rapport
client/serveur

mohamed*rida***hanti**
mohammed**mahdoute**

LES TUBES

A-ECHANGE DE CIVILITE ENTRE PERE ET FILS

Le programme suivant effectue un échange de chaines de caractères entre un processus père et ses cinq fils, la génération des processus est faite par la primitive *fork(),* tandis que la communication est établie par des tubes anonymes (*pipe()*).

Nous allons désormais détailler quelques fragements de code source du projet :

```
int i, status, tmp;
pid_t pID;
int pP[CHILDS_NBR][2], pC[CHILDS_NBR][2];
char str[15] ;
```

Dans le programme principal, on déclare les variables nécessaires parmis lesquelles on note deux tableaux bidimensionnels d'entiers, qui ne sont autres que les tubes que nous allons initialiser par la suite.
pP[CHILDS_NBR][2] : pour « parent pipes » qui établi un lien entre le père et ses diffèrents fils, de façon à ce que, pour chacun de ces tubes, le père pourra alors écrire à son propre fils et le fils lire sur le même tube.

pC[CHILDS_NBR][2] : pour « childs pipes » qui lui établi un lien mutuel entre les fils et leurs père, de façon à ce que chacun des fils puisse écrire sur un tube, donnant ainsi la main à son parent de lire sur le même tube.

```
for(i = 0 ; i < CHILDS_NBR ; i++) {
      if(pipe(pP[i]) != 0 || pipe(pC[i]) != 0) {
            printf("impossible de créer les tubes\n");
            exit(0);
      }
}
```

Grâce à une boucle, on installe les tubes entre père et fils.
La primitive *pipe()* retourne 0 si elle est executée avec succès, en cas d'échec, elle retourne -1.

Le processus père enregistre son pid dans le tube convenant, le processus va, alors, lire la valeur depuis le tube puis enregistrer le message « merci !! » dans le tube convenant afin que père puisse le lire, finalement, ce dernier, de la même manière, envoie le message « Aurevoir !! » à son fils, puis attend la terminaison du fils, grâce à la primitive *wait(&status)*.

B-INCRÉMENTATION

Dans ce programme et contrairement au précédent, la communication se fait entre processus fils, le processus père, alors, ne fait que génèrer ces derniers.

On commence toujours par installer les tubes entres les processus de la même manière que l'exercice précédent.

Puisqu'il sagit de la création de cinq fils, la meilleur façon est de mettre le traitement dans une boucle. Puis on appel la primitive *fork()*

Le traitement ne doit être effectuer que lorsqu'il sagit d'un processus fils

```
if(pID == 0) {
      printf("processus: %d | pid: %d\n", i, getpid());
```

On accorde un traitement particulier au premier fils, puisque c'est lui qui doit initialiser le jeton et le placer dans le premier tube, qui est le sien.
Une variable *isInit* est initialisée à 0 lors du lancement du programme, grâce à cette variable on teste si le jeton a été initialisé ou pas.

S'il n'est pas initialisé, on l'initialise à 0, puis on change la valeur de la variable *isInit,* puis on le place dans le tube du processus courant.

```
      if(i == 0) {
            if(isInit == 0) {
                  JETON = 0;
                  isInit = 1;
                  close(pC[i][READ]);
                  write(pC[i][WRITE], &JETON, sizeof(JETON));
                  printf("jeton initialisé [OK]\n");
      }
```

Le processus fils alors entre dans une boucle ou il récupère le jeton depuis le fils qui le précède, l'incrémente et l'enregistre dans son propre tube.

Si jamais le jeton atteint 50, on change sa valeur à -1, et on l'enregistre tout de même dans le tube courant, cela déclenchera la fin des autres processus

```
      if(JETON > 50) {
            JETON = -1;
            printf("le jeton désormais est à -1\n");
      }
```

Enfin, On affiche la valeur qu'a atteint le jeton si cette dernière est différente de -1 .

L'algorithme qu'executeront les autres processus est similaire, tanque la valeur du jeton est différente de -1, le processus courant lit la valeur du jeton depuis le tube du processus le précède, et l'incrémente et l'écrit dans son propre tube.

Si la valeur du jeton est -1, le processus courant écrit comme même le jeton dans son tube et se termine .

C-PHRASES ALÉATOIRES

Le défi le plus important dans ce programme pour nous était de génèrer un nombre aléatoire.
En C comme en C++, cela se fait grâce aux primitives *srand(unsigned int seed)* qui permet d'initialiser le générateur de nombres pseudo-aléatoires avec une graine différente (1 par défaut), cette primitive n'est appellé qu'une seule fois avant l'appel de *rand()*.
En d'autres termes, la génèration d'un nombre aléatoire ne C doit se fait en fonction d'une référence dynamique afin que le nombre qui est génèrer dans un moment *t0* ne soit pas celui génèré au moment

t1. On voit alors automatique comment la notion de temps s'est introduite dans l'algorithme, c'est en récupèrant alors le temps du système que l'ont va génèrer nos nombre aléatoires. Cela ressemblera à ça :

```
...
#include<time.h>
...
...
srand(time(NULL)) ;
...
```

Sans oublier d'inclure l'entête *time.h*.

Le plus grand problème c'est qu'ici il sagit de la création de plusieurs processus, qui, chacun d'eux doit génèrer son propre nombre aléatoire. Chose qui se fait simultanément, on voit alors que le temps dans ce cas ne nous sert plus à faire grand chose, car tous les processus génèreront le même nombre aléatoire. Il faut alors introduire un paramètre qui varie d'un processus à un autre qui créera donc la différence du temps de création.
Pour répondre à ce problème nous avons choisi le pid du processus comme paramètre supplémentaire, il suffit donc d'ajouter le pid au nombre génèrer par la primitive time(time_t) et on obtiendra donc un nombre aléatoire propre au processus qui le génère.

Je vous laisse avec le bout de code :

```
srand(((int)time(NULL)) + getpid());
if(i > 0) {
      close(p[i-1][WRITE]);
      read(p[i-1][READ], str, sizeof(str));
}
```

Pour le reste de l'algorithme il sagit seulement de récupèrer la chaine de caractères en fonction du nombre génèrer depuis les tableaux que l'on a déclaré.

```
char str[255], subjects[3][20], verbs[3][20], cod[3][40];
```

Et la mettre dans le tube du processus courant afin de permettre au processus suivant de refaire le même traitement.

LES SOCKETS

A-ÉCHANGE CLIENT/SERVEUR

Ce programme implémente l'échange de chaines de caractères entre un serveur et son client.

Regardons le serveur de plus près.

Dans un premier temps on définit le contexte d'adressage du serveur,

```
SOCKADDR_IN sin;
SOCKET sock;
socklen_t recsize = sizeof(sin);
```

On a définit `SOCKADDR_IN` pour remplacer struct sockaddr_in, de même pour `SOCKET` qui remplace struct socket.
On peut voir la variable sin comme étant le serveur, c'est dans cette structure d'ailleurs que nous allons enregistrer la configuration du serveur (adresse IP, port)

```
SOCKADDR_IN csin;
SOCKET csock;
socklen_t crecsize = sizeof(csin);
```

Le deuxième contexte est reservé au client, c'est dans cette structure que nous allons enregistrer les paramètres du client qui demande la connexion au serveur.

```
sock = socket(AF_INET, SOCK_STREAM, 0);
```

On crée donc notre socket grâce à la primitive *socket()*.

```
[...]
sin.sin_addr.s_addr = htonl(INADDR_ANY);  /* Adresse IP automatique */
sin.sin_family = AF_INET;                 /* Protocole familial (IP) */
sin.sin_port = htons(PORT);               /* Listage du port */

if(bind(sock, (SOCKADDR*) &sin, recsize) < 0) error("ERREUR: bind()");
[...]
```

Puis grâce à la primite *bind()* on lie la configuration donnée plus haut à la socket que nous avons crée.
Il ne manque plus qu'écouter si un client essaye de se connecter.

```
sock_err = listen(sock, 5);
printf("en attente...\n");
```

Grâce à la primitive *listen()* on met notre serveur en écoute d'éventuelle connexion demandée par un client. Le premier paramètre de *listen()* est la socket serveur elle même, le deuxième par contre, est un entier qui indique la longueur de la file d'attente.

Lorsqu'un client demande une connexion, le serveur alors appelle la primitive
`csock = accept(sock, (SOCKADDR*)&csin, &crecsize);`
Ce qui nous mène à l'envoie et la reception des données avec le serveur.
L'envoie des données se fait par la primitive *send()* ou *write(),* elles ont le même fonctionnement.
Comme il est d'ailleurs le cas de *recv()* avec *read()*.

Au niveau du client pratiquement rien ne change, on crée aussi une socket, à la quelle on associe l'adresse et le port spécifiés en paramtres. La seul différence demeure dans la demande d'une connexion au serveur qui elle se fait par la primitive *connect()*

```
[...]
connect(sock, (SOCKADDR*)&sin, sizeof(sin));
[...]
```

B-INCRÉMENTATION

Nous avons parlé dans l'exercice précédent du fonctionnement des sockets, dans cet exercice nous allons aborder le principe de communication de plusieurs client avec le père.
Puisqu'il sagit de la connexion de trois client, nous avons choisi alors de créer un tableau de socket client.

```
[...]
SOCKADDR_IN csin[3];
SOCKET csock[3];
socklen_t crecsize = sizeof(SOCKADDR_IN);
[...]
```

Voyons le coté serveur :

```
i = 0;
while(i < 3) {
      sock_err = listen(sock, 5);
      printf("en attente...\n");
      [...]
            csock[i] = accept(sock, (SOCKADDR*)&csin[i], &crecsize);
            printf("client #%d connecté\n", i+1);
      [...]
      i++;
}
```

Puisqu'il sagit de trois client, on commence donc par les connecter au serveur, notez que *csin[i]* prend maintenant un indice puisqu'il sagit d'un tableau de trois sockets à créer.

Voyons comment la communication entre une instance du client et le serveur est faite :

```
i = 0;
JETON = -1;  (1)
while(JETON != -2) {
      n = send(csock[i], &JETON, sizeof(JETON), 0) ;  (2)
      if (n < 0) error("ERREUR: send()");

      n = recv(csock[i], &JETON, sizeof(JETON), 0) ;  (3)
      if (n < 0) error("ERREUR: read()");

      if(JETON != -2) printf("client #%d> %d\n", i+1, JETON);
      i++;
      if(i == 3) i = 0 ;  (4)
}
```

(1) On initialise la valeur du jeton à -1
(2) on envoie le jeton au client concernée, selon l'indice de i
(3) le client alors incrémente la valeur du jeton, l'envoie puis on la reçoi grâce à recv()
(4) on incrémente i pour qu'il passe que le serveur envoie le jeton au client suivant, tout en veillant à ce que i ne dépasse jamais 3, qui est le nombre de clients que l'on doit connecter.

Notez, que lorsque la valeur du JETON est à -2, on sort de la boucle, pour terminer le processus de communication entre le serveur et ses clients.

Voyons maintenant le client.

```
while(JETON != -2) {
            n = recv(sock, &JETON, sizeof(JETON), 0); (1)
            if (n < 0) error("ERREUR: recv()");

            JETON++ ; (2)

            if(((JETON) % 3) == 0) {
                  if(JETON > 50) JETON = -2; (3)
            }

      if(JETON != -2) printf("server> JETON = %d\n", JETON);

      n = send(sock, &JETON, sizeof(JETON), 0) ;  (4)
      if (n < 0) error("ERREUR: send()");
}
```

Alors tanque le JETON n'est pas à -2,

(1) On lit la valeur du jeton envoyée par le serveur qui était initialement de -1
(2) On l'incrémente
(3) Puisque c'est le premier client qui s'occupe de la terminaison du processus de communication, on teste si on est bien dans le premier client. Si c'est le cas on test si le JETON a dépassé 50, et si c'est le cas il remet sa valeur à -2, ce qui, après envoie au serveur, provoquera comme nous l'avons vu en haut la terminaison du processus de communication.
(4) Finalement et quoi qu'elle soit la valeur du jeton, on l'envoie au serveur.

C-SOCKETS & TUBES

Ce programme fusionne ce qu'on avait vu dans les tubes et dans la partie précédente des sockets, en effet dans cette partie, le serveur ne subira aucun changement par rapport à la précédente, nous allons donc nous focaliser sur les changements que nous avons apporté au client.
Alors que le client s'occupait de l'incrémentation du jeton dans le programme précédent, nous allons délèguer cette tâche à un processus fils que le client va génèrer.

```
pID = fork();

On génère donc un processus

if(pID > 0) { (1)
      if(connect(sock, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR) { (2)
      do {
      n = recv(sock, &JETON, sizeof(JETON), 0);  (3)
      [...]
      if(JETON != -2) {
            close(pipeParent[READ]);  (4)
            write(pipeParent[WRITE], &JETON, sizeof(JETON));

            close(pipeChild[WRITE]);  (5)
            read(pipeChild[READ], &JETON, sizeof(JETON));

            if(((JETON) % 3) == 0) { (6)
                  if(JETON > 50) JETON = -2;
            }
```

```
            if(JETON != -2) printf("client> JETON = %d\n", JETON);
      }

      n = send(sock, &JETON, sizeof(JETON), 0);  (7)
      [...]
      } while(JETON != -2);
      exit(1);
}
```

(1) traitement que le père doit effectuer
(2) connexion au serveur
(3) reception du jeton depuis le serveur
(4) envoie du jeton au fils pour l'incrémenter
(5) reception du jeton incrémenté
(6) vérification si c'est le premier client pour gérer ou non la terminaison des processus
(7) envoie du jeton au serveur

```
if(pID == 0) (1) {
do {
      close(pipeParent[WRITE]);  (2)
      read(pipeParent[READ], &JETON, sizeof(JETON));
      if(JETON != -2) {
            JETON++; (3)
            close(pipeChild[READ]);  (4)
            write(pipeChild[WRITE], &JETON, sizeof(JETON));
      }
} while(JETON != -2); (5)
exit(1);
}
```

(1) traitement concernant le fils
(2) réception du jeton envoyé par le père
(3) incrémentation
(4) envoie du jeton au client/père
(5) tanque la valeur du jeton est différente de -2, le cas échéant, on gère la terminaison du fils