TD 10 : Exercices de type Examen

Objectifs pédagogiques :

- **révisions**

Introduction : Dans ce TD, nous allons réviser les briques essentielles de la concurrence et du parallélisme vus dans l'UE. Les exercices[1] sont censés être accessibles sans assistant de TD.

1 Simulation des Pipes en Thread

Nous voulons offrir un ensemble de fonctions qui permet à des threads de communiquer par un tube. Autrement dit, un mécanisme de communication unidirectionnel où chacune des extrémités permet à une thread soit de lire dans un tube soit d'y écrire des caractères. A cette fin, nous avons défini la structure suivante qui regroupe des informations d'un tube donné :

```
class tube {
  char vect [TAILLE_PIPE]; /* vecteur pour sauvegarder le contenu du tube */
  int lect_off, ecr_off; /* indice de la prochaine case de vect à lire ou libre
      respectivement */
  int nb_char; /* nombre de caractères dans le tube */
public:
  tube():lect_off(0),ecr_off(0),nb_char(0){}
};
```

où :

- `vect [TAILLE_PIPE]` : vecteur de taille `TAILLE_PIPE` qui sauvegarde les caractères qui n'ont pas été encore lus dans le tube. Ce vecteur est géré de façon circulaire ;
- `lect_off et ecr_off` : indice de la prochaine case de vect à lire ou à écrire respectivement ;
- `nb_char` : nombre de caractères dans `vect` ;

On propose de réaliser les méthodes suivantes du tube, conçues pour un contexte multi-thread.

- Lecture de n caractères dans le tube : `int read (char *buf, int n);`.
  - Fonction bloquante qui lit au plus n caractères du tube.
  - Si le tube contient x caractères, la fonction extrait du tube $nb_lu = min(x, n)$ caractères qui sont alors copiés dans buf;
  - Si le tube est vide, la thread est mise en sommeil jusqu'à ce que le tube ne soit plus vide;
  - La fonction renvoie le nombre de caractères lus dans le tube (nb_lu caractères).
- Ecriture de n caractère dans le tube : `int write (char *buf, int n);`.
  - Fonction qui écrit de façon atomique n caractères dans le tube, s'il y a de la place.
  - S'il y a au moins n emplacements libres dans le tube, une écriture atomique est réalisée et le nombre de caractères écrits est renvoyé. Dans ce cas tous les threads lecteur en attente sur le tube seront réveillées.
  - Sinon la fonction renvoie -1.

Nous considérons que les programmes utilisent correctement les fonctions, c'est-à-dire qu'une thread ne peut que lire ou écrire dans un tube.

Question 1. Ajouter les attributs utiles à la classe tube et codez ces deux méthodes. On suggère de procéder caractère par caractère pour plus de facilité, ou de faire deux cas selon qu'on déborde ou non (stockage circulaire) et une à deux copies (avec `memcpy`) selon le cas.

[1] Les deux premiers exercices sont adaptés d'une annale de 2006, rédigée par L. Arantes, O. Marin et P.Sens.

Question 2. On suppose à présent que l'on ajoute un attribut nbThread comptabilisant le nombre de threads au total qui ont une référence vers une le tube. On suppose que cet attribut est correctement mis à jour au fil de la manipulation du pipe (par exemple en appui sur `shared_ptr`). Modifier la fonction write pour qu'elle délivre un signal SIGPIPE au thread qui invoque write s'il est le seul à pouvoir réaliser une lecture.

Question 3. Quel sera l'effet du signal sur la thread dans write ? Sur les autres threads du programmes ?

2 IPC

Un programmeur très (mais alors vraiment très) inexpérimenté a voulu rédiger une petite application dans laquelle un processus émetteur envoie un message mot par mot à deux processus récepteurs de sa famille. Le résultat recherché de l'application est de faire afficher (NB : une seule fois) le message original dans son ordre initial. Par exemple, si l'émetteur envoie successivement les mots "je", suis", "en", "examen", l'affichage final de l'application sera : "je suis en examen". Le code produit par notre programmeur égaré est le suivant :

lost.cpp

```
#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <sys/types.h>
#include <sys/wait.h>


int i, n = 6;
int pid;

void send(char *buf) {
      const char *msg[] = {"Je", "suis", "un", "vilain", "programme", "\n"};
      for(i = 0; i < n; i++) {
            strcpy(buf, msg[i]);
      }
      exit(0);
}

void receive(char *buf) {
      if ((pid = fork()) == -1) {perror("fork"); exit(1);}
      for(i = 0; i < (n/2); i++) {
            printf("%s ", buf);
      }
}

int main()
{
      char *shared = new char[50];

      if ((pid = fork()) == -1) {perror("fork"); exit(1);}

      if (pid == 0)
            send(shared);
      else
            receive(shared);

      if (pid == 0) exit(0);
      for (i = 0; i < 2; i++)
            wait(0);
```

```
      printf("fin du programme\n");
      delete [] shared;
      return 0;
}
```

Question 1. Expliquez la mise en place du parallélisme, en particulier identifier l'arborescence des processus engendrés et le code exécuté par chacun d'eux.

Question 2. Expliquez en deux points pourquoi ce programme ne peut pas fonctionner.

Question 3. Sans altérer les créations de processus, corrigez le code à l'aide d'IPC afin que l'application fonctionne.

Question 4. Quelles modifications faut-il apporter pour que chacun des processus récepteurs affiche le message dans son intégralité ? Voici un exemple de résultat attendu :

```
Récepteur1> Je
Récepteur2> Je
Récepteur2> suis
Récepteur1> suis
..
```

3 Primitive select

On fournit une classe `ServerSocket`basée sur le TD 7.

ServerSocket.h

```
#ifndef SRC_SERVERSOCKET_H_
#define SRC_SERVERSOCKET_H_
#include "Socket.h"
namespace pr {

class ServerSocket {
      int socketfd;
public :
      // Demarre l'ecoute sur le port donne
      ServerSocket(int port);

      int getFD() { return socketfd;}
      bool isOpen() const {return socketfd != -1;}

      Socket accept();
      void close();
};
} // ns pr
#endif /* SRC_SERVERSOCKET_H_ */
```

ServerSocket.cpp

```
#include "ServerSocket.h"
#include <cstring>
#include <unistd.h>
#include <iostream>
namespace pr {
ServerSocket::ServerSocket(int port) :socketfd(-1) {
      // create socket
      int fd = socket(AF_INET,SOCK_STREAM,0);
      if (fd == -1) {
            perror("create socket");
```

```
            return;
      }

      // bind
      struct sockaddr_in sin; /* Nom de la socket de connexion */
      memset(&sin, 0, sizeof(sin)); // utile ?
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = htonl(INADDR_ANY); // on attend sur n'importe quelle interface
          de la machine
      sin.sin_port = htons(port); // host to network

      /* nommage, meme probleme de typage sockaddr que dans la Socket */
      if (bind(fd, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
            perror("bind");
            // on veut le close de unistd, pas le close membre de cette classe
            ::close(fd);
            return;
      }

      // listen
      if (listen(fd, 50) < 0) {
            perror("listen");
            ::close(fd);
            return;
      }

      // success !
      socketfd = fd;
}

Socket ServerSocket::accept() {
      struct sockaddr_in exp;
      socklen_t len = sizeof(exp);
      // on qualifie sinon le compilo rale
      int scom = ::accept(socketfd, (struct sockaddr *) &exp, &len);
      if (scom < 0) {
            perror("accept");
      } else {
            // en appui sur operator << pour (sin_addr *) développé plus haut
            std::cout << "Accepted connection from " << &exp << std::endl;
      }
      return scom;
}

void ServerSocket::close() {
      if (socketfd != -1) {
            ::close(socketfd);
            socketfd = -1;
      }
}
} // ns pr
```

Question 1. Coder dans ServerSocket (fourni) un comportement tel que si on est bloqué dans `accept()` et qu'un autre thread appelle `close`, on sort de l'accept (en rendant une Socket non connectée `sockFD` vaut -1).

On référera aux numéros de lignes pour indiquer où faire les modifications.