

## Systèmes d'exploitation, 2ème année

Communication

Yves STADLER

Université Paul Verlaine - Metz

10 novembre 2011

### Plan du cours

- Tubes de communication
- Tubes nommés
- Partage de mémoire
- Signaux

1/22

2/22

## Motivations

### Contexte

- Exécution de `ls -l | cut -f 2 -d ' '`
- Création de deux processus concurrents
- `ls` écrit des données dans un 'pipe'
- `cut` lit des données dans un 'pipe'
- `ls` termine par un 'EOF'
- `cut` lit jusqu'à 'EOF'

### En programmation

- Dans le cas d'un pipe système : automatique
- Peut-on faire pareil en programmant ?
- Avoir un flux qui bloque tant que rien n'est à lire
- Pouvoir écrire dans ce flux avec un processus.

3/22

## Tubes

### Définition

- Le tube est une file d'attente FIFO
- Unidirectionnel
- Communication par flot
- Auto-synchronisé
- Appartient au système (comme les sémaphores)

### Comportement

- Table des fichiers ouverts (descripteur)
- La lecture est destructrice, tout ce qui est lu disparaît du tube
- Capacité finie, un tube peut être saturé.

4/22

### Représentation

- Identifié par un numéro `i-node` comme un fichier
- N'existe pas dans le système de fichier
- On utilise les blocs adressés directement
- File circulaires (double pointeurs)

### Aller plus loin

- On peut aussi nommer un tube
- On peut communiquer entre processus étrangers.

### Plan du cours

#### NAME

`pipe, pipe2` - create pipe

#### SYNOPSIS

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

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

### Plan du cours

- Créer un pipe unidirectionnel
- `pipefd` et rempli avec deux descripteurs
- `[0]` est la sortie du tube, on peut lire `pipefd[0]`
- `[1]` est l'entrée du tube, on peut écrire `pipefd[1]`
- Le processus qui ne lit pas fermera `pipefd[0]` (`close`)
- Le processus qui n'écrit pas fermera `pipefd[1]` (`close`)

### man 2 read

#### NAME

`read` - read from a file descriptor

#### SYNOPSIS

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

```
ssize_t read(int fd, void *buf, size_t count);
```

### Paramètres

- fd : descripteur de fichier
- buf : variable pour recevoir les données
- count : nombre d'octets à lire
- Attention, read renvoi le nombre d'octets effectivement lus (pas forcément ce que l'on a demandé)

### Paramètres

- fd : descripteur de fichier
- buf : variable pour recevoir les données
- count : nombre d'octets à lire
- Attention, read renvoi le nombre d'octets effectivement lus (pas forcément ce que l'on a demandé)

### man 2 write

#### NAME

write - write to a file descriptor

#### SYNOPSIS

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

```
ssize_t write(int fd, const void *buf, size_t count);
```

### man 3 mkfifo

#### NAME

mkfifo - make a FIFO special file (a named pipe)

#### SYNOPSIS

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

```
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

### Dup

- `int desc2 = dup (int desc1);`
- Duplique une entrée de la table des descripteurs
- `int dup2(int src, int dest);`
- Copie dans dest les informations de src
- Exemple, remplacer stdin par un tube

13/22

## Controle des segments

### man 2 shmctl

#### NAME

`shmctl` - shared memory control

#### SYNOPSIS

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

### Notes

- Peu utile en général
- Lié à la structure `shmid_ds` décrite dans le man

15/22

### Utilisation

- Similaire à l'utilisation des files et sémaphores
- Besoin d'une clef (man `ftok`), ou `IPC_PRIVATE`
- Dispose d'une taille

### man 2 shmget

#### NAME

`shmget` - allocates a shared memory segment

#### SYNOPSIS

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int shmflg);
```

14/22

## Utilisation

### man shmatt, man shmdt

#### NAME

`shmatt`, `shmdt` - shared memory operations

#### SYNOPSIS

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

```
void *shmatt(int shmid, const void *shmaddr, int shmflg);
```

```
int shmdt(const void *shmaddr);
```

16/22

## Attacher, détacher

- `shmat` Attache `shmid` dans l'espace d'adressage du processus appelant
- `shmaddr == NULL` : cherche une adresse compatible.
- Si `shmaddr` n'est pas `NULL`, utiliser `SHM_RND` ou faire attention d'avoir une adresse alignée avec la page mémoire.
- renvoie l'adresse du segment attaché.
- `shmdt` détache `shmid`.

17/22

## Interruptions

- Interruptions logicielles
- Réagir à des événements sans continuellement attendre.
- Un processus peut dire au système ce qui doit se passer sur un signal.
  - Ignorer
  - Prendre en compte : exécuter une fonction (handler)
  - Appliquer le comportement par défaut (KILL)
- tous les signaux ne sont pas interceptable ni ignorable.

18/22

## man 7 signals

man 7 signal

Signal Value Action Comment

SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from <code>abort(3)</code>
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
...			

19/22

## Signal kill

- `int kill(pid_t pid, int sig);`
- Génère/Envoi un signal KILL pour `pid`

## Gestion de signaux

NAME

`sigaction` - examine and change a signal action

SYNOPSIS

`#include <signal.h>`

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

20/22

```
struct sigaction {  
    void      (*sa_handler)(int);  
    void      (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t   sa_mask;  
    int       sa_flags;  
    void      (*sa_restorer)(void);  
};
```

### Masquage

- Les processus dispose d'un masque qui définit l'ensemble des signaux bloqués.
- sigprocmask gère cet ensemble
- sigemptyset, sigfillset, sigaddset, ...

### Attente d'un signal

- pause()
- alarm()