Chap. IX : Communication entre processus (signaux & tubes nommés)

Laurent Poinsot

UMR 7030 - Université Paris 13 - Institut Galilée

Cours “Architecture et Système”

Les processus ne sont pas des entités indépendantes. Ils doivent partager les ressources de l'ordinateur. Dans certains cas, ils doivent communiquer entre eux pour se synchroniser ou pour communiquer de l'information. Il existe de nombreuses façons de communiquer. Dans ce chapitre, nous allons étudier les différents moyens qu'ont a leur disposition les programmeurs en langage C pour faire communiquer des processus.

Un signal est un moyen de communication indiquant une action à entreprendre à partir de conventions préétablies. Chaque signal a une signification particulière qui détermine le comportement du processus. Il n'y a pas de communication de données.
Un processus ne peut connaître l'identité du processus qui lui a envoyé le signal. Les signaux sont identifiés par un numéro entier et un nom symbolique décrit dans `signal.h`. La commande `kill -l` donne la liste des signaux d'un système Unix.

Un signal peut être envoyé

1. lors de la constation d'une anomalie matérielle (voir le chapitre sur les signaux matériels) ;
2. suite à la frappe d'une combinaison de touches (par ex. `CTRL-C` pour envoyer le signal `SIGINT`) ;
3. par un autre processus utilisant la commande `kill` ou l'appel système `kill`.

Lors de la réception d'un signal, il y a trois actions par défaut suivant le signal : ignorer le signal, terminer le programme ou terminer en créant un fichier `core`. Un processus peut changer son comportement par défaut lors de la réception d'un signal en déroutant le signal, c'est-à-dire en indiquant la fonction à exécuter lors de la réception du signal.

Les signaux `SIGKILL` et `SIGSTOP` ne peuvent pas être redéfinis. Un signal n'est pas reçu lors de son envoie (asynchronisme) ; il existe un délai entre l'envoi et la réception. Liste des principaux :

- `SIGINT` 2 signal d'interruption (provoqué par `CTRL-C` par exemple) ;
- `SIGQUIT` 3 quitter et créer un fichier `core` ;
- `SIGKILL` 9 fin du processus (non déroutable) ;
- `SIGPIPE` 13 écriture dans un tube sans lecteur (donc écriture inutile) ;
- `SIGALRM` 14 signal déclenché après un certain nombre de secondes (horloge) ;
- `SIGTERM` 15 fin de processus ;
- `SIGSTOP` 19 suspension du processus (non déroutable) ;
- `SIGUSR1` 30 signal utilisateur 1 (disponible pour les applications) ;
- `SIGUSR2` 31 signal utilisateur 2 (disponible pour les applications).

L'appel système `int kill (pid_t pid, int sig)` envoie le signal `sig` au processus `pid`. La réception d'un signal entraîne un comportement par défaut du processus. Celui-ci peut être modifié en indiquant une autre action à effectuer (voir `signal ()` ci-après). Un processus ne peut envoyer un signal à un autre que si le propriétaire (réel ou effectif) est le même pour les deux processus.

Exemples usuels de mise en œuvre de l'appel système `signal ()`.

- `signal (SIGINT,SIG_IGN);` : la réception du signal `SIGINT` devient inopérante ; le signal est ignoré ;
- `signal (SIGINT,SIG_DFL);` : le signal `SIGINT` retrouve son action par défaut ;
- `signal (SIGINT,queFaire);` : la fonction `queFaire()` est exécutée à la réception du signal `SIGINT`. On doit avoir une déclaration de la forme `void queFaire(int sig);`.

Exemple de programme non interruptible avec `CTRL-C` ; `SIGINT` est ignoré. Le programme doit être arrêté avec `kill -9 no.PID`. Par contre, si l'appel `signal()` est mis en commentaire, le programme s'arrête bien avec `CTRL-C`.

```
#include<signal.h>
void main (){
signal (SIGINT,SIG_IGN);
for (;;); }
```

Exemple de déroutement de signaux vers une fonction `queFaire()` qui imprime le numéro et le libellé du signal dérouté ; s'il s'agit du signal `SIGINT`, on arrête le programme.

```
#include<stdio.h>
#include<string.h>
#include<signal.h>
void quefaire(int sig){
printf("Signal reçu %d %s\n",sig,
strsignal(sig));
if (sig==SIGINT){
printf("Fin volontaire");
exit(1);}
}
```

Puis le `main` :

```
void main (){
/* CTRL-C : signal 2 */
signal (SIGINT, quefaire);
/* CTRL-\ : signal 3 */
signal (SIQUIT, quefaire);
/* CTRL-Z : signal 20 */
signal (SIGTSTP, quefaire);
/* Signal définit par l'utilisateur */
signal (SIGUSR1, quefaire);
for (;;); /* ad vitam aeternam */
}
```

L'appel système `int pause (void);` suspend le processus courant jusqu'à réception d'un signal quelconque non ignoré.
L'appel système `unsigned int alarm (unsigned int seconds);` active une horloge qui déclenche un signal `SIGALRM` au bout du nombre de secondes demandé en paramètre.

Par exemple, le programme ci-dessous déclenche une alarme au bout de dix secondes.

```
#include<unistd.h> /* pause, alarm */
...
void chrono (int sig) {
printf("Bien reçu %d
%s\n",sig,strsignal(sig));
printf("Dring dring ... temps écoulé\n");}
```

Puis le `main` :

```
void main () {
signal (SIGALRM,chrono); /* signal 14*/
printf("Prévenez-moi dans 10 secondes\n);
printf("Top chrono\n");
alarm(10);
}
```

L'exécution de ce programme provoque les affichages suivants :

```
Prévenez-moi dans 10 secondes
Top chrono
Bien reçu 14 Minuterie d'alarme
Dring dring ... Temps écoulé
```

Dans une application de gestion de fichiers ou dans une base de données classique, différents processus peuvent accéder en consultation ou en modification aux mêmes fichiers en même temps. Il en résulte un éventuel problème de conflits et d'intégrité des données. Les conflits peuvent être réglés par la pose de verrous. Une autre façon de régler les conflits consiste à n'autoriser qu'un seul processus à consulter et à modifier les données. Chaque processus client désirant accéder aux données doit soumettre une requête au processus serveur responsable de la bonne gestion des données. Le serveur examine la requête et fournit les données correspondant à la requête. Dans une application client-serveur, il y a un processus chargé de consulter ou modifier les données (le serveur), les autres processus (les clients) adressant des demandes (de consultation ou de modification) au serveur.

Les clients et le serveur doivent être en mesure de communiquer à l'aide des requêtes qui jouent le rôle d'interface entre les deux processus. Il n'y a pas de problème de conflits d'accès aux données car un seul programme (le serveur) peut consulter ou modifier les données. La zone servant d'interface pour les requêtes et les réponses est une sorte de boîte aux lettres où on dépose les requêtes et où on récupère les réponses. Il peut y avoir :

- une seule boîte aux lettres pour tout le monde ;
- ou une pour le serveur et une pour les clients ;
- ou une pour le serveur et une pour chaque client.

La boîte aux lettres peut être réalisée sous Unix de différentes façons : tubes nommés, files de messages, mémoires partagées et sémaphores.

Un tube nommé est un tube qui a un nom dans le système de fichiers. Tout processus peut l'ouvrir à condition d'en connaître le nom et d'en avoir les droits d'accès. Un processus ayant ouvert un tube en écriture (resp. lecture) est suspendu tant qu'il n'existe pas de processus lecteur (resp. écrivain). Les processus n'ont pas forcément de lien de parenté (comme c'est le cas pour les tubes anonymes déjà rencontrés).
L'appel système `mkfifo()` crée un tube nommé : `mkfifo (nom, 0666);`.

Contrairement aux tubes anonymes, d'une part les tubes nommés possèdent un nom dans le système de fichier et d'autre part ils peuvent être utilisés comme moyen de communication entre processus qui n'ont pas de lien de parenté.

Exemple : Deux programmes `fifo1` (dans `fifo1.pdf`) et `fifo2` (dans `fifo2.pdf`). Le premier crée un tube nommé - dont le nom est passé en argument de la commande - l'ouvre en écriture et écrit la chaîne " BONJOOUR ". Le second programme ouvre en lecture ce même tube, lit son contenu et l'affiche. On doit lancer en arrière plan le premier programme car un processus ayant ouvert un tube en écriture est suspendu tant qu'il n'y a pas de lecteur.

Client-serveur avec des tubes nommés

La gestion d'une application peut se faire en client-serveur en utilisant des tubes nommés comme moyen de communication interprocessus. Un tube dont le nom est connu des client reçoit des requêtes. Le serveur fournit la réponse en créant un tube pour chaque client. Les processus client et serveur sont indépendants mais communiquent au travers de tubes nommés. S'il n'y a aucun client, le processus serveur est bloqué en attente de lecture dans le tube. Il y'a autant de tubes nommés pour les réponses que de clients. Quand un client se termine, son tube nommé est détruit. L'envoi d'un message client (une requête) consiste en une écriture dans le tube nommé du serveur. La réception d'un message client consiste en la lecture dans le tube nommé du client.

Client-serveur avec des tubes nommés

Exemple : On suppose qu'un serveur a accès à un fichier `notes.txt` contenant les notes d'étudiants. Chaque client envoie ses requêtes dans le tube nommé `accesspoint` : il écrit dans le tube par `write(accesspoint,"PID-210+Toto",12);` signifiant qu'il souhaite connaître la note de l'étudiant Toto ("210" est son numéro de PID). Le serveur lit dans le tube `read(accesspoint,tamponreq,100);` (où `tamponreq` est un tableau de caractères). Il cherche la note correspondant à Toto dans `notes.txt`. Puis il crée le tube nommés `rep210` et écrit par `write(rep210,"14",2);` la note de Toto (14 en l'occurrence). Enfin, le client récupère la note par `read(rep210,tableaunote,20);` où `tableaunote` est un tableau de caractères.

Remarque

Dans cet exemple on a utilisé le protocole de communication suivant : les clients envoient leur requête sur le tube `accesspoint` puis reçoivent les réponses sur le tube `repPID` (où `PID` est le numéro de pid du client).