IF 219 – Programmation Systèmes

Laurent Réveillère

Enseirb-Matmeca
Département Télécommunications

Laurent.Reveillere@bordeaux-inp.fr
http://www.labri.fr/perso/reveille/if219/

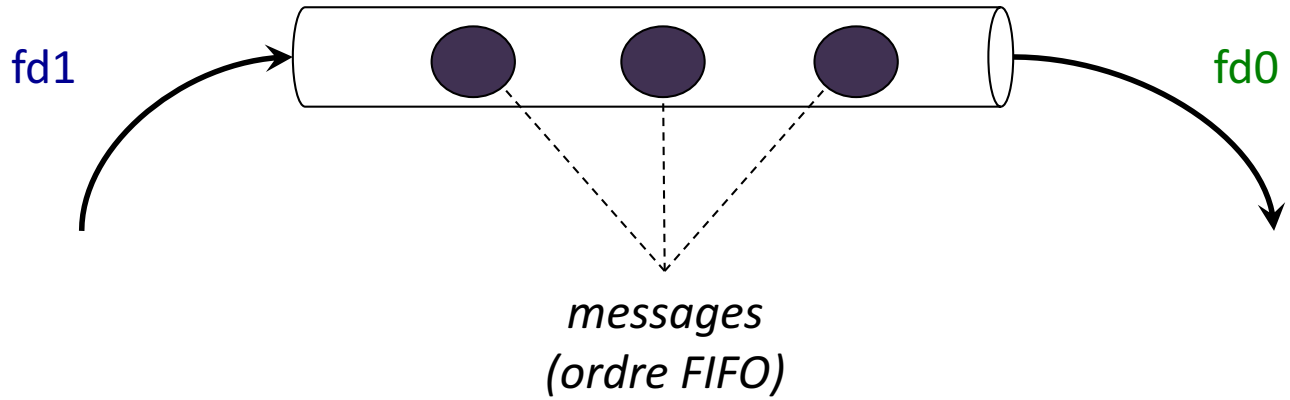
Communication entre processus : Les tubes

IF 219

Programmation Système

Communication par tubes

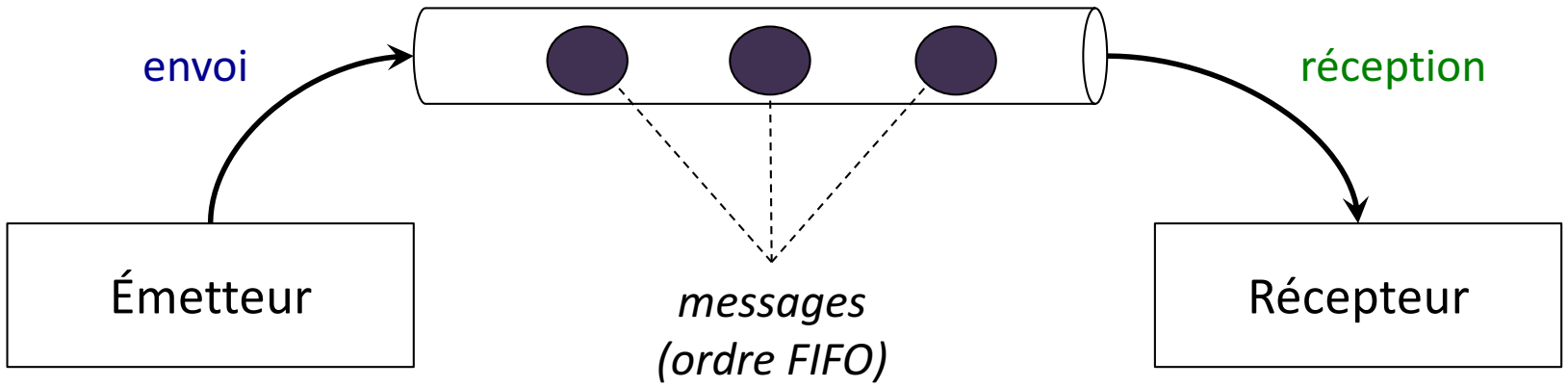
- Tube (*pipe*) : permet l'échange de données entre processus
  - Modèle de programmation producteur-consommateur
  - Tampon intermédiaire de capacité inconnue mais finie
  - Réception dans l'ordre d'émission (réception dite FIFO)



Caractéristiques des tubes

- Mécanisme de communication pour des entités appartenant à la même machine
  - Communication unidirectionnelle
  - Communication bidirectionnelle nécessite de mettre en place une paire de tubes
- Représentation sous forme de fichiers
  - Manipulation sous forme de descripteurs de fichier
  - Lecture/écriture avec les appels système read/write
  - Utilisables pour mettre en place des redirections d'entrées/sorties

Tubes anonymes



- Création d'un tube anonyme par un processus
  - `int pipe (int fd[2])`
- Donne accès à deux descripteurs
  - fd[0] - lecture à partir d'un tube
  - fd[1] - écriture dans le tube

Utilisation des tubes

- Partage d'un tube anonyme entre processus
  - Création par un ancêtre commun
- Chacun des deux processus ferme le descripteur dont il n'a pas besoin
  - Économie de ressources systèmes (descripteurs)
  - Évite les inter-blocages

Exemple - tubes

```
int main() {
   int  fd[2];
   int  pid;
   int  c;

   pipe(fd);
   if ( (pid = fork()) == 0) {    /* Le fils est l'écrivain */
      close(fd[0]);               /* Donc il ferme la lecture */
      write(fd[1], "A", 1);
   } else if (pid > 0) {.         /* Le père est le lecteur */
      close(fd[1]);               /* Donc il ferme l'écriture */
      read(fd[0], &c, 1);
      printf("Le père a reçu \"%d\".\n", c);
   }

   return 0;
}
```

Sémantique différentes des fichiers

- Modèle de programmation "producteur / consommateur"
  - Processus travaillant de façon partiellement asynchrone
- Taille de tampon finie … et petite
  - Pas nécessairement de correspondance entre la taille des données à lire / écrire et la capacité restante dans le pipe

Sémantique de *read(…, x)*

- S'il existe des octets dans le tube
  - Read(…, x) renvoie au plus x octets
  - Différence avec les fichiers classique
- S'il n'y a plus d'octets dans le tube (tube vide)
  - Plus d'écrivain potentiel
    - » Read() renvoie 0
    - » Analogue à une fin de fichier classique
  - Au moins un écrivain potentiel
    - » Read() bloque et endort le processus jusqu'à retrouver un des cas ci-dessous
    - » Rien à faire de particulier (automatique)

Sémantique de *write()*

- S'il reste au moins un lecteur potentiel
  - Write(…, x) ne retournera qu'une fois que les x octets seront effectivement écrits dans le tube
    - Analogue à l'écriture dans un fichier classique
    - Peut conduire à l'endormissement du processus s'il reste moins de x octets libres dans le tube
    - Pas de prise en charge particulière (automatique)

Sémantique de *write()* - suite

- S'il n'existe plus aucun lecteur potentiel
  - Envoie d'un signal SIGPIPE au processus
  - Write(…, x) retournera une erreur
    - Écrire dans un tube sans lecteur est une anomalie
    - Comme personne n'écoute, cela ne sert à rien de parler
      - Comportement par défaut à la réception d'un SIGPIPE
      - Exemle de "ls | more" où l'utilisateur termine le "more"

Tubes nommés

- Permet la communication entre processus n'ayant pas de liens de parenté
- Création

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

  - Apparait dans le système de fichiers

- Doit avoir été ouvert par au moins un lecteur et un écrivain pour réaliser une communication
  - Par défaut, l'ouverture du tube nommé bloque jusqu'à ce que l''autre extrémité ait été ouverte

Exemple

```
int main() {
   int  fd;
   int  pid;
   int  c;

   mkfifo("/tmp/tube.fifo", 0600);
   if ( (pid = fork()) == 0) {    /* Le fils est l'écrivain */
      fd = open("/tmp/tube.fifo", O_WRONLY);
      write(fd, "A", 1);
   } else if (pid > 0) {.          /* Le père est le lecteur */
      fd = open("/tmp/tube.fifo", O_RDONLY);
      read(fd, &c, 1);
      printf("Le père a reçu \"%d\".\n", c);
   }

   return 0;
}
```

Caractéristiques

- Appel *open* bloquant pour un tube nommé
- Un processus demandant l'ouverture en lecture attend l'arrivée d'écrivains, et vice-versa
- On peut rendre l'ouverture non bloquante (O_NONBLOCK dans le mode d'ouverture)
  - Dans ce cas, une demande en écriture sans lecteurs échoue
  - Cf. pipe2 pour passer des arguments lors de la création d'un tube anonyme

Communication entre processus : Les signaux

IF 219

Programmation Système

Communication par signaux

- Occurrence d'un évènement asynchrone
  - Nécessite un traitement spécifique
  - Arrivé d'un bloc disque, frappe clavier, etc.
- Niveau matériel
  - Mise en œuvre d'interruptions
  - Déroutement du processeur et exécution du traitant ad-hoc
- Niveau logiciel
  - Mise en œuvre des signaux
  - Traitement par le processus lui-même

Représentation d'un signal

- Identifiés par des noms et codés par des entiers
  - SIGTERM, SIGKILL, SIGSTOP, SIGALARM, SIGPIPE, …
  - Page de man pour liste des signaux disponibles
- Ne véhicule aucune information
  - Sauf son numéro
  - Ne sert qu'à indiquer à un processus qu'un certain type d'évènement s'est produit

Utilisation des signaux

- Mise en place de traitants de signaux (*signal handlers*)
  - Définit la réaction du processus lors de la notification d'un évènement
- Comportement
  - Traitant par défaut pour chaque signal (possiblement vide)
  - Possibilité de de redéfinir le traitement
    - Sauf pour certains (ex. SIGKILL, SIGSTOP)
  - Possibilité d'ignorer temporairement des signaux
    - Sauf pour certains (ex. SIGKILL, SIGSTOP)

Principaux signaux

Signal	Action par défaut	Événement déclenchant
SIGABRT	A	Généré par `abort()`.
SIGALRM	T	Expiration d'une alarme.
SIGCHLD	I	Fils terminé ou stoppé (ou continué [XSI]).
SIGCONT	C	Continue l'execution, si stoppé.
SIGFPE	A	Opération arithmetique erronée.
SIGHUP	T	Hangup, fin de session.
SIGINT	T	Interruption au terminal (C-c).
SIGKILL	T	Termine le processus (*).
SIGPIPE	T	Écriture sur tube sans lecteurs.

(*) ne peut être ni capté ni ignoré

- T : termine le processus normalement.
- A : termine le processus anormalement (eg, avec fichier `core`).
- I : ignoré.
- C : continue un processus stoppé.

Principaux signaux

Signal	Action par défaut	Événement déclenchant
SIGQUIT	A	Quit au terminal (C-\).
SIGSEGV	A	Référence mémoire invalide.
SIGSTOP	S	Stoppe l'execution (*).
SIGTERM	T	Termine l'exécution.
SIGTSTP	S	Envoi Stop au terminal (C-Z).
SIGTTIN	S	Processus en background essayant de lire.
SIGTTOU	S	Processus en background essayant d'écrire.
SIGUSR1	T	À la disposition du programmeur.
SIGUSR2	T	À la disposition du programmeur.

(*) ne peut être ni capté ni ignoré.

- T : termine le processus normalement.
- A : termine le processus anormalement (eg, avec fichier `core`).
- I : ignoré.
- S : stoppe le processus.

État d'un signal

- Délivré (ou pris en compte)
  - Traitant associé (peut être fonction vide) a été exécuté
  - Intervient lorsque le processus passe du mode actif noyau au mode actif utilisateur
- Pendant (*pending*)
  - Signal émis mais pas encore pris en compte
- Masqué (ou bloqué)
  - Signal pour lequel le processus retarde volontairement et temporairement sa délivrance
  - Le temporaire peut durer toujours!

Signaux en rafales

- Si on envoie à un processus n occurrences d'un signal
  - le processus détectera qu'il a reçu au moins une occurrence du signal...
  - mais ne peut détecter avec certitude n occurrences
  - Mécanisme de communication non fiable!

Cas particulier des appels système

- Appels interruptibles
  - Appels systèmes qui peuvent bloquer un temps arbitrairement long
    - Exemple : wait, read dans un tube vide, write dans un tube plein...
    - Si un signal non ignoré est généré pendant que le processus est bloqué, l'appel échoue (erreur : EINTR)
- Appels non interruptibles
  - Autres appels systèmes (attente normalement bornée)
  - Exemple : getpid, umask, read/write dans un fichier
- Retenir
  - Appels systèmes bloquants, tel que read sur un tube vide avec écrivains, sont interruptibles par un signal

Signaux et appels systèmes

- Garantir qu'un appel système ne soit pas interrompu par un signal
  1. Bloquer le signal avant d'effectuer l'appel système
     - » Débloquer au retour de l'appel
  2. Positionner le drapeau SA_RESTART pour le traitant
     - » Réexécution automatique l'appel interrompu
       - Ne fonctionne pour toutes les fonctions
  3. Recommencer manuellement l'appel

```
do {
    ret = appel(…);
} while ( (ret == -1) && (errno == EINTR));
```

     - » Indispensable en prog. Réseau avec : accept, connect, select, poll, listen, read, write, …

Émission d'un signal

- Émission depuis le terminal

```
kill -<numéro de signal> <pid>
```

- Émission depuis un processus

```
int kill (pid_t pid, int signum);
```

Paramètre 2: Si signum ==0, test l'existence du processus

Code retour : -1 si problème (cf. errno) / 0 si succès

- S'envoyer un signal à soi-même

```
int raise(int signum);
```

Réception d'un signal

- Mécanisme asynchrone
  - Pas de fonction à appeler
  - Enregistrement du traitant ad-hoc

- Attendre explicitement l'arrivée d'un signal

```
int pause(void);
```

  - Code de retour : -1 et errno == EINTR (une erreur est le comportement normal de pause)

Mise en place d'un handler

- Opération en 3 étapes
  1. Déclaration du traitant pour le signal concerné
     - » Fonction qui sera exécuté
  2. Déclaration et initialisation d'une structure de type struct sigaction
     - » Transmettre au systèmes le comportement souhaité
  3. Appel de la fonction sigaction pour mise en place effective du traitant
     - » Ne provoque pas d'émission et ne sert pas à recevoir
- Mise en place jusqu'à une nouvelle demande de changement

Structure *sigaction*

- Prototype

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

- Champ `sa_handler`
  - Pointeur de fonction du traitant a exécuté
    - » SIG_DFL : traitant par défaut
    - » SIG_IGN : ignorer le signal

Structure *sigaction*

- Prototype

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

- Champ `sa_sigaction`
  - Pointeur de fonction alternatif en lieu et place de sa_handler
    - » Permet d'avoir plus d'information sur le signal

Structure *sigaction*

- Prototype

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

- Champ `sa_mask`
  - Sert à positionner un masque de signaux qui seront automatiquement bloqué lors de l'exécution du traitant
  - En plus du signal pour lequel le traitant est mis en place qui est systématiquement bloqué

Structure *sigaction*

- Prototype

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

- Champ `sa_flags`
  - SA_NOCLDSTOP et sig==SIGCHLD : ne pas générer SIGCHLD si un fils est stoppé
  - SA_RESTART : redémarre les appels systèmes interruptibles, interrompus avant d'avoir pu commencer (XSI)
  - SA_RESETHAND : rebascule le traitant à SIG_DFL lors de la délivrance
  - SA_NODEFER : ne pas masquer dans le handler le signal délivré

Fonction sigaction

```
#include <signal .h>
int sigaction (int sig , const struct sigaction *act,
                                 struct sigaction *old_act);
```

- Paramètre 1 : numéro du signal
- Paramètre 2 :
  - Si non NULL, action à mettre en place
- Paramètre 3 :
  - sauvegarder l'ancien comportement du processus
  - NULL si sauvegarde non nécessaire

Règles d'écriture

- Appels bloquant
  - Processus mis en sommeil dans le traitant
    - Ne peut pas revenir dans le main
- Utilisation de fonctions *async-signal-safe*
  - Voir page de man de signal (section 7) pour la liste
- Fonctions non *async-signal-safe*
  - Possible mais potentiellement dangereux !
  - Ne pas utiliser *printf*, malloc, free ☹
- Atomicité des variables partagées entre le main et un traitant
  - Utilisation du qualificatif *volatile* et du type *sigatomic_t*

Installation d'un masque (1)

- Représentation d'un masque de signaux
  - Champ de bits (notion ensembliste)
  - Structure de donnée sigset_t

- Construction d'un masque

```
int sigemptyset( sigset_t * set);
int sigfillset( sigset_t * set);
int sigaddset( sigset_t *set , int signum );
int sigdelset( sigset_t *set , int signum );
int sigismember( const sigset_t *set , int signum );
```

Installation d'un masque (2)

- Mise en place effective

```
int sigprocmask (int how, const sigset_t *set,
                                  sigset_t *old_set);
```

- set : structure de masque de signaux à utiliser
- old_set : ancien masque à sauvegarder
  - NULL : ne rien sauvegarder
- Valeurs possible de how
  - SIG_BLOCK ajouter set au masque actuel
  - SIG_UNBLOCK enlever set du masque actuel
  - SIG_SETMASK installer comme masque exactement set

Attente d'un signal

- Calculer l'ensemble des signaux pendants :

```
int sigpending( sigset_t * set);
```

- `int pause(void);`
  - fait passer le processus appelant en sommeil

- `int sigsuspend(const sigset_t *sigmask) ;`
  - fait passer le processus appelant en sommeil,
  - installe le masque spécifié ...
  - ... le tout de façon **atomique**

Exemple : fonction sleep

- Un processus peut demander à recevoir le signal SIGALRM au bout d'un certain temps

- Les fonctions

```
unsigned int alarm (unsigned int seconds);
int setitimer(int which, struct itimerval *value,
                         struct itimerval *old);
```

- permettent de manipuler des chronomètres qui, à expiration, envoient des signaux (SIGALRM, SIGVTALRM, SIGPROF)

Signaux et fork/exec

- Lors d'un fork(), le processus fils hérite
  - du masque du père
  - des handlers installés

mais pas des signaux pendants !

- Lors d'un exec(), le processus garde
  - le masque avant l'exec
  - les signaux pendants

mais pas les handlers !