Les processus

Louis Mandel
louis.mandel@lri.fr
Laboratoire de Recherche en Informatique
Université Paris-Sud 11

IFIPS
Cycle ingénieur de la filière étudiant
année 2007/2008

Processus

- 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 nécessaire à un processus
  - Ressources matérielles : processeur, périphérique ...
  - Ressources logicielles :
    - code
    - contexte d'exécution : compteur ordinal, fichiers ouverts ...
    - mémoires
- Mode d'exécution
  - utilisateur
  - noyau (ou système ou superviseur)

Processus

- Information sur les processus
  - Commandes shells
    - `ps`
    - `pstree`
  - Système de fichier
    - `/proc`

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 group

    ```
    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);
    ```

Attributs d'un processus

- Répertoire de travail
  - Origine de l'interprétation des chemins relatifs

    ```
    char *getcwd(char *buf, size_t size);
    ```

  - Peut être changé

    ```
    int chdir(const char *path);
    ```

Attributs d'un processus

- Date de création : en secondes depuis le 1^{er} janvier 1970 (EPOCH)
- Temps CPU consommés
  - par le processus / par ses fils terminés
  - en mode utilisateur / en mode
  - `clock_t times(struct tms *buf);`
    - retourne le temps écoulé depuis une date fixe (démarrage du système par exemple)
    - `clock_t` : tics d'horloge survenus alors que le processus était actif
    -
```
struct tms {
  clock_t tms_utime;  /* durée utilisateur          */
  clock_t tms_stime;  /* durée système              */
  clock_t tms_cutime; /* durée utilisateur des fils */
  clock_t tms_cstime; /* durée système des fils     */
};
```
    - conversion en secondes par division par `_SC_CLK_TCK`

Exemple : mtime.c

```
static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;
float tics_to_seconds(clock_t tics) {
  return tics/(float)sysconf(_SC_CLK_TCK);
}
void start_clock() { st_time = times(&st_cpu); }
void end_clock() {
  en_time = times(&en_cpu);
  printf("Real Time: %.2f, User Time %.2f, System Time %.2f\n",
 tics_to_seconds(en_time - st_time),
 tics_to_seconds(en_cpu.tms_utime - st_cpu.tms_utime),
 tics_to_seconds(en_cpu.tms_stime - st_cpu.tms_stime));
}
```

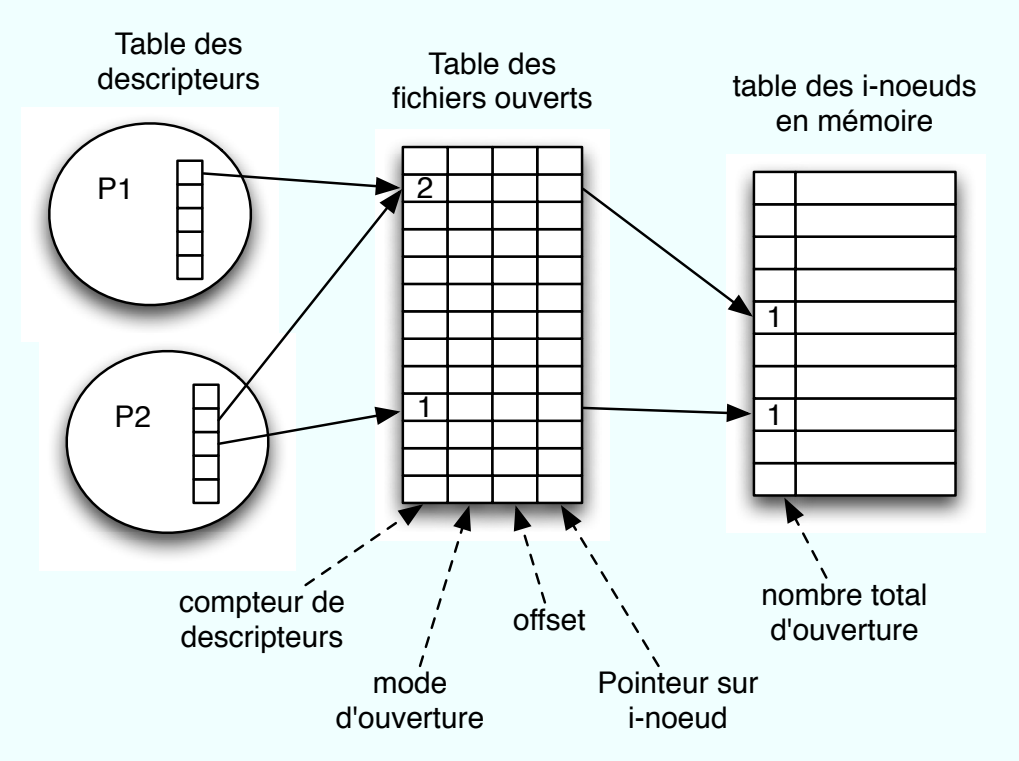
Attributs d'un processus

- Masque de création des fichiers
  - `mode_t umask(mode_t mask);`
  - fixe le masque de création de fichiers à la valeur `mask & 0777`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char **argv) {
  if (argc != 3) return 1;
  if (creat(argv[1], S_IRWXU|S_IRWXG|S_IRWXO ) == -1) {
    perror("creat"); return 1;
  }
  umask(S_IRWXG|S_IRWXO);
  if (creat(argv[2], S_IRWXU|S_IRWXG|S_IRWXO ) == -1) {
   perror("creat"); return 1;
  }
  return 0;
}
```

Attributs d'un processus

- Table des descripteurs de fichiers



- Limite : `OPEN_MAX`
- Acquisition : `open`, `creat`, `dup`, `dup2`, `fcntl`, `pipe` et `socket`
- Libération : `close`

Attributs d'un processus

- Variables d'environnement
  - `extern char **environ`
  - Exemple :

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
int main (int argc, char *argv[]) {
  char **envp = environ;
  while (*envp != NULL) { printf("%s\n", *envp); envp++; }
  exit(EXIT_SUCCESS);
}
```

  - Récupérer la valeur d'une variable :

```
char *getenv(const char *var_name);
```

  - Changer la valeur d'une variable :

```
int putenv(const char *string);
```

Attributs d'un processus

- Consultation et changement d'attributs variables
  - `int getrlimit(int resource, struct rlimit *rlim);`
  - `int setrlimit(int resource, const struct rlimit *rlim);`
    - Exemple de ressources : `RLIMIT_CORE`, `RLIMIT_CPU`, etc.
    - ```
      struct rlimit {
        rlim_t rlim_cur; /* limite souple */
        rlim_t rlim_max; /* limite stricte (plafond de rlim_cur) */
      };
      ```
  - `long sysconf (int name);`
    - Exemple de noms : `_SC_CLK_TCK`, `_SC_OPEN_MAX`, etc.
      ```

Gestion de processus

- Parallélisme asynchrone
  - Plusieurs processus tournent en parallèle sur la machine.
  - Sur une machine mono-processeur, le système se charge d'entrelacer les processus en donnant un peu de temps à chacun (ordonnancement) : donne l'illusion du parallélisme.
  - L'entrelacement est imprévisible et non reproductible : l'exécution d'un programme peut être interrompue de façon autoritaire par le système pour donner la main à un autre.
  - *Race condition* : Deux entrelacements possibles de deux processus changent le sens du programme (de façon non voulue) : accès aux ressources, lecture/écriture d'une même donnée, etc.
  - Changer la priorité (ajoute une valeur de gentillesse, i.e. diminue la priorité. Valeurs négatives possibles avec privilège).
    - `int nice(int inc);`

Création de processus

Création : `fork` (`fork1.c`)

- La création de processus se fait par clonage
- ```
  #include <sys/types.h>
  #include <unistd.h>
  pid_t fork(void);
  ```
- Exemple :

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

- 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
- Exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  pid_t id;
  id = fork();
  printf("id = %d, pid = %d, ppid = %d\n",
         id, getpid(), getppid());
  return 0;
}
```

fork : utilisation typique (fork3.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.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 père est %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());
}
```

- Comme la mémoire est copiée :
  - les données sont copiées

```
#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 :
    sleep(1);
    printf("Fils : (%d, %d)\n", glob, loc);
    break;
  default :
    glob++; loc++;
    printf("Pere : (%d, %d)\n", glob, loc);
  }
  printf("[%d] Je termine\n", getpid());
}
```

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

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

  - Il faut vider les tampons avant `fork` (par un appel à `fflush`)

- Comme la mémoire est copiée :
  - les références vers les ressources systèmes sont copiées
  - les ressources systèmes sont partagées.

```
int main(int argc, char *argv[]) {
  int desc;  char buf [10];
  if ((desc = open(argv[1], O_RDONLY)) == -1) { perror("open"); exit(1); }
  switch (fork()) {
  case -1 : perror("Creation de processus"); return 1;
  case 0 :
    read(desc, buf, 3); buf[3] = '\0';
    printf("Fils : '%s'\n", buf);
    break;
  default :
    sleep(1);
    read(desc, buf, 3); buf[3] = '\0';
    printf("Pere : '%s'\n", buf);
  }
  close(desc); exit(0);
}
```

`fork`

- Attributs non copiés :
  - Numéro de processus
  - Numéro de processus du père
  - Temps d'exécution
  - Priorité du processus
  - Verrous sur les fichiers
- Coût de la copie mémoire ?
  - Sémantique = copie
  - Performance = pas de copie systématique
  - *copy on write* des pages mémoires

Terminaison des fils : `wait`

- Terminaison d'un processus
  - Appel système : `_exit`
  - Appel à la fonction de bibliothèque : `exit`
  - Positionnement d'une valeur de retour
- Le processus père peut consulter la valeur de retour
- Attente de la terminaison d'un fils
  - `pid_t wait(int *pstatus);`
    - retourne le PID de fils ou -1 en cas d'erreur (n'a pas de fils)
    - bloquant si aucun fils n'a terminé
    - `*pstatus` renseigne sur la terminaison du fils

Terminaison des fils : `wait`

- Renseignements concernant la terminaison d'un fils
  - Rangés dans l'entier `status` pointé par `pstatus`
  - Raison de la terminaison
    - le processus a fait un `exit` : `WIFEXITED(status)`
    - le processus a reçu un signal : `WIFSIGNALED(status)`
    - le processus a été stoppé : `WIFSTOPPED(status)`
  - valeur de retour (si `WIFEXITED(status)`)
    - 8 bits de poids faibles
    - accessible par la macro `WEXITSTATUS(status)`
  - numéro du signal ayant provoqué la terminaison (si `WIFSIGNALED(status)`) / l'arrêt (si `WIFSTOPPED(status)`)
    - accessible par les macros `WTERMSIG(status)` / `WSTOPSIG`

Terminaison des fils : wait (wait1.c)

```
int main(int argc, char **argv) {
  int status;
  pid_t pidz;
  switch (fork()) {
  case -1 : perror("Creation de processus"); return 1;
  case 0 : printf("[%d] fils eclaire\n", getpid()); exit(2);
  default :
    pidz = wait(&status);
    if (WIFEXITED(status)) {
      printf("[%d] mon fils %d a termine normalement\n", getpid(), pidz);
      printf("[%d] code de retour : %d\n", getpid(), WEXITSTATUS(status));
    } else
      printf("[%d] mon fils %d a termine anormalement\n", getpid(), pidz);
  }
}
```

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

Processus orphelins (wait2.c)

```
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(2);
    printf("[%d] Pere : %d\n", getpid(), getppid());
    exit(0);
  default :
    sleep(1);
    printf("[%d] fin du pere\n", getpid());
    exit(0);
  }
}
```

Processus zombis

- Zombi = état d'un processus
  - ayant terminé
  - non réclamé par son père
- Il faut éviter les zombis
  - Le système doit garder des informations relatives aux processus pour les retournées aux pères.
  - Encombre la mémoire

- Comment éviter les zombis si le père ne s'intéresse pas à la terminaison de ses fils ?