins
- Le processus initial (PID 1) récupère les processus orphelins

Illustration d'un processus orphelin

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

int main(int argc, char **argv) {

    switch (fork()) {
        case -1 : perror("Creation de processus"); return 1;
        case 0 :
            printf("[%d] Pere : %d\n", getpid(), getppid());
            sleep(5);
            printf("[%d] Pere : %d\n", getpid(), getppid());
            exit(0);
        default :
            sleep(1);
            printf("[%d] fin du pere\n", getpid());
            exit(0);
    }
}
```

Processus zombies

- Zombie = état d'un processus
  - ayant terminé;
  - non réclamé par son père (par l'exécution d'un *wait*)
  - La redirection du fils vers le processus initial *init* se fait à la mort du père
  - Le processus *init* exécute une boucle d'attente (avec *wait* de ses processus fils pour tuer les « zombis »
- Il faut éviter les zombies.
  - Le système doit garder des informations relatives aux processus pour les retournées aux pères.
  - Encombre la mémoire
- Comment éviter les zombies si le père ne s'intéresse pas à la terminaison de ses fils ?
  - Solution du « double fork »

« Double fork »

- Le processus père ne s'intéresse pas à la terminaison de son fils.
- Dès que le fils termine, il passe dans un état zombie.
- La redirection vers le processus initial *init* ne se fait qu'à la mort du père.

Le mécanisme du « Double fork »

- Le fils ne vit que le temps de créer le petit fils puis meure
- Ainsi, le petit fils n'a plus de père et est rattaché au processus *init*
- Le processus *init* surveille ses fils (avec wait) pour éviter qu'ils ne restent dans l'état « zombie »
- Le père peut libérer immédiatement son fils (attente courte)

Exemple avec « Double fork »

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    pid_t status;
    status = fork();
    switch (status) {
        case -1: perror("Erreur de creation de processus"); return -1;
        case 0: // Code du fils
            switch (fork()) {
                case -1: perror("Erreur de creation de processus intermediaire");return -1;
                case 0 :
                    printf("Petit fils : processus [%d], Mon Pere : [%d] \n", getpid(), getppid());
                    sleep(5);
                    printf("Petit fils : processus [%d], Mon Pere : [%d] \n", getpid(), getppid());
                    break;
                default :
                    sleep(1);
                    printf("Pere : processus [%d], Mon Pere : [%d] \n", getpid(), getppid());return
                        0;
            };
        break;
        default :
            printf("Grand Pere : processus [%d], Mon Pere : [%d] \n", getpid(), getppid());
            wait(&status);
      printf("fin du grand pere\n");
      break;
    };
    exit(EXIT_SUCCESS);
```

Sommaire

Les primitives exec()

- Il s'agit d'une famille de primitives permettant le lancement de l'exécution d'un programme externe
- Il n'y a pas création d'un nouveau processus, mais simplement changement de programme
- Il y a six primitives exec() que l'on peut répartir dans deux groupes :
  1. les execl(), pour lesquels le nombre des arguments du programme lancé est connu
  2. les execv() où il ne l'est pas
- toutes ces primitives se distinguent par le type et le nombre de paramètres passés

Les primitives exec() : Premier groupe

☛ Premier groupe d'exec(), les arguments sont passés sous forme de liste

```
1  int execl(char *path, char *arg0, char *arg1,..., char *argn,NULL)
2
3  /* exécute un programme */
4
5  /* path : chemin du fichier programme */ /* arg0 : premier argument */
6
7  /* ... */
8
9  /* argn : (n+1)ieme argument */
10
11 int execle(char *path,char *arg0,char *arg1,...,char *argn,NULL,char *envp[])
12
13 /* envp : pointeur sur l'environnement */
14
15 int execlp(char *file,char *arg0,char *arg1,...,char *argn,NULL)
16
```

Les primitives exec() : Premier groupe

- Dans execl et execle, path est une chaine indiquant le chemin exact d'un programme. Un exemple est "/bin/ls"
- Dans execlp, le « p » correspond à path et signifie que les chemins de recherche de l'environnement sont utilisés.
- Par conséquent, il n'est plus nécessaire d'indiquer le chemin complet.
- Le premier paramètre de execlp pourra par exemple être "ls"
- Le second paramètre des trois fonctions exec est le nom de la commande lancée et reprend donc une partie du premier paramètre
  - Si le premier paramètre est "/bin/ls", le second doit être "ls"
- Pour la troisième commande, le second paramètre est en général identique au premier si aucun chemin explicite à été donné

Les primitives exec() : Second groupe

☛ Second groupe d'exec() : les arguments sont passées sous forme de tableau :

```
1  int execv(char *path,char *argv[])
2
3  /* argv : pointeur vers le tableau contenant les arguments */
4
5  int execve(char  *path,char  *argv[],char *envp[])
6
7  int execvp(char  *file,char  *argv[])
8
```

Les primitives exec() : Second groupe

```
1  int execv(char *path,char *argv[])
2
3  /* argv : pointeur vers le tableau contenant les arguments */
4
5  int execve(char  *path,char  *argv[],char *envp[])
6
7  int execvp(char  *file,char  *argv[])
8
```

- ☛ path et file ont la même signification que dans le premier groupe de commandes
- ☛ Les autres paramètres du premier groupe de commandes sont regroupés dans des tableaux dans le second groupe
- ☛ La dernière case du tableau doit être un pointeur nul, car cela permet d'identifier le nombre d'éléments utiles dans le tableau
- ☛ Pour execle et execve, le dernier paramètre correspond à un tableau de chaînes de caractères, chacune correspondant à une variable d'environnement

Recouvrement

- ☛ Lors de l'appel d'une primitive exec(), il y a recouvrement du segment d'instructions du processus appelant
- ☛ Ce qui implique qu'il n'y a pas de retour d'un exec() réussi (l'adresse de retour a disparu)
- ☛ Le code du processus appelant est détruit

```c
#include <stdio.h>
int main() {
    execl("/bin/ls","ls",NULL) ;
    printf ("je ne suis pas mort\n") ;
    return 0;
}
```

- ☛ On note que la commande ls est réalisée, contrairement à l'appel à printf(), ce qui montre que le processus ne retourne pas après exec()
- ☛ D'où l'intérêt de l'utilisation de la primitive fork()

Recouvrement

- Lors de l'appel d'une primitive exec(), il y a recouvrement du segment d'instructions du processus appelant
- Ce qui implique qu'il n'y a pas de retour d'un exec() réussi (l'adresse de retour a disparu)
- Le code du processus appelant est détruit

```c
#include <stdio.h>
int main() {
    execl("/bin/ls","ls",NULL) ;
    printf ("je ne suis pas mort\n") ;
    return 0;
}
```

- On note que la commande ls est réalisée, contrairement à l'appel à printf(), ce qui montre que le processus ne retourne pas après exec()
- D'où l'intérêt de l'utilisation de la primitive fork()

Recouvrement : exec() avec de la primitive fork()

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {

    if ( fork()==0 ) execl( "/bin/ls","ls",NULL) ;

    else {

        sleep(2) ; /* attend la fin de ls pour exécuter printf() */

        printf ("je suis le p're et je peux continuer") ; }
}
```

☛ Dans ce cas, le fils meurt après l'éxécution de ls, et le père continue à vivre et exécute printf().

Recouvrement : exec() avec de la primitive fork()

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {

    if ( fork()==0 ) execl( "/bin/ls","ls",NULL) ;

    else {

        sleep(2) ; /* attend la fin de ls pour exécuter printf() */

        printf ("je suis le p're et je peux continuer") ; }
}
```

☛ Dans ce cas, le fils meurt après l'éxécution de ls, et le père continue à vivre et exécute printf().

Chapitre 03 (CM/TP) : Mise en Oeuvre de Processus (UNIX/Linux)

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Diplôme Universitaire de Technique (DUT, 2^e année)
Diplôme Supérieure de Technologie (DST, 2^e année)
Option : Informatique

ECOLE SUPERIEURE POLYTECHNIQUE
www.esp.sn