Examen Réparti 2 PR
Master 1 Informatique Jan 2020

UE 4I400

Année 2019-2020

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones. . .)

Introduction

- Le barème est sur 20 points et est donné à titre indicatif et susceptible d'être modifié.
- Il est demandé de répondre précisément et concisément aux questions.
- Dans le code C++ demandé vous vous efforcerez d'ecrire du code correct et compilable, mais les petites typos ne seront pas sanctionnées.
- De même les problèmes d'include et de namespace ne sont pas pertinents (pas la peine de qualifier les `std::`, ni de citer tous les `#include`)
- On suppose également que **tous les appels système se passent bien**, on ne demande pas de contrôler les valeurs de retour (donc pas de `perror` ou `errno.h`).

Le sujet est composé de quatre exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite.

1 Shell et redirections (4,5 points)

On considère la réalisation d'une partie d'un interprète de commande type bash. On souhaite traiter l'exécution d'une commande arbitraire avec redirection de la sortie dans un fichier.
Un exemple d'invocation : `./redirect /bin/ls -l > log.txt`.

> Evidemment en ligne de commande, penser à protéger le ">" avec des guillemets, sinoon bash va l'interpréter.

Question 1. (3,5 points) Ecrire un programme `main` qui :

- contrôle que l'avant dernier argument est bien une chaîne ">"
- considère que le dernier argument est un nom de fichier, qui sera créé s'il n'existe pas. Dans ce cas, les droits du fichier seront positionnés sur : `-rw-rw-r-` (read et write pour user et group, read pour other).
- invoque la commande correspondant aux arguments précédant le ">" dans un nouveau processus en *redirigeant* la sortie du processus dans ce fichier
- attend la fin de cette exécution et sort proprement

redirect.cpp

```cpp
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <cstring>
#include <cstdio>

int main (int argc, char ** argv) {
      if (strcmp(argv[argc-2],">")) {
            exit(1);
      }
      // le nom de fichier
      const char * file = argv[argc-1];
      // On prépare les args pour exec
      char * const * commande = argv+1;
      // garde avec une sentinelle les arguments 1 à argc-2
      argv[argc-2] = nullptr;

      // fork
      if (fork()==0) { // cas fils
            // ouverture du fichier : 00664
            int fd = open(file, O_WRONLY | O_CREAT , 00664);
            if (fd < 0) perror("open");

            // redirection
            dup2(fd,STDOUT_FILENO);
            // close du fichier
            close(fd);

            // exec
            if (execv (commande[0], commande)<0) {
                  perror("exec");
                  exit(1);
            }
            // fils ne revient pas ici si exec fonctionne
      }
      // pere
      wait(nullptr);
      return 0;
}
```

Barème :

20 % fork + wait

30 % open, dans fils ou pere

10 % close, au moins un dans fils ou pere

20 % exec

20 % dup2

Question 2. (1 point) Que faut il modifier dans le programme pour que si le fichier existe, on ajoute à la fin du fichier plutôt que d'écraser son contenu ? (i.e. la redirection symbolisée par ">>" en shell).

Simplement le “open” avec le flag `O_APPEND` qui va bien.
On peut aussi proposer d’utiliser une variante de lseek.
Barème :
binaire

2 Serveur TCP (5,5 points)

On souhaite construire un serveur de diffusion utilisant le protocole TCP. Le serveur attend les connections de clients sur le port “1664”; il peut traiter plusieur clients simultanément.

Chaque fois qu’un client écrit un message au serveur (les messages ont une taille fixe de 128 octets), celui-ci doit renvoyer le message reçu à tous les clients connectés.

L’objectif est de réaliser ce serveur de diffusion sans aucune concurrence (processus ou thread).

Question 1. (1,5 point) Sans fournir de code, expliquer informellement les étapes (appels système utiles) nécessaires avant de pouvoir *accepter* des connexions TCP sur le port donné. Expliquez en une ligne le rôle de chaque appel système utilisé, et la nature de ses arguments.

On attend le trio : socket, bind, listen.
socket : ouvre une socket, avec le domaine INET, le type STREAM et le protocole TCP. Fournit un filedescriptor pour cette socket.
bind : nomme la socket, et la déclare au niveau réseau. Arguments : une addresse IP (INADDR_ANY + un numéro de port (1664). Quelques vilains cast sont nécessaires (sockaddr, sockaddr_in...) mais ce n’est pas la question.
listen : déclare que c’est une socket d’attente de connexion et donne la taille de la file d’atttente.
Barème : 10 % chacun = citer la fonction
10 % chacun = expliquer les arguments
15 % (10 pour listen) = expliquer à quoi ça sert.
Soit :
35 % socket.
35 % bind.
30 % listen.

Question 2. (1 point) On souhaite stocker dans un `vector<int>` les filedescriptor correspondant aux sockets des clients conectés au serveur. Ecrivez une fonction : `void handleClientConnexion(int fdattente, vector<int> & clients)` qui traite l’arrivée d’un nouveau client qui vient de tenter une connexion sur le port 1664.

L’argument *fdattente* désigne le filedescriptor représentant la socket à laquelle les clients se connectent, et l’argument *clients* doit être mis à jour dans la fonction pour stocker le filedescriptor de la socket de discussion avec le client.

```
clients.push_back(accept(fdattente,nullptr,nullptr));
```

Barème :
60 % On invoque accept (on peut passer une struct sockaddr pour l’expéditeur).
40 % Le résultat est ajouté dans clients

Question 3. (1,5 points) Donner le code de la fonction `void readAndBroadcast(int readfd, const vector<int> & clients)`, qui lit un message sur `readfd` et le rediffuse sur tous les clients représentés dans `clients`.

```
char [128] buff;
read(readfd,buff,128);
for (int cli : clients) {
write(cli,buff,128);
}
```

Barème :
30 % read ou recv correct sur la socket readfd
40 % la boucle itère pour envoyer à tous
30 % l'écriture vers les autres clients.
-20 % si on envoie/lit autre chose que 128 bytes.

Question 4. (1,5 points) Sans fournir de code, expliquer informellement les étapes (appels système utiles) nécessaires pour pouvoir attendre simultanément une demande de connexion ou un message d'un des clients connectés avec un seul processsus, et le traiter. Expliquez en une ligne le rôle de chaque appel système utilisé, et la nature de ses arguments.

Il nous faut un select ou un poll. Select permet d'attendre plusieurs fd simultanément : on passe des ensembles de fd à surveiller.
Au réveil, on peut savoir quels sockets sont prêts et les traiter avec la question 2 ou la question 3.
On attend donc que l'une des sockets client soient prêtes à read, ou que la socket d'attente de connection soit prête pour un accept.
Barème :
50 % on cite select ou poll
50 % on explique ses arguments = toutes les sockets(fd) concernées et comment on traite.

3 Famille de processus (6 points)

Question 1. (1,5 points) Ecrire un programme qui crée trois processus au total : le grand-père, le père et le petit-fils. Le grand-père affichera "GP" et son pid, le pere "P" et son pid et le petit-fils "PF" et son pid. On souhaite que l'ordre de ces affichages ne soit pas contraint ; cependant le programme doit se terminer proprement quand tous les affichages sont finis.

on fork en cascade, on wait son fils si on en a. getpid pour le pid.

```
if (fork()==0)
{
  if (fork()==0)
  {
   printf("PF %d", getpid()) ;
    exit(0);
  } else {
    printf("P %d", getpid()) ;
    wait(0);
  }
} else {
  printf("GP %d", getpid()) ;
  wait(0);
}
```

Barème :
40 création de trois processus en chaîne
20 chacun affiche son pid + message approprié
20 l'ordre des affichages est arbitraire (précède les wait).
20 les wait (2 occurrences) attendent bien le fils avant de terminer.

Question 2. (1,5 points) Sans fournir de code, expliquer informellement les étapes (appels système utiles) nécessaires pour pouvoir afficher dans le petit-fils le pid de son grand-père, et dans le grand-père le pid de son petit-fils. Expliquez en une ligne le rôle de chaque appel système utilisé, et la nature de ses arguments.

NB: on suppose dans la suite que *gpid* et $pfid$ sont des variables désignant les pid du grand-père et du petit-fils, définies et acessibles dans tous les processus.

GP-> PF : Donc il faut simplement stocker dans une variable le pid du GP avant de fork la première fois. On fait $intgpid = getpid()$; avant le premier fork de question 1.
PF-> GP : Il faut un IPC, le plus simple est sans doute un pipe ici. Donc, on ouvre un pipe avant le premier fork.
pipe : rend deux filedescriptor extrémités read et write d'un tube anonyme.
soit le P (en capturant la valeur de retour de fork) soit le PF (avec getpid) écrit dans l'extrémité write du pipe le pid du petit-fils.
Le GP read le pid dans l'extrémité read du pipe.
Tout le monde close gentiment les extrémités du pipe qu'il n'utilise pas ou plus (après le read/write).
Barème :
30 % la façon de passer *gppid* à petit-fils : stocker avant le fork
30 % on a besoin d'un IPC pour le sens PF->GP : on suggère pipe et on décrit son appel système. Ou un autre IPC (attention si on propose fichier ou shm, il faudra aussi un sémaphore).
30 % on explique qui doit écrire le pid (PF ou P) et qui le lit (GP)
10 % on parle de close

Question 3. (1,5 points) On souhaite garantir que l'affichage du grand-pere soit réalisé *après* l'affichage du petit-fils. Expliquez comment réaliser ce comportement à l'aide de signaux: donner le code à exécuter par le grand-père pour que le signal reçu déclenche un affichage, et par le petit-fils pour envoyer le signal.

PF : kill(gpid,SIGUSR1);
GP : `signal(SIGUSR1, [](int){ printf("GP");})`
Barème :
50 % utilisation de kill, après l'affichage du petit-fils
50 % position d'un handler : avec signal ou sigaction.

Question 4. (1,5 points) Sans fournir de code, expliquer informellement les étapes (appels système utiles) nécessaires, position dans le code déjà élaboré, pour que le grand-père ne puisse recevoir le signal que *après* avoir réussi à obtenir le pid de son petit-fils. Expliquez en une ligne le rôle de chaque appel système utilisé, et la nature de ses arguments.

Il faut que le grand-pere se masque jusqu'a avoir fini de read sur le pipe.
Donc avant le premier fork (sinon c'est trop tard, dès le premier fork on peut commencer à se prendre le signal), on place un masque qui protège de SIGUSR1.

sigprocmask : met à jour le masque de signaux du processus, un bitset qui retarde les traitements associés. On masque SIGUSR1.
sigsuspend : remplace le masque par l'argument, et attend un signal/traite le signal pending éventuel.
Enfin après le read réussi du pid du pf, on démasque avec un sigsuspend.
Barème :
30 % sigprocmask : cité (10) + 20 décrire arguments
30 % sigsuspend : cité (10) + 20 décrire arguments
20 % on masque bien AVANT le fork
20 % on démasque/sigsuspend bien APRES avoir lu le pid du fils et AVANT de faire son propre affichage.

gppf.cpp

```cpp
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <iostream>

pid_t GP,P,PF;

int main () {

	GP = getpid();

	int pdesc[2];
	if (pipe(pdesc) != 0) {
		perror("pipe error");
		return 1;
	}

	signal(SIGUSR1,[](int) {std::cout << "Je suis :" << getpid() << " GP=" << GP << " P=" << P << " PF=" << PF << std::endl; });

	sigset_t set;
	sigset_t old;
	sigemptyset(&set);
	sigaddset(&set, SIGUSR1);
	sigprocmask(SIG_BLOCK, &set, &old);

	P = fork();
	if (P < 0) {
		perror("fork error");
		return 1;
	} else if (P == 0){
		P = getpid();
		// Pere
		PF = fork();
		if (PF < 0) {
			perror("fork error");
			return 1;
		} else if (PF == 0){
			// PF
			PF = getpid();
			close(pdesc[0]);
			if (write(pdesc[1],&PF,sizeof(pid_t)) == sizeof(pid_t)) {
				std::cout << "Je suis :" << getpid() << " GP=" << GP << " P=" << P << " PF=" << PF << std::endl;
				kill(GP,SIGUSR1);
			} else {
				perror("erreur write");
```

```
                        return 1;
                  }
                  close(pdesc[1]);
                  return 0;
            } else {
                  // PERE
                  close(pdesc[1]);
                  close(pdesc[0]);
                  wait(0);
                  return 0;
            }
      } else {

            // GP
            if (read(pdesc[0],&PF,sizeof(pid_t)) == sizeof(pid_t)) {
                  sigset_t mask;
                  sigfillset(&mask);
                  sigdelset(&mask, SIGUSR1);
                  sigsuspend(&mask);
                  // via signal à la fin
                  // std::cout << "Je suis :" << getpid() << " GP=" << GP << " P=" << P
                      << " PF=" << PF << std::endl;
            } else {
                  perror("erreur read");
                  return 1;
            }
            close(pdesc[0]);
            close(pdesc[1]);
            wait(0);
            return 0;
      }
}
```

4 Questions de synchronisation (4 points)

Question 1. (2 points) On vous demande le code de `void multilock (vector<mutex> & mutexes, vector<int> indexes)` où l'objectif est d'acquérir le sous ensemble des mutex dans le vector *mutexes* désignés par les indices dans *indexes*. Cette fonction est invoquée dans un contexte concurrent et ne doit jamais provoquer d'interblocage.

Par exemple on peut demander à acquérir les lock $indexes = \{8, 3, 5\}$ soit les verrous $mutexes[8]$, $mutexes[3]$, $mutexes[5]$.

Donnez le code de cette fonction *multilock* et de son symétrique `void multiunlock (vector<mutex> & mutexes, vector<int> indexes)` qui libère les mutex aux indices désignéss.

> On doit trier indexes avant de travailler, en C++ : `std::sort(indexes.begin(),indexes.end());`
> Ensuite un bon foreach : `for (int index:  indexes) mutexes[index].lock();`
> Le unlock n'a pas besoin de faire de tri vu que ce n'est pas bloquant.
> Barème :
> 40 % on dit qu'il faut trier les indexes
> 10 % syntaxe du sort correcte
> 20 % on lock bien tout le monde

30 % unlock de tout le monde.

Question 2. (2 points) On suppose une interaction entre deux tâches qui échangent des données complexes (contenant des pointeurs, taille variable) que l'une produit et l'autre consomme. En max deux lignes par réponse, quels mécanismes faut-il mettre en place dans un contexte où :

1. Les deux tâches sont des threads au sein d'un seul processus
2. Les deux tâches sont des processus hébergés sur la même machine
3. Les deux tâches sont des processus distants, sur des machines différentes d'un même réseau

1. Les deux tâches sont des threads au sein d'un seul processus : des mutex et/ou des atomic pour protéger les accès des data-race; des conditions_variable pour que l'un puisse attendre la production/consommation de l'autre (cf exo sur la file en TD)
2. Les deux tâches sont des processus hébergés sur la même machine : si on peut borner la taille des données, préparer le terrain, on peut s'en sortir avec du shared memory + semaphore pour protéger les accès et gérer les notifications. Mais c'est vite compliqué. Avec des données qui sont vraiment complexes, on préfères souvent une copie plutot : donc on utilise un pipe + une (dé)sérialisation des données (plutot que shm+sem).
3. Les deux tâches sont des processus distants, sur des machines différentes d'un même réseau : on utilises des sockets + une (dé)sérialisation des données

Barème :

40 % thread : 20% il faut des mutex et/ou atomic, 20 % il peut aussi falloir des cond si on s'attend

30 % processus : 10 % il faut un IPC 20% on propose soit shm+sem, soit pipe

30 % intermachine : sockets (15) + sérialisation (15)