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 se termine 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 terminé.

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 terminé ; par défaut il est ignoré.

TP1 : Processus et signaux

I. Boîte à outils réseau et Makefile

Important : le but de ce cours étant aussi d'apprendre à exploiter les outils Unix et la ligne de commande, il vous est demandé de ne pas utiliser les éditeurs : Eclipse, QtCreator, XCode.

Créer un répertoire `TP-reseau` ; récupérer sur la page de l'UE (indiquée dans le lien en bas à droite de cette feuille) les fichiers `bor-util.h` et `bor-util.c` de la "Boîte à Outil Réseaux" qui accompagne ce cours, ainsi que le fichier `Makefile`, puis les enregistrer dans `TP-reseau/`.

Tous les programmes C que vous écrirez seront placés dans ce répertoire. Étant donné un programme `toto.c`, pour le compiler il faut d'abord éditer le fichier `Makefile` et rajouter le nom de l'exécutable (ici `toto`) après le symbole `=` qui suit l'une des variables `EXECS` (compilation sans `bor-util.c`) ou `EXECSUTIL` (avec `bor-util.c`). Taper ensuite :

`make toto`	pour compiler `toto.c` et créer l'exécutable ;
`make all`	pour compiler tous les fichiers qui ont été modifiés ;
`make clean`	pour effacer les éventuels fichiers `.o` ;
`make distclean`	pour effacer aussi les exécutables.

Tester sur un exemple, puis utiliser systématiquement ce `Makefile` pour tous les programmes que vous écrirez en TP réseau.

II. Comptes à rebours et comptage de fils

1) Écrire un programme C `rebourfils.c` qui lit un entier n au clavier puis recommence, en bouclant ainsi jusqu'à ce que n soit nul. 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 (ils seront lus par le père) pendant l'affichage des comptes à rebours (par les fils).

2) Après la lecture de 0 au clavier, le père attend que *tous* ses fils soient morts (en faisant un `wait` par fils créé), puis le père affiche le message "fin de tous les fils détectée".

3) Découper le programme en fonctions, avec des noms de fonctions explicites.

III. Signaux et timeout

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 placer le nom de l'exécutable dans la variable `EXECSUTIL` du `Makefile`.

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.

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. Lecture et écriture

On se propose d'étendre les primitives `read` et `write` en rajoutant un affichage lors de toute erreur, ou de la détection de fin de fichier. Écrire les fonctions :

`ssize_t bor_read (int fd, void *buf, size_t count);` lit au plus `count` octets dans `fd` et les mémorise dans `buf`, puis renvoie le résultat de `read`.

`ssize_t bor_read_str (int fd, char *buf, size_t count);` appelle la fonction `bor_read` pour `count-1` caractères, puis rajoute un `'\0'` terminal en cas de succès.

`ssize_t bor_write (int fd, const void *buf, size_t count);` écrit au plus `count` octets dans `fd` provenant de `buf`, puis renvoie le résultat de `write`.

`ssize_t bor_write_str (int fd, const char *buf);` écrit la chaîne de caractères `buf` dans `fd`, puis renvoie le résultat de `write`.

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 fils redirige la sortie standard sur le tube, le père 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. 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. 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.

III. 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 erreur ou 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 *encore 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 éligible, 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 éligible, il fait une lecture 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. Fonctions utilitaires

On introduit quelques fonctions utiltaires de `bor-util.c` à la page suivante.

1) Écrire la fonction `void bor_set_sockaddr_un (const char *path, struct sockaddr_un *sa)` qui construit une adresse de socket `sa` du domaine `AF_UNIX` avec le chemin `path`.

2) Écrire la fonction `int bor_create_socket_un (int type, const char *path, struct sockaddr_un *sa)` qui crée une socket du domaine `AF_UNIX` et du type `type`, puis construit une adresse locale dans `sa` avec le chemin `path`, et enfin attache la socket à cette adresse locale. La fonction renvoie la socket $\geqslant 0$ en cas de succès, sinon renvoie -1 et affiche un message d'erreur.

II. 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 compiler à l'aide du `Makefile`. 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).

Rappels

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

T.S.V.P.

Fonctions utilitaires

```
void bor_perror (const char *funcname)
{
    int e = errno; perror (funcname); errno = e;
}

int bor_bind_un (int soc, struct sockaddr_un *sa)
{
    int r = bind (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_un));
    if (r < 0) bor_perror (__func__);
    return r;
}

ssize_t bor_recvfrom_un (int soc, void *buf, size_t count, struct sockaddr_un *sa)
{
    socklen_t adrlen = sizeof(struct sockaddr_un);
    ssize_t  r = recvfrom (soc, buf, count, 0, (struct sockaddr *) sa, &adrlen);
    if (r < 0) bor_perror (__func__);
    return r;
}

ssize_t bor_recvfrom_un_str (int soc, char *buf, size_t count, struct sockaddr_un *sa)
{
    ssize_t r = bor_recvfrom_un (soc, buf, count-1, sa);
    if (r >= 0) buf[r] = '\0';
    return r;
}

ssize_t bor_sendto_un (int soc, const void *buf, size_t count,
                       const struct sockaddr_un *sa)
{
    ssize_t r = sendto (soc, buf, count, 0, (struct sockaddr *) sa,
        sizeof(struct sockaddr_un));
    if (r < 0) bor_perror (__func__);
    return r;
}

ssize_t bor_sendto_un_str (int soc, const char *buf, const struct sockaddr_un *sa)
{
    return bor_sendto_un (soc, buf, strlen (buf), sa);
}
```

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 peut-être 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_listen (int soc, int max_pending) {
    int r = listen (soc, max_pending);
    if (r < 0) bor_perror (__func__);
    return r;
}
int bor_accept_un (int soc, struct sockaddr_un *sa) {
    socklen_t adrlen = sizeof(struct sockaddr_un);
    int r = accept (soc, (struct sockaddr *) sa, &adrlen);
    if (r < 0) bor_perror (__func__);
    return r;
}
int bor_connect_un (int soc, const struct sockaddr_un *sa) {
    int r = connect (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_un));
    if (r < 0) bor_perror (__func__);
    return r;
}
```

TD7 : Clients TCP/IP

I. Fonctions utilitaires

1) Écrire la fonction `void bor_set_sockaddr_in (int port, uint32_t ipv4, struct sockaddr_in *sa)` qui construit une adresse de socket `sa` du domaine `AF_INET` à partir d'une adresse IP `ipv4` et un port `port`.

2) Écrire la fonction `int bor_create_socket_in (int type, int port, struct sockaddr_in *sa)` qui crée une socket du domaine `AF_INET` et du type `type`, puis construit une adresse locale dans `sa` avec le port `port`, et enfin attache la socket à cette adresse locale. La fonction renvoie la socket $\geqslant 0$ en cas de succès, sinon renvoie -1 et affiche un message d'erreur.

3) Écrire la fonction `int bor_resolve_address_in (const char *host, int port, struct sockaddr_in *sa)` qui résout une adresse `host` puis la stocke avec le port `port` dans `sa`. La fonction renvoie 0 en cas de succès, sinon renvoie -1 et affiche un message d'erreur.

II. 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.

III. Défragmentation de lignes

Le protocole TCP/IP ne préserve pas les limites des messages, si bien qu'en lecture, les messages peuvent êtres 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.

TP7 : Clients TCP/IP

I. Défragmentation de lignes : tests

1) Reprenez le client `defrag.c` du TD7-II et testez-le ainsi : dans un terminal 1 on simule un serveur daytime avec la commande : `while true ; do date | netcat -lv4 13000 ; done`
Dans un terminal 2 on simule un client en exécutant plusieurs fois : `netcat localhost 13000`
Dans le terminal 2 on exécute enfin `defrag.c` avec les arguments `localhost` et 13000.

2) Dans le terminal 1 interrompez la commande et tapez à la place `netcat -lv4 13000` et dans le terminal 2 relancez `defrag.c`. Tapez ensuite des lignes dans le terminal 1, elle apparaitront dans le terminal 2 via votre client `defrag.c`. Pour interrompre le dialogue, tapez `^C` ou `^D`.

3) Testez la défragmentation proprement dite. Dans le terminal 1 tapez `stty cbreak` pour passer en mode caractère (à la fin de l'exercice on remettra le terminal en mode ligne en tapant `stty -cbreak`) puis relancez `netcat`.

Tapez des caractères dans le terminal 1, ils seront transmis un à un au client. Pour testez la défragmentation en envoyant en une fois une chaîne comportant des retours chariots, il suffit d'ouvrir un éditeur, de copier un bloc de texte, puis de coller dans le terminal 1 ce bloc en faisant [CTRL][SHIFT]v.

II. Aspirateur Web

1) Copiez 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 requète (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.dil.univ-mrs.fr` sur le port `80` et envoyer la requète `"GET /~thiel/essai.txt HTTP/1.0\r\n\r\n"`.

Comparez en exécutant `netcat -C sol.dil.univ-mrs.fr 80` puis en tapant la requète `GET /~thiel/essai.txt HTTP/1.0` et 2 fois sur [Entrée].

Dans `aspiweb.c`, remplacez `~thiel/essai.txt` par le chemin de fichier donné en argument 3 (attention, bien conserver dans la requète le `/` qui est devant) et testez avec l'argument `'~thiel/essai2.txt'` (en conservant les `''` qui protègent le `~`).

Remarque : si la requète 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.

Fonctions utilitaires

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

```
int bor_bind_in (int soc, struct sockaddr_in *sa)
{
    int r = bind (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_in));
    if (r < 0) bor_perror (__func__);
    return r;
}
int bor_connect_in (int soc, const struct sockaddr_in *sa)
{
    int r = connect (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_in));
    if (r < 0) bor_perror (__func__);
    return r;
}
int bor_getsockname_in (int soc, struct sockaddr_in *sa)
{
    socklen_t adrlen = sizeof(struct sockaddr_in);
    int r = getsockname (soc, (struct sockaddr *) sa, &adrlen);
    if (r < 0) bor_perror (__func__);
    return r;
}
```

TD8 : Serveur TCP/IP

I. Serveur générique

On se propose d'écrire un serveur générique `serveur1.c` par scrutation, que l'on spécialisera ensuite en serveur web.

1) Le serveur doit être capable de gérer plusieurs connexions simultanées avec des clients. On se donne le type `Slot` suivant pour mémoriser les données de connexion avec un client (on y rajoutera des champs par la suite). Chaque slot possède un état, permettant au serveur de savoir s'il doit scruter en lecture (pour une requête) ou en écriture (pour une réponse).

```
typedef enum { E_LIBRE, E_LIRE_REQUETE, E_ECRIRE_REPONSE } Etat;

typedef struct {
    Etat etat;
    int  soc;                  /* Socket de service, défaut -1 */
    struct sockaddr_in adr;    /* Adresse du client */
} Slot;
```

Écrire les fonctions suivantes : `void init_slot (Slot *o)` initialise les champs de `o`, en particulier spécifie que `o` est libre. `int slot_est_libre (Slot *o)` renvoie vrai si le slot est libre. `void liberer_slot (Slot *o)` ne fait rien si `o` est libre, sinon ferme la socket et réinitialise `o`.

2) Pour mémoriser les informations du serveur on définit le type :

```
#define SLOTS_NB 32
typedef struct {
    Slot slots[SLOTS_NB];
    int soc_ec;               /* Socket d'écoute */
    struct sockaddr_in adr;   /* Adresse du serveur */
} Serveur;
```

Écrire les fonctions suivantes : `void init_serveur (Serveur *ser)` initialise tous les slots dans `ser` et la socket d'écoute à -1. `int chercher_slot_libre (Serveur *ser)` renvoie l'indice du premier slot libre, sinon -1.

3) Écrire `int demarrer_serveur (Serveur *ser, int port)` qui initialise le serveur, puis crée une socket TCP/IP attachée au `port`, ouvre le service, et mémorise la socket d'écoute et son adresse dans `ser`.
Écrire `void fermer_serveur (Serveur *ser)` qui ferme la socket d'écoute et libère tous les slots.

4) Écrire la fonction `int accepter_connexion (Serveur *ser)`, qui accepte une connexion d'un client puis cherche un slot libre, l'y mémorise et passe l'état à `E_LIRE_REQUETE`, sinon elle ferme la connexion (trop de clients). Cette fonction sera appelée lorsque la socket d'écoute sera éligible.

5) On suppose disposer de la fonction `int proceder_lecture_requete (Slot *o)` (qui commence par faire un `read`) et de `int proceder_ecriture_reponse (Slot *o)` (qui commence par faire un `write`). Ces fonctions seront réalisées en TP.
Écrire `void traiter_slot_si_eligible (Slot *o, fd_set *set_read, fd_set *set_write)`, qui teste l'égibilité de `o->soc` en fonction de l'état du slot `o`, puis selon le cas appelle l'une ou l'autre des fonctions `proceder_...` et récupère le résultat. S'il est $\leqslant 0$ alors `traiter_slot_si_eligible` libère le slot `o` (ce qui ferme la connexion). Cette fonction sera appelée lors de la scrutation.

6) On peut maintenant passer à la préparation de la scrutation. Écrire la fonction `void preparer_select (Serveur *ser, int *maxfd, fd_set *set_read, fd_set *set_write)` qui initialise et place dans les listes `set_read` ou `set_write` les descripteurs à scruter en lecture ou en écriture, et mémorise leur valeur maximale dans `*maxfd`.

TP8 : Serveur TCP/IP

I. Serveur générique (suite)

1) Écrire la fonction `int faire_scrutation (Serveur *ser)` qui scrute une seule fois les sockets en lecture ou en écriture, puis accepte éventuellement une connexion et enfin traite tous les slots éligibles.

2) Écrire le programme principal, qui accepte en argument le numéro de port, puis démarre le serveur. Dans une boucle interruptible par SIGINT, il fait une scrutation des sockets. Enfin il ferme le serveur.

3) Écrire la fonction `int proceder_lecture_requete (Slot *o)` qui fait un `read` dans la socket de service, puis passe le slot dans l'état `E_ECRIRE_REPONSE` si le résultat du `read` est $\geqslant 0$. La fonction renvoie le résultat du `read`.

4) Écrire la fonction `int proceder_ecriture_reponse (Slot *o)` qui fait un `write` dans la socket de service, puis passe le slot dans l'état `E_LIRE_REQUETE` si le résultat du `write` est $\geqslant 0$. La fonction renvoie le résultat du `write`.

5) Tester le programme avec `netcat localhost` *port* dans plusieurs terminaux ; testez le comportement du serveur lorsque la capacité en slots est dépassée.

Testez ensuite le programme avec un navigateur web : tapez l'URL `"http://localhost:`*port*`"`. Vous constaterez que le navigateur n'affiche rien et reste en attente.

6) Recopiez `serveur1-tcpip.c` en `serveur2-tcpip.c` et modifiez `proceder_ecriture_reponse` de façon à ce qu'il coupe la communication après écriture (il suffit de ne pas renvoyer 1). Vous pourrez constater que le navigateur affiche maintenant le message du serveur.

Remplacez la réponse dans `proceder_ecriture_reponse` par le code HTML suivant :

```
<html><body><h1>Serveur en construction !!</h1></body></html>\r\n
```

Vous constaterez que le navigateur affiche la réponse mais sans l'interpréter comme du code HTML.

Remplacez la réponse dans `proceder_ecriture_reponse` par le code HTML suivant :

```
HTTP/1.1 500 Erreur du serveur\r\n\r\n
<html><body><h1>Serveur en construction !!</h1></body></html>\r\n
```

Cette fois-ci vous devriez voir le message affiché en gras.

Fonctions utilitaires

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

```
int bor_accept_in (int soc, struct sockaddr_in *adr)
{
    socklen_t adrlen = sizeof(struct sockaddr_in);
    int r = accept (soc, (struct sockaddr *) adr, &adrlen);
    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;
}
```

TD9 : Serveur TCP/IP

I. Serveur web

On se propose de modifier le programme `serveur2-tcpip.c` réalisé au TP8 en un serveur web `serweb1.c`.

1) On rajoute dans `Slot` les champs `char req[REQSIZE]` pour stocker la requête, `int req_pos` pour retenir la position d'insertion lors de la prochaine lecture, et `int fin_entete` pour mémoriser la position de la fin de l'entête dans la requête. Ces champs seront initialisés dans `init_slot`.

Écrire `int lire_suite_requete (Slot *o)`, qui fait une lecture dans la socket, mémorise dans `o->req` à partir de la position d'insertion `o->req_pos` puis met à jour cette position. Elle renvoie le résultat de `read`.

2) On rajoute dans `Slot` les champs `char rep[REPSIZE]` pour stocker la réponse, et `int rep_pos` pour retenir la position non déjà envoyée de la réponse. Ces champs seront initialisés dans `init_slot`.

Écrire `int ecrire_suite_reponse (Slot *o)`, qui fait dans la socket une écriture de la réponse à partir de `o->rep_pos`, puis met à jour cette position. Elle renvoie le résultat de `write`.

3) Écrire la fonction `int chercher_fin_entete (Slot *o, int debut)` qui cherche le marqueur de fin de l'entête HTTP à partir de la position `debut` dans la requête `o->req`. Elle renvoie la position du marqueur si elle le trouve, sinon -1.

4) On se donne les types suivants, qui vont servir à mémoriser le résultat de l'analyse de l'entête HTTP, afin de générer la réponse appropriée.

```
typedef enum {
    M_NONE,
    M_GET,
    M_TRACE,
} Id_methode;

typedef enum {
    C_OK             = 200,
    C_BAD_REQUEST    = 400,
    C_NOT_FOUND      = 404,
    C_METHOD_UNKNOWN = 501,
} Code_reponse;
```

```
typedef struct {
    char methode[REQSIZE],
         url[REQSIZE],
         version[REQSIZE],
         chemin[REQSIZE];
    Id_methode   id_meth;
    Code_reponse code_rep;
} Infos_entete;
```

Écrire la fonction `char *get_http_error_message (Code_reponse code)` qui en fonction du code reçu (`C_OK`, etc), renvoie une chaîne de caractères le traduisant (`"OK"`, etc).

Écrire la fonction `Id_methode get_id_methode (char *methode)` qui, en fonction de la `methode` reçue (`"GET"`, etc), renvoie le code correspondant (`M_GET`, etc) sinon `M_NONE`.

5) On suppose disposer de la fonction `void analyser_requete (Slot *o, Infos_entete *ie)` qui analyse l'entête de la requête `o->req` (pour le moment elle met `ie->code_rep` à `C_NOT_FOUND`). On suppose disposer également de la fonction `void preparer_reponse (Slot *o, Infos_entete *ie)` qui en fonction de `ie`, construit et mémorise dans `o->rep` l'entête de la réponse HTTP.

Modifier la fonction `int proceder_lecture_requete (Slot *o)` afin qu'elle lise la suite de la requête, puis cherche la position du marqueur de fin d'entête HTTP. Si le marqueur est trouvé elle analyse la requête, prépare la réponse puis passe à l'étape `E_ECRIRE_REPONSE`.

6) Modifier la fonction `int proceder_ecriture_reponse (Slot *o)` afin qu'elle écrive la suite de la réponse HTTP, puis renvoie 1 si la réponse n'a pu être complètement écrite.

TP9 : Serveur TCP/IP

I. Serveur web (suite)

1) Taper et tester le programme du TD9, avec `netcat` puis avec un navigateur web.

2) On introduit le champ `fic_fd` dans le type `Slot` ; on l'initialise à -1 dans `init_slot`, enfin on ferme `fic_fd` dans `liberer_slot` s'il est différent de -1.

On suppose que l'URL contenu dans `ie->url` est sous la forme `"chemin"` ou `"chemin?paramètres"`. Écrire la fonction `int preparer_fichier (Slot *o, Infos_entete *ie)` qui extrait et mémorise dans `ie->chemin` le chemin contenu au début de `ie->url`. La fonction ouvre ensuite le fichier `ie->chemin` en lecture, puis mémorise le descripteur dans `o->fic_fd` et renvoie 0, sinon -1.

3) Écrire `void analyser_requete (Slot *o, Infos_entete *ie)` qui reçoit une entête complète dans `o->req`, l'analyse et mémorise dans `ie` les résultats, en particulier le code de la réponse dans `ie->code_rep`. Pour le moment on se contente d'analyse la première ligne qui doit avoir la forme `"methode /url version"`, où `methode` est soit `"GET"` soit `"TRACE"`, et `version` est soit `"HTTP/1.0"` soit `"HTTP/1.1"` (se servir de `sscanf`, `get_id_methode` et `strcasecmp`).

Si la forme est erronée, `ie->code_rep` sera mis à `C_BAD_REQUEST` ; si la méthode est inconnue, `ie->code_rep` sera mis à `C_METHOD_UNKNOWN` ; si la méthode est `"TRACE"`, `ie->code_rep` sera mis à `C_OK` ; si la méthode est `"GET"`, alors on préparera le fichier puis on mettra `ie->code_rep` à `C_OK` ou `C_NOT_FOUND`.

4) Écrire la fonction `void preparer_reponse (Slot *o, Infos_entete *ie)` qui en fonction de `ie->code_rep` construit et mémorise dans `o->rep` une réponse HTTP appropriée, de la forme

```
HTTP/1.1 code_rep message_erreur_ou_succès
Date: la_date
Server: serweb2
Connection: close
Content-Type: type_du_contenu

Corps de la réponse ou fichier
```

Si le code de la réponse n'est pas `C_OK` on construira un texte HTML d'erreur, de type `"text/html"`.

Si la méthode est `M_TRACE`, on construira en réponse le contenu original de la requête ; le type du contenu sera `"message/http"`.

Si la méthode est `M_GET`, on se contentera de construire un texte HTML disant que le fichier a été trouvé (la partie envoi du fichier sera abordée au TP suivant) ; le type du contenu sera `"text/html"`.

Rappels

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

TD10 : 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 stoqués 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 reconnaitre 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 `multime.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 micro-secondes avec la date absolue par rapport à l'Epoch.

TP10 : Serveur TCP/IP et timers

I. Serveur web (fin)

1) Sur la page de l'UE (indiquée dans le lien en bas à droite de cette feuille), récupérer les fichiers `bor-timer.c` et `bor-timer.h`, puis tester le programme `multime.c` du TD10.

2) Modifier le programme `serweb2.c` du TP9 en `serweb3.c`.

Pour chaque connexion, rajouter un timer, qui déconnectera le slot si sa transaction n'est pas complète au bout de 30s. Conseil : lors de l'ajout du timer, donner le slot en paramètre `data`, ce qui permettra de le retrouver directement à l'échéance avec `bor_timer_data`.

3) On se consacre maintenant à l'envoi de fichiers pour le méthode `GET`. Cet envoi sera fait par tronçons, lus dans le fichier et écrits dans la socket du slot concerné, chaque fois que les descripteurs seront éligibles en lecture ou écriture, respectivement.

On rajoute donc deux états `E_LIRE_FICHIER` et `E_ENVOYER_FICHIER`, puis dans le type `Slot` les champs suivants : un buffer `fic_bin` de capacité `FICSIZE` pour mémoriser un tronçon de fichier, la position courante `fic_pos` dans le tronçon, et la taille `fic_len` du tronçon. Les deux derniers champs sont initialisés à 0.

4) Modifier `preparer_select` de façon à tenir compte des deux nouveaux états : pour l'état `E_LIRE_FICHIER` on insère `o->fic_fd` en lecture, tandis que pour l'état `E_ENVOYER_FICHIER` on insère `o->soc` en écriture.

5) On suppose disposer de la fonction `int proceder_lecture_fichier (Slot *o)` et de la fonction `int proceder_envoi_fichier (Slot *o)`. Modifier `traiter_slot_si_eligible` de façon à ce que pour les deux nouveaux états ces fonctions soient correctement appelées.

6) Actuellement, `proceder_ecriture_reponse` appelle `ecrire_suite_reponse` puis renvoie 1 si la réponse n'a pas été complètement écrite, sinon -1 (ce qui aura pour effet de libérer le slot).

Modifier la fin de la fonction ainsi : si le fichier `o->fic_fd` est ouvert (c'est-à-dire $\neq$ -1), on passe à l'état `E_LIRE_FICHIER` et on renvoie 1, sinon on renvoie -1. Le passage à l'état `E_LIRE_FICHIER` provoquera l'appel de `proceder_lecture_fichier` par `traiter_slot_si_eligible` lors de la prochaine scrutation.

Écrire `int proceder_lecture_fichier (Slot *o)` qui lit un tronçon de `o->fic_fd`, le mémorise dans `o->fic_bin`, initialise `o->fic_pos` à 0 et `o->fic_len` à la longueur du tronçon lu. En cas de succès la fonction passe à l'état `E_ENVOYER_FICHIER` et renvoie 1, sinon renvoie -1. Le passage à l'état `E_ENVOYER_FICHIER` provoquera l'appel de `proceder_envoi_fichier` par `traiter_slot_si_eligible` lors de la prochaine scrutation.

Écrire `int proceder_envoi_fichier (Slot *o)` qui envoie dans la socket le contenu de `o->fic_bin` à partir de la position `o->fic_pos`, puis met à jour cette position. Si `o->fic_bin` n'a pas encore été entièrement envoyé, la fonction affiche un message et renvoie 1 (elle sera rappelée lors de la prochaine scrutation pour envoyer la suite). Sinon, on passe à l'état `E_LIRE_FICHIER` et on renvoie 1, ce qui aura pour effet de procéder à la lecture du tronçon suivant du fichier.

Finalement quand s'arrête ce va-et-vient entre ces deux états ? Lors de `proceder_lecture_fichier` : lorsque `read` renvoie 0, cela veut dire que l'on a fini de lire le fichier et déjà envoyé intégralement son contenu dans la socket ; on renvoie alors 0 ce qui libère le slot (et ferme le fichier).

7) Pour que le navigateur affiche correctement les pages web il faut lui envoyer des réponses qui décrivent correctement le type et la longueur du contenu envoyé.

Écrire la fonction `char *get_extension (char *chemin)` qui renvoie la dernière extension du `chemin`, sinon `""`. Par exemple si `chemin` est `"pub/chap1/figure1.ppm.gz"` alors la fonction renverra `".gz"` .

Insérer le champ `type_mime[100]` dans `Infos_entete`. Écrire la fonction `void chercher_type_mime (Infos_entete *ie)` qui récupère l'extension de `ie->chemin` puis mémorise dans `ie->type_mime` le type MIME correspondant à l'extension : `"text/html"` pour `".html"`, `"text/css"` pour `".css"`, `"image/png"` pour `".png"`, `"image/jpeg"` pour `".jpg"`, sinon `"application/octet-stream"` par défaut.

Appeler `chercher_type_mime` dans `analyser_requete` pour la methode `M_GET`.

Enfin, dans `preparer_reponse` pour la méthode `M_GET`, afficher dans l'entête de la réponse HTTP la clé `"Content-Type:"` suivie de la valeur stockée dans `ie->type_mime`.

8) Actuellement, `preparer_fichier` ouvre le fichier et mémorise le descripteur dans `o->fic_fd`, puis renvoie 0 en cas de succès, sinon -1. Insérer le champ `off_t file_size` dans `Infos_entete` puis modifier la fonction de telle manière que `ie->file_size` contienne la taille du fichier ouvert (sinon 0). La taille sera obtenue avec la fonction `stat` ou `fstat`.

Ces fonctions permettent aussi de tester si un chemin correspond à un fichier régulier ou à un répertoire. On pourra ainsi encore améliorer le traitement : si le chemin correspond à un répertoire, on concaténera `index.html` au chemin avant de procéder au test de l'ouverture. (En réalité pour que cela fonctionne bien il faut limiter cette possibilité au cas où le chemin finit par un `'/'` ; dans le cas contraire on est supposé rajouter une propriété `"Location: http://host/chemin/"` dans la réponse, qui nécessite de récupérer la propriété `"Host:"` dans la requête).

Pour finir, rajouter dans `preparer_reponse` pour la méthode `M_GET` l'affichage de la clé `"Content-Length:"` et de la valeur `ie->file_size`.