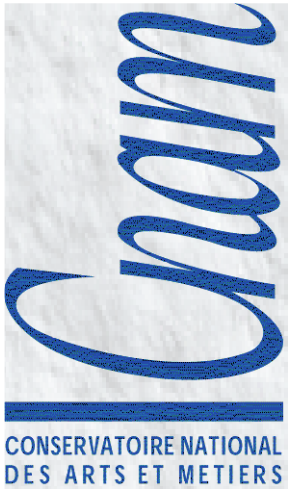

Tubes & Signaux

Samia Bouzefrane & Ziad Kachouh

http://cedric.cnam.fr/~bouzefra

Les Tubes

- Les tubes ordinaires
- Lecture/écriture dans un tube ordinaire
- Les tubes nommés
- Lecture/écriture dans un tube nommé

Introduction aux tubes

- Tube : mécanisme de communication *unidirectionnel*.
  - Possède deux extrémités, une pour y lire et l'autre pour écrire
  - La lecture dans un tube est destructrice: l'info lue est supprimée du tube
  - Les tubes permettent la communication d'un flot continu de caractères (mode stream)
  - Un tube a une capacité finie
  - La gestion des tubes se fait en mode FIFO

Les tubes ordinaires

- A un tube est associé un nœud du système de gestion de fichiers (son compteur de liens est = 0 car aucun répertoire ne le référence).
- Le tube sera supprimé et le noeud correspondant libéré lorsque plus aucun processus ne l'utilise.
- Un tube ordinaire n’a pas de nom.
- L'existence d'un tube correspond à la possession d'un descripteur acquis de deux manières:

  -un appel à la primitive de création de tube *pipe*;

  -par héritage: un processus fils hérite de son père des descripteurs de tubes, entre autres.

La création d'un tube ordinaire

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

crée un tube, c-à-d:

- un noeud sur le disque des tubes,
- deux entrées dans la table des fichiers ouverts (1 en lecture, 1 en écriture), c-à-d deux descripteurs dans la table des processus appelant.

pipe retourne dans *p* respectivement les descripteurs de lecture et d'écriture:

p[0] est le descripteur en lecture

p[1] celui en écriture.

La lecture dans un tube ordinaire

La lecture se fait grâce à la primitive ***read***:

Exemple:

char *buf* [10];

int *nbr_car*=10;

int *nb_lu*;

***nb_lu*= read(*p*[0], &*buf*, *nbr_car*);** correspond à la lecture d'au plus *nbr_car* caractères qui seront rangés dans une zone pointée par *buf*.

Fonctionnement d'un ***read***

Si ***tube non vide*** et contient *nbr* caractères **alors**

la primitive extrait du tube *nb_lu*=min (*nbr*, *nbr_car*) caractères et les place à l'adresse *buf*;

Sinon

Si ***nombre d'écrivains = 0***, la fin de fichier est atteinte **alors**

aucun caractère n'est lu et la primitive renvoie la valeur *nb_lu*=0;

Fsi

Si ***nombre d'écrivains != 0*** **alors**

Si ***lecture bloquante*** **alors** le processus est bloqué jusqu'à tube non vide;

Fsi;

Si lecture non bloquante alors

le retour est immédiat et la valeur de retour est -1 et errno=EAGAIN

Fsi;

Fsi

Fsi

L’écriture dans un tube ordinaire

Elle se fait à l'aide de la primitive ***write***.

Exemple:

int *nb_ecrit*;

char *buf*[8]="Bonjour\0"; int *n*=8;

nb_ecrit= **write**(*p*[1], *buf*, *n*); demande l'écriture dans le tube de descripteur *p*[1] de *n* caractères accessibles à l'adresse *buf*.

Remarque: *n* doit être ≤ PIPE_BUF défini dans <limits.h>.

Fonctionnement d'un ***write***

Si nombre de lecteurs dans le tube = 0 alors

le signal SIGPIPE est envoyé au processus, ayant pour handler de terminer ce processus;

Sinon

si écriture bloquante alors

le retour de la primitive (avec la valeur n) n'a lieu que lorsque les *n* caractères ont été écrits (le processus passe à l'état *bloqué* dans l'attente que le tube se vide);

sinon

si n > PIPE_BUF **alors** le retour est un nombre < n éventuellement -1 **fsi**

si n ≤ PIPE_BUF **et nbre emplacements libres dans le tube >= n alors**

une écriture atomique est réalisé; **fsi**

si n ≤ PIPE_BUF **et nbre emplacements libres dans le tube < n alors**

le retour est immédiat *sans écriture* avec la valeur de retour 0 ou -1 **fsi**

fsi

Exemple d'utilisation

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

int tube[2];
char buf[20];

main() {
        pipe(tube);
        if (fork()==0) { /* fils */
                close(tube[0]);
                write(tube[1], "bonjour", 8);
        }
        else { /* pere */
                wait(NULL);
                close(tube[1]);
                read(tube[0], buf, 8);
                printf("%s bien recu\n", buf);
        }
}
```

```
$ ./tube_exo1
$ bonjour bien recu
$
```

Les signaux

- Introduction
- Les types de signaux
- L’envoi d’un signal
- Prise en compte d’un signal
- Installation de nouveau handlers
- L’attente d’un signal
- Signaux particuliers

Introduction

- **Signal**
  - ***Interruption***: événement extérieur au processus
    - Frappe au clavier
    - Signal déclenché par programme: primitive kill, …
  - ***Déroutement***: événement intérieur au processus généré par le hard
    - FPE (floating point error)
    - Violation mémoire
- Il existe NSIG (<signal.h>) signaux différents, identifiés par un numéro de 1 à NSIG

Les types de signaux

Signal		Traitement par défaut
SIGHUP	Terminaison du processus leader de session	(1)
SIGINT	Frappe du car **intr** sur terminal de contrôle	(1)
SIGQUIT	Frappe du car **quit** sur terminal de contrôle	(2)
SIGILL	Détection d'une instruction illégale	(2)
SIGABRT	Terminaison provoquée en exécutant **abort**	(1)
SIGFPE	Erreur arithmétique (division par zéro, …)	(1)
SIGKILL	Signal de terminaison	(1) * non modifiable
SIGSEGV	Violation mémoire	(2)
SIGPIPE	Écriture dans un tube sans lecteur	(1)
SIGALARM	Fin de temporisation (fonction alarm)	(1)
SIGTERM	Signal de terminaison	(1)
SIGUSR1	Signal émis par un processus utilisateur	(1)
SIGUSR2	Signal émis par un processus utilisateur	(1)
SIGCHLD	Terminaison d'un fils	(3)
SIGSTOP	Signal de suspension	(4) *
SIGSUSP	Frappe du car susp sur terminal de contrôle	(4)
SIGCONT	Signal de continuation d'un signal d'un processus stoppé	(5) *

- La commande kill –l donne la liste des signaux du système
- **Traitements par défaut**: (1) terminaison de processus (2) terminaison de processus avec image mémoire (fichier core) (3) signal ignoré (sans effets) (4) suspension du processus, (5) continuation (reprise d'un processus stoppé).

Envoi d'un signal

- Les processus peuvent communiquer par le biais des signaux. La commande *kill* envoie un signal à un processus (ou groupe de processus) :
- int **kill** (pid_t *pid*, int *sig*);
- *sig* est le nom d'un signal ou un entier entre 1 et NSIG.
- On peut utiliser *sig* = 0 pour tester l'existence d'un processus.

Valeur de *pid*	**Signification**
> 0	Processus d'identité *pid*
0	Tous les processus dans le même groupe que le processus
< -1	Tous les processus du groupe |*pid*|

Exemple d’envoi de signal

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

void main(void) {
   pid_t p;
   int etat;
   if ((p=fork()) == 0) { /* processus fils qui boucle */
      while (1);
      exit(2);
   }
   /* processus pere */
   sleep(10);
   printf("envoi de SIGUSR1 au fils %d\n", p);
   kill(p, SIGUSR1);
   // bloque l'appelant (pere) et selectionne le fils
   p = waitpid(p, &etat, 0);
   printf("etat du fils %d : %d\n", p, etat >> 8);
}
```

Traitements par défaut

- A chaque type de signal est associé un *handler* par défaut appelé **SIG_DFL**.
- Les traitements par défaut sont:

  1- terminaison du processus (avec/sans image mémoire : fichier *core*)

  3- signal ignoré (sans effet)

  4- suspension du processus

  5- continuation : reprise d'un processus stoppé.
- Un processus peut ignorer un signal en lui associant le handler **SIG_IGN**.
- Les signaux **SIGKILL**, **SIGCONT** et **SIGSTOP** ne peuvent avoir que le handler **SIG_DFL**.

Traitements spécifiques

- Les signaux (autres que SIGKILL, SIGCONT et SIGSTOP) peuvent avoir un handler spécifique installé par un processus:
- La primitive ***signal***() fait partie du standard de C et non de la norme de POSIX:

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

- Elle installe le handler spécifié par *p_handler* pour le signal *sig*. La valeur retournée est un pointeur sur la valeur ancienne du *handler*.

Exemple d'installation de nouveau handler

Signal généré par des touches du clavier (CTL/C= SIGINT)

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

void hand(int signum)
{
   printf("appui sur Ctrl-C\n");
   printf("Arret au prochain coup\n");
   signal(SIGINT, SIG_DFL);
}

int main(void)
{
   signal(SIGINT, hand);
   for (;;) { }
   return 0;
}
```

Attente d’un signal

- La primitive ***pause***() bloque en attente le processus appelant jusqu’à l’arrivée d’un signal.

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

- A la prise en compte d'un signal, le processus peut :
  - se terminer car le handler associé est SIG_DFL;
  - passer à l'état stoppé; à son réveil, il exécutera de nouveau ***pause()*** et s’il a été réveillé par SIGCONT ou SIGTERM avec le handler SIG_IGN, il se mettra de nouveau en attente d'arrivée d'un signal;
  - exécuter le handler correspondant au signal intercepté.
- ***pause***() ne permet pas d'attendre un signal de type donné ni de connaître le nom du signal qui l'a réveillé.

Exemple d’attente de signal

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

int nb_req=0;
int pid_fils, pid_pere;

void   Hand_Pere( int sig ){
 nb_req ++;
 printf("\t le pere traite la requête numero %d du fils \n",
   nb_req);
}

void   main() {
```

Exemple d’attente de signal (suite)

```
if ((pid_fils=fork()) == 0) { /* FILS */
        pid_pere = getppid();
        sleep(2);  /* laisser le temps au pere de se mettre en pause */
        for (i=0; i<10; i++) {
                printf("le fils envoie un signal au pere\n");
                kill (pid_pere, SIGUSR1); /* demande de service */
        }
        exit(0);
   }
   else { /* PERE */
        signal(SIGUSER1,Hand_Pere);
        while(1) {
                pause();  /* attend une demande de service */
                sleep(5); /* realise le service */
                }
        }
}
```

Déroutement d'un signal: SIGFPE

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

void Hand_sigfpe() {
   printf("\nErreur division par 0 !\n");
   exit(1);
}

main() {
        int a, b, Resultat;

        signal(SIGFPE, Hand_sigfpe);
        printf("Taper a : "); scanf("%d", &a);
        printf("Taper b : "); scanf("%d", &b);
        Resultat = a/b;
        printf("La division de a par b = %d\n", Resultat);
}
```

Bibliographie