OS 302: les processus

L. Gonnord (remplacement O. Aktouf)

Grenoble INP/Esisar

2022-2023

Les processus

Objectif :

- Connaître les caractéristiques système d'un processus.
- Programmation C de processus.

1. Notions de base
2. Commandes utilisateur·rice·s sous Linux pour les processus
3. Gestion Système des processus
4. Processus, programmation C

Programme et processus

Définition

Un processus est un programme en cours d'exécution.

- Chaque exécution d'un programme (éventuellement plusieurs fois le même) donne lieu à un processus différent ;
- A tout instant, un processeur exécute au plus un processus ;
- Dans un OS multi-tâches, le système alterne entre l'exécution de plusieurs processus ;
- Seuls les systèmes multi-cœur ou multi-processeur exécutent réellement plusieurs processus en même temps (un par cœur/processeur) ;
- Plus généralement, les processus partagent l'accès à différentes ressources : processeur mais aussi mémoire et périphériques.

Propriétés d'un processus

A un instant donné, un processus est caractérisé par de nombreuses informations, dont :

- Son état : exécution, suspendu, etc. ;
- Son identificateur ;
- Son compteur ordinal : indique la prochaine instruction à exécuter ;
- Sa pile d'exécution : mémorise l'empilement des appels de fonction ;
- Ses données en mémoire ;
- Toutes autres informations utiles à son exécution (E/S, fichiers ouverts, . . .)

Mode d'exécution

Deux modes d'exécution d'un processus :

- Mode noyau : accès sans restriction (manipulation de la mémoire, dialogue avec les contrôleurs de périphériques, . . .)
- Mode utilisateur : accès restreint, certaines instructions sont interdites (pas d'accès direct aux périphériques). Il peut être interrompu par d'autres processus.

⇒ Les appels systèmes permettent à un processus en mode utilisateur d'accéder (temporairement) à des fonctions nécessitant le mode noyau.

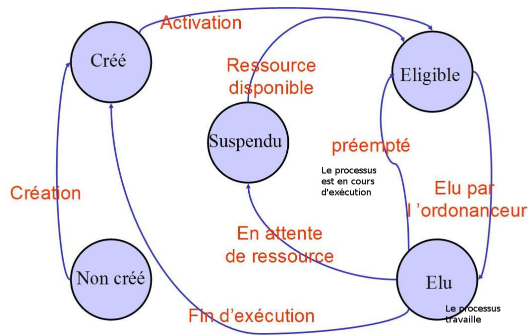
Identificateurs d'un processus

Plusieurs identificateurs sont associés à un processus (UNIX) :

- Numéro de processus (`pid`) ;
- Numéro du processus père (`ppid`) ;
- Identificateurs d'utilisateur, et de groupe, ...

Possibilité de connaître ces informations dynamiquement via des appels systèmes. ▶ : on a donc une hiérarchie de processus.

Evolution de l’état au cours de l’exécution



Etat d'un processus

- non créé : le code n'est pas en mémoire ;
- créé : le code est en mémoire, en attente d'activation ;
- éligible : l'activation a été demandée. En attente de l'accès au processeur ;
- élu : le processus s'exécute ;
- suspendu : une ressource est indisponible (allouée à un autre processus).

Préemption

- Le système interrompt l'exécution d'un processus pour en exécuter un autre (ressource indisponible, processus prioritaire, etc.) ;
- Il sauve le contexte du processus et le remplace par celui d'un autre ;
- Il rétablira plus tard le contexte du processus préempté.

Contexte d'un processus : valeur des registres, des variables, où on en est dans son exécution . . .

▶ Une préemption prend du temps : le temps de **commutation**

Classification des systèmes

- Gestion des processus
  - mono-tâche : CPU dédié à un processus ;
  - multi-tâche : CPU partagé entre les processus ;
  - multi-tâche préemptif : possibilité de suspendre puis rétablir un processus en cours d'exécution.
- Gestion des utilisateurs
  - mono-utilisateur : pas de cohabitation entre utilisateurs ;
  - multi-utilisateur : cohabitation possible.

Exemples de système d'exploitation

- MS-DOS : mono-utilisateur, mono-tâche ;
- Windows : mono-utilisateur, multi-tâche ;
- WinNT : mono-utilisateur, multi-tâche préemptif ;
- UNIX : multi-utilisateur, multi-tâche préemptif.

1. Notions de base

2. Commandes utilisateur·rice·s sous Linux pour les processus

3. Gestion Système des processus

4. Processus, programmation C

Visualisation (statique) des processus 1/2

Sous UNIX :

- l'information sur les processus en cours peut être obtenue par :

  `ps [options]`

- Attention les options sont de deux types (bsd/gnu).
- Quelques exemples :
  - `ps -ef` : every,full.
  - `ps -eo pid,ppid,tt,user,fname | grep laure`

Création de processus

En Unix :

- Au lancement du système, un unique processus ;
- L'appel système fork crée un nouveau processus ;
- Le processus père et le processus fils s'exécutent en parallèle ;
- Le père et le fils peuvent à nouveau effectuer un fork ;

⇒ Hiérarchie arborescente de processus.

Visualisation (statique) des processus 2/2

```
laure@sorlin:~$ ps
  PID TTY TIME CMD
18220 pts/13 00:00:00 bash
18270 pts/13 00:00:00 ps
laure@sorlin:~: ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 07:59 ? 00:00:00 /sbin/init
root 2 0 0 07:59 ? 00:00:00 [kthreadd]
root 3 2 0 07:59 ? 00:00:00 [ksoftirqd/0]
...
laure 19717 1909 0 10:34 ? 00:00:00 evince ...
laure 19856 3014 0 10:34 pts/14 00:00:00 bash
laure 20093 5867 3 10:35 ? 00:00:01 /usr/lib/chromium-browser/chro
laure 20238 19856 0 10:36 pts/14 00:00:00 ps -ef
```

Visualisation des processus

Sous UNIX :

- visualisation dynamique des processus :

`top [options]`

```
laure@sorlin: ~
top - 10:18:26 up  2:19,  5 users,  load average: 0,14, 0,21, 0,23
Tasks: 238 total,   3 running, 235 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3,9 us,  1,2 sy,  0,0 ni, 94,9 id,  0,1 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   8059004 total,  3594672 used,  4464332 free,   205888 buffers
KiB Swap: 16538620 total,        0 used, 16538620 free,  1431232 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 8413 laure     20   0  825m 175m  54m S   3,7  2,2   1:05.49 chromium-browse
 1068 root      20   0  470m 137m 119m S   2,7  1,7   2:11.29 Xorg
 2353 laure     20   0 1572m 104m  31m S   2,7  1,3   3:30.85 compiz
 2407 laure     20   0  645m 162m  49m S   2,3  2,1   3:25.17 skype
 1002 root      20   0  343m 1468 1048 S   1,3  0,0   0:24.23 glided
 3014 laure     20   0  664m  25m  15m S   1,3  0,3   0:38.79 gnome-terminal
```

Interruption des processus

Sous UNIX :

`kill` [numéro_de_signal] numéro_de_processus

- `kill -l` donne l'ensemble des signaux disponibles
- exemple de signaux :
  - SIGHUP (1) : émis à tous les processus associés à un terminal lorsque celui-ci se déconnecte
  - SIGINT (2) : émis à tous les processus associés à un terminal lorsque <`ctrl + C` > est tapé
  - SIGKILL (9) : tue un processus quel que soit son état. C'est l'arme absolue.
  - SIGTERM (15) : signal de terminaison normale d'un processus

Lancement en arrière plan

- Il est possible de lancer une commande sans que le shell courant en attende la terminaison :

  commande &

  - Elle ne peut plus lire au clavier ;
  - Les sorties standards sont toujours associées, par défaut, au terminal ;
  - Elles ne sont plus interruptibles à partir du clavier (`ctrl-c` par exemple).
- Connaître les processus lancés en arrière plan (pour un shell donné) :

  `jobs`

Changement d'état d'un processus

Sous UNIX (%n est le numéro obtenu par `jobs`)

Exécution en avant plan

fg [%n]

^Z

fg [%n]

bg [%n]

^C

suspendu

Exécution en arrière plan

kill %n

(*)

kill %n

Tué

(*)
stop [%n] en csh
kill -SIGSTOP %n en sh

Démo

Les PID seront récupérés avec la commande `ps -ef` lancée dans un autre terminal.

- Lancement du logiciel `eog` et terminaison avec CTRL-c ou `kill -SIGINT`.
- Lancement du logiciel `eog` en arrière plan. CTRL-c ne permet pas de terminer. `kill -SIGINT` le peut (ou `kill` sans option (SIGTERM).
- Lancement du logiciel `eog` puis suspendu (CTRL-z) puis en arrière plan (`bg`), puis suspendu avec `kill -SIGSTOP`, puis `fg`.

eog, ou `xclock -update 1` c'est plus drôle. :-)

1. Notions de base

2. Commandes utilisateur·rice·s sous Linux pour les processus

3. Gestion Système des processus
   - Ordonnancement

4. Processus, programmation C

Problématiques de la gestion de processus

Difficulté principale : gestion de l'accès concurrent à diverses ressources :

- Accès au processeur : ordonnancement (now)
- Lecture/écriture de données : (plus tard)
  - Assurer la cohérence des données ;
  - Gérer les synchronisations lecteur/écrivain ;
  - Eviter les blocages.

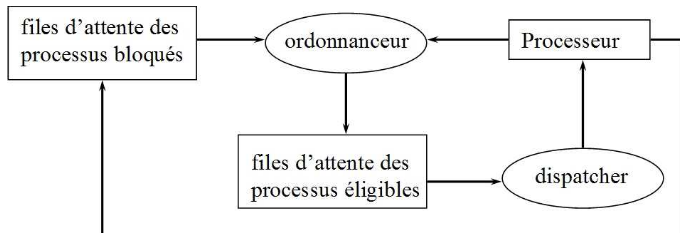
1. Notions de base
2. Commandes utilisateur·rice·s sous Linux pour les processus
3. Gestion Système des processus
   - Ordonnancement
4. Processus, programmation C
   - Utilitaires C de gestion de processus
   - Création d'un nouveau processus: `Fork`
   - Recouvrement d'un processus: `Exec`
   - Communication par tuyau: `Pipe`

Définition

- L'ordonnancement (scheduling) est une fonction fondamentale d'un système d'exploitation ;
- A la base de la programmation multi-tâche :
  - Plusieurs processus en mémoire en même temps ;
  - Assignation du processeur à un processus en fonction de certains critères ;
- Objectif = optimiser l'utilisation concurrente du processeur.

Ordonnanceur vs dispatcher

- L'ordonnanceur est responsable de l'organisation des file d'attente des tâches éligibles ;
- Le dispatcher réalise l'élection d'un processus et le changement de contexte associé.



Décisions de l'ordonnanceur

Les décisions de l'ordonnanceur peuvent avoir lieu dans les circonstances suivantes :

- Quand un processus passe de l'état "élu" à "suspendu" (attente de disponibilité d'une ressource) ;
- Quand un processus passe de l'état "élu" à l'état "éligible" (à cause d'une interruption) ;
- Quand un processus passe de l'état "suspendu" à l'état "éligible" (libération d'une ressource) ;
- Quand un processus se termine.

Pour aller plus loin

Différents algorithmes (FIFS, tourniquet), ... (voir la littérature).

http://fr.wikipedia.org/wiki/Ordonnancement_dans_les_syst%C3%A8mes_d'exploitation

À retenir de cette partie

- Qu'est-ce qu'un ordonnanceur ? (partie de l'OS)
- Quels sont les problèmes qui se posent lorsque deux processus communiquent (programmation système) ?

1. Notions de base
2. Commandes utilisateur·rice·s sous Linux pour les processus
3. Gestion Système des processus
4. Processus, programmation C
   - Utilitaires C de gestion de processus
   - Création d'un nouveau processus: `Fork`
   - Recouvrement d'un processus: `Exec`
   - Communication par tuyau: `Pipe`

Observation des processus

Sous Unix les processus sont organisés de façon hiérarchique:

- Chaque processus est identifié par son PID (Process Identifier)
- Chaque processus peut créer des processus appelés fils.
- Tous les processus ont accès à l'id de leur père PPIC (Parent Process Identifier).

Arborescence de processus

- Au lancement du système, un unique processus ;
- Création d'un nouveau processus :
  - Duplication = création d'un fils identique au père (fork);
  - Le fils a les mêmes propriétés que le père (code, données, pile, droits, ...)
  - Recouvrement du code du fils par le programme voulu (exec);
  - Éventuellement, communication entre père et fils (pipe).

1. Notions de base

2. Commandes utilisateur·rice·s sous Linux pour les processus

3. Gestion Système des processus
   - Ordonnancement

4. Processus, programmation C
   - Utilitaires C de gestion de processus
   - Création d'un nouveau processus: `Fork`
   - Recouvrement d'un processus: `Exec`
   - Communication par tuyau: `Pipe`

Accès aux propriétés du processus

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

- `pid_t getpid(void)` : l'identifiant du processus ;
- `pid_t getppid(void)` : l'identifiant du père du processus ;
- `char *getcwd(char *buf, size_t size)` : le répertoire de travail ;
- `int chdir(const char *path)` : changer le répertoire de travail ;
- ...

Divers

- `unsigned int sleep(unsigned int seconds)` : endormir le processus pour quelques secondes ;
- `void exit(int status)` : termine le processus courant + renvoie un code de retour.

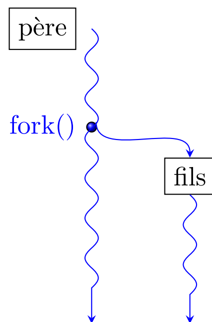
1. Notions de base
2. Commandes utilisateur·rice·s sous Linux pour les processus
3. Gestion Système des processus
   - Ordonnancement
4. Processus, programmation C
   - Utilitaires C de gestion de processus
   - Création d'un nouveau processus: `Fork`
   - Recouvrement d'un processus: `Exec`
   - Communication par tuyau: `Pipe`

Primitive

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

- Crée dynamiquement un processus ;
- Processus qui exécute le fork = père du processus créé (fils) ;
- Fils = <u>copie exacte</u> du père ;
- Exécution concurrente du père et du fils (entrelacement possible des instructions).

Schéma d'exécution (1)



Propriétés du fils

Le fils hérite de son père :

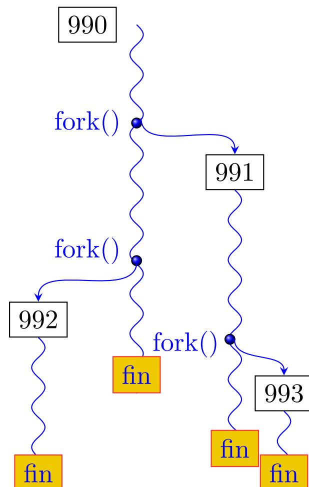
- le même code ;
- une copie des données (variables et valeurs) ;
- le même environnement (e.g. répertoire de travail) ;
- la même priorité ;
- les mêmes droits ;
- les descripteurs de fichiers ;
- la gestion des signaux.

Exemple (création)

```c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf(" 1 -proc %d, pere %d\n", getpid(), getppid());
  fork(); /* 2 processus */
  printf(" 2 -proc %d, pere %d\n", getpid(), getppid());
  fork(); /* 4 processus ! */
  printf(" 3 -proc %d, pere %d\n", getpid(), getppid());
}
```

Schéma d'exécution de l'exemple

```
laure@sorlin:~/.../Cours/$ ./fork1
 1 -proc 990, pere 11539
 2 -proc 990, pere 11539
 3 -proc 990, pere 11539
 2 -proc 991, pere 990
 3 -proc 992, pere 990
 3 -proc 991, pere 1909
 3 -proc 993, pere 991
laure@sorlin:~/.../Cours/$
```



Exemple `fork1`- 3/3

Sur l’exemple précédent, bien observer les dates d’impression. D’autres exécutions sont possibles, lesquelles ?

Terminaison d’un processus

Les processus terminent lorsqu’ils arrivent à la fin de leur code. Dans l’exemple précédent, le processus de pid 990 termine avant que le processus 991 n’exécute son `printf(3...)`, le père de 991 est alors le processus init (ou init-user).

Distinguer père et fils

Grâce à la valeur de retour du fork :

- Le PID du fils est retourné au père ;
- 0 est retourné au fils ;
- -1 est retourné au père si erreur.

Exemple (distinction père/fils)

```
#include <sys/types.h> /*programme creerproc */
int main() {
  pid_t n= fork();
  if (n == -1) { perror ("creation de processus"); exit(1); }
  if (n == 0) { /* seul le processus fils execute ceci */
    printf("dans fils n = %d\n", n );
    printf("PID fils %d\n", getpid());
  } else{ /* seul le processus pere execute ceci */
    printf("dans pere n = %d\n", n );
    printf("PID pere %d\n", getpid());
    printf("fin proc pere\n");
  }
```

Exemple (distinction père/fils) (2)

```
laure@sorlin:~/../Cours$ ./creeproc
dans pere n = 2577
PID pere 2576
fin proc pere
dans fils n = 0
PID fils 2577
laure@sorlin:~/../Cours$
```

- Ici (dans cette exécution) le père se termine avant le fils ;
- Le fils est alors “adopté” par le processus init (PID 1) ou init-user.

Synchronisation

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

- `pid_t wait (int *status)`
  - Suspend le processus jusqu'à ce qu'un de ses fils termine ;
  - Valeur de retour :
    - -1 si pas de fils ;
    - Le PID du fils terminé sinon.
- `pid_t waitpid(pid_t pid, int *status, int opts)`
  - Attend le fils de PID pid.
  - Le wait peut être bloquant ou non (opts)
- status “décrit” l'état de terminaison du fils.

Exemple (synchronisation)

```
void main() /* sync1.c */
{
  if (fork() == 0)
    {
      printf("fils -PID = %d\n", getpid());
      exit(3);
    } else {
      pid_t m; int n;
      m = wait(&n);
      printf("fin proc %d avec code %d\n", m, n);
    }
}
```

Exemple : synchronisation (2)

```
void main() { /* sync2.c */
  if (fork() == 0) {
    printf ("PID fils = %d\n", getpid());
    for (;;); /* on va terminer ce processus par un kill */
  }
  else {
   int n ; pid_t m ;
   m = wait(&n); // on récupère dans n le code de retour
   printf("fin proc %d avec code %d\n", m, n);
  }
}
```

Avec waitpid

```
waitpid(-1,&n, 0); // -1 pour un fils quelconque
// voir le man pour le troisième argument
```

1. Notions de base
2. Commandes utilisateur·rice·s sous Linux pour les processus
3. Gestion Système des processus
   - Ordonnancement
4. Processus, programmation C
   - Utilitaires C de gestion de processus
   - Création d'un nouveau processus: `Fork`
   - Recouvrement d'un processus: `Exec`
   - Communication par tuyau: `Pipe`

Primitives

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
...
```

- Exécute un nouveau programme ;
- Ce programme recouvre l'ancien (pas de processus créé) ;
- Conservation des propriétés système :
  - PID, PPID, priorité ;
  - descripteurs de fichiers ;
  - ...

Un exemple simple

```c
int main() /* my_ls.c */
{
  if (execlp("ls", "ls", "-l", NULL) ==-1)
    {
      perror ("echec execlp\n");
      exit(1);
    }
  printf("cette ligne ne sera jamais affichee\n");
  exit(0);
}
```

- Liste des arguments, comme argc/argv.

Shell simple

Schéma simplifié d’un Shell :

```
tant que vrai faire
   lire(commande)
   Si (fork()==0)
       exec(commande)
   Sinon
       wait(lefils)
fin_tq
```

Si vous voulez implémenter, suivez le lien

https://nlouvet.gitlabpages.inria.fr/lifasr5/tp/lifasr5-tp08.pdf

commande system()

res = system("emacs");

- A utilise `fork()` pour se dupliquer et créer B
  - **A attend la fin de B**
  - B lance la commande
  - B se termine
- **A récupère le retour de B**
- A reprend son cours

1. Notions de base
2. Commandes utilisateur·rice·s sous Linux pour les processus
3. Gestion Système des processus
   - Ordonnancement
4. Processus, programmation C
   - Utilitaires C de gestion de processus
   - Création d'un nouveau processus: `Fork`
   - Recouvrement d'un processus: `Exec`
   - Communication par tuyau: `Pipe`

Duplication de descripteur

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- Duplique oldfd ;
- dup retourne la plus petite valeur possible parmi les fd disponibles;
- dup2 :
  - ferme newfd ;
  - copie oldfd vers newfd ;
- Retourne -1 si échec ;
- Utilisation : redirection d'E/S standard.

Exemple

```
main() {
  int desc;
  if ( (desc = creat("erreur.txt", 0666)) == -1){
    perror("creation fichier erreur impossible");
    exit(1);
  }
  /* redirection de la sortie erreur standard */
  close(2);
  dup(desc);
  fprintf(stderr, "message d'erreur redirige dans erreur.txt\n");
}
```

Communication par FIFO (file/tube/pipe)

- Communication typique 1 producteur $\rightarrow$ 1 consommateur ;
- Taille limitée ;
- On écrit d'un côté...
- ...et on lit de l'autre (dans le même ordre que l'écriture) ;
- Le système bloque :
  - La lecture si tube vide ;
  - L'écriture si tube plein.

Tubes en POSIX

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

- Tube = 1 fichier, mais pas de référence dans le système ;
- On lit/écrit avec les primitives classiques sur les fichiers ;
- Tube = 2 descripteurs (sur le même fichier) :
  - fd[0] pour la lecture
  - fd[1] pour l'écriture
- Retourne 0 si ok, -1 si erreur (ex : trop de descripteurs ouverts) ;
- S'utilise typiquement avec fork (communication père↔fils) ;
- Il faut systématiquement fermer les descripteurs de tubes non utilisés

Exemple

```
int main(){
  int tube[2]; int i, pid , ret;
  if (pipe(tube)<0) { perror ("erreur ouverture pipe"); exit(1);}
  if ((pid=fork())<0) { perror ("erreur fork"); exit(1); }
  if (pid !=0) {
    close(tube[0]); /*pere ne lit pas ds le tube */
    for (i=1; i<=1000 ; i++) write(tube[1], &i, sizeof(int));
    close(tube[1]);
    wait(&ret);}
  else {
    close(tube[1]); /*fils n'ecrit pas ds tube */
    while(read(tube[0],&i,sizeof(int))==sizeof(int)) printf("%d ", i
```

Conclusion de cette partie

Le langage C fournit des primitives pour :

- Créer, inspecter des processus.
- Exécuter des commandes système.
- Communiquer entre processus (pipe, dup).

Ce que l'on n'a pas vu :

- La notion de processus léger (thread) : partage d'une même zone de "mémoire virtuelle", et son utilisation en C.