Chap. VII : Les processus en C

Laurent Poinsot

UMR 7030 - Université Paris 13 - Institut Galilée

Cours "Architecture et Système"

Un programme est un fichier exécutable tel que `date` (qui affiche la date courante) ou `less` (qui affiche le contenu d'un fichier). Un processus est un programme en cours d'exécution. Un même programme (par exemple, `less`) peut être lancé plusieurs fois par un même utilisateur (dans des fenêtres différentes) ou par des utilisateurs différents. Il y aura donc en mémoire plusieurs processus exécutant le même programme. On peut distinguer deux sortes de processus : les processus système qui accomplissent une tâche pour le compte du système (gérer l'imprimante par exemple) et les processus utilisateurs.

Les processus exécutent leur code et demandent des services au noyau par l'intermédiaire des appels systèmes (`open`, `read`, `write`, etc.). Ils doivent cependant partager l'unité centrale (CPU) : à un instant donné, un seul processus est exécuté par le processeur. Les processus doivent donc passer rapidemment et à tour de rôle (sauf s'ils sont bloqués) en unité centrale jusqu'à ce que leur tâche soit accomplie. Les processus peuvent être indépendants ou au contraire devoir se synchroniser et communiquer de l'information pour accomplir une tâche comme dans le cas des architectures " client-serveur ". Dans ce dernier cas, un processus (le client) effectuant des demandes (ou requêtes) à un autre processus (le serveur) chargé de recueillir les informations et de les lui faire parvenir. Un processus peut aussi sous-traiter un travail à un processus fils qu'il crée et attendre ou non la fin de ce travail.

Dans tous les cas, les processus doivent partager les ressources communes de l'ordinateur (unité centrale, mémoire, périphériques, fichiers, etc.) et attendre si les ressources nécessaires à leur exécution ne sont pas disponibles. Le noyau de l'O.S. est chargé du bon déroulement des processus. Il doit également veiller à ce qu'un processus n'en détruise pas un autre ou n'accède pas (sauf autorisation) à certaines données en mémoire ou sur disque. Le noyau doit répartir le temps d'exécution entre les processus et décider du prochain processus à faire passer en unité centrale suivant un algorithme dit d'" ordonnancement ". La partie du noyau chargée de cette tâche s'appelle le " scheduler " (ou " ordonnanceur ").

La commande `ps` liste, avec plus ou moins de détails, tous les processus ou seulement ceux de l'utilisateur, suivant différentes options. Par exemple :

```
ps -fu ZZ
```

UID	*PID*	*PPID*	*S*	*STIME*	*TTY*	*TIME*	*CMD*
ZZ	533	1	0	09 : 44	*pts*/0	00 : 00 : 03	*xemacs*
ZZ	532	1	0	09 : 44	*pts*/0	00 : 00 : 01	*konsole*
ZZ	536	532	0	09 : 44	*pts*/3	00 : 00 : 00	*/bin/bash*
ZZ	624	536	0	11 : 16	*pts*/3	00 : 00 : 00	*ps −fu ZZ*
ZZ	531	1	0	09 : 44	*pts*/0	00 : 00 : 01	*konsole*
ZZ	535	531	0	09 : 44	*pts*/2	00 : 00 : 00	*/bin/bash*
ZZ	602	535	0	11 : 04	*pts*/2	00 : 00 : 00	*less client.dat*
ZZ	530	1	0	09 : 44	*pts*/0	00 : 00 : 00	*konsole*
ZZ	534	530	0	09 : 44	*pts*/1	00 : 00 : 00	*/bin/bash*

L'exemple précédent présente des processus listés par la commande `ps`. Le champ `UID` indique le nom du propriétaire du processus (ZZ) ; `PID` (process ID) indique le numéro du processus ; `PPID` (Parent PID) indique le numéro processus père (celui qui a crée le processus correspondant au PID) ; `STIME` indique l'heure de lancement du processus ; `TTY` indique le terminal ou la fenêtre du processus (`pts/0` est une fenêtre ; `pts/1`, une autre fenêtre) ; `CMD` indique la commande qui a permis de lancer le processus. L'option " `-u ZZ` ", de la commande `ps`, indique de lister les processus de l'utilisateur ZZ.

Toujours dans l'exemple précédent, le processus 1 est père des processus 530, 531, 532 (correspondant à des fenêtres `XWINDOWS`) et 533. Dans chacune de ces fenêtres, un processus `bash` (interpréteur de commandes) a été lancé. Dans une des fenêtres, le processus `less` est en cours d'exécution alors que `ps` s'exécute dans la fenêtre 532.

Vous pouvez utiliser le programme `/usr/bin/passwd` qui appartient à root et bénéficier des mêmes droits que root afin de changer votre mot de passe en écrivant dans le fichier des mots de passe. Unix distingue deux utilisateurs de processus : l'utilisateur réel et l'utilisateur effectif. Quand vous modifiez votre mot de passe, vous restez l'utilisateur réel de la commande `passwd`, mais l'utilisateur effectif est devenu root. Un bit particulier du fichier exécutable permet d'effectuer ce changement temporaire d'identité : c'est le bit s.

Un utilisateur non propriétaire d'un programme exécutable peut obtenir, le temps de l'exécution du programme, l'identité du propriétaire de façon à bénéficier des droits d'accès du propriétaire. Le propriétaire doit positionner un droit `s` au lieu de `x` pour les droits d'accès. Lors de l'exécution du programme, il y a donc un utilisateur réel (`uid` : user id) et un utilisateur effectif (`euid` : effective uid). La même possibilité existe pour les groupes : un utilisateur peut prendre le groupe du propriétaire du fichier, le temps de l'exécution d'un programme. Il y a donc un `gid` (group id) et un `egid` (effective gid).

Les primitives `getuid` et `geteuid`

getuid et geteuid donnent l'identité de l'utilisateur.

```
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
```

`getuid` donne le numéro de l'usager réel à qui appartient le processus effectuant l'appel ; `geteuid` donne le numéro de l'usager effectif associé processus. Ce numéro peut être différent si le bit s du fichier est positionné. Le type `uid_t` est généralement un `int`. De même `getgid()` fournit le numéro du groupe réel, alors que `getegid()` renvoie le numéro du group effectif.

Le programme suivant `litmdp.c` lit le fichier `motpas.dat` et l'affiche.

```
#include <stdio.h>
#include<unistd.h>
void main (){
int c;
FILE* fe;
printf("uid :%d, euid :%d, gid :%d,
egid :%d\n",getuid(),geteuid(),getgid(),getegid());
fe=fopen("/usr/local/bin/motpas.dat", "r");
if (fe==NULL) { perror("fopen");exit(1);}
while ((c=fgetc(fe))!=EOF) {
putchar(c);}
}
```

On recopie ensuite le fichier `litmdp` dans le répertoire `/usr/local/bin`, le mettant ainsi à la disposition de tous les utilisateurs. Le fichier `motpas.dat` est en lecture pour le propriétaire `ZZ`. Les autres utilisateurs ne peuvent y accéder directement. Le droit d'accès en exécution de `litmpd` est `s` pour l'utilisateur. Ceci indique qu'un utilisateur autorisé à exécuter le programme `litmdp` prendra, le temps de l'exécution du programme, l'identité (effective) du propriétaire `ZZ`. Il pourra donc lire le fichier `motpas.dat` en utilisant le programme `litmdp`. Vérifions le positionnement des droits par la commande `ls` :

```
> ls -li /usr/local/bin
```

82355	$-r-----$	1	*ZZ*	*prof*	...	*motpas.dat*
14504	$-rws--x--x$	1	*ZZ*	*prof*	....	*litmdp*

L'utilisateur DD peut exécuter `litmdp`. Son `uid` est 501 (DD) ; son `euid` est 500 (ZZ) :

```
> litmdp
uid :501, euid :500, gid :502, egid :502
ZZ : toto
Alain : tata
Nadia : tutu
> id
uid=501(DD) gid=502(etudiant)
groupes=502(etudiant),503(amis),506(projet)
```

Le fichier des mots de passe du système Unix `etc/passwd` est géré de façon similaire en utilisant l'exécutable `/usr/bin/passwd` qui a un droit d'accès `s` pour le propriétaire (`root`) seul autorisé à modifier le fichier. Les utilisateurs de `/usr/bin/passwd` acquièrent, le temps de l'exécution du programme, les droits d'accès de `root` sur `/etc/passwd`.

La primitive `getlogin`

```
#include <unistd.h>
char * getlogin ( void );
```

Cette commande donne le nom - sous forme d'une suite de caractères - de l'utilisateur connecté au système sur un terminal de contrôle.

Nous avons déjà vu qu'un processus est un programme qui s'exécute. Un processus possède un identificateur qui est un numéro : le pid. Il s'agit d'un entier du type `pid_t` déclaré comme synonyme du type `int` ou `unsigned long int`. Un processus incarne - exécute - un programme ; il appartient à un utilisateur ou au noyau. Un processus possède une priorité et un mode d'exécution. Nous distinguerons deux modes : le mode utilisateur (ou esclave) de basse priorité et le mode noyau (ou maître) de plus forte priorité.

Rappelons que généralement un processus appartient à la personne qui a écrit le programme qui s'exécute, mais ce n'est pas toujours le cas. Quand vous exécutez la commande `ls` vous exécutez le programme `/bin/ls` qui appartient à root. Regardez le résultat de la commande ls -l :

```
% ls -l /bin/ls
-rwxr-xr-x 1 root root 29980 Apr 24 1998
/bin/ls
```

Un processus naît, crée d'autres processus (ses fils), attend la fin d'exécution de ceux-ci ou entre en compétition avec eux pour avoir des ressources du système et enfin meurt. Pendant sa période de vie, il accède à des variables en mémoire suivant son mode d'exécution. En mode noyau, tout l'espace adressable lui est ouvert ; en mode utilisateur, il ne peut accéder qu'à ses données privées.

Chaque processus possède un identificateur unique nommé pid. Comme pour les utilisateurs, il peut être lié à un groupe. Citons les différentes primitives permettant de connaître ces différents identificateurs :

```
pid_t getpid() /* retourne l'identificateur du
processus */
pid_t getpgrp() /* retourne l'identificateur
du groupe de processus */
pid_t getppid() /* retourne l'identificateur
du père du processus */
```

Exemple :

```
/* fichier test_idf.c */
#include <stdio.h>
main()
{ printf("je suis le processus %d de père %d
et de groupe %d\n",getpid(),
getppid(),getpgrp());
}
```

Résultat de l'exécution :

```
Systeme> ps
PID TTY TIME COMMAND
6658 ttyp5 0 :04 csh
Systeme> test_idf
je suis le processus 8262 de père 6658 et de
groupe 8262
```

Notons que le père du processus exécutant `test_idf` est csh.

Descripteurs de fichier

Nous avons vu que le noeud d'index d'un fichier est la structure d'identification du fichier vis-à-vis du système. Lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur au noeud d'index du fichier se fait lors de l'appel à la primitive `open()`. Le descripteur devient alors le nom local du fichier dans le processus. Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19.

Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;
- le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives open(), write(), read(), close().

Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrées/sorties fopen, fread, fscanf, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

- `stdin` qui pointe vers le tampon de l'entrée standard (généralement le clavier) ;
- `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
- `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

Primitive fork()

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork() /* création d'un fils */
```

Valeur retournée : 0 pour le processus fils, et l'identificateur du processus fils pour le père, -1 dans le cas d'épuisement de ressource. Cette primitive est l'unique appel système permettant de créer un processus. Le processus fils diffère du père uniquement par ses numéros de PID et PPID. Juste après le fork (), chaque processus (le père et le fils) dispose des mêmes valeurs de données et de pile. Cependant, les variables ne sont pas partagées. Chaque processus a sa propre copie des données et de la pile ; si un processus modifie une variable, cela ne modifie pas la variable équivalente de l'autre processus.

Après exécution du `fork ()`, il y a donc deux processus qui s'exécutent concurremment. Les processus père et fils partagent le même code. Le fils hérite d'un double de tous les descripteurs de fichiers ouverts par le père ; les pointeurs de fichier associés sont partagés (si le fils se déplace dans le fichier, la prochaine manipulation du père se fera à la nouvelle adresse). Le père et le fils peuvent lire ou écrire dans le même fichier si celui a été ouvert avant le `fork ()`.

Exemple introductif

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main(){
pid_t p1;
printf("Debut de fork\n");
p1 = fork();
printf("Fin de fork %d\n", p1);}
```

Résultat de l'exécution :

```
% ./a.out
Debut de fork
Fin de fork 16099
Fin de fork 0
```

Dans cet exemple, on voit qu'un seul processus exécute l'écriture `Debut de fork`, par contre, on voit deux écritures `Fin de fork` suivies de la valeur de retour de la primitive `fork()`. Il y a bien apparition d'un processus : le fils, qui ne débute pas son exécution au début du programme mais à partir de la primitive `fork`.

Primitive exit()

```
void exit(int status)
/* terminaison du processus */
/* status : état de sortie */
```

Tous les descripteurs de fichier ouverts sont fermés. Si le père meurt avant ses fils, le père du processus fils devient le processus `init` de pid 1. Par convention, un code de retour égal à zéro signifie que le processus s'est terminé normalement, et un code non nul (généralement 1 ou -1) signifie qu'une erreur s'est produite.

Primitive wait ()

L'appel système `wait ()` bloque un processus en attente de la fin de l'exécution d'un fils qu'il a crée. La syntaxe est la suivante :

```
pid_t wait (int* status);
```

La fonction retourne le numéro de PID du fils qui vient de se terminer ; `status` peut recevoir des informations s'il est non `NULL` sur la façon dont s'est terminé le processus (valeur du `exit` du fils). Si on a créé plusieurs fils, l'appel `wait` ne permet pas d'attendre la fin d'un fils particulier. C'est le rôle de l'appel système `waitpid` (voir la documentation en ligne : `man waitpid`).

Primitive sleep ()

La fonction sleep () suspend un processus pendant le nombre de secondes indiqué en paramètre. La fonction peut être interrompue par un signal (comme on le verra plus loin dans le cours) ; dans ce cas, la fonction fournit le nombres de secondes restant avant la fin programmée, 0 sinon. La syntaxe est :

```
unsigned int sleep (unsigned int seconds)
```

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 : les `execl()`, pour lesquels le nombre des arguments du programme lancé est connu, puis les `execv()` où il ne l'est pas. En outre toutes ces primitives se distinguent par le type et le nombre de paramètres passés.

Premier groupe d'`exec()`. Les arguments sont passés sous forme de liste :

```
int execl(char *path, char *arg0, char
*arg1,..., char *argn,NULL)
/* exécute un programme */
/* path : chemin du fichier programme */
/* arg0 : premier argument */
/* ... */
/* argn : (n+1)ième argument */
int execle(char *path,char *arg0,char
*arg1,...,char *argn,NULL,char *envp[])
/* envp : pointeur sur l'environnement */
int execlp(char *file,char *arg0,char
*arg1,...,char *argn,NULL)
```

Dans `execl` et `execle`, `path` est une chaîne 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 n'a été donné.

Second groupe d'`exec()`. Les arguments sont passés sous forme de tableau :

```
int execv(char *path,char *argv[])
/* argv : pointeur vers le tableau contenant
les arguments */
int execve(char *path,char *argv[],char
*envp[])
int execvp(char *file,char *argv[])
```

`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. On trouvera plus de détails dans le manuel en ligne.

Exemple : Soit le fichier `test_exec_fork.c`

```
#include<stdio.h>
main() {
if ( fork()==0 ) execl( "/bin/ls","ls",NULL) ;
else {
sleep(2) ; /* attend la fin de ls pour exécuter printf() */
printf ("je suis le père et je peux continuer") ;
}
```

Résultat de l'exécution :

```
Systeme> test_exec_fork
fichier1
fichier2
test_exec
test_exec.c
test_exec_fork
test_exec_fork.c
je suis le père et je peux continuer
```

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