IUT Orsay

DUT Informatique
Module Système S4-L
2010 / 2011

Travaux Dirigés nº 2 : Synchronisation par tubes

Objectifs : savoir gérer les tubes (pipes) Unix et les utiliser dans le cas de la synchronisation de plusieurs processus.

1 Notes de cours

1.1 Avertissement

Ces notes de cours sont simplifiées à l'essentiel. Il est vivement conseillé de se référer à votre cours de Système S2 et au polycopié *Primitives Système sous UNIX* distribué en première année.

1.2 Tubes

Les tubes (*pipes* en anglais) sont des canaux de communication unidirectionnels manipulés à l'aide de descripteurs de fichiers. Tout processus ayant accès à ces descripteurs peut lire ou écrire dans un tube : seuls les processus descendant du créateur d'un tube, et le créateur lui-même pourront l'utiliser.
Un tube se comporte comme une file *fifo* (*first in, first out*) : les premières données entrées dans le tube seront les premières à être lues. Toute lecture est destructive : si un processus lit une donnée dans le tube, celle-ci n'y sera plus présente, même pour les autres processus.
Il y a simplement quatre primitives fondamentales pour les tubes :

```
#include <unistd.h>
int pipe (int descripteur[2]);
int write(int descripteur, char * buffer, int longueur);
int read (int descripteur, char * buffer, int longueur);
int close(int descripteur);
```

1. `pipe(descripteur)` crée un tube et retourne 0 en cas de succès, -1 en cas d'echec. Après appel :
   – `descripteur[0]` est le descripteur du tube créé *en lecture*,
   – `descripteur[1]` est le descripteur du tube créé *en écriture*.
2. `write(descripteur,buffer,longueur)` écrit dans le tube ayant `descripteur` pour descripteur en écriture les `longueur` octets commençant en mémoire à l'adresse `buffer`. Par exemple si le descripeur d'un tube en écriture est `p[1]` et qu'on veut y écrire la variable `v` de type `float`, on écrira `write(p[1],&v,sizeof(float))`. La primitive renvoie le nombre d'octets écrits ou -1 en cas d'echec.
3. `read(descripteur,buffer,longueur)` lit `longueur` octets (au maximum) dans le tube ayant `descripteur` pour descripteur en lecture et place ces octets en mémoire à partir de l'adresse `buffer`. Par exemple si le descripeur d'un tube en lecture est `p[0]`

et qu'on veut y lire un nombre flottant et le placer dans la variable `v`, on utilisera l'instruction `read(p[0],&v,sizeof(float))`. La primitive renvoie le nombre d'octets lus, ou 0 si le tube est vide ***et*** qu'il n'y a plus d'écrivains sur le tube ou enfin -1 en cas d'erreur. S'il reste un écrivain mais que le tube est vide, le processus se bloque en attente d'information.

4. `close(descripteur)` ferme le tube en lecture ou en écriture (en fonction du descripteur passé en argument), pour le processus ayant appelé `close` seulement. Elle renvoie 0 en cas de succès ou -1 en cas d'echec.

2 Exercices

2.1 Exercice 1 : tube simple

Une application est composée d'un processus père et d'un fils. Le fils réalise le travail suivant :

```
lire un réel x au clavier
tant_que x <> 0 faire
| si x < 0 alors
| | message d'avertissement
| sinon
| | transmettre f(x) au père
| demander un réel x au clavier
```

La fonction `f` est définie dans le code du fils. Le père reçoit les valeurs `y` (= `f(x)`). Lorsque le dernier `y` est reçu, il calcule la moyenne des `y` et l'affiche à l'écran. Les valeurs sont transmises par un tube.

1. On suppose que $\forall x \in \mathbb{R}, f(x) > 0$. Comment le père peut-il détecter la fin de la suite des valeurs reçues ? Écrire les algorithmes et les programmes C du père et du fils dans cette hypothèse.
2. On ne fait aucune hypothèse sur `f`. Proposer une solution pour la détection de fin d'arrivée des valeurs reçues par le père et en donner les algorithmes puis les programmes C correspondants.

2.2 Exercice 2 : interblocages

Une application est composée d'un processus père et d'un fils. Le père écrit dans le tube `p` et le fils lit à partir de ce tube. Après initialisations et création du fils, les codes du père et du fils sont :

```
/* Père */
write(p[1],"abcdef",6) ;
sleep(10) ;
pid_fils = wait(&etat) ;
exit(0) ;
```

```
/* Fils */
r = read(p[0],&c,1) ;
while (r != 0) {
| printf("...") ;
| r = read(p[0],&c,1) ;
}
exit(0) ;
```

Cette application est erronée. Indiquer ce qui se produit et expliquer pourquoi. Proposez une solution.

2.3 Exercice 3 : synchronisation

Un travail à réaliser comporte une partie initiale globale (notée `TI`), deux travaux `T1` et `T2` chacun décomposé en 3 parties (`T1a`, `T1b`, `T1c` et `T2a`, `T2b`, `T2c`) et une partie finale `TF`. Les travaux `T1` et `T2` peuvent être réalisés simultanément avec les réserves suivantes, où «début» et «fin» indiquent les instants de début et de fin d'une partie donnée :
– début(`T2a`) $\geq$ fin(`T1a`),
– début(`T1c`) $\geq$ fin(`T2b`).

1. En supposant qu'on travaille sur une machine mono-processeur, donner deux exemples d'exécution où les travaux $\texttt{T1}_i$ et $\texttt{T2}_j$ ne sont pas exécutés dans le même ordre.
2. Dessiner le graphe de précédence de l'application.
3. Implanter ce travail à réaliser à l'aide de deux processus (un père et un fils), le fil réalisant le travail `T2`, la synchronisation étant réalisée grâce à des tubes.

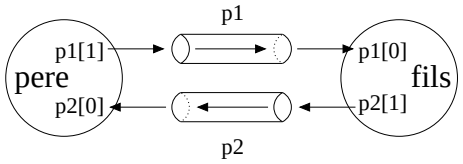
3 Entraînement : exercice corrigé

3.1 Énoncé : communication bidirectionnelle

Donner l'organisation d'une application de transmission bidirectionnelle d'informations entre un processus père et un de ses fils via des tubes : le père envoie 5 entiers au fils qui les affiche et renvoie ces entiers multipliés par 2. Le père affiche ces doubles. Écrire ensuite le programme C correspondant.

3.2 Correction (essayez d'abord ! ! !)

Dans ce problème, on crée deux tubes `p1` et `p2` pour faire communiquer les deux processus :
– le père a accès en écriture sur `p1` et en lecture sur `p2`,
– le fils a accès en écriture sur `p2` et en lecture sur `p1`.



Un programme possible est le suivant. Pour des raisons de place, les tests d'echec des diverses primitives ne sont pas réalisés, ils sont bien sûr à ajouter dans un travail sérieux.

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

#define  NB_ENTIERS 5

void fils(int p1[2], int p2[2])
{
  int i, nombre, nb_lus;

  close(p1[1]);    // Fermeture du tube p1 en ecriture pour le fils
  close(p2[0]);    // Fermeture du tube p2 en lecture pour le fils
  nb_lus = read(p1[0], &nombre, sizeof(int));  // Lecture nombre sur p1
  while(nb_lus == sizeof(int))
  {
    nombre = nombre*2;                          // Calcul du double
    printf("Les doubles sont :%d\n", nombre);
    write(p2[1], &nombre, sizeof(float));       // Ecriture double sur p2
    nb_lus = read(p1[0], &nombre, sizeof(int));// Lecture nombre sur p2
  }
  close(p1[0]);    // Fermeture du tube p1 en ecriture pour le fils
  close(p2[1]);    // Fermeture du tube p2 en lecture pour le fils
  exit(0);
}

int main()
{
  int i, nombre,
      p1[2],       // Descripteurs du tube p1
      p2[2];       // Descripteurs du tube p2

  pipe(p1);        // Creation du tube p1
  pipe(p2);        // Creation du tube p2

  if (fork() == 0)// Creation du fils
    fils(p1,p2);  // Code du fils
  else
  {
    close(p1[0]); // Fermeture du tube p1 en lecture pour le pere
    close(p2[1]); // Fermeture du tube p2 en ecriture pour le pere
    for (i=0; i<NB_ENTIERS; i++)
    {
      printf("Entrez un entier\n");       // Demande d'un entier
      scanf(" %d", &nombre);              // Saisie d'un entier
      write(p1[1], &nombre, sizeof(int));// Ecriture de l'entier sur p1
    }
    close(p1[1]); // Fermeture du tube p1 en ecriture pour le pere

    printf("Les doubles sont :\n");
    for (i=0; i<NB_ENTIERS; i++)
    {
      read(p2[0], &nombre, sizeof(int)); // Lecture d'un entier sur p2
      printf("%d ", nombre);             // Affichage de l'entier
    }
    printf("\n"); // Important pour forcer l'affichage
    close(p2[0]); // Fermeture du tube p2 en lecture pour le pere
  }
  return 0;
}
```