Systèmes d'Exploitation

Communications entre processus

Stefan Balev

`Stefan.Balev@univ-lehavre.fr`

UFR Sciences – Université du Havre

Plan (1)

- Processus UNIX
- Communication par tubes et par signaux
- Problème de synchronisation
  - étude des problèmes classiques
  - sémaphores
  - moniteurs

Plan (2)

- IPC System V
  - files de messages
  - mémoire partagée
  - sémaphores
- Threads POSIX
- Programmation réseau
  - protocoles, services et ports
  - utilisation de sockets

Bibliographie

- C. Blaess. Programmation système sous Linux
- J.-M. Rifflet. La programmation sous Unix
- J.-M. Rifflet. La communication sous Unix
- A. Silberschatz, P. B. Galvin. Principes des systèmes d'exploitation
- doc en ligne, pages man, . . .

Processus

Introduction

Processus – programme *en exécution*

processus $\neq$ programe

- programme – entité passive
- processus – entité active

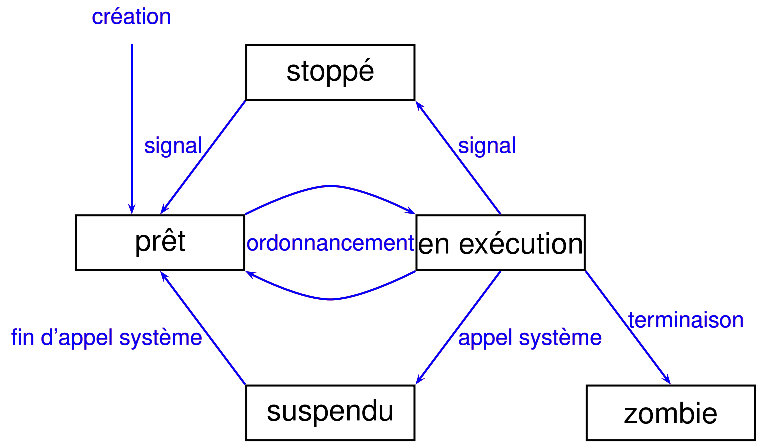
Intérêt :

- multi-tâche
- multi-utilisateur
- utilisation plus efficace de ressources

Contexte

- segment de code (programme)
- valeurs des registres (y compris compteur ordinal)
- pile
- segment de données
- ressources associées
- attributs (pid, priorité,. . .)

Etat d'un processus



Création d'un processus

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)      – duplication du processus appelant
pid_t getpid(void)    – pid de processus appelant
pid_t getppid(void)   – pid du père
```

$$\texttt{fork()}\ \text{renvoie} \begin{cases} 0 & \text{au fils} \\ \text{pid de fils} & \text{au père} \\ -1 & \text{en cas d'échec} \end{cases}$$

Structure arborescente de processus. Racine – processus 1 (init)

Exemple

```
int main(void)
{  pid_t pid;

   pid = fork();
   if (pid == -1)
   {  perror("fork"); exit(1);
   }
   if (pid == 0)
   {  printf("fils: moi %d, mon père %d\n",
         getpid(), getppid());
   }
   else
   {  printf("père: moi %d, mon fils %d\n",
         getpid(), pid);
   }
   return 0;
}
```


Terminaison d'un processus

```
#include <stdlib.h>
void exit(int status)
```

- libération de ressources
- les fils sont rattachés au processus 1 (init)
- le signal `SIGCHLD` est envoyé au père

Attente d'un fils

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options)
```

`options`:

`WNOHANG` – retour immédiat

`WUNTRACED` – prise en compte de changement d'état des fils

Interpretation de `status`: `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`

```
wait(&status);
printf("le code de retour de fils est %d\n", WEXITSTATUS(status));
```

Exemple

```
n = 1;
pid = fork();
if (pid)
{   wait(NULL);
    printf("%d\n", n);
}
else
{   n++;
    printf("%d\n", n);
}
```

NB : Le père et le fils ne partagent pas l'espace d'adressage

Exemple

```
fd = creat("toto",0660);
if (fork())
{   wait(NULL);
    write(fd,"père\n",5);
}
else
    write(fd,"fils\n",5);
```

NB : Le fils hérite les descripteurs de fichiers du père

Changement de code

```
#include <unistd.h>
execl(const char *path, const char *arg,...,NULL);
execle(const char *path, const char *arg,...,NULL,
       char *const envp[]);
execlp(const char *path, const char *arg,...,NULL);
execv(const char *path, char *const argv[]);
execve(const char *path, char *const argv[],
       char *const envp[]);
execvp(const char *path, char *const argv[]);
```

NB : En cas de succès les fonctions `exec` ne retournent jamais.

Exemple

```
pid = fork();
if (pid == -1) { perror("fork"); exit(1); }
if (pid == 0)
{   execlp("ls", "ls", "-l", NULL);
    perror("execlp"); exit(1);
}
else
{   wait(&st);
    if (WIFEXITED(st) && WEXITSTATUS(st) == 0)
       printf("ok\n");
    else
       printf("échec\n");
    exit(0);
}
```

Signaux

Introduction (1)

Signal : mécanisme permettant aux processus de réagir aux événements

signal $\leftrightarrow$ événement

Génération d'un signal :

- exception matérielle – `SIGSEGV`, `SIGFPE`
- par l'utilisateur – `SIGINT` (Ctrl-C), `SIGSTP` (Ctrl-Z)
- par l'appel système `kill`
- par le noyau – `SIGSTOP`, `SIGALARM`

Réception non synchrone $\Rightarrow$ delai important possible

Introduction (2)

Action par défault :

- ignorer le signal
- terminer le processus
- terminer et générer fichier core
- suspendre le processus

Le processus peut modifier le comportement par défault.

NB : `SIGKILL`, `SIGSTOP` – comportement non modifiable

`SIGUSR1`, `SIGUSR2` – à la disposition du programmeur

Exemple

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

int main(void)
{  int s;

   for (s = 1; s < NSIG; s++)
      printf("%d:%s\n", s, strsignal(s));
   return 0;
}
```

Fonctions de gestion

Emission d'un signal

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Attente d'un signal

```
#include <unistd.h>
int pause(void);
```

Alarme (SIGALRM)

```
#include <unistd.h>
unsigned int alarm(long seconds);
```


Déroutement d'un signal

```
#include <signal.h>
void (*signal(int sig, void (*handler)(int)))(int);
```

Exemple :

```
void myhandler(int);
signal(SIGUSR1, myhandler);
```

`handler`

- `SIG_IGN` – ignorer
- `SIG_DFL` – comportement par défaut
- une fonction définie par l'utilisateur

Valeur de retour : `handler` ou `SIG_ERR`

Exemple

```
void handler(int sig)
{  printf("Signal reçu : %s\n", strsignal(sig));
}

int main(void)
{  if (signal(SIGUSR1, handler) == SIG_ERR)
   {  perror("signal");
      exit(1);
   }
   if (signal(SIGUSR2, handler) == SIG_ERR)
   {  perror("signal");
      exit(1);
   }
   signal(SIGKILL, hangler);
   while(1) pause();
}
```


Exemple – exécution

```
[stefan@ecrins examples]$ ex1 &
[7] 3381
[stefan@ecrins examples]$ kill -USR1 3381
Signal reçu :  User defined signal 1
[stefan@ecrins examples]$ kill -USR2 3381
Signal recu :  User defined signal 2
[stefan@ecrins examples]$ kill -KILL 3381
[7]+ Processus arrêté ex1
[stefan@ecrins examples]$
```

Exemple – gestion des alarmes

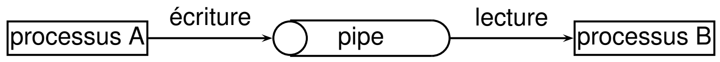
```
void handler(int);
int t;

int main(void)
{  signal(SIGALRM, handler);
   printf("Vous avez 10s pour frapper Entrée\n");
   t = 10; alarm(2);
   getchar(); alarm(0);
   printf("Bien joué!\n");
   return 0;
}

void handler(int sig)
{  alarm(2); t -=2;
   if (t) printf("Il vous reste %ds\n", t);
   else
   {  printf("Trop tard!\n");
      exit(1);
   }
}
```


Tubes de communication (pipes)

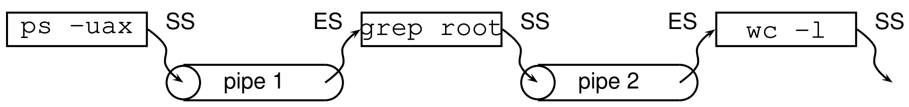
Introduction



- file d'octets (FIFO)
- communication non structurée
  - lecture et écriture indépendantes
  - pas d'info sur la taille, l'emetteur ou destinaire
  - permet une lecture pour plusieures écritures et vice versa
- gestion - integrée dans le système de fichiers
  - acces – par descripteurs de fichiers
  - lecture/ecriture – par `read/write`

Exemple (shell)

```
[stefan@ecrins examples]$ ps -uax|grep root|wc -l
     29
```



Création d'un tube

```
#include <unistd.h>
int pipe(int fd[2]);
```

`fd[0]`, `fd[1]` – descripteurs de fichiers

Ils représentent les deux extrémités du tube

`fd[0]` – lecture

`fd[1]` – écriture

Lecture / écriture

`read(fd[0],...)`

- renvoie 0 si le tube est vide et `fd[1]` est fermé
- est bloquant si le tube est vide et `fd[0]` est ouvert

`write(fd[1],...)`

- génère `SIGPIPE` si `fd[0]` est fermé
- est bloquant si le tube est plain

Exemple

```
int fd[2], in, out;

pipe(fd);
if (fork())
{   close(fd[0]);
    in = 123;
    printf("père envoie %d\n", in);
    write(fd[1], &in, sizeof(int));
    close(fd[1]);
}
else
{   close(fd[1]);
    read(fd[0], &out, sizeof(int));
    close(fd[0]);
    printf("fils reçu %d\n", out);
}
```


Exemple

```
pipe(fd);
if (fork())
{   write(fd[1], "abc", 3);
    close(fd[1]);
    wait(NULL);
}
else
{   while(read(fd[0], &c, 1))
        putchar(c);
}
```

NB : Toujours fermer les descripteurs non utilisés

Tubes nommés

Avantage : permettent communication entre processus indépendants

Création :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode)
```

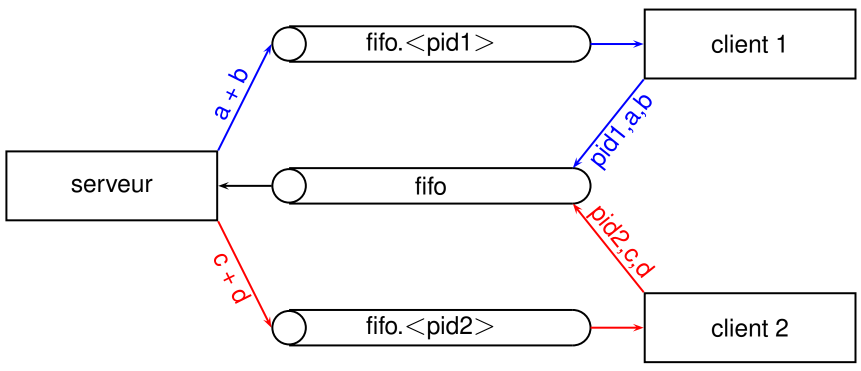
Suppression :

```
#include <unistd.h>
int unlink(const char *path)
```

Exemple :

```
[stefan@ecrins examples]$ mkfifo fifo
[stefan@ecrins examples]$ ll fifo
prw-rw-r--  1 stefan   stefan    0 Sep 29 16:43 fifo
```


Exemple : Client – Serveur



Serveur

```
int fdq, fdr, buf[3], res;
char nom[20];

mkfifo("fifo", 0666);
fdq = fopen("fifo", O_RDONLY);
while(1)
{  read(fdq, buf, 3 * sizeof(int));
   res = buf[1] + buf[2];
   sprintf(nom, "fifo.%d", buf[0]);
   fdr = open(nom, O_WRONLY);
   write(fdr, &res, sizeof(int));
   close(fdr);
}
```


Client

```
int fdq, fdr, buf[3], res;
char nom[20];

buf[0] = getpid();
buf[1] = random(); buf[2] = random();
sprintf(nom, "fifo.%d", buf[0])
mkfifo(nom);
fdr = open(nom, O_RDONLY);
fdq = open("fifo", O_WRONLY);
write(fdq, buf, 3 * sizeof(int));
close(fdq);
read(fdr, &res, sizeof(int);
close(fdr);
unlink(nom);
if (res == buf[1] + buf[2])
   printf("Le serveur marche bien!\n");
else
   printf("Le serveur est NUL!\n");
```