Vie et mort d'un processus (point de vue local)

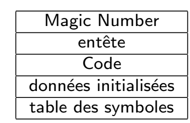
Licence MIAGE — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 6 — 2012-2013

Définition d'un processus

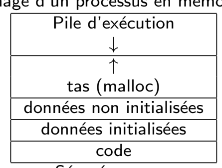
Un processus est l'abstraction d'un programme exécuté en mémoire.

Image d'un programme sur le disque



Séparé en bloc

Image d'un processus en mémoire



Séparé en page

Comme exemple de *magic number*, signalons que si un fichier commence par `#!`, il est exécutable par un interpréteur (ces 2 caractères sont suivit par le chemin d'accès à l'interpréteur).

Les informations attachées à un processus

L'OS dispose de renseignement concernant un processus :

- le pid i.e. numéro identificateur du processus ;
- le ppid i.e. numéro identificateur du processus père ;
- l'uid i.e. numéro identificateur du propriétaire du processus ;
- le numéro du groupe auquel appartenait le propriétaire du processus (il peut appartenir à plusieurs groupe) ;
- l'heure et la date de création du processus ;
- des statistiques sur les ressources (processeur, etc.) utilisées ;
- la taille mémoire actuellement utilisée par le processus (en distinguant chaque composante : swap, tas, code, données, etc).

Mais aussi,

- un masque indiquant la sensibilité aux signaux ;
- son état (en cours d'exécution, zombie, arrêté, suspendu, etc.) ;
- la priorité d'exécution ;

qui sont des notions que nous expliciterons plus loin ou lors du cours sur l'ordonnancement.

Chaque processus est représenté dans l'OS par un *bloc de contrôle de processus* contenant les informations suivantes :

- l'état du processus (nouveau, prêt, en cours d'exécution, en attente, arrêté, etc.) ;
- les numéros associés au processus ;
- l'état des registres du microprocesseur associés au processus et notamment :
  - les differents numéros de segments (mémoire, code, pile, etc.) ;
  - un pointeur sur la prochaine instruction à exécuter ;
  - les registres à usage généraliste.
- la liste des fichiers ouvert et l'ensemble des informations associées aux entrées-sorties ;
- des statistiques sur l'utilisation des ressources de la machine (temps d'occupation du processeur, etc.) ;
- les limites maximales de la mémoire (début et fin des ces espaces).

Les commandes shell de manipulation des processus

L'interpréteur de commandes propose des commandes externes de gestion des processus permettant :

- de récupérer de l'information :
  - **ps** permet d'afficher l'ensemble des informations disponibles sur l'ensemble des processus existant sur une machine. Cette commande utilise le système de fichier `/proc` ;
  - **top** encapsule des appels successifs à ps afin d'afficher l'évolution de la communauté des processus ;
- d'agir sur les processus :
  - **at** diffère temporellement l'exécution d'un processus ;
  - **batch** diffère — suivant les ressources — l'exécution d'un processus ;
  - **Ctrl-z** envois le signal de suspension au processus courant ;
  - **fg** réactive le processus suspendu en avant plan ;
  - **bg** réactive le processus suspendu en arrière plan ;
  - **kill** permet d'envoyer l'ensemble des signaux à un processus.

Les signaux

La commande interne `kill -Valeur pid` permet d'envoyer le signal `Valeur` ($1 \leq \text{Valeur} \leq 31$) aux processus d'identificateur `pid`.

Signal	Valeur	Action	Commentaire
SIGINT	2	Term	Terminaison depuis le clavier (CTRL-C)
SIGQUIT	3	Core	Quit depuis le clavier (CTRL-\)
SIGKILL	9	Term	Kill signal
SIGCONT	18		Continue if stopped (bg)
SIGSTOP	19 et 20	Stop	Stop process (CTRL-Z)

Les effets de la réception de ces signaux sur un processus sont :

Term : le processus se termine ;
Core : Term + copie de la mémoire dans un fichier core ;
Stop : suspend l'exécution du processus.

L'appel système fork

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>  /* La structure de ce code est typique du  */
#include <unistd.h>    /* clonage d'un processus.                 */
#include <stdlib.h>
int main(void){
  int status,pid = 0 ;
  pid = fork() ;      /* L'appel syst\'eme permettant le clonage */
  if (pid == 0){
    printf("fils :%d\n",getpid()) ; /* le code du processus fils */
    exit(1) ;  /* un appel permettant la terminaison du fils  */
  }
  else {  /* le code du processus p\'ere qui utilise */
    printf("pere :%d\n",getpid()) ; /* un appel pour conna\^itre */
    printf("mon fils :%d\n",pid) ;  /* son pid */
    wait(&status) ;                 /* Un appel permettant au  */
                                    /* p\'ere d'attendre la mort */
                                    /* de son fils /*
   exit(++status) ;                 /* avant de terminer. */
  }
  return 0 ;  /* cette instruction n'est jamais ex\'ecut\'ee */
}
```


- L'appel `pid_t fork(void);` provoque le *clonage* du père en fils :
  - les segments de mémoire sont dupliqués (code, donnée, etc) ;
  - cela implique que père et fils partagent les données définies avant le clonage (variables mais aussi flux ouvert, etc.) ;
  - l'exécution des processus père et fils poursuivent leurs exécutions juste après le clonage.
- L'appel `pid_t getpid(void);` (resp. `pid_t getppid(void);`) retourne l'identificateur (resp. du père) du processus courant.
- L'appel `void exit(int status);` provoque la terminaison normale du processus et retourne l'entier status au père. Dans l'exemple précédant le fils envoit 1 au père qui retourne 2 au shell.
- L'appel `pid_t wait(int *status);` force le père à attendre la terminaison d'un processus de sa parenté (qui lui envoit un signal stocké dans l'espace pointé par status) dont le pid est retourné par l'appel.

La famille d'appels système exec

Un processus existant peut subir une mutation par un appel système en un processus construit à partir d'un exécutable existant.

```
#include <unistd.h>
int main(void){
  execl("/bin/ls","ls",NULL) ;
  return 1 ; /* Cette commande n'existe plus */
}           /* du fait de la mutation       */
```

Il existe 6 primitives que l'on peut répartir en deux groupes :

- les primitives `execl` pour lesquels le nombre d'arguments du programme lancé est connu. Les arguments sont passés sous forme de liste ;

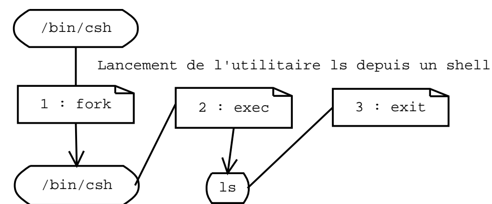
```
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg  , ...,
              char * const envp[]);
```

- les primitives `execv` pour lesquels le nombre d'arguments ne l'est pas. Les arguments sont passés sous formes de tableau.

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


Lancement d'une commande depuis un shell

Le clonage et la mutation peuvent se combiner. Ainsi que se passe-t-il lorsque le shell interprète la commande `ls`.



Comment l'emplacement du répertoire courant — connu au niveau du shell — est-il transmis à l'exécutable ?

1. le `fork` duplique la mémoire, donc transmission directe ;
2. deux possibilités pour l'`exec` :
   - 2.1 soit on le passe comme argument (du `main` en C) ;
   - 2.2 soit on le passe comme variable d'environnement.

La généalogie des processus

La commande `pstree` permet d'afficher l'arbre généalogique dans lequel prennent placent tous les processus tournant sur une machine :

```
init-+-atd
     |-4*[automount]  // configure mount points for autofs
     |-bdflush        // kernel daemon to flush dirty buffers back to
     |-crond          // daemon to execute scheduled commands
     |-6*[mingetty]
     |-portmap     // converts RPC program numbers into DARPA protoco
     |-postmaster---postmaster---postmaster
     |-sshd                              // Open SSH daemon
     |-xdm-+-X
     |     `-xdm--wmaker-+-rxvt.bin--csh-+-mozilla-bin--mozilla-bin--
     |                   |               `-mutt
     |                   |-rxvt.bin--csh-+-pstree
     |                   |               `-xemacs--xdvi--xdvi.bin--gs
     |                   |-ssh-agent
     |                   `-wmCalClock-Wind
     |-xfs                               // X font server
     etc.
```


Quelques règles régissant la communauté des processus

Au démarrage de la machine un premier processus est créer par le super utilisateur. Ce processus porte le nom `init` et le pid 1.

Le processus `init` ne se termine qu'à l'extinction de la machine.

Si le père disparaît avant ses fils, l'OS fait adopter les processus orphelins par le processus `init`.

Un processus ne sait que son père a disparu qu'à sa propre terminaison.

Un processus fils crée un nouveau groupe en se retirant du groupe auquel il appartient.

La commande `jobs` permet d'afficher l'ensemble des processus engendrés exclusivement dans un shell avec un numéro de job %num.

On peut envoyer un signal à un job en utilisant la commande `kill` et son numéro de job.

En tuant un processus père, on rend les processus fils orphelins mais en tuant un job père, on tue tous les jobs fils.

Processus ?
Signaux à destinations des processus
Clonage d'un processus
Mutation d'un processus
Terminaison d'un processus
Compléments

L'appel système `uid_t getuid(void)` (resp. `int setgid(gid_t gid);`) permet de déterminer le propriétaire (resp. le groupe) du processus.

L'appel système `int setuid(uid_t uid)` (resp. `int setgid(gid_t gid);`) permet de modifier le propriétaire (resp. le groupe) d'un processus.

Le système d'exploitation lance un grand nombre de processus dont la fonction est d'assurer les tâches de gestions, par exemple :

- cups est un gestionnaire d'impression ;
- xscreensaver un gestionnaire pour économiser des ressources ;
- gpm un gestionnaire de la sourie.

Processus ?
Signaux à destinations des processus
Clonage d'un processus
Mutation d'un processus
Terminaison d'un processus
Compléments

Les étapes de la mort d'un processus

L'appel système `exit` permet de *terminer* un processus :

- l'ensemble des descripteur de fichiers ouvert sont fermés ;
- un entier `status` est *envoyé* au père. Cet entier peut être récupéré par ce dernier grâce à l'appel système `wait`. Cet appel bloque l'exécution du père.

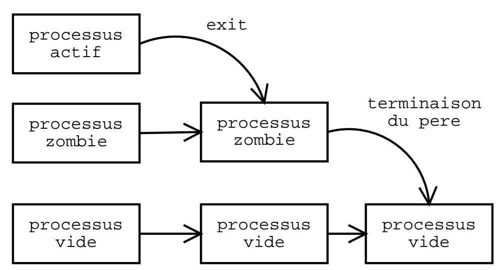
```
#include <stdlib.h>              #include <sys/types.h>
void exit(int status);           #include <sys/wait.h>
                                 pid_t wait(int *status);
```

Considérons le code :

```
#include <stdlib.h>  #include <unistd.h>#include <sys/wait.h>
int main(void){
        int res = 0, status = 0 ;
        res = fork() ;            /* clonage          */
        if(!res) exit(status) ;   /* le fils termine  */
        else wait(&status) ;      /* le p\'ere attend */
        return 0 ;                /* la fin du fils   */
}
```


Processus ?
Signaux à destinations des processus
Clonage d'un processus
Mutation d'un processus
Terminaison d'un processus
Compléments

À la terminaison d'un processus fils, un lien avec le père demeure jusqu'à ce que ce dernier se termine à son tour ou appelle wait. L'entrée du processus fils dans la table des processus ne se libère pas immédiatement à sa terminaison. Bien qu'inactif, le processus fils demeure dans le système en vue d'un éventuel appel wait. C'est un processus zombie.



Processus ?
Signaux à destinations des processus
Clonage d'un processus
Mutation d'un processus
Terminaison d'un processus
Compléments

Les processus légers : les threads

Le modèle de processus lourd — décrit jusqu'à présent — repose sur une entité exécutant un flot — thread en anglais — d'instructions.

Mais un processus peut être associé à une organisation multiflot :

- d'unique segments de code, de données, etc. ;
- mais des flots d'exécution — des threads — multiples.

À chaque flot doit être associé le contenu des registres, une pile d'exécution mais une seule copie des données — code, etc. — est partagée. Les avantages de cette organisation sont :

- le partage des ressources permet d'économiser l'espace mémoire nécessaire et les manipulations sur cette espace ;
- l'utilisation plus efficace d'une architecture multiprocesseurs.

Les désavantages dépendent de la gestion de la communication interprocessus par l'OS. Les threads sont une solution intéressante pour les OS ayant une commutation de contexte coûteuse (Windows).

Processus ?
Signaux à destinations des processus
Clonage d'un processus
Mutation d'un processus
Terminaison d'un processus
Compléments

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

pthread_t pthread_id[3] ; /* On se pr\'epare \'a cr\'eer~$3$
                             flots d'instructions */
void *FredLife(void *i){  /* le code ex\'ecut\'e par les flots */
  printf("Je suis le %d-i\\`eme thread de num\\'ero %d.%l\n",
 * (int *)i, getpid(), pthread_self()) ;
  return NULL ;
}

int main(void){
  int i ;
  printf("Je suis le flot initial %d.%lu\n",getpid(),pthread_self());
  for(i=0; i<3 ; i++ )
    if(pthread_create(&(pthread_id[i]), NULL, &FredLife, &i)==-1)
      fprintf(stderr,"Impossible de cr\\'eer le %d-i\\`eme thread\n",
              i) ;
  pthread_join(pthread_id[2],NULL) ; /* Attend la terminaison */
  return 0 ;                         /* d'un autre flot      */
}
```


Processus ?
Signaux à destinations des processus
Clonage d'un processus
Mutation d'un processus
Terminaison d'un processus
Compléments

D'autres formes de clonage

Il existe plusieurs façons de créer un nouveau processus en dehors de l'utilisation de `fork`.

Par exemple, l'appel système `pid_t vfork(void)` se comporte comme `fork` si ce n'est qu'il n'y a ni copie de la pile ni des données : ces dernières sont partagées avec le processus père.

De plus, l'activité du processus père est suspendu jusqu'à ce que le fils termine (`exit` ou mute `execve`). Les signaux ne parviennent au père qu'après sa réactivation.

Cette commande évite principalement la copie de la table des pages — cf. le cours sur la gestion de la mémoire — qui est la tâche la plus coûteuse effectuée par l'appel `fork`.