TP 2– fork(), execv() et serveur http

Sovannara Hak et Vincent Padois
hak@isir.upmc.fr

Concepts abordés

- Manipulation de l'appel système fork()
- Concept de serveur multi-utilisateur
- Lecture de fichiers

Fonctions utiles

- atoi ()
- execv(), execvp(), execlp()
- fork ()
- fopen(), fread (), fclose ()
- getpid()
- malloc(), free ()
- printf ()
- stat ()
- strcat ()
- system()

N'hésitez pas à utiliser la commande man pour obtenir la documentation de chaque fonction.

Contraintes et remise des TPs

- Le code de chaque exercice doit être dans un répertoire séparé (exercice1, exercice2, etc.).
- Chaque répertoire d'exercice doit contenir un fichier Makefile permettant de compiler les réponses à chaque question.
- Lorsque c'est nécessaire, ajoutez dans le repertoire de l'exercice des fichiers texte ou image pour répondre aux questions (avec un nom explicite).
- N'oubliez pas de rendre une archive « propre » (pas de .o, pas binaires, pas de fichier temporaires *.swp ou *~). La cible « clean » de vos makefiles est faite pour cela !
- Indentez vos fichiers (commande `indent` ou éditeur de texte civilisé).
- N'oubliez pas les "gardes" (#ifndef ...) dans vos fichier `.h`.
- La correction tiendra compte de la briéveté des fonctions que vous écrivez (évitez les fonctions de plus de 25 lignes) ; n'hésitez pas à découper une fonction en plusieurs sous-fonctions plus courtes.
- Vos enseigants vérifieront avec Valgrind l'absence d'erreurs (fuites mémoires, lectures invalides, ...).
- La convention de nommage des fonctions est la suivante : une fonction doit avoir un nom explicite et si son nom se compose de plusieurs mots, ces derniers sont écrits en

minuscule et séparés d'un tiret bas _. Ainsi une fonction qui affiche un tableau sera appelée `affiche_tableau`.
- Le code source doit être envoyé forme d'une seule archive tar.gz [1]
- Votre archive doit être propre : pas d'exécutables, pas de .o, pas de fichiers temporaires. **au maximum 30 minutes après la fin du TP**
- Envoyez votre archive par e-mail à `hak@isir.upmc.fr`
- Les fichiers utiles et les transparents de cours sont sur `http://pages.isir.upmc.fr/~hak`

Exercice 1 – préliminaires (maximum : 2 heures)

1. Écrivez un programme générant trois processus fils en utilisant des appels à fork() dans une boucle **for** et qui affiche le PID du processus en cours dans la partie « fils » du code. Exemple (vos obtiendrez bien sûr d'autres PIDs ; le $ représente l'invite de commande du shell) :

```
$ ./question1
PID : 4242
PID : 4243
PID : 4244
$
```

2. En utilisant la fonction system(), écrivez un programme permettant d'ouvrir avec le programme emacs un fichier dont le nom a été passé en argument d'entrée de l'exécutable et qui rend ensuite la main à l'utilisateur. Exemple :

```
$ ./question2 question2.c
(le fichier question2.c s'ouvre dans emacs)
$
```

3. Écrivez un programme C équivalent à la commande shell who & ps & ls -l *sans utiliser* system() (les commandes séparées par « & » s'exécutent en parallèle).
   Exemple (la sortie que vous obtiendrez sera différente) :

```
$ ./question3
mandor   tty7         2008-10-05 13:23 (:0)
  PID TTY          TIME CMD
total 4
 9612 pts/4    00:00:00 zsh
drwxr-xr-x 2 mandor mandor 4096 2008-10-03 10:43 sujet
 9667 pts/4    00:00:00 evince
 9699 pts/4    00:00:02 emacs
[1] - Done                    who
$  9852 pts/4    00:00:00 sh
 9870 pts/4    00:00:00 ps
```

1. Pour faire une archive tar.gz : `tar zcvf mon_archive.tar.gz mon_repertoire`

4. Écrivez un programme C équivalent à la commande shell `who; ps; ls -l` *sans utiliser* `system()` (les commandes séparées par « ; » s'exécutent successivement). Exemple :

```
$ ./question4
8-10-05 13:23 (:0)
  PID TTY          TIME CMD
 9612 pts/4    00:00:00 zsh
 9667 pts/4    00:00:00 evince
 9699 pts/4    00:00:03 emacs
 9852 pts/4    00:00:00 sh
 9894 pts/4    00:00:00 ps
total 4
drwxr-xr-x 2 mandor mandor 4096 2008-10-03 10:43 sujet
$
```

5. Écrivez un programme C qui prend un nom de fichier en argument et qui essaie de le compiler avec `gcc -c` . Si la compilation échoue, le programme doit écrire sur la sortie standard « erreur », sinon il doit « compilation OK ». Vous ne devez pas utiliser la fonction `system` et nous vous suggérons de lire attentivement la documentation de `waitpid` et de `exit` . Exemple :

```
$./question5 test4.c
gcc: test4.c: No such file or directory
gcc: no input files
erreur
$./question5 question5.c
compilation OK
```

Exercice 2 – serveur HTTP

Un serveur HTTP (par exemple Apache, IIS ou Tomcat) est l'application qui répond à votre navigateur lorsque celui-ci demande un fichier (une page web, une image, etc.) à un serveur, identifié par son adresse IP (par exemple le serveur `www.polytech.upmc.fr` a pour adresse `134.157.105.43`). Un même serveur doit pouvoir répondre à plusieurs navigateurs en même temps (figure 1). Si ce n'était pas le cas, vous devriez attendre votre tour pour accéder à chaque page web, image, etc. La méthode classiquement utilisée sous Unix pour programmer de tels serveurs multi-utilisateurs repose sur l'appel système `fork()` .

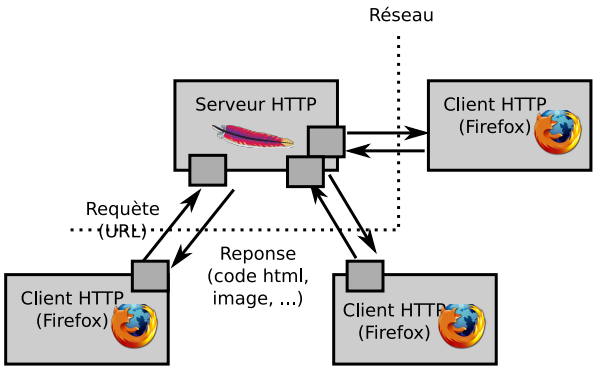

FIGURE 1 – Principe d'un serveur HTTP (serveur web). Plusieurs clients peuvent se connecter en même temps en demandant certaines pages web. Le serveur doit leur répondre en leur envoyant le contenu du fichier demandé.

Le but de ce TP est de vous guider dans l'implémentation d'un serveur HTTP simple multi-utilisateur en utilisant `fork()`, les concepts étant similaires pour la plupart des serveurs. Nous vous fournissons le code pour créer des connexions réseau et des paquets HTTP, votre travail se situe essentiellement dans la programmation de la fonction `main()` du serveur.

Principe d'un serveur multi-utilisateur

La majorité des serveurs multi-utilisateurs suivent la séquence suivante (les noms de fonction entre parenthèses correspondent au code source fourni ; reportez vous à l'annexe) :

1. créer une connexion serveur (`new_server_socket`)
2. passer en mode d'écoute (`server_accept`)
3. quand un nouveau client se connecte, utiliser l'appel système `fork()` pour créer un fils
4. dans le fils :
   - lire la requête du client (`read_request`)
   - envoyer le contenu demandé (par exemple une chaîne de caractères ou le contenu d'un fichier) ou un message d'erreur (`send_http_buffer`)
5. dans le père :
   - retourner en mode d'écoute

1. **Serveur mono-utilisateur et mono-fichier.** Le but de cette étape est de créer un serveur capable d'envoyer la chaîne de caractères `"<html><body>Hello World!</body></html>"` à une instance de Firefox. Comme un seul utilisateur peut se connecter à la fois, il n'est pas nécessaire d'utiliser `fork()`. Vous devez utiliser le code fourni (voir l'explication dans l'annexe A).

   Voici la séquence que vous devez implémenter :

   1. créer une connexion serveur (`new_server_socket`)
   2. passer en mode d'écoute (`server_accept`)
   3. lire la requête du client (`read_request`)

4. envoyer la chaîne de caractères `"<html><body>Hello World!</body></html>"` (send_http_buffer), de type HTML
5. fermer la connection du client (close sur le socket client).
6. retourner en mode d'écoute

Vous devez créer un Makefile contenant la règle all construisant votre application.

Votre application doit prendre en unique argument le numéro du port sur lequel votre serveur doit écouter ; la fonction atoi peut vous être utile. Vous devez afficher un message d'erreur si le port n'est pas spécifié.

Pour tester :

– lancez votre application dans un terminal sur un port *supérieur à 1024* (par exemple 4242 : ./question1 4242
– connectez vous avez Firefox sur `http://localhost:4242`
– vous devriez voir s'afficher "Hello World !"

2. **Serveur multi-utilisateur et mono-fichier.** Ajoutez la possibilité de connecter plusieurs clients en même temps en utilisant fork. Il s'agit donc d'implémenter la séquence :

   1. créer une connexion serveur (new_server_socket)
   2. passer en mode d'écoute (server_accept)
   3. quand un nouveau client se connecte, utiliser l'appel système fork () pour créer un fils
   4. dans le fils :
      – lire la requête du client (read_request)
      – envoyer la chaîne de caractères `"<html><body>Helloworld!</body></html>"` (send_http_buffer), de type HTML
   5. dans le père :
      – retourner en mode d'écoute

   Pour tester :

   – ajoutez un délai (par exemple 10 secondes) en utilisant la fonction sleep() avant d'envoyer la réponse ;
   – avant que la réponse n'arrive, essayez d'ouvrir une fenêtre dans un autre navigateur (par exemple konqueror) sur la même adresse ;
   – faites le même test avec le code de la question1 (mono-utilisateur).

3. **Serveur multi-utilisateurs et multi-fichiers.** Le but de cette dernière étape est d'envoyer le contenu du fichier demandé par le client. Le nom du fichier demandé est donné par read_request(). Vous devez alors utiliser les fonctions système suivantes :
   – stat () pour connaître la taille du fichier ;
   – malloc() pour allouer un espace mémoire de la bonne taille pour le contenu du fichier ;
   – fopen() pour ouvrir le fichier ;
   – fread () pour lire le fichier ;
   – fclose () pour fermer le fichier.

   Vous devez renvoyer une page d'erreur lorsque le fichier demandé est introuvable.

Pour tester

- créez un fichier « hello.html » contenant une page html simple ;
- connectez vous sur `http://localhost:4242/hello.html` ;
- pour tester le multi-utilisateur, vous pouvez essayer d'envoyer un gros fichier (par exemple en copiant /boot/vmlinuz).

4. **(bonus) : mode « démon ».** Modifier votre programme pour que le serveur rende la main à l'utilisateur dès qu'il est lancé mais continue à tourner en arrière plan. Vous devez donc faire en sorte que
./question4 −bg 4242 soit équivalent à ./question3 4242 & .

A Code fourni

Vous devez récupérer les fichiers network.c et network.h sur la page web du cours.
Le fichier network.h contient les prototypes des fonctions réseaux nécessaires :

- **int** new_server_socket(**int** port); crée une connexion serveur sur le port `port`. Attention, pour vos tests, vous devez utiliser un port supérieur à 1024 (Linux vous interdira de créer une connexion serveur sur un port inférieur à 1024). Un port est une sorte de portail à laquelle un client doit s'annoncer pour qu'on lui crée sa propre connection directe au serveur (le socket renvoyé par server_accept).
- **int** server_accept(**int** server_socket); fonction bloquante qui renvoie un identifiant pour une connexion client sur la connexion server_socket . Le processus reste bloqué tant que aucun client ne demande de connexion.
- read_request(**int** client_socket, **char**∗∗ filename); lit le nom du fichier demandé par un client et le place dans filename après l'avoir alloué avec malloc . Vous n'avez pas besoin d'allouer de mémoire pour filename .
- send_http_buffer(**int** client_socket, **const char**∗ content,
int content_size,**int** type); envoie la chaîne de caractère content , de taille content_size , sur la connexion client_socket (obtenue via un server_accept()). Le type peut être HTML ou BIN , suivant si c'est un document texte ou html ou si c'est un fichier binaire.