iUT ORSAY
Département Informatique

DUT Informatique
Module Système S4 C
2009 / 2010

Travaux Dirigés n° 4 : Multiplexage

Objectifs : savoir créer une application capable de gérer plusieurs connections sans recourir à plusieurs processus avec `select`.

1 Notes de cours

La manière traditionnelle de construire une application serveur réseau est de constuire un bloc principal attendant une connection par la primitive `accept`. Dès qu'une demande de connexion survient, on crée un processus fils par `fork` : le fils prend en charge la communication et le père se remet en position d'attente d'une nouvelle connexion.
La primitive `select` permet au contraire de construire un serveur autour d'un seul processus qui *multiplexe* les demandes de connexion et qui les prend en charge aussi bien qu'il le peut. L'avantage de l'utilisation de `select` est que le serveur n'est qu'un seul processus, il n'y a donc pas besoin de primitives de synchronisations ou de communication entre processus. Le principal désavantage est bien sûr de devoir gérer plusieurs connexions en même temps alors que chaque processus travaillait avec une seule connexion dans la solution utilisant `fork`.
La primitive `select` est basée sur le concept de `fd_set` qui sont des ensembles de descripteurs de fichiers (File Descriptor Sets). On manipule ce type de données en utilisant les macros standard suivantes :

```
fd_set ensemble;
FD_ZERO(&ensemble);                 /* Vide ensemble. */
FD_SET(descripteur,&ensemble);      /* Ajoute descripteur a ensemble. */
FD_CLR(descripteur,&ensemble);      /* Retire descripteur de ensemble. */
FD_ISSET(descripteur,&ensemble); /* Vrai si descripteur dans ensemble. */
```

Son prototype est le suivant :

```
#include <sys/select.h>
int select(
    int nb_descripteurs,
    fd_set * ensemble_lecture,
    fd_set * ensemble_ecriture,
    fd_set * ensemble_exceptionnel,
    struct timeval * delai
);
```

1. `nb_descripteurs` : nombre de descripteurs à examiner. Il est égal au plus grand descripteur de tous les ensembles + 1 (et non au nombre total de descripteurs, en gros on prend le dernier créé + 1).
2. `ensemble_lecture` : pointeur vers l'ensemble de descripteurs à examiner en lecture (`NULL` est l'ensemble vide).
3. `ensemble_ecriture` : pointeur vers l'ensemble de descripteurs à examiner en écriture (`NULL` est l'ensemble vide).

4. `ensemble_exceptionnel` : pointeur vers l'ensemble de descripteurs à examiner pour un état exceptionnel, par exemple les messages urgents (`NULL` est l'ensemble vide).
5. `delai` : pointeur vers une structure `timeval` donnant le delai d'attente maximum. Mettre `NULL` pour un temps infini.

Cette primitive retourne le nombre de descripteurs *prêts* et les trois ensembles sont modifiés pour contenir les ensembles de descripteurs prêts (cela signifie aussi qu'il faudra remettre les ensembles comme il faut avant un nouvel appel). Elle est bloquante jusqu'à ce que le delai d'attente soit fini ou qu'un des événements attendus se produise. En cas de réception d'un signal par le processus, elle retourne -1.

2 Exercices

2.1 Exercice 1 : test simple

On cherche à disposer d'une fonction `isready` pour tester si une socket donnée est prête en lecture. Cette fonction prendra comme unique paramètre un descripteur (type entier `int`) et renverra 1 si le descripteur passé en paramètre est prêt en lecture, 0 sinon ou -1 en cas de problème. Indication : le code suivant prépare une structure `delai` de type `struct timeval` à zéro secondes :

```
struct timeval delai; /* Declaration. */
delai.tv_sec  = 0;    /* Zero seconde... */
delai.tv_usec = 0;    /* Et zero micro-seconde. */
```

2.2 Exercice 2 : multiplexage *

Pour des raisons pratiques, un serveur dédié au calcul de puissances de deux (requêtes sous la forme d'entiers non nuls n de type `int` et réponses 2^n sous la forme d'entiers de type `int`) doit être disponible à la fois depuis le réseau *via* une connexion de type socket datagramme et depuis l'entrée et la sortie standard sur la machine où il s'exécute. Réalisez un tel serveur sans recourir à `fork` pour ne pas polluer la table des processus.
Pour éviter tout interblocage il peut être utile de rendre la lecture sur un fichier (une socket par exemple) non bloquante. Cela peut se faire de la manière suivante, à l'aide du descripteur du fichier :

```
#include <fcntl.h>
fcntl(descripteur,F_SETFL,O_NDELAY); /* Lecture non bloquante */
```

Voir la documentation de `fcntl` (file control) pour plus de détails.

3 Entraînement : exercice corrigé

3.1 Énoncé : tubes spécialisés

Un processus père crée deux processus fils. Le premier fils envoie à son père un flux de caractères au travers d'un tube (ce fils réalise une boucle infinie d'écritures d'un caractère sur le tube et s'endort une seconde entre deux écritures). Le second fils réalise la même opération mais avec un flux d'entiers, sur un autre tube. Le processus père se mettra en attente d'informations sur les deux tubes à l'aide de la primitive `select()` et affichera chaque élement reçu, caractère ou entier. Réalisez cette application.

3.2 Correction (essayez d'abord ! ! !)

Il s'agit simplement d'une application des notes de cours. Le processus père crée les deux tubes et les deux fils (attention à ne pas oublier de fermer tous les descripteurs inutiles, en particulier ceux du tube inutile pour le fils qui ne s'en sert pas). Il crée un ensemble de descripteurs, y place ceux des deux tubes, utilise la primitive select pour s'endormir en attente d'information sur l'un des deux tubes (`select()` s'applique sur des descripteurs de fichier, donc tubes, sockets, fichiers etc. ensembles ou mélangés ce qui rend cette primitive extrêmement utile). À la sortie de `select()`, le père teste quel descripteur présente une information, la lit et l'affiche.

```c
#include <stdio.h>                          /* Fichiers d'en-tete classiques */
#include <stdlib.h>
#include <sys/types.h>
#include <sys/select.h>

int tube_car[2];
int tube_int[2];

void fils_car(void) {                       /* Code du fils caractere */
  char c = 'A';
  close(tube_car[0]);                       /* Fermeture tube en lecture */

  while(1) {
    write(tube_car[1],&c,sizeof(char));     /* Envoi d'un caractere */
    sleep(1);
    c++;
  }
}

void fils_int(void) {                       /* Code du fils entier */
  int i = 0;
  close(tube_car[0]);                       /* Fermeture descripteurs inutiles */
  close(tube_car[1]);
  close(tube_int[0]);

  while(1) {
    write(tube_int[1],&i,sizeof(int));      /* Envoi d'un entier */
    sleep(1);
    i++;
  }
}

int main(int argc, char *argv[]) {          /* Code du pere */
  int i, c;
  fd_set ensemble;                          /* Declaration de l'ensemble */

  pipe(tube_car);                           /* Creation tube caractere */
  if (fork())                               /* Creation fils caractere */
    fils_car();
  else {
    pipe(tube_int);                         /* Creation tube entier */
    if (fork())                             /* Creation fils entier */
      fils_int();
  }

  close(tube_car[1]);                       /* Fermeture descipteurs inutiles */
  close(tube_int[1]);

  while(1) {
    FD_ZERO(&ensemble) ;                    /* On initialise l'ensemble a vide */
    FD_SET(tube_car[0],&ensemble);          /* Ajout du tube caracteres */
    FD_SET(tube_int[0],&ensemble) ;         /* Ajout de tube entier */

    select(tube_int[0]+1,&ensemble,NULL,NULL,NULL);

    if (FD_ISSET(tube_car[0],&ensemble)) { /* Si le tube caractere est pret */
      read(tube_car[0],&c,sizeof(char));
      printf("Reception d'un caractere : %c.\n",c);
    }
    if (FD_ISSET(tube_int[0],&ensemble)) { /* Si le tube entier est pret */
      read(tube_int[0],&i,sizeof(int));
      printf("Reception d'un entier : %d.\n",i);
    }
  }

  return 0;
}
```