Examen de LIFASR5, le 21/05/2019 — 2ème partie (≈ 45 minutes)

Vous devez répondre lisiblement sur votre copie, indenter votre code, et éviter le plus possibles les ratures. Le correcteur ne passera pas de temps sur les réponses confuses et leur attribuera 0.

1ère partie des consignes Le but est d'écrire le code d'un serveur TCP/IP, qui n'utilise qu'un seul processus pour prendre en charge des clients arrivant successivement (si deux clients arrivent en même temps, le système en mettra un en attente : vous n'avez pas à vous préoccuper de cela).

Le serveur doit ouvrir une socket d'écoute sur le port 8000 de la machine, puis se mettre en attente d'une demande de connexion d'un client. Lorsqu'un client se connecte, le serveur sort de cette attente et échange avec le client *via* la socket de dialogue qu'il obtient. L'échange est composé de 3 étapes :

1. le serveur commence en envoyant (avec la chaîne `"HELLO\r\n"`) ;
2. le serveur se met en attente d'une requête de la part du client (c'est une chaîne de caractères terminée par `\r\n`) ;
   - si la requête est la chaîne `"DATE"`, alors le serveur répond en envoyant la date du jour (utilisez `my_date()`),
   - si la requête est la chaîne `"HOUR"`, alors le serveur répond en envoyant l'heure actuelle (utilisez `my_hour()`),
   - si la requête n'est ni `"DATE"` ni `"HOUR"`, alors le serveur ne fait rien.
3. le serveur conclut l'échange en envoyant la chaîne `"BYE\r\n"` au client traité, et ferme la socket de dialogue,

A la fin de l'échange, le serveur se remet en attente de la connexion suivante. Le serveur doit afficher `"arrivée d'un nouveau client"` quand le cas se présente. Les chaînes envoyées et reçues doivent se terminer par `"\r\n"`.

2ème partie des consignes Vous devez mettre en place un gestionnaire pour signal `SIGINT`, pour gérer proprement la fin du serveur. À la réception de ce signal, le serveur peut être :

1. bloqué en attente de connexion d'un nouveau client,
2. bloqué en attente de la requête du client en cours,
3. ailleurs dans la boucle de traitement d'un client.

Dans le 1er cas, le serveur sort de l'attente d'un client sans aucune connexion. Il doit alors sortir de la boucle d'attente, fermer la socket d'attente et se terminer. Dans le 2ème cas, il a bien un client mais pas de requête, il doit alors envoyer `"BYE\r\n"` avant de poursuivre comme dans le 1er cas. Dans le 3ème cas, il finit de traiter la requête en cours avant de se terminer.

Comme durant les TPs, vous êtes autorisés à utiliser une variable booléenne globale `stop` afin de passer des informations entre le gestionnaire de signal et le reste du programme.

Les questions.

1. (8 pts) Donnez le code de la fonction `main()` du serveur répondant aux consignes. **Il vous est conseillé d'utiliser un brouillon**. Vous pouvez décider de ne traiter que la 1ère partie des consignes (5 pts). Si vous traitez également la 2ème partie (3 pts), ne donnez votre code qu'une seule fois. **Si vous traitez la 2ème partie alors que vous avez déjà répondu à la 1ère, vous devrez recopier votre code afin que votre réponse soit lisible.**
2. (2 pts) Expliquez comment vous devriez modifier votre programme pour permettre au serveur de traiter plusieurs clients simultanément : on ne vous demande pas de code ; une réponse lisible, claire, précise et concise est attendue.

Les fonctions. En dehors des fonctions standards, vous disposez des fonctions suivantes.

```
// Tente de créer une socket côté serveur (localhost), à l'écoute sur toutes les interfaces, sur le port passé
// en paramètre. Retourne la socket crée en cas de succès, -1 en cas d'échec.
int create_server_socket(const char* port);

// Côté serveur, se met en attente bloquante d'une connexion sur la socket d'écoute s. Retourne la socket de
// dialogue crée en cas de succès, -1 en cas d'échec.
int accept_connection(int s);

// Reçoit une chaîne de caractères du type string sur la socket s, jusqu'à rencontrer le caractère c. Ce caractère
// est consommé sur la socket, et il n'est pas recopié dans la chaîne résultat. Retourne : -1 en cas d'échec, 0 si
// l'on n'a rien lu sur la socket, le nombre de caractères lus sinon.
int recv_line(int s, string &line, char c = '\n');

// Envoie une chaîne de caractères du type string sur la socket s. Retourne 0 en cas de succès, -1 en cas d'échec.
int send_str(int s, const std::string &str);

// Affiche un message d'erreur, puis termine le processus avec comme valeur de retour EXIT_FAILURE.
void exit_error(const std::string &msg = "");

// Retourne un objet de la classe string donnant la date.
string my_date(void);

// Retourne un objet de la classe string donnant l'heure.
string my_hour(void);

// Supprime les éventuels caractères de retour à la ligne ('\r', '\n' ou "\r\n") en fin de la string line.
void strip_line(std::string &line);
```

Antisèche :

NAME open -open a file
SYNOPSIS int open(const char *pathname, int flags);
DESCRIPTION
The open() system call opens the file specified by pathname. The return value of open() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (read(), write(), etc.) to refer to the open file. The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.
RETURN VALUE
open() returns the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

==========

NAME read -read from a file descriptor
SYNOPSIS ssize_t read(int fd, void *buf, size_t count);
DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
RETURN VALUE
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately.

==========

NAME write -write to a file descriptor
SYNOPSIS ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
RETURN VALUE
On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this number is smaller than the number of bytes requested; this may happen for example because the disk device was filled. On error, -1 is returned, and errno is set appropriately.

==========

NAME fork -create a child process
SYNOPSIS pid_t fork(void);
DESCRIPTION
fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
RETURN VALUE
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

==========

NAME waitpid -wait for process to change state
SYNOPSIS pid_t waitpid(pid_t pid, int *wstatus, int options);
DESCRIPTION
waitpid() is used to wait for state changes in a child of the calling process. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

If pid > 0, then the call will wait for the children whose PID equals pid.

If wstatus is not NULL, then waitpid() stores status in informations in the int it points to. If wstatus is NULL, then this parameter is ignored. The value of options is an OR of zero or more of the following constants : WNOHANG, WUNTRACED, WCONTINUED.
RETURN VALUE
On success, waitpid() returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

==========

NAME getpid, getppid -get process identification
SYNOPSIS pid_t getpid(void);
pid_t getppid(void);
DESCRIPTION
getpid() returns the process ID (PID) of the calling process. getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork(), or, if that process has already terminated, the ID of the process to which this process has been reparented.
ERRORS
These functions are always successful.

==========

NAME pipe -create pipe
SYNOPSIS int pipe(int pipefd[2]);
DESCRIPTION
pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see end-of- file and will return 0. If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write fails with the error EPIPE. An application that uses pipe and fork should use suitable close calls to close unnecessary duplicate file descriptors; this ensures that end-of- file and SIGPIPE/EPIPE are delivered when appropriate.
RETURN VALUE
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

==========

NAME sigaction -examine and change a signal action
SYNOPSIS int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
DESCRIPTION
The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP. If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact. The sigaction structure is defined as something like:

```
struct sigaction {
  void     (*sa_handler)(int);
  ... ;
};
```

sa_handler specifies the action to be associated with signum and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.
RETURN VALUE
sigaction() returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.