

TD1 : Processus et signaux

I. Le premier processus qui meurt

- 1) Écrire un programme C `primeurt.c` qui accepte deux entiers `a` et `b` en arguments puis se duplique. Le père et le fils affichent leur identité, s'endorment respectivement pendant `a` et `b` secondes, puis affichent un message de réveil et se terminent.
- 2) Modifier le programme de telle sorte que chacun des processus affiche s'il meurt en premier ou en dernier. On peut commencer par traiter le cas (facile) où $a \neq b$, puis celui où $a = b$.

II. Signaux utilisateurs

La fonction `signal` n'étant pas portable, on se donne une fonction `bor_signal` qui simplifie l'usage de `sigaction` pour placer un handler de signal :

```
#include <signal.h>

/* Place le handler de signal void h(int) pour le signal sig avec sigaction().
Le signal est automatiquement masqué pendant sa délivrance.
Si options = 0,
- les appels bloquants sont interrompus avec retour -1 et errno = EINTR.
- le handler est réarmé automatiquement après chaque délivrance de signal.
si options est une combinaison bit à bit (opérateur |) de
- SA_RESTART : les appels bloquants sont silencieusement repris.
- SA_RESETHAND : le handler n'est pas réarmé.
Renvoie le résultat de sigaction. Verbeux.
*/
int bor_signal (int sig, void (*h)(int), int options)
{
    int r; struct sigaction s;
    s.sa_handler = h; sigemptyset (&s.sa_mask); s.sa_flags = options;
    r = sigaction (sig, &s, NULL);
    if (r < 0) bor_perror (__func__); /* cf bor-util.c en TP */
    return r;
}
```

- 1) Écrire un programme C `sigaz.c` qui crée un fils. Le fils fait un compte à rebours en secondes de 10 à 1 et se termine. Chaque fois que le fils reçoit le signal `SIGUSR1`, le compte est augmenté de 5 secondes, et chaque fois qu'il reçoit `SIGUSR2` le compte est diminué de 2 secondes.
- 2) Modifier le père afin qu'il lise une suite de caractères entrés au clavier, et se termine à la frappe de `^D` (fin de fichier). Lorsque le caractère lu est `'a'` (respectivement `'z'`), le père envoie le signal `SIGUSR1` (resp. `SIGUSR2`) à son fils.
- 3) Que se passe-t-il si à l'exécution du père on tape `aaaaaaaaaa` suivi d'un retour chariot ?
- 4) Modifier le père pour qu'il se termine dès que son fils est mort.

Rappels

- ▷ `sleep(n)`; endort un processus pendant `n` secondes.
- ▷ La fonction `int atoi(char *s)`; de `stdlib.h` convertit le début de la chaîne pointée par `s` en entier de type `int`.
- ▷ Les signaux `SIGUSR1` et `SIGUSR2` sont des signaux utilisateur et n'ont pas de signification prédéterminée. Le signal `SIGCHLD` est reçu par le père lorsque son fils (en anglais *child*) est mort; par défaut il est ignoré.

TP1 : Processus et signaux

I. Comptes à rebours et comptage de fils

Écrire un programme C `rebourfils.c` qui dans un premier temps, lit en boucle un entier n au clavier jusqu'à ce que n soit 0; à chaque itération, le père crée n fils; chaque fils fait un compte à rebours en seconde de 10 à 1 puis se termine (on peut donc entrer des entiers au clavier pendant l'affichage des comptes à rebours).

Dans un deuxième temps (c'est-à-dire une fois que l'on a tapé 0 au clavier) le père attend que *tous* ses fils soient morts (en faisant un `wait` par fils créé), puis affiche un message.

II. Signaux et timeout

Pour faciliter cet exercice, récupérer sur <http://www.lif.univ-mrs.fr/~thiel/ens/rezo> les fichiers `bor-util.h` et `bor-util.c` de la "Boîte à Outil Réseaux" qui accompagne ce cours.

On va se servir ici de la fonction `bor_signal` qui simplifie l'usage de `sigaction` pour placer un handler de signal, et évite l'emploi de `signal` qui n'est pas portable.

Il suffit d'inclure "`bor-util.h`" (qui inclut lui-même les `.h` classiques, vous dispensant de les inclure) et de compiler votre `exercice.c` avec :

```
gcc -Wall exercice.c bor-util.c -o exercice
```

ou, si vous préférez compiler par morceaux :

```
gcc -Wall -c exercice.c
gcc -Wall -c bor-util.c
gcc exercice.o bor-util.o -o exercice
```

- 1) Écrire un programme C `triosig.c` qui crée 2 fils. Le fils 2 envoie le signal `SIGUSR1` au fils 1, qui le renvoie au père, qui le renvoie au fils 2, etc. Pourquoi doit-on envoyer les signaux dans cet ordre?
- 2) Modifier le programme de telle sorte que, dès qu'un de ces processus n'a pas reçu de signal `SIGUSR1` depuis 5 secondes (timeout), il affiche un message et se termine.

III. Makefile

Récupérer <http://www.lif.univ-mrs.fr/~thiel/ens/rezo/Makefile>, l'enregistrer dans votre répertoire `TP-reseau` (dans lequel seront placés tous les programmes que vous ferez dans cette UE), puis dans un éditeur de texte, décommenter les noms de vos programmes dans la macro-variable `EXECS` (modifier ces noms au besoin) et sauver. Taper `make all` pour compiler tout, `make clean` pour effacer les éventuels fichiers `.o`, `make distclean` pour effacer aussi les exécutable.

Rappels

- ▷ La fonction `alarm(unsigned int n);` de `unistd.h` provoque l'envoi d'un signal `SIGALRM` au processus en cours au bout de n secondes. Chaque nouvel appel à `alarm` annule et remplace le précédent.

TD2 : Tubes anonymes

I. Héritage des descripteurs du tube

Écrivez un programme C `heritub.c` qui crée un tube, puis écrit dans ce tube 10 caractères lus au clavier. Le programme lit ensuite 3 caractères dans le tube et les affiche. Enfin, le programme crée un fils qui lit les 7 caractères restants dans le tube et les affiche.

II. Détection de la fin d'un tube

1) Écrire un programme C `fintub.c` qui crée un tube puis se duplique. Le père lit des caractères au clavier et les écrit dans le tube; il se termine à la frappe de `^D` (fin de fichier). Le fils lit les caractères dans le tube, les met en majuscule puis les affiche; il se termine à la fin du tube.

2) Que se passe-t-il si l'on tue le père, ou si l'on tue le fils ?

III. Redirection d'un tube

Écrire un programme C `redirtub.c` qui crée un tube puis se duplique. Le père redirige la sortie standard sur le tube, le fils redirige l'entrée standard sur le tube, puis ils se recouvrent respectivement avec la commande `ls` et la commande `wc -l` (comptage de ligne) de manière à réaliser `ls | wc -l`.

Rappels

▷ Redirection : la fonction `int dup(int fd)` de `unistd.h` crée une copie du descripteur `fd` (c'est-à-dire de la case d'indice `fd` dans la table des descripteurs du processus appelant), et renvoie l'indice de la copie ou `-1` en cas d'erreur. La copie est mémorisée dans la case de plus petit indice disponible.

Pour rediriger l'entrée standard `0` sur l'entrée `p[0]` par un tube, il suffit donc de faire `close(0)` qui libère le descripteur `0`, puis de faire `dup(p[0])` qui duplique le descripteur `p[0]` dans la place du plus petit descripteur disponible (qui est `0`), puis enfin `close(p[0])`. Cela a pour effet que le descripteur `0` pointe maintenant sur l'entrée par le tube au lieu de pointer sur le clavier.

TP2 : Tubes anonymes

I. Redirections de plusieurs tubes

Écrire un programme C `redir2tub.c` qui crée deux tubes puis deux fils. Le fils1 redirige la sortie standard sur le tube 1, le fils 2 redirige l'entrée standard sur le tube 1 et la sortie standard sur le tube 2, le père redirige l'entrée standard sur le tube 2. Ensuite, les trois processus se recouvrent respectivement avec la commande `ls -t`, la commande `sort` et la commande `head -3` de manière à réaliser `ls -t | sort | head -3`.

Attention, il faut veiller à fermer *tous* les descripteurs inutiles avant de faire les recouvrements, sinon les commandes ne détecteront pas les fins de fichiers. Taper `ps` pour vérifier que les 3 processus sont terminés.

II. Taux de transmission dans un tube

Écrire un programme C `debitub.c` qui reçoit en argument une taille de buffer `bufsize`. Le programme alloue un tableau de caractères `s` d'au moins `bufsize` éléments. Le programme crée ensuite un tube puis un fils.

Le père entre dans une boucle infinie dans laquelle, à chaque itération, il écrit en une seule opération les `bufsize` premiers caractères de `s` dans le tube. Le fils procède de même en lecture dans le tube. Tester la détection de fin de tube en tuant le père ou le fils avec `kill` dans un deuxième terminal (penser à capter `SIGPIPE`).

Chaque seconde, le fils affiche le nombre total de caractères lus (en kilo-octets) pendant la seconde écoulée (utiliser `alarm()`). Faire des essais avec `bufsize` valant 1, 10, 100, 1000, 10000, 20000, 100000. Que conclure ?

Rappels

▷ Redirections : `man dup`

TD3 : Scrutation

I. Scrutation de l'entrée standard et d'un tube

Écrire un programme C `scrutatub.c` qui crée un tube puis un fils. Le fils écrit chaque seconde un caractère dans le tube, et meurt au bout de 30 secondes. Le père scrute en lecture l'entrée standard et le tube; chaque fois qu'un descripteur est prêt en lecture, le père fait une lecture puis affiche sur la sortie standard ce qu'il a lu ainsi que la provenance. Le père se termine dès qu'il détecte une fin de fichier.

II. Scrutation de deux tubes avec un timeout

Écrire un programme C `scrutimeout.c` qui crée deux tubes puis deux fils. Le fils 1 écrit toutes les 4 secondes un caractère dans le tube 1, et meurt au bout de 5 fois. Le fils 2 écrit toutes les 6 secondes un caractère dans le tube 2, et meurt au bout de 5 fois. Le père scrute en lecture les tubes *ouverts*; chaque fois qu'un descripteur est prêt en lecture, le père fait une lecture puis affiche sur la sortie standard ce qu'il a lu ainsi que la provenance. Chaque fois que le père n'a rien lu pendant 3 secondes, il affiche un message, et vérifie s'il y a encore au moins un descripteur ouvert, sinon il se termine.

III. Signal et tube de réveil

Écrire un programme C `scrutsig.c` qui crée un tube, installe un handler pour `SIGUSR1`, puis scrute en lecture l'entrée standard et le tube. Le handler de `SIGUSR1` écrit à chaque appel 1 caractère dans le tube. Le programme ne fait rien si l'appel à `select` revient pour cause de réception de signal (voir remarque). Chaque fois qu'un descripteur est prêt en lecture, le programme fait une lecture puis affiche sur la sortie standard ce qu'il a lu ainsi que la provenance; si le descripteur concerné est le tube, il ne lit qu'un seul caractère et affiche le message "un signal a été reçu".

Remarque : on appelle un tel tube un *wake-up pipe* (tube de réveil); il permet de détecter de façon portable les signaux au niveau du `select`, tout en évitant le cas où un signal serait délivré entre deux appels à `select`, et donc non détecté par celui-ci (une autre solution est d'utiliser `pselect`).

Rappels

▷ La fonction `select` attend que des descripteurs soient éligibles, où la survenue d'un signal, ou la fin d'un timeout, puis renvoie le nombre > 0 de descripteurs éligibles, ou 0 si le timeout est fini, sinon -1 ; dans ce dernier cas, la variable `errno` indique si un signal est arrivé (valeur `EINTR`), si un descripteur est fermé (`EBADF`, retour immédiat) ou s'il s'agit d'une autre erreur.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n,                // le max des fd dans les listes, +1
           fd_set *readfds,     // liste des fd à scruter en lecture
           fd_set *writefds,    // liste des fd à scruter en écriture
           fd_set *exceptfds,   // en général NULL
           struct timeval *timeout); // .tv_sec=0 .tv_usec=0 : retour immédiat
                                     // si NULL : pas de timeout

FD_ZERO(fd_set *set);          // vide la liste
FD_SET(int fd, fd_set *set);   // ajoute fd à la liste
FD_CLR(int fd, fd_set *set);   // enlève fd de la liste
FD_ISSET(int fd, fd_set *set); // vrai si fd est dans la liste
```

TP3 : Scrutation

I. Taux de transmission avec plusieurs tubes

Écrire un programme C `debitntub.c` qui reçoit en argument une taille de buffer `bufsize` et un nombre de tubes `nbtubes`. Le programme alloue un tableau de caractères `s` d'au moins `bufsize` éléments. Le programme crée ensuite `nbtubes` tubes puis un fils.

Le père scrute en écriture les tubes ; chaque fois qu'un descripteur est prêt, il écrit en une seule opération les `bufsize` premiers caractères de `s` dans le tube correspondant.

Le fils procède de même en lecture. De plus, il affiche chaque seconde le nombre total de caractères lus (en kilo-octets) pendant la seconde écoulée (utiliser `alarm`).

Faire des essais avec différents taille de buffer et nombre de tubes.

II. Traducteur avec deux tubes

Écrire un programme C `tradu2tub.c` qui crée deux tubes puis un fils.

Le père scrute en lecture les descripteurs *ouverts* parmi l'entrée standard et le tube 2, et s'arrête lorsqu'ils sont *tous* fermés. Chaque fois qu'un descripteur est prêt, il lit en une seule opération les `bufsize` premiers caractères de `s` dans le descripteur correspondant. Dans le cas où il lit dans l'entrée standard, il recopie les caractères vers le tube 1 ; dans le cas où il lit dans le tube 2, il recopie les caractères vers la sortie standard.

Le fils joue le rôle de traducteur en se servant des deux tubes pour communiquer : il redirige l'entrée standard sur le tube 1, la sortie standard sur le tube 2, puis se recouvre avec la commande `cat` (recopie de caractères). Lorsque le programme est bien au point, remplacer cette commande avec `tr a-z A-Z` (conversion des minuscules en majuscules).

Une différence entre ces deux commandes (sur la plupart des systèmes) est que `cat` procède caractère par caractère, tandis que `tr` attend d'avoir lu plusieurs milliers de caractères avant d'écrire le résultat ; il faut donc taper beaucoup de caractères (ou copier-coller dans le terminal) pour voir le résultat, ou taper `^D` dans le terminal pour fermer l'entrée standard du père, ce qui provoque en cascade la fin de la traduction, l'affichage par le père et sa terminaison. Bien tester ces cas de figure.

Rappels

▷ Scrutation : `man select`, `man select_tut`

TD4 : Tubes nommés

I. M. et Mme ont un fils

On se propose d'écrire un serveur de « M. et Mme » et son client, qui communiquent par l'intermédiaire de tubes nommés. Le client lit sur l'entrée standard un nom de famille et l'envoie au serveur. Le serveur cherche ce nom de famille dans le fichier `mEtMme.txt` qui a cette forme :

```
ABA ont un fils : Bart
BALMASKE ont un fils : Alonzo
ENFAILLITE ont une fille : Melusine
NAREF ont deux fils : Michel Paul
...
```

Si le serveur trouve le nom, il renvoie le prénom du fils, de la fille ou des enfants, sinon il renvoie le message « Non trouvé ». Le client affiche le message reçu du serveur, puis recommence à lire un nom de famille sur l'entrée standard.

1) Écrire le client `mEtMme-cli.c` .

Le client reçoit en argument le nom du tube nommé du serveur, que l'on appelle le *tube d'écoute*. Le client crée deux tubes nommés (dont les noms sont fonctions de son PID), que l'on appelle *tubes de service*, envoie leur nom au serveur par son tube d'écoute, puis ferme celui-ci. Il ouvre ensuite l'un des tubes en écriture pour envoyer des noms de famille, et l'autre en lecture pour recevoir le résultat de chaque demande. Le client se termine à la fermeture de l'entrée standard.

2) Écrire le *squelette* du serveur `mEtMme-ser.c` .

Le serveur reçoit en argument le nom de son tube d'écoute et du fichier de noms de famille. Après création et ouverture en lecture, le serveur attend une « connexion » par un client puis se duplique. Le père retourne en attente d'une nouvelle connexion, tandis que le fils se charge du dialogue avec ce client : il décode le message contenant le nom des tubes de service, extrait les noms de famille, effectue la recherche puis répond chaque fois au client. Le fils se termine à la déconnexion du client.

TP4 : Tubes nommés

I. M. et Mme ont un fils (suite)

1) Tester le client `mEtMme-cli.c` .

Pour la mise au point, on simule le serveur. Pour cela il faut ouvrir 4 terminaux. Dans le 1^{er}, taper `mkfifo tub-ec.tmp` puis `cat < tub-ec.tmp`; dans le 2^e, taper `./mEtMme-cli tub-ec.tmp` : on obtient dans le 1^{er} terminal les noms des deux tubes de service, par exemple `tub-sc-x.tmp` et `tub-cs-x.tmp`. Dans le 3^e, taper `cat < tub-cs-x.tmp` et dans le 4^e, taper `cat > tub-sc-x.tmp`. Taper maintenant dans le 2^e un nom de famille, par exemple `ABA`; il doit s'afficher dans le 3^e. Répondre alors `Bart` dans le 4^e; il doit s'afficher dans le 2^e.

2) Écrire le serveur `mEtMme-ser.c`, puis tester avec 1 ou plusieurs clients.

Rappels

▷ Par défaut, l'ouverture d'un tube nommé est bloquante, jusqu'à ce que l'autre extrémité soit ouverte.

TD5 : Sockets UDP/UN

I. Serveur de date UDP/UN

On se propose d'écrire un client-serveur en mode datagramme avec le protocole UDP.

- 1) Écrire le serveur `date-ser.c` qui crée une socket, l'attache à une adresse donnée en argument, puis entre dans une boucle sans fin, dans laquelle il attend un message d'un client et lui répond en lui envoyant la date courante.
- 2) Écrire le client `date-cli.c` qui crée une socket, l'attache à une adresse donnée en argument 1, puis construit l'adresse du serveur à partir de l'argument 2 et lui envoie un message (par exemple "Hello"). Le client attend la réponse du serveur, l'affiche puis se termine.

TP5 : Sockets UDP/UN

I. Compteur de messages UDP/UN

- 1) Tapez le serveur de date du TD5-I, puis : `gcc -Wall date-ser.c bor-util.c -o date-ser` pour compiler. Faites de même avec le client et testez (avec un client ou plusieurs, terminaisons, etc).
- 2) Modifiez le client en un fichier `nhello-cli.c` de telle sorte qu'il envoie à la suite 10 000 messages "HELLO" au serveur, puis le message "NUMBER". Le serveur `nhello-ser.c` compte le nombre de "HELLO" reçus, et l'envoie au client lorsqu'il reçoit le message "NUMBER".
- 3) Modifiez le serveur en `nhello-ser2.c` pour qu'il puisse compter séparément les "HELLO" selon leur expéditeur (par un test sur l'adresse du client).

Fonctions utilitaires et rappels

▷ On introduit les fonctions suivantes de `bor-util.c` :

```
void bor_perror (const char *funcname) /* Conserve errno */
{
    int e = errno; perror (funcname); errno = e;
}
int bor_bind_un (int soc, struct sockaddr_un *adr)
{
    int r = bind (soc, (struct sockaddr *) adr, sizeof(struct sockaddr_un));
    if (r < 0) bor_perror (__func__);
    return r;
}
int bor_sendto_un (int soc, void *buf, size_t len, struct sockaddr_un *adr)
{
    int r = sendto (soc, buf, len, 0, (struct sockaddr *) adr,
        sizeof(struct sockaddr_un));
    if (r < 0) bor_perror (__func__);
    return r;
}
int bor_recvfrom_un (int soc, void *buf, size_t len, struct sockaddr_un *adr)
{
    socklen_t adrlen = sizeof(struct sockaddr_un);
    int r = recvfrom (soc, buf, len, 0, (struct sockaddr *) adr, &adrlen);
    if (r < 0) bor_perror (__func__);
    return r;
}
```

▷ La fonction `time(time_t *t)` de `<time.h>` mémorise dans `*t` la date courante depuis l'*Epoch* (le 1/1/1970 à 00:00:00), mesurée en secondes. La fonction `char *ctime(const time_t *t)` convertit `*t` en une chaîne de caractères en zone statique et renvoie son adresse.

TD6 : Sockets TCP/UN

I. Écho de nombres pairs TCP/UN

On se propose d'écrire un client-serveur en mode connecté avec le protocole TCP.

1) Écrire le client `pair-cli.c` qui crée une socket, l'attache à une adresse donnée en argument 1, puis construit l'adresse du serveur à partir de l'argument 2 et s'y connecte. Le client entre dans une boucle sans fin dans laquelle il lit une suite de digits au clavier, l'envoie au serveur, récupère sa réponse et l'affiche ; il se termine à la frappe de `^D` ou à la déconnexion du serveur.

2) Écrire le serveur `pair-ser.c` qui crée une socket d'écoute, l'attache à une adresse donnée en argument, puis entre dans une boucle sans fin, dans laquelle il attend une connexion sur la socket d'écoute ; à chaque connexion, il crée un fils qui dialogue avec le client sur la socket de service, en lui renvoyant les digits reçus qui sont pairs. Le père se termine à la frappe de `^C`, les fils se terminent à la déconnexion de leur client.

TP6 : Sockets TCP/UN

I. Écho de nombres pairs TCP/UN avec scrutation

1) Reprenez le client et le serveur de nombres pairs du TD6-I et testez (avec un client ou plusieurs, terminaisons, message sans nombres pairs). Vous constaterez que certaines réponses du serveur sont reçues en plusieurs morceaux (et donc en plusieurs fois) au niveau du client.

2) Modifiez le client en un fichier `scrutpair-cli.c` afin qu'il scrute en lecture l'entrée standard et la socket. Vous constaterez que tous les morceaux d'une réponse du serveur peuvent être lus à la suite, et que la détection de la déconnexion du serveur est immédiate.

3) Modifiez le serveur en un fichier `scrutpair-ser.c` de telle sorte que le même processus scrute en lecture la socket d'écoute et les sockets de service, et donc peut dialoguer avec tous les clients sans créer de fils. Il faut pour cela déclarer un tableau de sockets de service, insérer la nouvelle socket lors d'une acceptation de connexion à un emplacement libre du tableau, ne scruter que les sockets nécessaires, et mettre à jour le tableau lors des déconnexions.

Fonctions utilitaires

▷ On introduit les fonctions suivantes de `bor-util.c` :

```
int bor_connect_un (int soc, struct sockaddr_un *adr)
{
    int r = connect (soc, (struct sockaddr *) adr, sizeof(struct sockaddr_un));
    if (r < 0) bor_perror (__func__);
    return r;
}

int bor_accept_un (int soc, struct sockaddr_un *adr)
{
    socklen_t adrlen = sizeof(struct sockaddr_un);
    int r = accept (soc, (struct sockaddr *) adr, &adrlen);
    if (r < 0) bor_perror (__func__);
    return r;
}
```

TD7 : Clients TCP/IP

I. Client du service daytime

Le service `daytime` (port 13) donne la date courante sur une machine distante ; en général il est assuré par le démon serveur `inetd`, qui accepte la connexion d'un client, lui envoie un message contenant la date courante, puis le déconnecte.

Écrire le client `daytime.c` qui crée une socket, l'attache à l'adresse locale et au port 0, puis construit l'adresse du serveur à partir du nom donné en argument et du port 13. Le client se connecte et lit les caractères envoyés par le serveur, les affiche au fur et à mesure puis se termine à la déconnexion.

II. Défragmentation de lignes

Le protocole TCP/IP ne préserve pas les limites des messages, si bien qu'en lecture, les messages peuvent être agglutinés ou fragmentés. Il est donc nécessaire dans un protocole d'échange de messages de *défragmenter les messages* après l'opération de lecture. On se propose de réaliser cette opération dans un protocole orienté *lignes de texte* (i.e chaque message est terminé par un `'\n'`).

Écrire le client `defrag.c` qui crée une socket, l'attache à l'adresse locale et au port 0, puis construit l'adresse du serveur à partir du nom donné en argument 1 et du port en argument 2. Le client se connecte, lit les caractères envoyés par le serveur, affiche les messages du serveur *ligne à ligne*, puis se termine à la déconnexion.

Pour afficher les messages du serveur ligne à ligne, le client gère un buffer contenant la dernière ligne incomplète. Chaque fois que le client lit un groupe de caractères, il les concatène dans le buffer et recherche les lignes terminées par un retour chariot `'\n'`. Chaque ligne détectée est affichée puis retirée du buffer. Lors d'un dépassement de capacité, d'une déconnexion ou d'une erreur de lecture, le buffer est vidé.

Fonctions utilitaires

▷ On introduit les fonctions suivantes de `bor-util.c` :

```
int bor_bind_in (int soc, struct sockaddr_in *adr)
{
    int r = bind (soc, (struct sockaddr *) adr, sizeof(struct sockaddr_in));
    if (r < 0) bor_perror (__func__);
    return r;
}

int bor_connect_in (int soc, struct sockaddr_in *adr)
{
    int r = connect (soc, (struct sockaddr *) adr, sizeof(struct sockaddr_in));
    if (r < 0) bor_perror (__func__);
    return r;
}

int bor_getsockname_in (int soc, struct sockaddr_in *adr)
{
    socklen_t adrlen = sizeof(struct sockaddr_in);
    int r = getsockname (soc, (struct sockaddr *) adr, &adrlen);
    if (r < 0) bor_perror (__func__);
    return r;
}
```

TP7 : Clients TCP/IP

I. Défragmentation de lignes : tests

Reprenez le client `defrag.c` du TD7-II et testez-le d'abord sur le port 13 d'une machine, puis avec le client/serveur graphique TCP/IP suivant : dans une console, tapez `~thiel/helium/demo/tcp &`, cliquez sur [Start]; tapez ensuite dans la console `./defrag localhost x` en remplaçant `x` par le numéro de port du serveur, qui est affiché à gauche de [Start]. Vous devriez alors voir dans la fenêtre une ligne `Stat: (5) localhost:y` où `y` est le port du client. Tapez dans le champ de saisie `Send:` des messages, comprenant éventuellement des `^J` (le code d'un retour chariot). Tapez sur [Entrée] ou cliquez sur [Send] pour envoyer le message au client. Cliquez sur [Stop] pour le déconnecter. Testez aussi le dépassement de buffer.

II. Aspirateur Web

1) Reprenez le client `daytime.c` du TD7-I en un fichier `aspiweb.c` afin qu'il reçoive en argument 1 une adresse de machine et en argument 2 un numéro de port. Le client se connecte, envoie une requête (pour le moment une chaîne de caractères quelconque), puis lit les caractères envoyés par le serveur, les affiche au fur et à mesure, et enfin se termine à la fermeture de la socket.

2) Testez en contactant la machine `sol` sur le port 80 et envoyer la requête `"GET /~thiel/essai.txt HTTP/1.0\n\n"`.

Remplacez `~thiel/essai.txt` par le chemin de fichier donné en argument 3 (attention, bien conserver dans la requête le `/` qui est devant) et testez avec `'~thiel/essai2.txt'` .

Remarque : si la requête est erronée (absence de `GET` ou de `/` devant le chemin de fichier, absence du protocole `HTTP/1.0` ou mauvais numéro, ou encore absence du double retour chariot final), le serveur vous délivrera un message d'erreur en HTML, voire fermera la connexion sans vous répondre.

3) Modifiez le programme pour qu'il recopie la réponse du serveur dans un fichier texte dont le nom est donné en argument 4.

TD8 : Serveur TCP/IP

I. Serveur web

On se propose d'écrire un serveur web rudimentaire `serweb1.c`. On considère le type `Client` et la fonction `main` d'un serveur TCP/IP avec scrutation, qui gère dynamiquement les connexions et déconnexions avec les clients, et s'interrompt à réception de `SIGINT` :

```
#define MAXCLI 32
typedef struct {
    struct sockaddr_in adr; /* Adresse du client */
    int soc;                /* Socket de service, défaut -1 */
    char req[4096];        /* Buffer requête */
    int req_pos;           /* Position courante dans req */
} Client;

int main (int argc, char *argv[])
{
    int soc_ec = -1, maxfd, res, nc; Client tc[MAXCLI]; fd_set set_read;

    for (nc = 0; nc < MAXCLI; nc++) init_client (tc, nc);
    soc_ec = creer_socket_ecoute (0,8); if (soc_ec < 0) goto fin_serveur;
    bor_signal (SIGINT, capte_fin, SA_RESTART);

    while (boucle_princ) {
        init_select (soc_ec, tc, &maxfd, &set_read);
        res = select (maxfd+1, &set_read, NULL, NULL, NULL);
        if (res > 0) { /* Recherche des sockets éligibles */
            if (FD_ISSET (soc_ec, &set_read))
                if (nouvelle_connexion (soc_ec, tc) < 0) goto fin_serveur;
            for (nc = 0; nc < MAXCLI; nc++)
                if (tc[nc].soc != -1 && FD_ISSET (tc[nc].soc, &set_read))
                    traitement_requete (tc, nc);
        } else if (res < 0) {
            if (errno == EINTR) ; /* Signal reçu, on ne fait rien */
            else { perror ("select"); goto fin_serveur; }
        }
    }

    fin_serveur: printf ("Serveur: fermeture sockets ...\n");
    if (soc_ec != -1) close (soc_ec);
    for (nc = 0; nc < MAXCLI; nc++) raz_client (tc, nc);
    exit (0);
}
```

- 1) Écrire la fonction `void init_client (Client *tc, int nc)` qui initialise les champs du client numéro `nc`, et la fonction `void raz_client (Client *tc, int nc)` qui ferme la socket de service si elle est ouverte et réinitialise le client.
- 2) Écrire la fonction `void init_select (int soc_ec, Client *tc, int *maxfd, fd_set *set_read)` qui calcule le maximum `maxfd` des valeurs des sockets ouverts, et remplit l'ensemble `set_read` des sockets ouverts à scruter.
- 3) Écrire la fonction `int nouvelle_connexion (int soc_ec, Client *tc)` qui appelle la fonction `bor_accept_in` sur la socket d'écoute et renvoie `-1` en cas d'erreur, puis insère la socket de service dans le tableau de clients et renvoie `0`, ou s'il n'y a plus de place, refuse la connexion et renvoie `0`.
- 4) Écrire la fonction `void traitement_requete (Client *tc, int nc)` qui fait une lecture sur la socket de service du client `nc`, concatène les caractères lus dans le buffer du client, puis recherche un double retour chariot. S'il n'est pas trouvé, la fonction est interrompue, sinon la requête est considérée complète; un message est alors envoyé au client, puis il est déconnecté.

TP8 : Serveur TCP/IP

I. Serveur web (suite)

- 1) Écrire la fonction `int creer_socket_ecoute (int nport, int maxpend)` qui crée une socket, l'attache à l'adresse locale et au port `nport`, affiche le port ouvert, la déclare en socket d'écoute avec `maxpend` connexion pendantes. Renvoie la socket en cas de succès, -1 sinon, et dans ce cas referme la socket.
- 2) Tester le programme à l'aide du client `telnet localhost port` sur le port du serveur (pour déconnecter le client, taper `Ctrl+AltGr+`] puis taper `quit`) ou avec `~thiel/helium/demo/tcp`.
- 3) Modifier le programme en `serweb2.c` et rajouter l'analyse de la requête (principalement la recherche du nom de fichier demandé), l'envoi de l'entête de réponse suivi du fichier en question (en supposant qu'il soit au format HTML et dans un chemin autorisé).
- 4) Tester `serweb2` avec Mozilla (par exemple) sur l'adresse `http://localhost:port/fichier`.

Fonctions utilitaires et rappels

▷ On introduit les fonctions suivantes de `bor-util.c` :

```
int bor_accept_in (int soc, struct sockaddr_in *adr)
{
    socklen_t adrln = sizeof(struct sockaddr_in);
    int r = accept (soc, (struct sockaddr *) adr, &adrln);
    if (r < 0) bor_perror (__func__);
    return r;
}
char *bor_adrtoa_in (struct sockaddr_in *adr)
{
    static char s[32];
    sprintf (s, "%s:%d", inet_ntoa(adr->sin_addr), ntohs(adr->sin_port));
    return s;
}
```

▷ Une requête HTTP pour la méthode GET est de la forme :

```
GET /fichier HTTP/version <cr-lf>
Motclé: Valeur <cr-lf>
...
Motclé: Valeur <cr-lf>
<cr-lf>
```

La version de protocole est 1.0 ou 1.1; si c'est 1.1, il doit y avoir une ligne

`Host: adresse-serveur[:port] <cr-lf>` (par défaut le port 80). La fin de la requête est marquée par un double `<cr-lf>`. Un retour chariot `<cr-lf>` correspond à `"\n"` ou `"\r\n"`.

▷ Une réponse HTTP pour la méthode GET est de la forme :

```
HTTP/version code explication<cr-lf>
Motclé: Valeur <cr-lf>
...
Motclé: Valeur <cr-lf>
<cr-lf>
Corps de la réponse
```

La version de protocole est 1.0 ou 1.1. Le code est un entier de 3 chiffres (20x signifie succès, 30x redirection, 40x erreur du client, 50x erreur du serveur). L'explication est un message quelconque (Ok, not found, etc). La fin de l'entête est marquée par un double `<cr-lf>`; le corps de la réponse suit immédiatement.

▷ RFC de HTTP/1.0 ou 1.1 : <http://www.faqs.org/rfcs/rfc1945.html> ou [rfc2616.html](http://www.faqs.org/rfcs/rfc2616.html)

TD9 : Temporisation

I. Liste de timers

Dans un client/serveur, on a parfois besoin de plusieurs timers (exemple : délai de réponse pour chaque client, délai pour une tâche répétitive, etc) avec de plus une précision de l'ordre de la milliseconde. Une solution est d'utiliser `select` avec un timeout en argument, et de calculer ce timeout à partir des dates d'expiration d'une liste de timers.

On crée un module `bor-timer.c` dans lequel on définit les types et variables suivants (le fichier `bor-timer.h` ne contiendra que les prototypes des fonctions) :

```
typedef struct {
    int handle;
    void *data;
    struct timeval expiration;
} bor_timer_struct;

#define BOR_TIMER_MAX 1000
bor_timer_struct bor_timer_list[BOR_TIMER_MAX];
int bor_timer_nb = 0, bor_timer_uniq = 0;
```

Les timers sont stockés dans un tableau global `bor_timer_list`, de taille courante `bor_timer_nb`. Chaque timer a : un numéro unique `handle`, obtenu en incrémentant à chaque création de timer la variable globale `bor_timer_uniq`; une donnée associée `data` quelconque; une date d'expiration absolue `expiration` en secondes et microsecondes.

1) Écrire la fonction `int bor_timer_add (unsigned long delay, void *data)` qui ajoute un timer dont l'échéance sera dans `delay` millisecondes. `data` est l'adresse d'une donnée quelconque, que l'on pourra récupérer lorsque le timer arrivera à échéance. La fonction renvoie le `handle` du timer, qui permettra de reconnaître quel est le timer arrivé à échéance, ou encore de le supprimer.

L'insertion se fait par dichotomie dans une liste triée sur la date d'expiration; le prochain timer est donc toujours en position 0.

2) Écrire la fonction `void bor_timer_remove (int handle)` qui supprime un timer à partir de son `handle`. On maintient le tableau trié.

3) Écrire la fonction `struct timeval *bor_timer_delay ()` qui renvoie le délai entre le prochain timer (c'est-à-dire en position 0) et la date courante. Cette fonction pourra être passée directement en paramètre à `select`.

4) Écrire les fonctions `int bor_timer_handle ()` et `void *bor_timer_data ()` qui renvoient respectivement le `handle` ou la donnée du prochain client, sinon `-1` ou `NULL`.

II. Application : délais multiples

Écrire un programme `multitime.c` qui crée quatre timers avec délais respectifs de 2, 5, 10 et 20 secondes, puis boucle sur `select`. Le programme affiche l'échéance de chacun des timers; le timer 1 est réarmé toutes les 2 secondes; le timer 4 provoque l'arrêt du programme.

Rappels

▷ La fonction `gettimeofday(struct timeval *tv, NULL)`; définie par `sys/time.h` et `time.h` remplit les champs (entiers) de `struct timeval { time_t tv_sec; suseconds_t tv_usec; }`; en secondes et microsecondes avec la date absolue par rapport à l'Epoch.

TP9 : Serveur TCP/IP et timers

I. Serveur web (fin)

- 1) Récupérer sur www.lif-sud.univ-mrs.fr/~thiel/ens/rezo/tp08-serweb2.c la correction du TP8 et tester par exemple avec Mozilla sur l'adresse `http://localhost:port/index.html` (au besoin créez un répertoire `~/public_html` et un fichier `index.html`).
- 2) Récupérer sur www.lif-sud.univ-mrs.fr/~thiel/ens/rezo/ les fichiers `bor-timer.c` et `bor-timer.h`, puis tester le programme `multime.c` du TD9.
- 3) Modifier le programme `serweb2.c` du TP8 en `serweb3.c` et pour chaque client connecté rajouter un timer, qui le déconnecte si sa requête n'est pas complète au bout de 30s de connexion (ne pas oublier d'annuler le timer lorsque la requête est complète).
- 4) Dans le `main` donné au TD8, la sortie du `select` garantit le droit de faire un `read`, mais pas celui de faire un ou plusieurs `write` : si le client se bloque en lecture, le serveur est paralysé! La solution est de scruter aussi en écriture.

Modifier le programme en `serweb4.c` de telle sorte que le serveur utilise une liste de descripteurs en lecture et une seconde liste en écriture. Chaque client aura un état valant une des constantes `E_LIT_REQUETE`, `E_ECRIT_ENTETE` ou `E_ECRIT_FICHER`, utilisé et modifié par `traitement_requete`, et permettant à `init_select` de calculer les deux listes :

```
#define E_LIT_REQUETE    1
#define E_ECRIT_ENTETE  2
#define E_ECRIT_FICHER  3

typedef struct {
    ...
    int etat;                /* Etat : E_LIT_REQUETE, E_ECRIT_ENTETE, ... */
    char rep[4096];         /* Buffer réponse */
    int rep_a, rep_b;       /* Début et fin dans rep */
    int fd;                 /* Fichier à transmettre */
} Client;

void init_client (Client *tc, int nc)
{
    ...
    tc[nc].etat = E_LIT_REQUETE;
    tc[nc].rep[0] = 0;
    tc[nc].fd = -1;
    tc[nc].rep_a = 0;
    tc[nc].rep_b = 0;
}

void traitement_requete (Client *tc, int nc)
{
    switch (tc[nc].etat) {
        case E_LIT_REQUETE : lit_requete (tc, nc); break;
        case E_ECRIT_ENTETE : ecrit_entete (tc, nc); break;
        case E_ECRIT_FICHER : ecrit_fichier (tc, nc); break;
    }
}
```

- 5) Rajouter un timer pour chaque client en cours d'émission afin qu'il soit déconnecté s'il n'a pas fini de lire sa réponse au bout de 2 minutes.

TD10 : Client TCP/IP avec Flex

I. Aspirateur Web

On se propose d'améliorer l'aspirateur Web `aspiweb.c` du TP7-II en utilisant le générateur d'analyseur lexical `flex`.

On suppose disposer d'une fonction `creer_socket_connectee` (`char *adrS_nom`, `int nport`) qui crée une socket, l'attache à l'adresse locale et au port 0, puis construit l'adresse du serveur à partir du nom `adrS_nom` et du port `nport`, et enfin se connecte au serveur. La fonction renvoie la socket connectée, sinon renvoie -1 et affiche un message d'erreur.

1) Réécrire le programme en un fichier `aspiweb2.lex`. Le programme reçoit en argument l'adresse du serveur, le numéro de port web et le chemin d'un fichier, se connecte au serveur avec la fonction `creer_socket_connectee`, puis envoie la requête `"GET /chemin HTTP/1.0\r\n\r\n"`. Le programme associe ensuite un `FILE*` à la socket et lance l'analyseur lexical. Celui-ci affiche tout ce qui est reçu sur la sortie standard.

2) Ajouter des règles pour que les voyelles accentuées HTML (par exemple `´`; ```; `ˆ`; `¨`;) soit traduites par le caractère correspondant (é è ê ë); de même pour ` `; `<`; `>`; traduits par un espace, '`<`' et '`>`'.

3) Ajouter des règles pour que les balises ne soient pas affichées.

4) Ajouter les états `HEADER` (première ligne et suivantes, terminé par un double retour chariot), `NORMAL` (texte normal, contenant des caractères HTML), `BALISE` (une balise); écrire les règles de transition entre états, et afficher chaque fois un message.

5) Déclarer en global des tableaux de tokens et de chaînes, y mémoriser les tokens et mots présents dans une balise. Ces tableaux sont réinitialisés chaque fois que l'on entre dans l'état `BALISE`; ils sont analysés chaque fois que l'on sort de cet état. Utiliser ce mécanisme pour afficher les liens hypertexte.

6) Dans la question précédente, remplacer l'affichage des liens hypertexte par l'appel d'une fonction qui crée un fils, qui se recouvre par `./aspiweb2` sur le serveur et la page correspondante; ajouter un test de profondeur récursive.

TP10 : Client TCP/IP avec Flex

I. Aspirateur Web (suite)

1) Taper le programme et écrire la fonction `creer_socket_connectee` puis tester. Pour générer le fichier C : `flex -oaspiweb2.yy.c aspiweb2.lex`; pour le compiler : `gcc -Wall aspiweb2.yy.c bor-util.c -lfl -o aspiweb2`. Exemple d'utilisation : `./aspiweb2 sol 80 'index.html'`.

2) Ajouter des arguments optionnels dans la ligne de commande pour afficher ou non l'entête, le texte, les balises, enregistrer le fichier, ou faire un parcours récursif en précisant la profondeur.

Rappels

▷ La fonction `FILE *fdopen (int fd, const char *mode)`; définie par `stdio.h` associe un flux à un descripteur ouvert `fd`. Le `mode` ("`r`", "`w`", "`a`", etc) doit être compatible avec le mode d'ouverture original de `fd`. Fermer ensuite le flux et le descripteur avec `fclose`.