M4101C - Programmation système avancée
Implémentation d'un serveur web

Michaël Hauspie Jean-Marie Place Damien Riquet

26 janvier 2016

Table des matières

Chapitre 1

Description générale

L'objectif du module est de continuer à vous faire découvrir les primitives système offertes par les noyaux UNIX de type POSIX. Afin d'appliquer ces connaissances à un exemple concret, vous allez développer votre propre serveur web. A la fin du module, votre serveur sera capable :

1. d'accepter des connexions en TCP sur IPv4 et sur IPv6 ;
2. de servir les requêtes GET du protocole HTTP ;
3. de servir plusieurs clients simultanément ;
4. de servir du contenu statique (html, css, js, images, son et vidéo) ;
5. de recevoir des commandes de configuration d'un autre processus via des outils de communications entre processus offerts par un noyau UNIX.

Pour les plus efficaces d'entre vous, le serveur pourra être amélioré, par exemple :

1. en ajoutant la gestion des `VirtualHost` comme dans Apache ;
2. en ajoutant le support de pages dynamiques écrites en langage C ;
3. en ajoutant le support de pages dynamiques à l'aide de la norme CGI ;
4. en ajoutant le support des requêtes POST, PUT, DELETE, HEAD, OPTIONS du protocole HTTP ;
5. ...

1.1 Pré-requis

Les pré-requis de ce module sont bien évidemment ceux du module M3101 et plus particulièrement :

1. savoir écrire un programme en langage C organisé en plusieurs fichiers et le compiler à l'aide de l'outil `make` ;
2. savoir manipuler les tableaux, les chaînes de caractères C, les structures et les pointeurs ;
3. connaître la différence entre les données allouées statiquement, les données allouées sur la pile et les données allouées dynamiquement à l'aide de la fonction `malloc` ;
4. savoir manipuler un fichier (au sens général) à l'aide des primitives dites de « bas niveau » (`open`, `read`, `write`, `close`) et connaître la signification d'un descripteur de fichier ;
5. savoir utiliser les tubes ;
6. savoir effectuer des redirection à l'aide de la fonction `dup` ;
7. savoir gérer des processus à l'aide de `fork`, `wait`, `waitpid` et les fonctions `exec*`.

Vous aurez également besoin de connaissances issues d'autres modules :

1. savoir utiliser l'outil de contrôle de version `git` ;
2. connaître un minimum de notion réseau : adresse IP (v4, v6), protocole TCP, port ;

1.2 Consignes générales

Vous devrez vous organiser par groupe de deux[1]. L'évaluation du module sera faite sur la base du serveur web que vous aurez développé. Vous devrez nous rendre votre dépôt git (cf. chapitre 2 pour la mise en place) sous la forme d'une archive ou éventuellement d'un lien vers une plateforme d'hébergement de dépôt git nous permettant de cloner le dépôt (serveur git de l'IUT, github...). Votre note sera évaluée en fonction de la version finale de votre serveur, mais également de l'historique de développement. Vous suivrez donc **scrupuleusement** les indications des différents chapitres concernant les branches ou les tags à créer sur votre dépôt.

1. Si votre groupe contient un nombre impair d'étudiants, un groupe de trois est autorisé.

Chapitre 2

Mise en place de l'environnement de travail

Le programme que nous allons écrire dans ce module est plus complexe que ce que vous avez développé jusqu'à présent en C. Nous allons donc organiser l'environnement de travail pour simplifier la tâche.

2.1 Contrôle de version

Que vous développiez seul ou à plusieurs, s'imposer l'utilisation d'un système de contrôle de version est une bonne habitude.

Si vous êtes seul, vous ne perdez rien de votre travail passé. Vous pouvez tester des fonctionnalités dans de nouvelles branches et les intégrer une fois terminées ou les abandonner en revenant facilement à une version stable.

Si vous collaborez avec d'autres développeurs, en plus des avantages précédents, les systèmes de contrôle de version permettent de fusionner facilement le travail de tout le monde.

2.1.1 Mise en place de git

Commencez par créer un répertoire pour votre projet et faites en un dépôt git à l'aide des commandes [1] :

```
$ mkdir pawnee && cd pawnee && git init
```

2.2 Organisation du dépôt

2.2.1 Premiers fichiers

Créez et ajouter les fichiers suivants à la racine de votre dépôt

1. `README` : vous indiquerez dans ce fichier au fur et à mesure du développement les instructions nécessaire à qui voudra compiler, configurer et utiliser votre serveur ;
2. `AUTHORS` : vous indiquerez dans ce fichier les noms, prénoms et adresse mail des auteurs (vous) ;
3. `LICENSE` : ce fichier devra contenir le texte relatif à la licence d'utilisation de votre logiciel. Sauf volonté contraire de votre part, vous utiliserez la licence GPL.

1. Vous devez trouver un autre nom pour votre serveur web, un peu d'imagination.

2.2.2 Organisation des sources

Créez le répertoire `webserver`. Ce répertoire contiendra les sources du serveur web. Par la suite, nous ajouterons d'autres utilitaires dont les fichiers source seront situés dans d'autres répertoires.

2.2.3 Fichiers `Makefile`

Pour l'instant, nous allons créer deux fichiers `Makefile`, l'un à la racine du dépôt, l'autre dans le répertoire `webserver`. Le premier fichier permettra, à terme, de construire tous les programmes liés au serveur web : le serveur ainsi que les programmes de gestion.

Listing 2.1 – Fichier `Makefile` racine

```
# Fichier Makefile racine

# Cette variable contient la liste des sous répertoires
# pour lesquels il faudra construire un programme.
# Pour l'instant, seul le serveur web est dans la liste.
FOLDERS=webserver

# Indique à make les règles qui ne correspondent pas à la création
# d'un fichier
# make lancera toujours la construction de cette règle si elle est
# demandée, même si un fichier/répertoire du nom de la cible existe
# On indique ici la règle all ainsi que les répertoires
.PHONY: all $(FOLDERS)

# La règle qui sera exécutée si on lance make sans argument
all: $(FOLDERS)

# Cette règle va lancer make dans le répertoire webserver
# option -C de make
webserver:
	make -C webserver
```

Pour le serveur web, nous allons utiliser un makefile plus compliqué, mais qui nous rendra la tâche plus simple par la suite. Il fera usage des règles génériques, de l'inclusion de fichiers et de la génération automatique de dépendances.

Listing 2.2 – Fichier `Makefile` du serveur web

```
# Ces variables servent à préciser le compilateur que l'on veut utiliser
# ainsi que les paramètres de compilation
CC=gcc
LD=gcc
CFLAGS=-Wall -W -Werror
LDFLAGS=

# Le nom de l'exécutable à fabriquer
EXE=pawnee

# Les variables HEADERS, CFILES et OBJS vont contenir respectivement
# la listes des fichiers .h, .c et le nom des fichiers .o à fabriquer
# On utilise la directive particulière $(wildcard ...) qui permet
# de construire automatiquement une liste de fichiers
HEADERS=$(wildcard *.h)
CFILES=$(wildcard *.c)
# Cette construction de variable veut dire: remplacer la chaine ".c" par
# ".o" dans la variable CFILES
# Ceci nous permet de construire la liste des fichiers .o à fabriquer
OBJS=$(CFILES:.c=.o)

# Même utilisation que précédemment. On précise les règles qui
# ne fabrique pas de fichier du nom de leur cible
.PHONY: all clean mrproper

# La règle par défaut déclenche la fabrication de l'exécutable
# par dépendance
all: $(EXE)

# Cette règle permet de construire l'exécutable. Elle dépend des fichiers
# .o et effectue l'édition de lien. Rien de nouveau ici
$(EXE): $(OBJS)
        $(LD) $^ $(LDFLAGS) -o $@

# Cette règle permet de construire automatiquement les règles
# de compilation pour chacun des fichiers .c
# l'option -MM de gcc analyse un fichier .c et
# affiche sur sa sortie standard une règle compatible
# make pour le compiler.
# Ceci permet d'ajouter automatiquement les fichiers .h aux dépendances
# des fichiers .o à construire. Ainsi, en modifiant un fichier .h
# tous les fichiers .c qui incluent ce fichier sont recompilés
# Après votre première compilation, regardez le contenu du fichier
# makefile.dep généré afin de comprendre exactement de quoi il retourne.
makefile.dep: $(CFILES) $(HEADERS)
        $(CC) -MM $(CFILES) > $@

# Cette règle efface le fichier de dépendances et les fichiers .o
clean:
        $(RM) $(OBJS) makefile.dep

# Cette règle effectue la précédente et efface en plus l'exécutable
mrproper: clean
        $(RM) $(EXE)

# On inclue le fichier de dépendance qui va contenir les règles
# de construction des fichiers .o
```

```
# S'il n'existe pas, make invoque automatiquement la règle
# qui l'a pour cible
include makefile.dep
```

2.2.4 Test du `Makefile`

Créez le fichier `webserver/main.c` suivant et compilez le à l'aide du makefile que nous venons d'écrire.

Listing 2.3 – Premier fichier source

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  /* Arnold Robbins in the LJ of February '95, describing RCS */
  if (argc > 1 && strcmp(argv[1], "-advice") == 0) {
    printf("Don't Panic!\n");
    return 42;
  }
  printf("Need an advice?\n");
  return 0;
}
```

Le fichier compile et s'exécute correctement ? Vous avez terminé la première étape. Après avoir ajouté tous les fichiers à votre dépôt et avoir « commité » toutes les modifications, créez un « tag » `mise_en_place` à l'aide de la commande :

```
$ git tag mise_en_place
```

Cette commande a pour effet de donner un nom explicite à la référence du commit correspondant à la fin de ce chapitre. Ces tags serviront à l'évaluation, soyez stricts sur leur nommage et leur utilisation. N'attendez pas la fin du module pour demander de l'aide si vous ne parvenez pas à les mettre en place correctement.

Chapitre 3

Un premier serveur TCP

Dans cette deuxième étape, nous allons développer un serveur TCP chargé de répéter à un client qui s'y connecte tout ce que ce dernier lui envoie.

3.1 Le réseau en C

L'utilisation du réseau en C passe par l'utilisation de « sockets ». Une socket est un fichier [1] qui permet de fournir une abstraction aux protocoles réseaux, et en particulier celui qui nous intéresse, TCP.

Dans le cas du développement d'un serveur TCP, nous allons distinguer deux types de sockets :

1. la socket dite « serveur » : c'est elle qui nous permettra d'accepter des connexions ;
2. les sockets dites « clientes » : ces sockets permettent au serveur de communiquer avec les clients, c'est à dire de leur envoyer des données. Du point de vue de votre programme, cette socket est vue, et utilisée, à l'aide d'un descripteur de fichier. Vous pourrez donc utiliser les primitives `read` et `write` pour, respectivement, recevoir des données du client et envoyer des données au client. Comme vous pouvez le voir, une socket s'utilise comme n'importe quel autre fichier (fichier régulier, tube, ...) et vous savez donc déjà comment l'utiliser... à condition d'avoir le descripteur.

3.1.1 La socket serveur

La socket serveur est celle qui permet à votre processus de demander au système d'exploitation (et en particulier à sa pile réseau) d'accepter les connexions en TCP sur un port donné et de le notifier. Pour cela, le système a besoin de connaître :

1. le protocole que le serveur veut utiliser : dans notre cas TCP ;
2. le port sur lequel il veut écouter.

Création de la socket La première étape se fait à la création de la socket. Cette création se fait à l'aide de la primitive :

```
int socket(int domain, int type, int protocol);
```

Cette fonction crée une socket et retourne un descripteur vers la socket créée ou -1 en cas d'erreur. Les trois paramètres de la fonction permettent de préciser :

— le domaine de la socket. On précise ici si l'on souhaite une socket utilisant IPv4 (`AF_INET`) ou IPv6 (`AF_INET6`) par exemple ;

— le type de la socket. C'est ici que l'on précisera si l'on veut utiliser TCP (`SOCK_STREAM`), UDP (`SOCK_DGRAM`) ou encore avoir un accès complet au protocole réseau (`SOCK_RAW`) [2]

1. rappelez vous, tout est fichier sous unix
2. Pour ce dernier cas, il faut être !root ! sur la machine

— le protocole spécifique à utiliser pour le type et le domaine souhaité. Généralement, un seul protocole est disponible et ce paramètre peut-être 0. Ce sera le cas pour TCP sur IPv4 et IPv6, TCP est le seul protocole permettant de fournir une socket de type `SOCK_STREAM` sur IPv4 ou IPv6.

Créer une socket serveur revient donc à utiliser le code suivant :

Listing 3.1 – Création de la socket serveur en IPv4

```
int socket_serveur;

socket_serveur = socket(AF_INET, SOCK_STREAM, 0);
if (socket_serveur == -1)
{
  perror("socket_serveur");
  /* traitement de l'erreur */
}
/* Utilisation de la socket serveur */
```

Configuration de la socket La deuxième étape se fait à l'aide de la primitive `bind`. Elle permet d'attacher la socket à une interface réseau et à un port particulier. Attention, seule une socket à la fois peut être attachée à une même adresse et un même port. Le prototype de la primitive `bind` est le suivant :

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Ici, les choses se compliquent un peu. Le premier paramètre est le descripteur de la socket que l'on veut attacher. C'est la valeur retournée par la fonction `socket` utilisée précédemment.

Le deuxième et le troisième paramètre permettent de donner les paramètres d'attachement : la/les interfaces à utiliser et le port sur lequel le serveur va accepter des connexions. Le deuxième paramètre est donc un pointeur vers une structure qui contient ces informations et le deuxième est la taille de cette structure (en octets).

En réalité, le type `struct sockaddr` n'est pas utilisable tel quel. Chaque type de socket possède sa propre version de cette structure. Ceci est nécessaire car chaque type de socket n'utilise pas nécessairement les mêmes informations, comme, par exemple, des adresses de taille différentes. En effet, IPv4 utilise des adresses de 4 octets et IPv6 de 16 octets.

Pour IPv4, le type à utiliser est `struct sockaddr_in`. Cette structure contient les champs suivants :

Listing 3.2 – Structure `sockaddr_in` (extrait de la page de man `ip(7)`)

```
struct sockaddr_in {
  sa_family_t    sin_family; /* address family: AF_INET */
  in_port_t      sin_port;   /* port in network byte order */
  struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
  uint32_t       s_addr;     /* address in network byte order */
};
```

On retrouve bien dans cette structure le port (`sin_port`) et l'adresse de l'interface sur laquelle on veut accepter des connexions (`sin_addr`). Le champ `sin_family` doit être affecté à la valeur `AF_INET`.

La page de manuel indique que le champ `sin_port` doit être donné dans le sens « réseau » des octets (*network byte order*). Un port TCP est sur deux octets et les protocoles réseaux utilisent

généralement [3] un encodage de type « *big endian* ». Il faudra donc affecter ce champ avec un entier au format big endian. Heureusement pour nous, la librairie système nous fournit une fonction permettant de faire la conversion, quelque soit l'architecture de la machine qui exécute le code. Cette fonction est `htons` : *Host to Network Short*.

Enfin, le champ `sin_addr` permet de préciser l'adresse de l'interface réseau sur laquelle on veut accepter des connexions. Si l'on souhaite accepter des connexions depuis toutes les interfaces de la machine (ce qui est souvent le cas), on peut utiliser la valeur particulière `INADDR_ANY`.

Le listing suivant donne un exemple d'utilisation de la fonction `bind` pour IPv4. Le descripteur `socket_serveur` aura été créé au préalable par le code du listing précédent.

Listing 3.3 – Attachement de la socket serveur sur toutes les interfaces

```
struct sockaddr_in saddr;
saddr.sin_family = AF_INET; /* Socket ipv4 */
saddr.sin_port = htons(8080); /* Port d'écoute */
saddr.sin_addr.s_addr = INADDR_ANY; /* écoute sur toutes les interfaces */

if (bind(socket_serveur, (struct sockaddr *)&saddr, sizeof(saddr)) == -1)
{
  perror("bind socker_serveur");
  /* traitement de l'erreur */
}
```

Vous remarquerez que nous devons effectuer un « cast » de l'adresse de la structure en `(struct sockaddr *)`. En effet, le C n'est pas un langage objet, la notion de « classe de base » n'existe pas et est simulée par l'emploi du type `struct sockaddr`. Le cast doit donc forcement être explicite.

Démarrer l'attente de connexions La dernière étape de préparation de la socket serveur consiste à signaler au système d'exploitation que le processus souhaite accepter des connexions (écouter) sur la socket que nous venons de configurer. Cette dernière étape se fait avec la fonction :

```
int listen(int sockfd, int backlog);
```

Le premier paramètre est toujours le descripteur de la socket serveur. Le deuxième paramètre est la taille de la file d'attente de connexions. Ceci indique au système le nombre de connexions qu'il peut mettre en attente d'acceptation de la connexion par le serveur. Cela ne correspond pas au nombre de connexions simultanées que peut gérer le serveur, mais uniquement au nombre de connexions non encore acceptées que le système accepte de mettre en attente avant de les rejeter. Le choix de cette valeur va dépendre de la charge du serveur, de sa capacité à gérer plusieurs clients simultanément ou d'autres paramètres. Pour notre utilisation, nous pouvons le fixer arbitrairement à 10.

Listing 3.4 – Lancer l'attente de connexion

```
if (listen(socket_serveur, 10) == -1)
{
  perror("listen socket_serveur");
  /* traitement d'erreur */
}
```

3.1.2 Les sockets clientes

Après avoir créé la socket serveur, le programme peut accepter des connexions. C'est en acceptant des connexions qu'il obtient des descripteurs vers les sockets des clients. Pour accepter une connexion, le serveur utilise la primitive :

3. C'est le cas pour TCP en particulier

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Le premier paramètre est toujours notre socket serveur. Le deuxième et le troisième paramètre permettent d'obtenir des informations sur le client qui se connecte. Ce sont des pointeurs vers une structure et un entier qui seront remplies par la fonction `accept` pour donner l'adresse IP du client par exemple. Dans un premier temps, nous pourrons passer `NULL` à la fonction pour ces deux paramètres.

La fonction retourne un descripteur qui correspond à la socket permettant de dialoguer avec le client ou -1 en cas d'erreur. Cette fonction est **bloquante**, c'est à dire que le programme serveur restera bloqué sur cette fonction tant qu'aucun client ne tentera de se connecter au serveur.

Listing 3.5 – Accepter une connexion

```
int socket_client;
socket_client = accept(socket_serveur, NULL, NULL);
if (socket_client == -1)
{
  perror("accept");
  /* traitement d'erreur */
}
/* On peut maintenant dialoguer avec le client */
const char *message_bienvenue = "Bonjour, bienvenue sur mon serveur\n";
write(socket_client, message_bienvenue, strlen(message_bienvenue));
```

3.2 Mise en pratique

Créez les fichiers `socket.c` et `socket.h`. Le fichier `socket.h` devra être le suivant :

Listing 3.6 – Fichier `socket.h`

```
#ifndef __SOCKET_H__
#define __SOCKET_H__
/** Crée une socket serveur qui écoute sur toute les interfaces IPv4
    de la machine sur le port passé en paramètre. La socket retournée
    doit pouvoir être utilisée directement par un appel à accept.

    La fonction retourne -1 en cas d'erreur ou le descripteur de la
    socket créée. */

int creer_serveur(int port);
#endif
```

Le fichier `socket.c` doit implémenter la fonction `creer_serveur` à l'aide des primitives `socket`, `bind` et `listen`.

Modifiez ensuite votre fichier `main.c` pour qu'il créé un serveur et accepte indéfiniment des connexions sur le port 8080.

À chaque connexion, le serveur envoie un **long (au moins 10 lignes)** message de bienvenue (soyez créatifs !) au client puis, tant que le client n'est pas déconnecté, lui répète les messages envoyés par le client.

3.2.1 Test de votre programme

Pour tester votre programme, vous pouvez utiliser les commandes `nc` ou `telnet`. Répondez aux questions suivantes dans un fichier reponses.txt qui devra être à la racine de votre dépôt :

1. Quittez votre serveur (`Ctrl+C`) et relancez le. Que se passe t'il ?

2. Ajouter un petit délai avant l'envoi du message de bienvenue (1 seconde). Puis, exécutez la commande suivante : `nc -z 127.0.0.1 8080`. Que se passe t'il ?
3. Exécutez la commande suivante : `telnet ::1 8080`. Que se passe t'il ? Pourquoi ?
4. Lancez deux clients simultanément. Que se passe t'il ? Pourquoi ?

Quand votre programme fonctionne, assurez vous d'avoir bien commité les dernières modifications et créez le tag `serveur_tcp_simple`. Cela marque la fin de ce chapitre.

Chapitre 4

Premières améliorations

En répondant aux questions du chapitre précédent, vous avez du identifier plusieurs problèmes :

1. une fois votre serveur arrêté, vous devez patientez quelques temps avant de pouvoir le redémarrer (l'appel à `bind` échoue).
2. votre serveur « plante » s'il envoie des données à un client déconnecté ;
3. votre serveur n'accepte pas de connexion sur une adresse IPv6 ;
4. votre serveur ne peut gérer qu'une connexion à la fois ;

Nous allons maintenant régler ces problèmes un par un, à l'exception de l'IPv6 qui vous pourrez traiter par vous même plus tard.

4.1 Options des sockets

Quand vous arrêtez votre serveur par un `Ctrl+C` alors que des clients étaient connectés à celui-ci, une nouvelle tentative de démarrage se solde par un échec de l'appel à la fonction `bind` avec le message : « Address already in use ». La raison à ce message est que, par défaut, même quand le processus qui a créé la socket est terminé, le système maintient la socket au cas où des paquets se seraient perdus et devraient être retransmis.

Ceci assure la fiabilité à TCP si un paquet s'est perdu après que le processus est arrêté, mais empêche de redémarrer un serveur pendant une durée de 30 à 120 secondes en fonction de la configuration des piles TCP/IP. Or, quand on développe un serveur on est souvent amené à vouloir le redémarrer (par exemple, changer son fichier de configuration et le redémarrer). Il existe une option permettant de modifier ce comportement de façon à ce qu'un processus qui demande de lier une socket sur une adresse encore utilisée puisse le faire, à la condition que le processus qui a créé la première socket soit terminé.

La fonction qui permet de modifier des options relatives au comportement des sockets est :

```
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

Cette fonction prend 5 paramètres :

1. le descripteur de la socket que l'on souhaite modifier ;
2. le niveau auquel s'applique l'option. Elle peut s'appliquer au niveau socket `SOL_SOCKET` ou à d'autre niveau de protocole comme `IPPROTO_TCP` par exemple. Dans le cas qui nous intéresse, c'est une option au niveau socket et on passera donc `SOL_SOCKET` comme valeur pour le paramètre `level` ;
3. le nom de l'option. L'option qui nous intéresse est `SO_REUSEADDR` qui permet à un processus de faire un `bind` sur une même adresse/port (sauf si un processus est déjà en `listen` dessus) ;

4. la valeur de l'option. Comme la fonction `setsockopt` est générique (plusieurs options possibles), ce paramètre est un `void *`. Dans la plupart des cas, on utilisera un entier (et donc un pointeur vers un entier valide) ;
5. la taille de l'option. Cette est également nécessaire pour apporter de la généricité à la fonction. Comme dans notre cas, on va passer un pointeur de type `int*` pour le paramètre `option_value`, on utilisera `sizeof(int)` pour ce paramètre.

Nous voulons donc activer l'option `SO_REUSEADDR`, il faudra donc passer comme valeur de l'option un pointeur vers un entier qui contient un nombre différent de 0.

L'appel à la fonction `setsockopt` doit être fait **avant** l'appel à `bind`.

Listing 4.1 – Activation de l'option `SO_REUSEADDR`

```
int optval = 1;

if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(int)) == -1)
   perror("Can not set SO_REUSEADDR option");
```

4.1.1 Mise en pratique

Modifiez votre fonction `creer_serveur` pour activer l'option `SO_REUSEADDR` sur la socket que vous créez.

Essayez à nouveau de lancer votre serveur, immédiatement après l'avoir terminé par `Ctrl+C`. Il doit normalement redémarrer sans erreur.

Une fois la modification effectuée, testée et commitée, créez un tag `REUSEADDR` sur votre dépôt git.

4.2 Gestion du signal `SIGPIPE`

Lorsque votre programme envoie un message de bienvenue suffisamment long et que le client se déconnecte pendant la transmission de ce message, votre serveur s'arrête. L'explication peut se trouver dans la page de manuel de la fonction `write` dont voici un extrait :

```
[...]
RETURN VALUE
[...]
       EPIPE  fd  is  connected to a pipe or socket whose reading end
              is closed.  When this happens the writing process  will
              also receive a SIGPIPE signal.  (Thus, the write return
              value is seen only if the program  catches,  blocks  or
              ignores this signal.)
```

En clair, lorsque votre programme effectue un appel à la fonction `write` alors que le fichier correspondant au descripteur fermé, le processus reçoit un signal.

Un signal est un événement envoyé par le système à un processus. Un processus qui reçoit un signal peut réagir de différentes façon :

1. l'ignorer ;
2. s'arrêter ;
3. interrompre le programme en cours, exécuter un traitement spécifique et revenir au programme.

Par défaut, la plupart des signaux entraînent l'arrêt du processus. C'est la raison de l'arrêt de votre serveur. Comme on ne peut pas savoir à l'avance quand un client va se déconnecter, il est impératif d'ignorer ce signal.

Deux fonctions permettent de contrôler le comportement d'un processus à la réception d'un signal : `signal` et `sigaction`. Pour les réglage fin de comportement (et en particulier l'exécution d'un traitement spécifique), la fonction à utiliser est `sigaction` [1]

Néanmoins, dans notre cas, comme on souhaite uniquement ignorer le signal, l'utilisation de `signal` et possible. Le prototype de la fonction `signal` est le suivant :

```
sighandler_t signal(int signum, sighandler_t handler);
```

Le premier paramètre correspond au numéro de signal (`SIGPIPE` pour nous) et le deuxième au comportement que le processus doit adopter à la réception du signal : `SIG_IGN` pour ignorer, `SIG_DFL` pour restaurer le comportement par défaut.

Listing 4.2 – Ignorer `SIGPIPE`

```
if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
{
  perror("signal");
}
```

4.2.1 Mise en pratique

Ajoutez une fonction

```
void initialiser_signaux(void);
```

qui s'assure que votre processus ignorera à présent le signal `SIGPIPE`.

Testez à nouveau l'utilisation de `nc` avec l'option `-z` pour confirmer que votre serveur ne s'arrête plus.

Une fois la modification effectuée, testée et commitée, créez un tag `SIGPIPE` sur votre dépôt git.

4.3 Gestion de plusieurs connexions simultanées

Pour gérer plusieurs connexion simultanément, le serveur doit pouvoir, en parallèle, :

1. attendre des connexions ;
2. lire les données envoyées par les clients connectés.

Le véritable problème réside dans le fait que les appels à `accept` et à `read` sont **bloquants**. Pour résoudre ce problème, une solution serait de créer un nouveau processus à chaque connexion d'un nouveau client.

Nous allons mettre en œuvre cette solution et voir si elle peut convenir.

4.3.1 Mise en pratique

Modifiez votre serveur pour qu'il crée un nouveau processus à chaque nouvelle connexion. Le nouveau processus devra être créé après l'appel à `accept` et ce dernier effectuera la boucle de traitement du client. Le processus père devra fermer le descripteur de la socket du client après avoir créé le processus fils.

Testez maintenant votre serveur avec plusieurs connexions simultanées, faites un commit des modifications et créez le tag `fork1`.

1. cf page de manuel de `signal` : « The `sigaction` function provides a more comprehensive and reliable mechanism for controlling signals ; new applications should use `sigaction()` rather than `signal()`. »

4.4 Processus zombies

Après avoir testé quelques connexions, affichez la liste des processus. Vous devez normalement voir apparaître autant de processus zombies que de connexions acceptées puis terminées.

Pour faire disparaître ces processus zombies, il faudra faire un appel à `wait` dans le processus père. Cependant, il faudrait qu'il le fasse entre chaque connexion. Une solution serait d'utiliser `waitpid` avec l'option `WNOHANG` mais on peut faire mieux.

A la mort d'un de ces fils, le processus père reçoit un signal `SIGCHLD`. Par défaut, le comportement du processus est d'ignorer ce signal. Il est possible de changer ce comportement par défaut pour, cette fois, effectuer un traitement particulier. Dans ce traitement particulier, on pourra effectuer l'appel à `waitpid`.

Pour cela, on utilise la fonction `sigaction`. Cette fonction permet d'attribuer un comportement spécifique à la réception d'un signal. Le prototype de la fonction est le suivant :

```
int sigaction(int sig, const struct sigaction *act,
          struct sigaction *oact);
```

Le premier paramètre est le signal pour lequel on veut effectuer un traitement particulier. Le deuxième paramètre est un pointeur vers une structure qui décrit le traitement. Le dernier paramètre permet de sauvegarder le comportement précédent dans le but de le restaurer plus tard si nécessaire.

Le comportement est décrit dans une structure qui contient trois champs[2] :

- `sa_handler` : Ce champ est un pointeur de fonction. Un pointeur de fonction est simplement un pointeur dans lequel on peut stocker l'adresse d'une fonction. Le pointeur peut alors être utilisé comme un appel de fonction ;
- `sa_mask` : un ensemble de signaux qui doivent être bloqués pendant le traitement du signal. Dans notre cas, nous utiliserons un ensemble vide qui peut être construit à l'aide de la fonction `sigemptyset` ;
- `sa_flags` : qui permet de régler des options relative au traitement du signal. On utilisera l'option `SA_RESTART` pour redémarrer automatiquement les fonctions qui auraient du être interrompue par la réception d'un signal (comme la fonction `accept`).

Le code suivant permet d'installer un traitement (d'appeler une fonction) à la réception du signal `SIGCHLD`.

Listing 4.3 – Utilisation de `sigaction`

```
void traitement_signal(int sig)
{
   printf("Signal %d reçu\n", sig);
}

void initialiser_signaux(void)
{
   struct sigaction sa;

   sa.sa_handler = traitement_signal;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = SA_RESTART;
   if (sigaction(SIGCHLD, &sa, NULL) == -1)
   {
      perror("sigaction(SIGCHLD)");
   }
}
```

2. En réalité quatre, mais le détail du quatrième champ sort du cadre de ce document. Vous pouvez consulter la page de manuel pour en savoir plus.

4.4.1 Mise en pratique

Modifiez votre programme pour qu'il effectue les appels nécessaires à la fonction `waitpid` à la réception du signal `SIGCHLD`.

Vérifiez que vous n'avez plus de processus zombies après les fermetures de connexions.

Après le commit de vos modifications, créez le tag `SIGCHLD`.

Chapitre 5

Un premier serveur web

Nous allons maintenant commencer à implémenter le protocole HTTP. Ce protocole est décrit de manière exhaustive dans les RFC 7230 à 7237. Pour des raisons de simplicité et de temps, nous n'allons gérer qu'une partie de ce protocole. Dans un premier temps, nous allons implémenter la réponse à une requête `GET` en ignorant les éventuelles entêtes envoyées par le client (le navigateur).

5.1 Récupération des lignes de la requête

Le protocole HTTP est essentiellement basé sur une séquence de lignes, séparées par le caractère `'\r'` [1] suivi du caractère `'\n'` [2] (notés `CRLF` dans la RFC).

Bien que le protocole TCP offre à l'application une vue en flux des données, il est tout de même basé sur un protocole paquet (IP). De ce fait, il se peut qu'une ligne se retrouve « coupée » et soit récupérée par l'application par deux appels successifs à la primitive `read` (voire plus pour une ligne très longue). Avant de traiter une ligne, votre serveur doit donc s'assurer qu'elle est complète. Cette opération n'est pas très difficile à mettre en place, mais la librairie C nous offre une primitive qui le fait déjà : `fgets`.

La fonction `fgets`, ainsi que toutes les fonctions `f*` (`fread, fwrite, fprintf...`) n'utilise pas directement un descripteur mais une structure de type `FILE *`. Cette structure contient le descripteur correspondant au fichier, mais également des tampons d'entrée et de sortie.

Pour obtenir un pointeur vers une structure `FILE *` à partir d'un descripteur de fichier, on utilise la fonction `fdopen`.

```
FILE *fdopen(int fd, const char *mode);
```

Comme le descripteur que l'on va utiliser correspond à la socket cliente, le mode devra être `"w+"` [3].

Le pointeur retourné pourra donc être utilisé avec les fonctions `fgets`, `fread`, `fwrite` et même `fprintf` pour un envoi de chaîne formattée au client.

5.1.1 Mise en pratique

Modifiez votre serveur pour utiliser les primitives `fgets` et `fprintf` pour, respectivement, lire les données envoyées par le client et lui retransmettre, en faisant précéder ces dernières par la chaîne `"<Pawnee> "`, dans laquelle vous remplacerez `Pawnee` par le nom de votre serveur.

Après modification et commit, créez le tag `fdopen` avant de passer à la suite.

1. CR : *carriage return*
2. LF : *line feed*
3. n'oubliez pas de lire la page de manuel.

5.2 Requête `GET`

5.2.1 Manipulations préliminaires

1. Modifiez votre programme pour qu'il ne transmette plus rien au client et qu'il affiche sur sa sortie standard les données envoyées par le client ;
2. Lancez la commande `$ curl http://localhost:8080/` [4]
3. Quel est le message envoyé par le client web ?
4. À l'aide de la RFC, trouvez le *nom* donné aux trois constituant de la première ligne de la requête envoyée par le client.

5.2.2 Première implémentation

Le premier message envoyé par un navigateur est donc de la forme :

```
GET / HTTP/1.1
User-Agent: curl/7.40.0
Host: localhost:8080
Accept: */*
```

Sur ces quatre lignes, seules deux sont obligatoires pour le procotole HTTP dans sa version 1.1 : la première (la requête à proprement parlé) et la troisième.

Nous allons modifier le serveur pour qu'il réponde correctement à une requête simple.

1. Modifiez votre serveur pour qu'il analyse la première ligne que lui envoie le client. Il doit vérifier que cette ligne commence par le mot `GET` et qu'elle contient exactement 3 mots ;
2. vérifiez que le troisième mot est bien de la forme `HTTP/M.m` ou `M` et `m` sont deux chiffres. Vérifiez que `M` vaut 1 et que `m` vaut 0 ou 1 ;
3. ignorez les lignes envoyées par le client tant qu'il ne s'agit pas d'une ligne vide (et donc que la chaîne de la requête soit `"\r\n"` ou `"\n"`) ;
4. si la requête ne respecte pas un des critères précédents, transmettez la réponse suivante (chaque ligne doit se terminer par `"\r\n"`) :

   ```
   HTTP/1.1 400 Bad Request
   Connection: close
   Content-Length: 17

   400 Bad request
   ```

   Vous pouvez tester cette partie de votre code en utilisant `nc` ou `telnet`.
5. si la requête respecte les critères, transmettez votre message de bienvenue. Vous devrez adapter le message précédent en changeant le code HTTP de `"HTTP/1.1 400 Bad Request"` à `"HTTP/1.1 200 OK"` et la taille du contenu.

Testez votre programme avec `curl` et un navigateur graphique. Après avoir effectué un commit de vos modifications, créez le tag `simple_get`.

5.2.3 Servir un seul contenu et générer une erreur 404

1. Modifiez votre serveur pour qu'il ne réponde qu'à une requête sur l'url `'/'`. Dans ce cas, il doit retourner votre message de bienvenue comme dans la section précédente.

4. si nécessaire, utilisez l'option `-noproxy` de curl

2. Si l'url est différente, votre serveur doit retourner une erreur 404. Inspirez vous de l'erreur 400 précédente pour générer correctement l'erreur 404.

Après avoir testé et commité, créez le tag `url_or_404`.

Chapitre 6

Mise au propre

Il est fort probable que pour répondre au demandes du chapitre précédent vous ayez tout implémenté quasiment dans une fonction unique (voire même votre main...). Afin de vous faciliter la tâche, voici quelques indication pour nettoyer un peu tout ça.

Si vous avez déjà fait des choses similaires, bravo, vous pouvez directement passer à la suite après avoir bien lu ce chapitre et vérifié que votre code est organisé aussi clairement que ce qui vous est proposé dans ce chapitre. Attention, ce chapitre donne des précisions sur le protocole HTTP (tirées directement de la RFC) qu'il est important d'avoir comprises [1].

6.1 Lecture des lignes envoyées par le client

Afin de faciliter la gestion d'erreur, nous allons écrire une fonction qui lit les données en provenance du client et quitte le processus si le client se déconnecte ou si une erreur survient.

La fonction à écrire est la suivante :

```
char *fgets_or_exit(char *buffer, int size, FILE *stream)
```

La fonction reprends le prototype de la fonction `fgets`. Elle fera donc appel à fgets. Si fgets détecte que le client est déconnecté, votre fonction devra quitter le processus par un appel à `exit`. Vous vous servirez désormais exclusivement de cette fonction pour lire les données du client. Vous n'aurez donc plus besoin de tester son retour, elle s'occupera de quitter le processus si nécessaire.

6.2 Analyse de la requête

Écrivez une fonction qui permet d'analyser la première ligne de la requête. D'après la RFC, cette première ligne est de la forme :

Listing 6.1 – Première ligne de la requête

```
request-line   = method SP request-target SP HTTP-version CRLF
```

— `method` doit être `GET` dans notre cas ;
— `request-target` correspond à l'url ;
— `SP` représente **un seul et unique** espace ;
— `HTTP-version` correspond dans notre cas à `HTTP/1.0` ou `HTTP/1.1` ;
— `CRLF` est la succession du caractère `'\r'` et du caractère `'\n'` ou uniquement le caractère `'n'`.

On propose d'utiliser le prototype suivant pour la fonction :

1. La description du protocole est donnée au format ABNF, utilisé fréquemment dans les RFC. Ce format est relativement simple à comprendre et est défini lui même par la RFC 5234.

```
int parse_http_request(const char *request_line, http_request *request);
```

Le premier paramêtre est la première ligne envoyée par le client. Le deuxième paramètre est un pointeur vers une structure que nous allons décrire par la suite et qui contiendra les informations relative à la requête. Elle sera donc modifiée par la fonction. La fonction doit retourner 0 si la requête est invalide et 1 si le format de la requête est correct.

La structure `http_request` est définie ainsi :

```
enum http_method {
   HTTP_GET,
   HTTP_UNSUPPORTED,
};

typedef struct
{
   enum http_method method;
   int  major_version;
   int  minor_version;
   char *url;
} http_request;
```

Vous pouvez remarquer que la structure utilise un type énuméré pour stocker un identifiant de la méthode employé. Ainsi, pour une évolution future du serveur, il suffira de rajouter des valeurs possibles [2] dans le type énuméré et de modifier la fonction d'analyse.

6.3 Analyse des entêtes

Une nouvelle lecture de la RFC nous apprend qu'un message HTTP doit être de la forme :

Listing 6.2 – Message HTTP

```
HTTP-message    = start-line
                  *( header-field CRLF )
                  CRLF
                  [ message-body ]

avec,

start-line      = request-line / status-line
```

Nous nous sommes déjà occupé de la `start-line` [3] avec la fonction précédente. Il s'agit maintenant de s'occuper de la succession de `header-field`, les entêtes.

Dans un premier temps, notre analyse des entêtes vont simplement consister à les ignorer.

Écrivez une fonction qui lit les lignes envoyées par le client tant que celles ci ne sont pas `"\r\n"` ou `"\n"`. Le prototype d'une telle fonction peut être :

```
void skip_headers(FILE *client);
```

Au retour de cette fonction, vous êtes prêts à envoyer la réponse attendue par le client.

6.4 Réponse au client

La réponse au client étant un message HTTP, elle doit répondre au format donné par le listing 6.2. La différence avec la requête du client réside dans la première ligne, qui n'est plus une

2. Pour les autres méthodes HTTP comme `POST` par exemple

3. La `start-line` est une `request-line` lorsque que le message vient du client et une `status-line` quand le message est émis par le serveur

`request-line` mais une `status-line`. C'est la première ligne qui indiquera si le serveur répond avec succès à la requête. Elle doit être de la forme :

Listing 6.3 – Première ligne de réponse

```
status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

- `HTTP-version` doit être du même format que précédemment ;
- `status-code` est un code à 3 chiffres indiquant comment le serveur réagit à la requête. Voici quelques codes HTTP avec la `reason-phrase` conseillée par la RFC :
  - 200 : OK
  - 400 : Bad Request
  - 403 : Forbidden
  - 404 : Not Found
  - 405 : Method Not Allowed
  - 500 : Internal Server Error
  - 501 : Not Implemented
  - 505 : HTTP Version Not Supported

Écrivez la fonction

```
void send_status(FILE *client, int code, const char *reason_phrase);
```

qui transmet au client une ligne de status correspondant aux paramètres passés à la fonction.

Pour compléter la réponse, après la ligne de status, le serveur doit envoyer un certain nombre d'entêtes. En particulier le serveur doit indiquer la taille du contenu qu'il transmet à l'aide de l'entête `Content-Length`. La valeur de l'entête doit être le nombre d'octet du message qui suivra (`message-body` dans le listing 6.2).

Écrivez la fonction

```
void send_response(FILE *client, int code, const char *reason_phrase,
                   const char *message_body);
```

Cette fonction doit transmettre une réponse complète au client et doit donc utiliser la fonction `send_status` pour transmettre la première ligne.

Les réponses du serveur du chapitre précédent peuvent alors être envoyées à l'aide des deux appels suivants (en supposant que `motd` pointe vers une chaîne de caractères contenant le message de bienvenue) :

```
if (bad_request)
   send_response(client, 400, "Bad Request", "Bad request\r\n");
else if (request.method == HTTP_UNSUPPORTED)
   send_response(client, 405, "Method Not Allowed", "Method Not Allowed\r\n");
else if (strcmp(request.url, "/") == 0)
   send_response(client, 200, "OK", motd);
else
   send_response(client, 404, "Not Found", "Not Found\r\n");
```

Une fois le nettoyage terminé et commité, créez le tag `url_or_404_clean`.

Chapitre 7

Servir du vrai contenu

Il est maintenant temps de servir un vrai site web et donc, des fichiers. Pour rappel, le format d'un message http est :

```
HTTP-message    = start-line
                  *( header-field CRLF )
                  CRLF
                  [ message-body ]

avec,

start-line      = request-line / status-line
```

Dans une réponse HTTP, le fichier est transmis dans la partie `message-body`. Il faut donc envoyer d'abord la ligne de status, les entêtes éventuelles et, enfin, le fichier en lui même.

La première remarque que l'on peut faire est qu'il faut donc vérifier que le fichier demandé par le client existe avant de commencer à répondre au client, ne serait-ce que pour répondre avec un 200, un 404 ou un 403.

7.1 Ouverture du fichier souhaité

Dans la requête HTTP, la cible de la requête (`request-target`, *c.f.* listing 6.1) est de la forme :

```
request-target = absolute-path [ "?" query ]
```

La partie `query` ne nous intéresse pas pour l'instant, nous allons donc modifier la valeur de l'url.

Écrivez la fonction

```
char *rewrite_url(char *url);
```

qui modifie la chaîne passée en paramètre de façon à supprimer tous les caractères suivants le caractère `'?'` (y compris celui ci) s'il est présent. La fonction doit retourner un pointeur vers le début de l'url modifiée. Ainsi l'url `"/fichier?action=afficher&format=verbatim"` doit devenir `"/fichier"`.

Une fois cette modification effectuée, on obtient un chemin vers un fichier. Bien que dénommé `absolute-path`, il est évident que ce chemin ne correspond pas à un chemin absolu dans l'arborescence du système de fichiers. Il est absolu dans l'arborescence du site servi mais relatif au répertoire dans lequel ce site est situé dans votre système de fichiers.

Modifiez votre programme pour qu'il accepte un paramètre qui sera le répertoire racine du site. Si le chemin donné en paramètre n'est pas un répertoire ou qu'il n'est pas accessible, votre serveur devra s'arrêter avec un message d'erreur approprié.

Avec un répertoire et un chemin relatif à ce répertoire (l'url), on peut maintenant tenter d'ouvrir le fichier souhaité.

Écrivez la fonction

```
int check_and_open(const char *url, const char *document_root);
```

qui prend en paramètre la cible de la requête ainsi que le répertoire racine du site à servir. La fonction doit s'assurer que le fichier est un fichier régulier, ouvrir le fichier en lecture seule et retourner un descripteur vers ce fichier à l'aide de la fonction `open`. La fonction doit retourner -1 en cas d'erreur et un descripteur valide en cas de succès.

7.2 Transmission du fichier

Si l'ouverture du fichier a échoué, il faut l'indiquer au client par une erreur 404. Cette partie ayant déjà été traitée dans le chapitre précédent, vous devriez la réaliser facilement.

S'il a été ouvert avec succès, il faut transmettre le fichier au client. Le fichier devra être transmis dans la partie `message-body` du message HTTP de réponse. Le format de cette partie est simple :

```
message-body = *OCTET
```

Il s'agit donc d'une suite d'octets. Cette suite d'octets n'est néanmoins pas nécessairement le contenu direct du fichier. Toujours d'après la RFC, il y a deux possibilités pour envoyer le contenu :

1. l'utilisation de l'entête `Transfert-Encoding` ;
2. l'utilisation de l'entête `Content-Length`.

la première possibilité (`Transfert-Encoding`) permet principalement d'envoyer des données sans connaître à l'avance le nombre d'octets à transmettre en utilisant un *« chunked encoding »*. Comme nous transmettons des fichiers **réguliers**, la taille est connue à l'avance. On utilisera donc plutôt la deuxième solution.

7.2.1 L'entête `Content-Length`

Pour indiquer au client la taille des données, il suffit d'inclure l'entête `Content-Length` dans le message de réponse. C'est cette technique que nous avons déjà utilisé précédemment. La valeur de cette entête est simplement la taille en octet (et en décimal) des données associées à la réponse.

1. Commencez par écrire une fonction `int get_file_size(int fd)` qui retourne la taille d'un fichier déjà ouvert à partir de son descripteur. Vous utiliserez la fonction `fstat` qui est le pendant de la fonction `stat` mais utilisant un descripteur au lieu d'un chemin ;
2. transmettez le message de status ainsi que l'entête `Content-Length` au client suivi d'une ligne vide (`"\r\n"`) ;
3. puis, écrivez une fonction `int copy(int in, int out);` qui copie un fichier depuis le descripteur `in` vers le descripteur `out`. Cette fonction sera utilisée pour envoyer le fichier vers la socket ;

7.3 Test du programme

1. Créez un site minimaliste (une ou deux pages dont un fichier `index.html`, quelques images...) qui vous servira pour tester votre serveur ;

2. essayez d'accéder à votre site par l'url `http://localhost:8080/index.html`. Que se passe-t'il ? Lisez le paragraphe 3.1.1.5 de la RFC 7231 et trouvez un moyen de corriger le problème. Un coup d'oeil au fichier `/etc/mime.types` pourrait vous être utile ;
3. essayez maintenant d'accéder à la racine de votre site. Que ce passe-t'il ? Modifiez la fonction `rewrite_url` de façon à transformer l'url `"/"` en `"/index.html"` ;
4. enfin, essayez d'accéder à l'url `http://localhost:8080/../../../../../../../etc/passwd` en utilisant `nc` [1]. Que ce passe-t'il ? En quoi est-ce un problème ? Corriger votre programme pour qu'il transmette une erreur 403 dans ce cas.

Après avoir terminé toutes ces modifications et vérifié que tout fonctionne, créez le tag `static_site`. Bravo, vous avez maintenant un serveur web minimaliste, mais qui fonctionne.

1. curl ou firefox corrigeront l'url et ne vous permettront pas d'observer le résultat correct

Chapitre 8

Mise en place de statistiques

Nous allons maintenant permettre à notre serveur de collecter des statistiques sur les requêtes traitées.

Il va donc falloir :

1. compter le nombre de requêtes traitées, à la fois globalement, et par réponse générée par le serveur ;
2. donner ces statistiques en réponse à un client sur une URL particulière.

Avant de nous intéresser aux statistiques, nous allons faire une première ébauche de réponse au client.

8.1 Réponse dynamique à une URL particulière

Modifiez votre serveur pour qu'il appelle une fonction

```
void send_stats(FILE *client);
```

lorsque l'URL de la requête est `/stats`.

La fonction doit alors envoyer une réponse HTTP valide avec, pour l'instant, un contenu de votre choix. Vous pouvez transmettre ce contenu en tant que simple texte ou comme une page html. Adaptez le type mime transmis au contenu que vous générez.

Cette fonction servira par la suite à transmettre les statistiques au client.

8.2 Collecte de statistiques

8.2.1 Une première version naïve

Dans un fichier `stats.h` définissez la structure suivante :

```
typedef struct
{
   int served_connections;
   int served_requests;
   int ok_200;
   int ko_400;
   int ko_403;
   int ko_404;
} web_stats;
```

Cette structure nous permettra de collecter les statistiques. Chaque entier sera incrémenté en fonction des traitements des connexions et des requêtes.

Écrivez ensuite deux fonctions dans un fichier `stats.c` :

```
int init_stats(void);
web_stats *get_stats(void);
```

La première devra initialiser une variable de type `web_stats` en mettant tous les champs à 0. Cette variable devra être une variable globale, mais dont l'accès au nom doit être limité au fichier `stats.c`. Pour cela, nous utilisons l'attribut `static`. Ainsi, la déclaration de cette variable globale sera fera de la façon suivante dans le fichier `stats.c` :

```
static web_stats stats;
```

La deuxième fonction doit simplement retourner un pointeur vers cette variable globale. Ainsi, lorsque l'on voudra accéder aux statistiques pour les consulter ou les modifier, il suffira d'appeler la fonction `get_stats` pour obtenir le pointeur vers la variable globale.

L'utilisation d'une fonction, plutôt que l'accès direct à la variable globale, est un peu plus propre et, surtout, nous permettra de facilement modifier l'accès aux statistiques dans la version moins naïve qui suivra.

Modifiez maintenant votre serveur pour qu'il :

1. incrémente les champs en fonction du traitement qu'il effectue des requêtes ;
2. transmette les valeurs de ces champs lors d'un accès à l'url `/stats`.

Accédez à votre serveur avec un, puis deux navigateurs différents (ou le mode de navigation privé par exemple). Que constatez vous ? Pourquoi ?

8.3 Utilisation de mémoire partagée

8.3.1 Description du problème

Vous avez normalement constaté, lors du test précédent, que les différentes statistiques affichées par les clients ne sont pas du tout synchronisées. Ceci est logique car les processus fils, qui traitent les requêtes, ne partagent pas le même espace mémoire. En clair, notre variable globale `stats` est globale uniquement pour un processus, mais chaque processus dispose d'une copie différente de cette variable.

C'est le comportement normal de la création d'un processus. Le système créé une copie de l'espace mémoire du processus père pour créer le fils. Après l'appel à `fork()` c'est donc bien deux espaces mémoires différents qui vont contenir deux « versions » de la variable globale `stats`.

Pour comprendre en détail comment le système gère la mémoire à la création d'un processus, commençons par observer le comportement du programme suivant :

```
#include <stdio.h>
#include <unistd.h>

static int var;

int main(void)
{
   if (fork() == 0)
      printf("Adresse de var dans le fils: %p\n", &var);
   else
      printf("Adresse de var dans le père: %p\n", &var);

   return 0;
}
```

En exécutant ce programme, on obtient la sortie suivante :

```
Adresse de var dans le processus père: 0x6009dc
Adresse de var dans le processus fils: 0x6009dc
```

La valeur effective de l'adresse peut être différent si vous testez ce programme, mais ce sera toujours la même valeur qui sera affichée dans le processus fils **et** dans le processus père.

Les deux processus utilisent donc la même adresse pour accéder à la variable `var`. On pourrait alors supposer, comme les processus utilisent la même adresse, qu'il s'agit du même espace dans la mémoire.

Observons maintenant le comportement du programme suivant :

```
#include <stdio.h>
#include <unistd.h>

static int var;

int main(void)
{
   var = 3;
   if (fork() == 0)
   {
      var = 42;
      printf("Adresse de var dans le fils: %p, valeur: %d\n", &var, var);
   }
   else
   {
      sleep(1); /* attente d'une seconde
                   on s'assure que l'affichage se fait après l'affectation
                   var = 42 dans le fils
                */
      printf("Adresse de var dans le père: %p, valeur: %d\n", &var, var);
   }

   return 0;
}
```

La sortie de ce programme est la suivante :

```
Adresse de var dans le processus fils: 0x600a7c, valeur: 42
Adresse de var dans le processus père: 0x600a7c, valeur: 3
```

On voit donc bien que, même si elles sont visiblement à la même adresse, les deux processus utilisent une version différente de la variable `var`.

8.3.2 La mémoire virtuelle

L'explication à cela est que les adresses manipulées par le processeur dans les systèmes « modernes » n'est pas directement une adresse physique, c'est à dire une adresse qui désigne un emplacement mémoire dans la RAM réelle de la machine.

Le processeur manipule des adresses « virtuelles » qui sont ensuite traduites en adresses physiques. Cette traduction est configurée par le système d'exploitation, mais assurée par un composant matériel du micro-processeur, l'unité de gestion mémoire ou *« Memory Management Unit (MMU) »*.

La figure 8.1 donne une illustration de cette technique. Les données du processus père et du processus fils (et donc la variable `var`) sont vues par le processeur à la même adresse. Cependant, quand le système d'exploitation donne la main à l'un ou l'autre des processus, il configure la traduction d'adresses de façon à ce que cette même adresse virtuelle (ici `0x600a7c`) corresponde à l'adresse physique `0x800a7c` dans le processus père et `0xa00a7c` dans le processus fils.

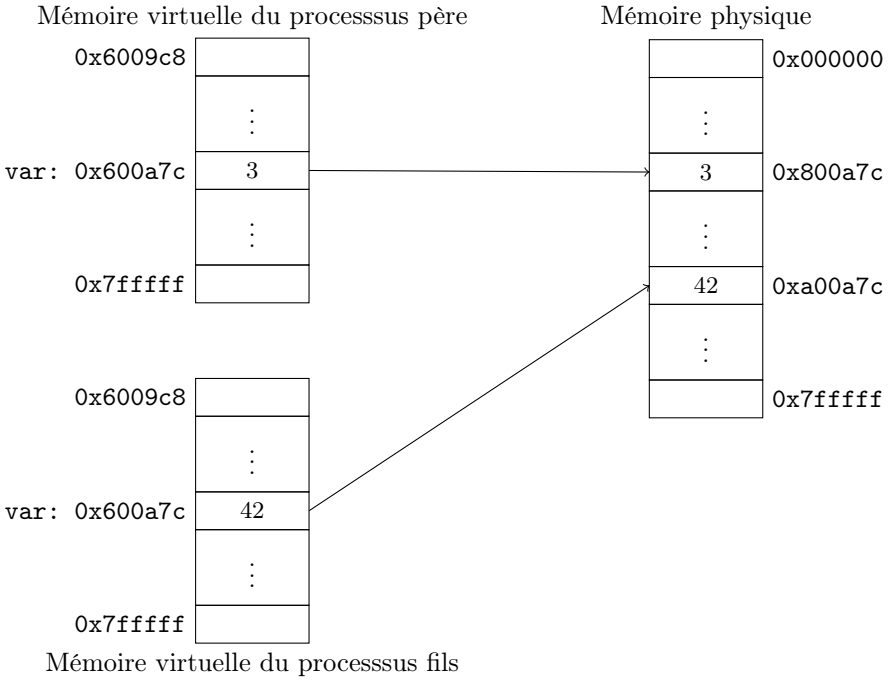

FIGURE 8.1 – Traduction d'adresse virtuelle vers adresse physique pour un processus père et son fils.

8.3.3 Partager de la mémoire entre les processus

Dans notre problème initial, nous voulons que la variable globale qui contient les statistiques du serveur soit partagée par l'ensemble des processus. C'est à dire que, si un processus modifie son contenu, cette modification doit être visible des autres processus.

Avec l'éclairage de la section précédente, cela signifie que, pour cette variable, nous devons faire correspondre l'adresse virtuelle de la variable globale de tous nos processus, vers la même adresse physique. De cette façon, les modifications apportées à la variable seront communes à tous les processus et nos statistiques seront enfin cohérentes. Ce que nous souhaitons obtenir est alors illustré par la figure 8.2.

Le noyau linux offre la possibilité (dans une certaine mesure) d'agir sur la transformation d'adresses virtuelles en adresses physiques. Pour être exact, on ne peut pas demander une traduction précise mais on peut demander explicitement à obtenir une zone mémoire qui sera partagée par les futurs processus fils [1]. À la création d'un processus fils, la traduction d'adresse sera conservée à l'identique et donc, les deux processus pourront accéder à la même zone de mémoire physique.

La primitive système permettant d'obtenir une zone de mémoire en précisant les propriétés de partage est :

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Cette fonction prends six paramètres et retourne un pointeur vers une zone mémoire qui respecte ce qui a été demandé à l'aide de ces paramètres :

— `addr` : l'adresse (virtuelle) que la fonction doit retourner. À utiliser si l'on souhaite obtenir une adresse particulière. Cependant, le noyau ne le considère que comme une contrainte faible et ne la respectera que s'il peut le faire. Il est conseillé de passer `NULL` pour ce paramètre et c'est alors le noyau qui décide lui même de l'adresse virtuelle à utiliser ;

1. ce qui est similaire à l'héritage de descripteurs de fichiers à la création d'un fils

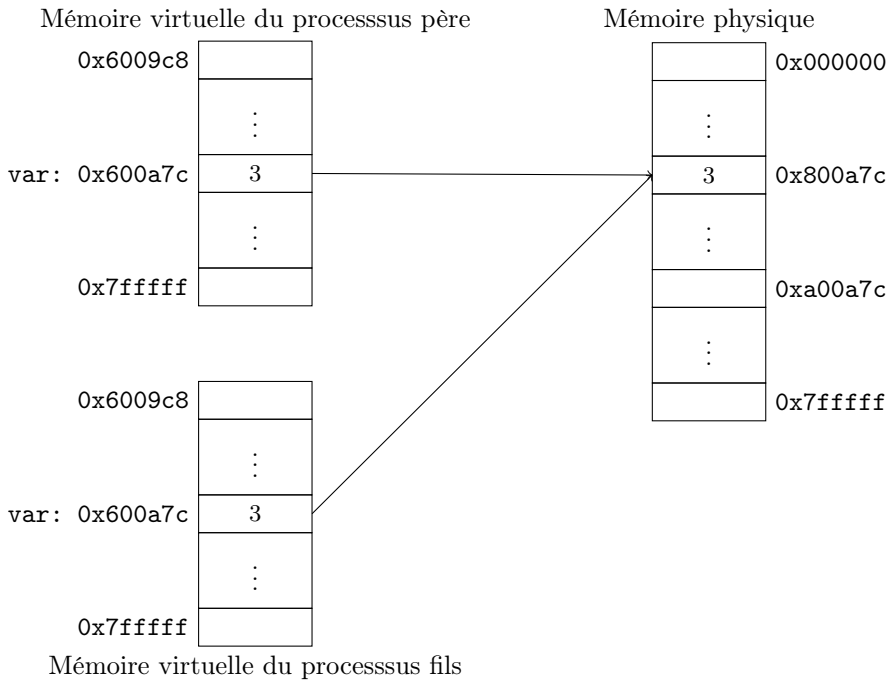

FIGURE 8.2 – Partage de mémoire entre un processus père et un processus fils.

— `length` : la taille de l'espace mémoire à réserver ;
— `prot` : permet, à l'aide des flags `PROT_EXEC`, `PROT_READ` et `PROT_WRITE` de contrôler les autorisations d'accès à la mémoire [2]. Par exemple, si on utilise la fonction avec le paramètre `PROT_READ` uniquement, une écriture dans la mémoire retournée se soldera par une erreur de segmentation [3] ;
— `flags` : ce paramètre permet de modifier les propriétés de la mémoire obtenue. Le paramètre doit inclure au minimum `MAP_SHARED` ou `MAP_PRIVATE`. La valeur qui nous intéresse est `MAP_SHARED` qui permet à plusieurs processus de partager l'espace mémoire. Ensuite, on peut préciser d'autres options. Celle qui va nous intéresser est l'option `MAP_ANONYMOUS` qui demande au noyau une zone mémoire « anonyme » [4]. Cette zone mémoire sera uniquement accessible par les processus fils par héritage. Nous allons devoir utiliser cette option ;
— `fd` : le descripteur d'un fichier à placer en mémoire. En réalité, la fonction première de la primitive `mmap` est de « placer » un fichier en mémoire. Plutôt que d'accéder à un fichier avec `read` ou `write`, `mmap` permet d'obtenir une adresse à laquelle un processus peut voir le fichier comme s'il s'agissait d'un tableau. Le système gère automatiquement le transfert des données du disque à la mémoire. C'est une fonction très utile pour traiter de gros fichiers. Néanmoins, ce n'est pas l'utilisation que nous allons en faire (à cause du `MAP_ANONYMOUS`), et on peut donc mettre `-1` pour ce paramètre.
— `offset` : dans le cas du placement d'un fichier en mémoire, ce paramètre permet de débuter le placement, non pas au début du fichier, mais après un nombre donné d'octets. Comme on ne se sert pas de `mmap` dans ce but, on peut passer 0 pour ce paramètre.

2. Ces protections peuvent être combinées à l'aide de l'opérateur `'|'`.
3. Attention cependant, l'architecture matérielle limite les possibilités de combinaisons des options. Par exemple, sur un processeur intel, la mémoire qui peut être écrite peut forcement être lue. Ainsi, `PROT_WRITE` est équivalent à `PROT_READ | PROT_WRITE`.
4. Cette fonctionnalité n'est pas strictement POSIX, c'est à dire que sur d'autre système qu'un linux, elle peut ne pas exister. Dans ce cas, l'établissement de mémoire partagée est plus complexe et il faut utiliser les « shared memory object » qui sont des fichiers particuliers qui représentent de la mémoire partagée.

Ainsi, dans l'utilisation que nous allons en faire, cette primitive est assez similaire à `malloc`[5].

8.3.4 Mise en pratique

Vous pouvez maintenant modifier vos fonctions `init_stats`, pour qu'elle initialise de la mémoire partagée à l'aide de la fonction `mmap`, et `get_stats` pour l'utilisation dans votre serveur. Vous devez donc appeler `mmap` pour créer une zone mémoire **partagée, anonyme** et accessible **lecture et écriture**.

Testez à nouveau votre serveur et observez l'évolution des statistiques. Vous devez observer que les statistiques sont maintenant partagées entre les processus. Il reste encore un problème sur ce partage, mais nous le traiterons dans le chapitre suivant.

Après avoir testé vos modifications, créez le tag `shared_memory` sur votre dépôt git.

8.4 Mémoire partagée et accès concurrents

Un dernier problème subsiste avec nos statistiques. Pour le mettre en lumière, nous allons utiliser un outil de test de serveur web : `siege`. Cet outil est une commande qui génère des requêtes http à un serveur. L'intérêt de cet outil est qu'il permet de générer ces requêtes de façon concurrente (*i.e.* en parallèle) et donc de « stresser » votre serveur.

8.4.1 Utilisation de l'outil `siege`

Lancez votre serveur et surtout **n'accédez à aucune page avec un navigateur** puis exécutez la commande suivante :

```
$ siege -c 800 http://localhost:8080
```

cette commande simule 800 requêtes simultanées, attends une seconde, puis relance 800 requêtes et ainsi de suite. Laissez tourner le programme quelques secondes (10 ou 20 tout au plus) et arrêtez le à l'aide de `Ctrl+C`.

`siege` s'arrête alors et affiche un message de ce type :

```
HTTP/1.1 200   0.00 secs:    4595 bytes ==> GET  /
HTTP/1.1 200   0.00 secs:    4595 bytes ==> GET  /
^C
Lifting the server siege...      done.

Transactions:                  19861 hits
Availability:                 100.00 %
Elapsed time:                  13.36 secs
Data transferred:              87.03 MB
Response time:                  0.04 secs
Transaction rate:            1486.60 trans/sec
Throughput:                     6.51 MB/sec
Concurrency:                   63.80
Successful transactions:       19861
Failed transactions:               0
Longest transaction:            7.05
Shortest transaction:           0.00
```

5. d'ailleurs, l'implémentation de `malloc` de la glibc utilise `mmap` quand elle doit faire des allocations de taille importante.

Ce rapport nous donne des informations sur la performance de notre serveur (comme le nombre de transactions traitées par seconde, le taux de transfert, la disponibilité, etc.) ainsi que le nombre de transactions correctes (réponse http 200).

Accédez à la page qui affiche les statistiques de votre serveur. Normalement, on s'attend à ce que le nombre de requêtes auxquelles votre serveur à répondu avec un code 200 doit correspondre au nombre de requête traité par `siege` plus 1 qui correspond à l'accès à la page de statistique (si votre serveur la compte).

Normalement, vous devez avoir un écart plus important, de l'ordre d'une dizaine de transaction d'écart. Par exemple, l'implémentation de référence que nous avons développée indique 19848 réponses avec un code 200 contre les 19862 attendues (19861 faites par `siege` et une pour l'accès à la page de statistiques en elle même). L'écart est faible, mais néanmoins présent !

8.4.2 Analyse du problème

Pour comprendre ce qu'il s'est passé, il faut se demander ce qu'il se passe vraiment quand un processus exécute le code suivant :

```
stats->ok_200 = stats->ok_200 + 1;
```

Voici le code compilé en assembleur d'un tel programme :

```
1 mov    0x6008c0,%rcx ; on charge dans rcx l'adresse contenue dans stats
2 mov    (%rcx),%edx    ; on charge la valeur du champ ok_200 dans edx
3 add    $0x1,%edx     ; on incrémente edx
4 mov    0x6008c0,%rcx ; on charge dans rcx l'adresse contenue dans stats
5 mov    %edx,(%rcx)   ; on charge dans le champ ok_200 la nouvelle valeur
```

Ce qu'on peut voir ici, même si l'on est pas à l'aise avec l'assembleur, est qu'une ligne de C est en réalité traduite par plusieurs lignes d'assembleur, et donc plusieurs instructions qui seront exécutées par le micro processeur.

Dans notre cas, l'adresse `0x6008c0` de l'exemple est **partagée** par plusieurs processus. Supposons que notre champ `ok_200` contienne la valeur 8. Un premier processus exécute les instructions des lignes 1 et 2. Il a alors la valeur 8 dans son registre `edx`. Le système décide à ce moment précis de stopper l'exécution du processus et de donner la main à un deuxième processus.

Ce deuxième processus a plus de chance et va exécuter les 5 lignes d'un coup. Il charge donc également la valeur 8 dans son registre `edx`, l'incrémente et la remet en mémoire à l'adresse `0x6008c0`. La valeur stockée en mémoire est alors 9. Le système décide alors de rendre la main au premier processus.

Ce premier processus avait éxécuté les lignes 1 et 2. Il reprend donc son exécution à la ligne 3. Il incrémente son registre `edx` (qui lui n'est pas partagé entre les processus) qui vallait 8. Le registre passe donc à 9. Puis, les lignes 4 et 5 sont exécutée et on vient ranger la valeur 9 dans la mémoire à l'adresse `0x6008c0`.

À première vue, tout peut sembler normal. Seulement, deux processus (qui partagent la même zone mémoire) ont voulu incrémenter la variable. Comme sa valeur initiale était 8, on devrait avoir $8 + 2 = 10$ à la fin de l'exécution des processus. Or, on vient de montrer qu'il était possible de trouver un ordonancement particulier de ces deux processus qui mène à la valeur 9 stockée en mémoire au lieu de la valeur 10.

Le problème est donc lié au fait que les deux processus peuvent exécuter l'incrémentation « en même temps ». Pour résoudre le problème, il nous faut donc une solution pour s'assurer qu'un seul processus peut exécuter l'incrémentation complète (les 5 instructions) et que les autres doivent attendre qu'il ait fini pour pouvoir, à leur tour, exécuter cette section de code. Une telle section d'un programme est appellée « section critique » et le système nous fourni des outils pour s'assurer qu'un seul processus peut rentrer en section critique.

8.4.3 Les sémaphores

L'outil principal à notre disposition est le **sémaphore**, inventé par Edsger Dijkstra. Un sémaphore est un type de variable particulier pour lequel on dispose de trois opérations principales : `Init`, `P` et `V`[6].

1. `Init` : initialise le sémaphore en lui donnant une valeur entière ;
2. `P` : essaye de décrémenter le sémaphore. Si la valeur du sémaphore est nulle, le processus qui utilise cette opération est bloqué jusqu'à ce que la valeur du sémaphore devienne positive ;
3. `V` : incrémente la valeur du sémaphore.

Les opérations `P` et `V` **doivent** être des opération **atomiques**, c'est à dire qu'un processus qui commence l'opération ne doit pas pouvoir être interrompu tant que l'opération n'est pas terminée. C'est donc le système d'exploitation qui doit fournir cette opération et il le fait soit en utilisant des instructions dédiées soit simplement en coupant le mécanisme d'interruption matérielle de façon temporaire.

Pour protéger notre section critique (la manipulation des statistiques), nous allons utiliser un sémaphore partagé par tous les processus. Comme on souhaite qu'un seul processus entre en section critique à la fois, on doit implémenter **l'exclusion mutuelle**.

Il suffit pour cela d'initialiser le sémaphore avec la valeur 1. Ainsi, dès qu'un processus aura exécuter l'opération `P`, si d'autres tentent l'opération, il seront bloqués tant que le premier n'aura pas utilisé l'opération `V`. Le pseudo code de la gestion des statistiques devient donc :

```
/* A l'initialisation des statistiques */
Init(1);

/* ... */

/* A la modification des statistiques */
P();
modifier_statistiques();
V();
```

L'utilisation de sémaphore paraît donc très simple pour effectuer de l'exclusion mutuelle. Il faut cependant faire **très attention** quand on utilise des sémaphores pour éviter ce que l'on appelle une situation d'interblocage. Les deux situations à éviter sont principalement :

1. deux processus se bloquent mutuellement : le processus P1 a pris un sémaphore S1 et tente de prendre également un sémaphore S2. Si un processus P2 a pris le sémaphore S2 et tente de prendre le sémaphore S1, les deux processus sont bloqués, sans espoir de se libérer ;
2. un processus prends un sémaphore S et oublie de le libérer avant d'essayer de le reprendre. Il se bloque alors lui même, indéfiniment.

Dans notre cas, le seul risque est le deuxième et il faudra donc bien faire attention à libérer le sémaphore après avoir modifié les statistiques.

8.4.4 Les sémaphores POSIX

Dans un système POSIX (comme Linux), les sémaphores sont fournis par le type `sem_t` et les opérations `Init`, `P` et `V` par les fonctions `sem_init`, `sem_wait` et `sem_post`. La page de manuel `sem_overview` donne des détails sur les sémaphores POSIX.

8.4.5 Mise en pratique

Vous allez maintenant modifier votre programme pour qu'il utilise un sémaphore partagé par les processus pour protéger la modification des statistiques. Vous êtes libres de choisir le bon moyen d'utiliser le sémaphore dans votre programme. Quelques indications :

6. P et V sont les initiales de *Proberen* et *Verhogen*, respectivement *essayer* et *augmenter* en néerlandais.

1. le sémaphore doit être partagé par les processus. La variable de type `sem_t` devra donc être dans la zone de mémoire partagée que nous avons mise en place dans la section précédente ;
2. il doit être initialisé au même moment que l'initialisation de la mémoire partagée, et donc dans le père.

Après avoir effectué la modification, procédez à nouveau au test avec `siege` et vérifiez que le résultats sont maintenant cohérents.

Enfin, créez un tag `semaphore` dans votre dépôt git.

Chapitre 9

Mot de la fin

Si vous êtes arrivés au bout, bravo vous avez réussi à implémenter un serveur web minimaliste. Pour ceux d'entre vous qui ont envie de continuer, voici quelques idées :

1. permettre l'utilisation d'un fichier de configuration pour paramétrer le serveur : port, adresse d'écoute, répertoire racine du site web, types mimes supportés... ;
2. générer des fichiers de logs, par exemple à la manière d'apache un fichier qui liste les requêtes avec la date, l'heure, l'adresse IP du client, l'url de la requête et un autre fichier qui contiendra les erreurs ;
3. gérer la notion de `VirtualHost` d'apache. Il faut dans ce cas gérer l'entête `Host` transmise par le client dans une requête `HTTP/1.1`. Le document racine change alors en fonction de la valeur de l'entête `Host`. Ceci permet à votre serveur de gérer plusieurs site ;
4. implémenter la norme CGI qui permet d'exécuter une commande externe, comme par exemple l'interpréteur PHP, pour répondre à une requête ;
5. gérer la norme HTTP complètement (lisez la RFC).

Ce ne sont que quelques idées, un coup d'œil à la documentation d'Apache ou de NGINX peut vous donner d'autres pistes qui pourront vous occuper pendant plusieurs années !