Cours de système

Communication inter-processus

Sébastien Paumier / Sylvain Cherrier

Le tube

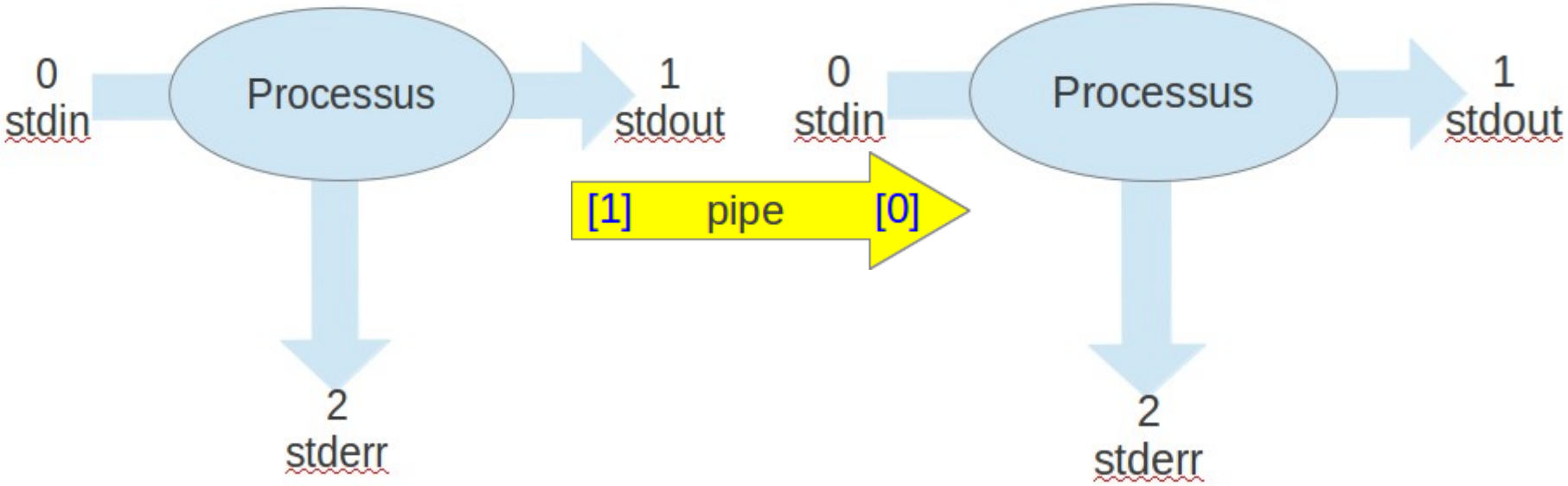
- moyen de transmettre un flot d'octets entre des processus
- représenté par une paire de descripteurs associés à une inode présente uniquement en mémoire et gérée de façon spéciale
- unidirectionnel: un côté du tube sert à l'écriture, le second pour la lecture
- la lecture est destructive !
- tous les détails dans *man 7 pipe*

Le tube

- ordre des octets conservés: FIFO
- capacité finie, qui dépend du système
- lecteur=processus possédant un descripteur sur le côté lecture
  - on ne peut pas écrire s'il n'y a pas au moins un lecteur !
- écrivain=processus possédant un descripteur sur le côté écriture
  - pas d'écrivain ? **`read`** renverra 0

Le tube dessiné

- Le tube a été conçu pour permettre l'implementation de | (pipe)



Création

- `int pipe(int p[2]);`
- crée deux descripteurs:
  - `p[0]` pour la lecture
  - `p[1]` pour l'écriture
- exemple: création d'un tube pour en mesurer la capacité maximum

pipe_capacity.cpp

Ecriture

- **`write`** est atomique si taille<**`PIPE_BUF`** (linux/limits.h), i.e. données non entrelacées si plusieurs processus écrivent

`fs/pipe.c`

- si pas de lecteur, l'écrivain reçoit le signal SIGPIPE, sinon:
  - si écriture bloquante, **`write`** bloque jusqu'à écriture complète
  - sinon, le comportement dépend de la taille, inférieure ou non à **`PIPE_BUF`**

Mode non bloquant

- pour passer un descripteur de tube en mode non bloquant, utiliser **`fcntl`** avec **`F_SETFL`** pour ajouter l'attribut **`O_NONBLOCK`**
- exemple: test de capacité d'un tube qui s'arrête tout seul

pipe_capacity2.cpp

Lecture

- si le tube n'est pas vide, **`read`** renvoie autant d'octets que possible
- sinon:
  - s'il n'y a plus d'écrivain, elle renvoie 0
  - sinon:
    - en mode bloquant, la fonction bloque an attente de données
    - en non bloquant, elle renvoie 0 et positionne **`errno`** à **`EAGAIN`**

Héritage

- les descripteurs d'un tube sont dupliqués par un **`fork`** et conservés à travers un exec
- conséquence, après un **`fork`**, chaque processus doit fermer le côté qu'il n'utilise pas, sinon il y aura 2 lecteurs et 2 écrivains

bad_fork_pipe.cpp
good_fork_pipe.cpp

Piper deux commandes

- principe: utiliser un tube pour connecter la sortie standard d'un processus à l'entrée standard d'un autre
- attention, avec un **`fork`** plus une redirection, on doit fermer les deux côtés du tube !

toupper.cpp

Piper des commandes

- ce que fait le shell avec plusieurs commandes et plusieurs tubes
- mais, il faut gérer plusieurs choses:
  - ne pas laisser de tubes ouverts si une des commandes échoue
  - récupérer le code d'erreur de la dernière
  - éviter les zombies
- pas si simple...

ush.cpp
tokenization.cpp
execute.cpp
vector_char.h

Interblocages

- cas n°1: si un processus crée un tube et lit dedans

deadlock_pipe1.cpp

cas n°2: deux processus ont deux tubes pour communiquer, mais:

- cas 2a: les deux lisent dans un tube vide
- cas 2b: les deux écrivent dans un tube plein

deadlock_pipe2.cpp

Tube nommé

- tube ayant un nom dans le système de fichiers
- commande shell **`mkfifo`**
- **`int mkfifo(const char *pathname, mode_t mode);`**
- crée un fichier spécial qu'on peut ouvrir avec **`open`**
- seul moyen de relier avec un tube des processus sans lien de filiation !

Tubes nommés

- une ouverture en lecture bloque jusqu'à une ouverture en écriture et inversement
  - sauf en mode non bloquant
- évite les interblocages

shell n°1:

```
$>mkfifo toto
$>ls -l > toto
(bloque jusqu'à ce
que le shell n°2
lise dans toto)
```

shell n°2:

```
$>cat toto
(affichage de ls -l)
```

Les signaux

- principe: mécanisme asynchrone permettant de "déclencher des sonneries" dans un processus (*man 7 signal*)
- l'envoi d'un signal à un processus revient à positionner un bit à 1 dans son bloc de contrôle
  - qui est dans le noyau, pour être sûr qu'un processus ne puisse pas ignorer des signaux abusivement

Les signaux

- traités par le noyau juste avant de repasser en mode utilisateur
  - pratique de le faire à ce moment-là pour des histoires de pile
- le noyau regarde s'il y a des signaux:
  - si oui, il invoque les handlers de signaux associés, s'il y en a
  - sinon, il effectue l'action par défaut pour ce signal (tuer, ignorer, créer un core, stopper, continuer)

`arch/x86/kernel/signal.c: do_signal`

Les signaux

- si aucun handler de signal n'a tué le processus, celui-ci reprend là où il en était au moment du changement de contexte vers le mode noyau
- possibilité de demander le redémarrage automatique d'un appel système interrompu (cf. plus loin)

Les signaux

Signal	Valeur	Action	Commentaire
SIGHUP	1	Term	Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle.
SIGINT	2	Term	Interruption depuis le clavier.
SIGQUIT	3	Core	Demande « Quitter » depuis le clavier.
SIGILL	4	Core	Instruction illégale.
SIGABRT	6	Core	Signal d'arrêt depuis abort(3).
SIGFPE	8	Core	Erreur mathématique virgule flottante.
SIGKILL	9	Term	Signal « KILL ».
SIGSEGV	11	Core	Référence mémoire invalide.
SIGPIPE	13	Term	Écriture dans un tube sans lecteur.
SIGALRM	14	Term	Temporisation alarm(2) écoulée.
SIGTERM	15	Term	Signal de fin.

Les signaux

Signal	Valeur	Action	Commentaire
SIGUSR1	30,10,16	Term	Signal utilisateur 1.
SIGUSR2	31,12,17	Term	Signal utilisateur 2.
SIGCHLD	20,17,18	Ign	Fils arrêté ou terminé.
SIGCONT	19,18,25	Cont	Continuer si arrêté.
SIGSTOP	17,19,23	Stop	Arrêt du processus.
SIGTSTP	18,20,24	Stop	Stop invoqué depuis tty.
SIGTTIN	21,21,26	Stop	Lecture sur tty en arrière-plan.
SIGTTOU	22,22,27	Stop	Écriture sur tty en arrière-plan.
SIGBUS	10,7,10	Core	Erreur de bus (mauvais accès mémoire).

- plus d'autres (cf. *man 7 signal*)
- SIGKILL et SIGSTOP ne peuvent pas être interceptés ni ignorés

Envoi

- certains signaux sont envoyés par le système:
  - SIGINT: Ctrl+C dans le terminal
  - SIGPIPE: écriture dans un tube sans lecteur
  - SIGSEGV: violation mémoire
  - SIGALRM: temporisation écoulée
  - ...
- on peut aussi en envoyer manuellement

Envoi

- commande **`kill`**
- **`int kill(pid_t pid, int sig);`**
- envoie un signal:
  - pid>0: au processus désigné
  - pid=0: à tous les processus du même groupe que le processus appelant
  - pid=-1: à tous les processus, sauf *init*
  - pid<-1: aux processus du groupe -pid

Envoi

- on peut utiliser **`kill`** avec **`sig`** nul pour tester l'existence d'un processus
- il faut être privilégié pour envoyer un signal à n'importe quel processus
  - mais même root ne peut envoyer de signal à *init,* sauf si *init* a un handler de prévu pour ce signal
- par défaut, on ne peut en envoyer qu'aux processus qu'on possède (uid réelles identiques entre l'appelant et la cible)

Interception d'un signal

- on peut intercepter un signal (sauf SIGKILL et SIGSTOP) en installant un gestionnaire (signal handler)
- `sighandler_t signal(int signum, sighandler_t handler);`
- DEPRECATED!
- ne l'utiliser que pour ignorer un signal avec `signal(xxx,SIG_IGN)`

Interception d'un signal

- **`int sigaction(int signum, const struct sigaction *act,struct sigaction *oldact);`**
- installe un handler en sauvant l'ancien, s'il existe, et si **`oldact`** est non **`NULL`**

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

struct sigaction

- attention: utiliser SOIT le champ **`sa_handler`** (utilisé par défaut), SOIT **`sa_sigaction`**
- bien penser à initialiser la structure avec **`memset`**:

```
struct sigaction sig;
memset(&sig,0,sizeof(struct sigaction));
```

- exemple: utilisation d'un handler sur SIGALRM pour écrire une horloge

signal_timer.cpp

struct sigaction

- le champ **`sa_mask`** permet de définir les signaux qu'on veut bloquer pendant l'exécution du handler
- se manipule avec **`sigemptyset`** et **`sigaddset`** (cf. *man sigemptyset* pour les autres fonctions sur les **`sigset_t`**)
- les signaux bloqués ne sont pas ignorés, mais juste post-traités:

signal_no_masking.cpp

signal_masking.cpp

struct sigaction

- le champ **`sa_flags`** est un OU binaire entre différentes valeurs, dont:
  - **`SA_SIGINFO:`** invoquera la fonction **`sa_sigaction`** et non pas **`sa_handler`**
  - **`SA_NOCLDWAIT`**: ne pas transformer les fils en zombie quand ils meurent
  - **`SA_RESTART`**: en cas d'interruption d'un appel système par le signal, le relancer automatiquement:

syscall_interrupt.cpp

syscall_interrupt2.cpp

Signaux multiples

- comme un signal n'est codé que par un bit, impossible de savoir s'il a été reçu plus d'une fois
- exemple: un père envoie des SIGUSR1 à son fils qui les compte dans un handler
  - la version qui marche de temps en temps:

signal_count_bug.cpp

  - une qui marche, et une totalement safe:

signal_count.cpp

signal_count2.cpp

pause

- `int pause(void);`
- endort le processus jusqu'à ce qu'un signal ait été reçu et traité par son gestionnaire, s'il existe
- moins consommateur de CPU que l'attente active quand on attend des signaux

```
while (1);
```

```
while (1) {
    pause();
}
```

Signaux synchrones

- **`pause`** attend n'importe quel signal
- mais, on peut attendre un ou plusieurs signaux précis
- **`int sigwait(const sigset_t *set, int *sig);`**
- **`int sigwaitinfo(const sigset_t *set, siginfo_t *info);`**

signal_pause.cpp

signal_sigwait.cpp

Les verrous de fichiers

- mécanisme permettant de gérer les accès concurrents aux fichiers
- verrouillage en lecture et/ou écriture sur tout ou partie d'un fichier
- un verrou est rattaché à une inode, donc visible par tout processus
- un verrou ne peut être relâché que par son propriétaire; il est relâché automatiquement à la mort de celui-ci

Les verrous de fichiers

- verrouillage consultatif (par défaut):
  - pas de contrôle d'accès, les processus doivent jouer le jeu
- verrouillage impératif:
  - contrôle d'accès sur **`read`** et **`write`**
  - le système de fichiers doit être compatible (**`mount`** avec **`-o mand`**) +réglages sur les permissions des fichiers concernés
  - pas fiable!!! (cf. BOGUES dans *man fcntl*)

flock

- **`int flock(int fd, int operation);`**
- verrouille tout le fichier en fonction de **`operation`**:
  - **`LOCK_SH`**: verrouillage partagé
  - **`LOCK_EX`**: verrouillage exclusif; au plus un processus peut en avoir un à un instant donné
  - **`LOCK_UN`**: déverrouillage

lock1.cpp

lockf

- interface au-dessus de **`fcntl`**, qui offre toutes les opérations possibles sur les verrous
- **`int lockf(int fd, int cmd, off_t len);`**
- idem, mais permet de définir la portion du fichier concernée par le verrou
- **`cmd`**=**`F_LOCK`**, **`F_ULOCK`**, **`F_TEST`**, ou **`F_TLOCK`**

lock2.cpp