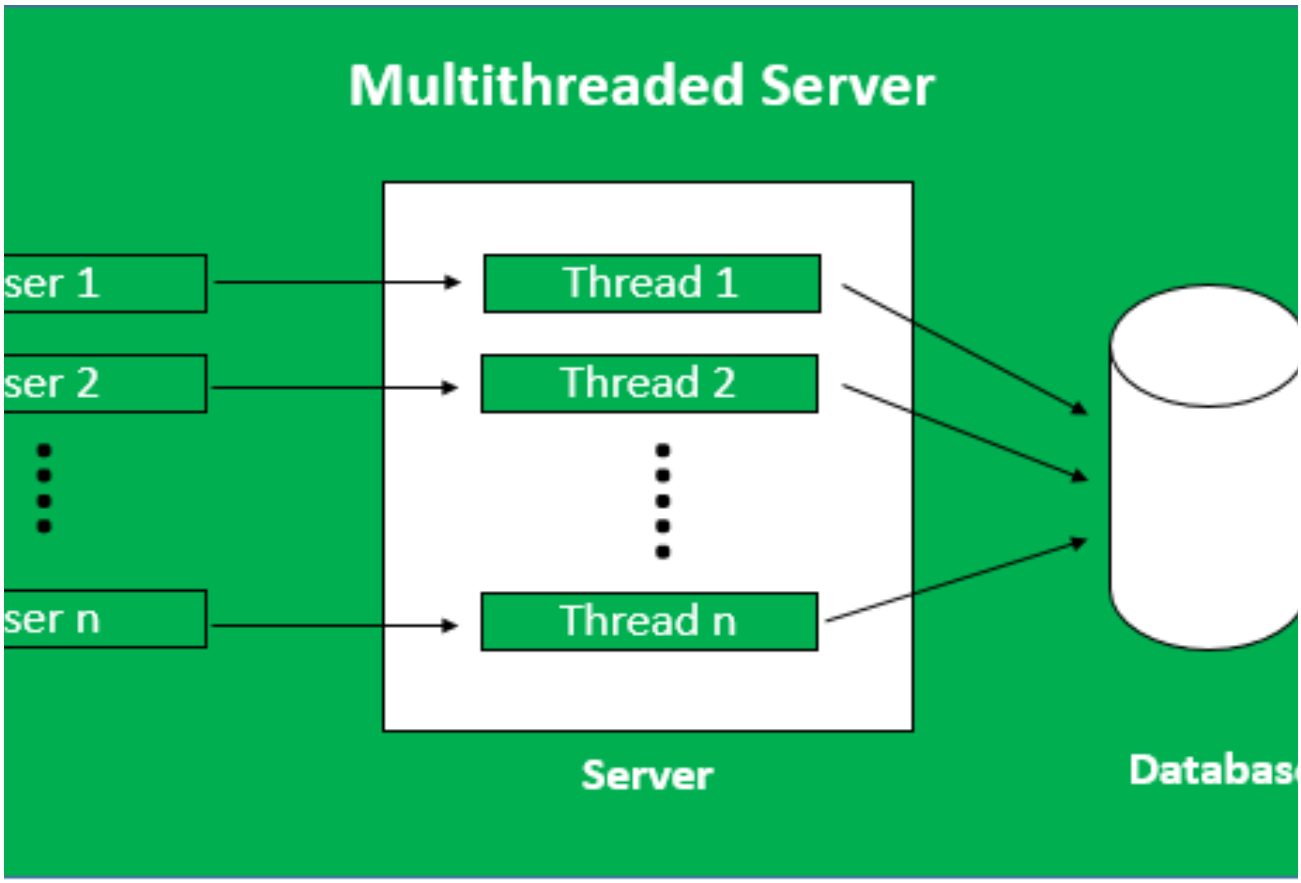
RAPPORT DU PROJET SYSTÈME D'EXPLOITATION

MISE EN PLACE D'UN SERVEUR WEB MULTITHREAD



Bakayoko Olivia
Achraf Jenzri

09/12/2021

UE SYSTÈME D'EXPLOITATION – CRÉNEAU A

SOMMAIRE

Objectifs et rappel de l'existant

Après avoir suivi un cours sur les systèmes d'exploitation, nous sommes invités au cours d'un projet à mettre en pratique nos connaissances en réalisant ce projet dénommé : "A concurrent web server" disponible sur un dossier partagé par les encadrants. Il consiste à réaliser un serveur Web Multithreader et rédiger un rapport sur nos avancements. Ce présent rapport explique donc les différentes étapes de la réalisation de notre projet ainsi que les méthodologies et techniques utilisées. De prime abord nous rappellerons les informations sur le code existant. Ensuite nous présenterons les fonctionnalités que nous avons réalisées. Enfin nous aborderons la phase pratique en présentant notre proof of concept puis les différentes interprétations qu'on en a tirées. Le code source de base est constitué d'un web serveur basique contenant plusieurs fonctions dont nous mentionnerons que celle qui nous interresse dans ce projet :

- wserver.c contient la fonction main() pour le serveur Web et la boucle de service de base.
- request.c : effectue la plupart des travaux de traitement des demandes dans le serveur Web à commencer par request_handle().
- wclient.c : permet d'effectuer les requêtes des clients
- Makefile : pour pouvoir compiler tous les fichiers d'un trait.

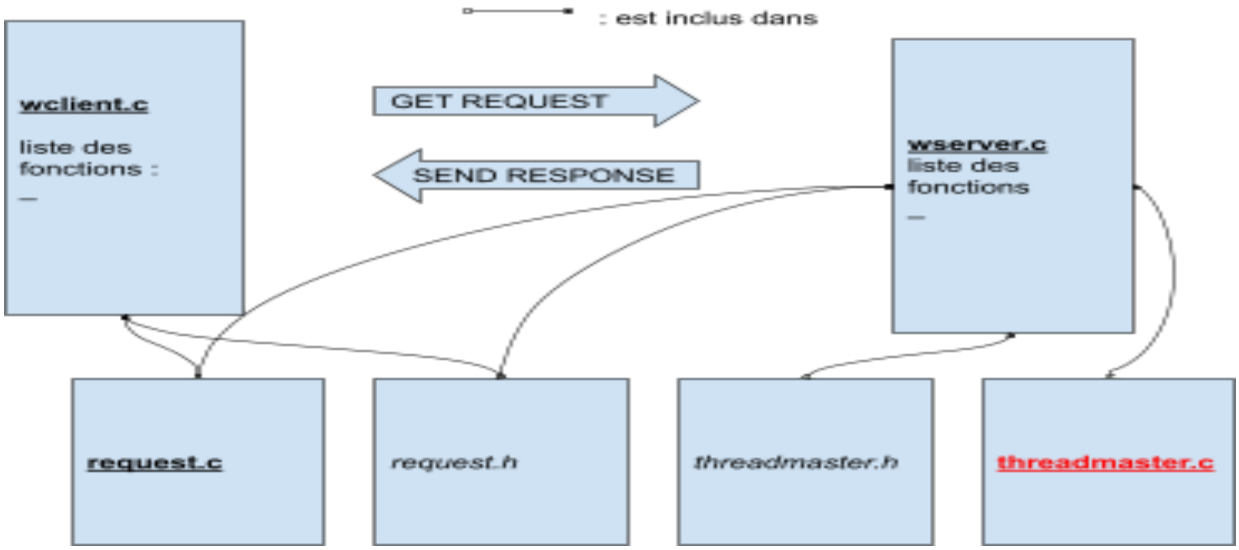
Ce serveur est simple de fonctionnement, chaque requête est gérée au fur et à mesure et une requête à la fois ainsi le serveur est bloqué lorsqu'il reçoit plusieurs requêtes en même temps.

D'où le but pour nous de réaliser un serveur multithread, en effet dans la vraie vie les serveurs HTTP reçoivent plusieurs requêtes simultanées auxquelles, il doit répondre de

façon simultanée et selon un ordonnancement au risque qu'un seul client ne bouffe toutes les requêtes.

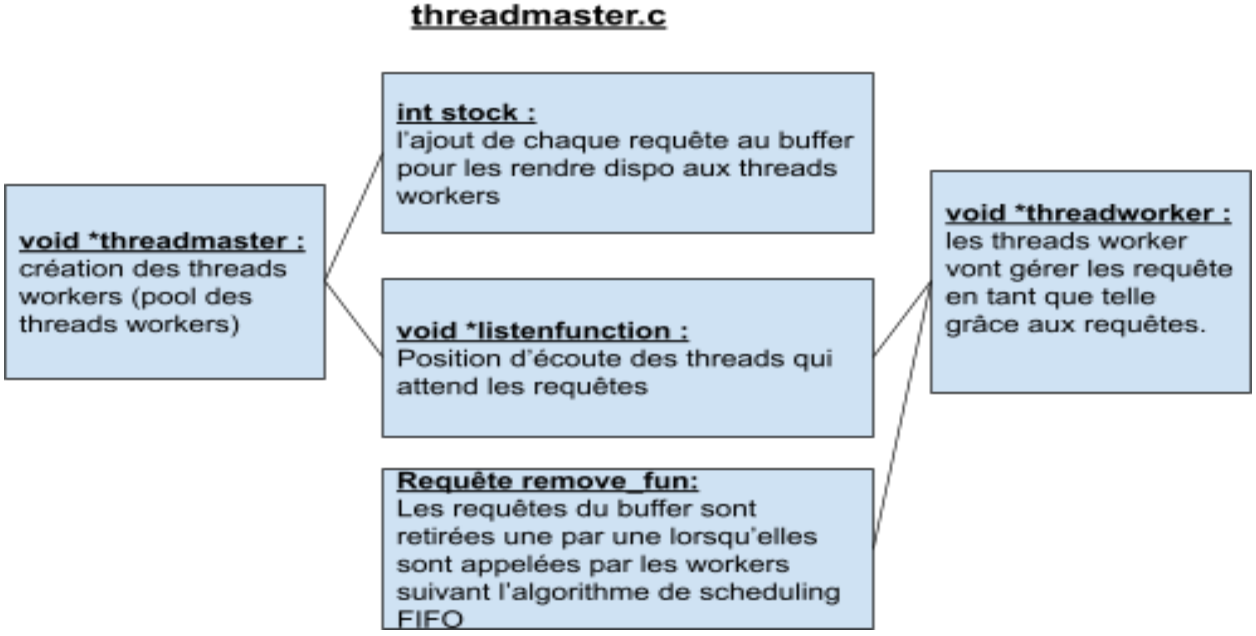
Diagramme des différents composants du code avec explication

le code que nous avons réalisé contient les différentes dépendances sur la figure suivante :



Overview du programme

L'amélioration majeure que nous avons apportée se trouve dans le fichier threadmaster.c dont les fonctions sont définies sur le diagramme suivant.



En effet dans ce fichier sont définit deux fonctions très importantes :

- **threadmaster :** qui a pour rôle de créer les threads workers, c'est-à-dire les threads ayant le rôle de consommateurs selon le modèle de 1 producteur pour N consommateurs. Ainsi le threadmaster ou le producteur va stocker les requêtes dans le buffer de requête pour les mettre à disposition des workers.
- **threadworker :** ceux ci aussi écoute afin de se préparer à se lancer en concurrence pour avoir accès à la ressource : le buffer de requête. L'algorithme d'ordonnancement que nous avons choisi est ***FIFO*** (first in first out). Ainsi chaque thread worker gérera une requête à la fois qu'il aura gagné dans la concurrence.

les autres fonctions utilisés sont :

- ***fonction listen_function :*** qui fait l'écoute sur le port choisi par l'utilisateur
- ***fonction remove_fun*** : qui fait l'extraction des requêtes de notre buffer pour les donner aux thread workers.
- ***fonction stock :*** qui sert à ajouter les requêtes dans notre buffer pour les rendre accessibles aux thread workers

Pour aussi s'assurer d'avoir une bonne concurrence des locks et conditions ont été ajoutés sur les variables critiques comme le buffer de requête.

Pseudo-code des différents algo essentielles de l'application

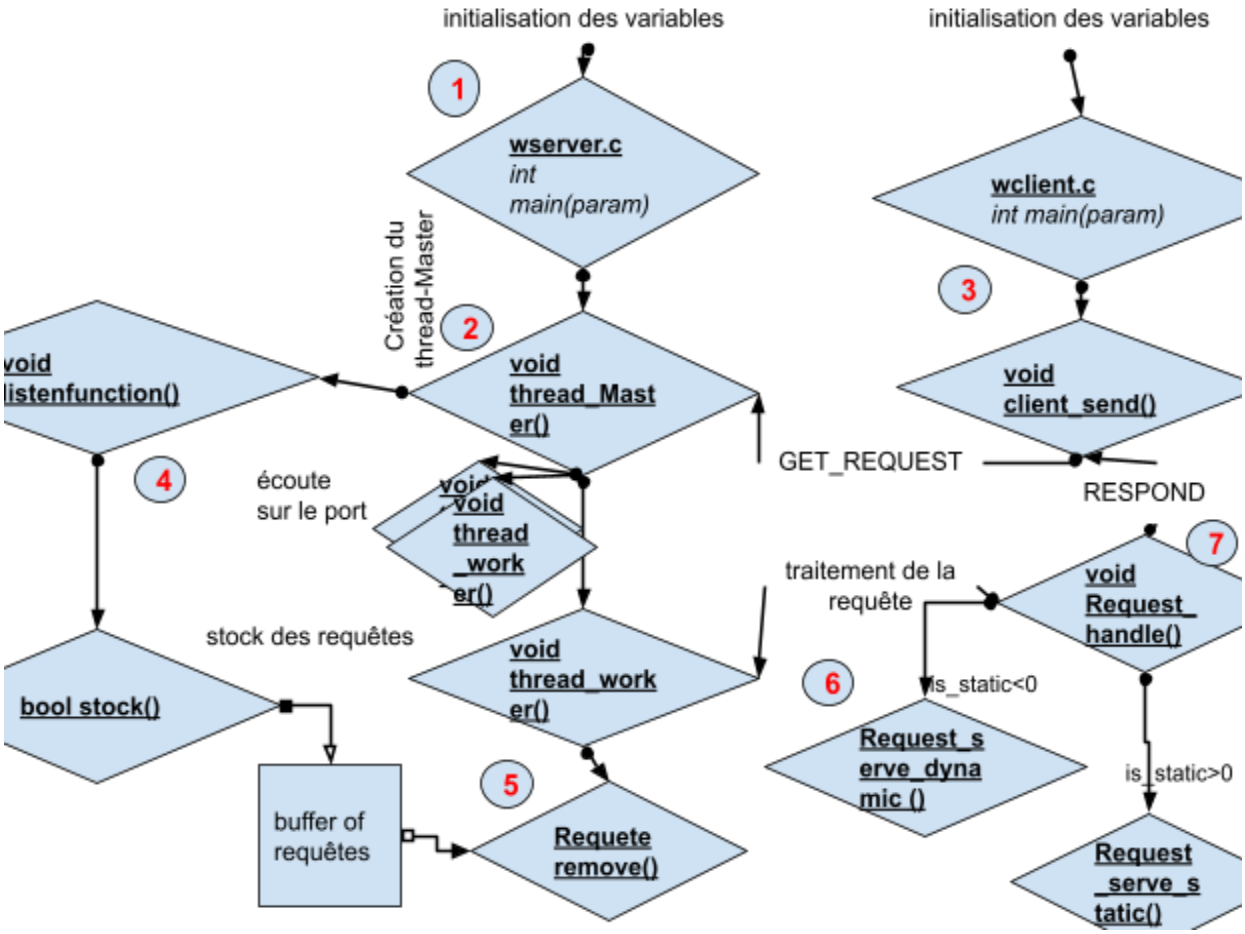
Le code peut être subdivisé en deux parties principales : le client et le serveur.
Concernant notre cas, nous avons choisi délibérément de ne pas modifier le client.
Ainsi il reste réduit à sa fonction de base c'est-à-dire d'envoyer des requêtes HTTP à un serveur. La partie serveur que nous avons alors développé de long en large contient plusieurs fonctionnalités présentés dans la partie précédente. Les liens entre eux sont définis sur la figure du pseudo code. Nous avons tenté de suivre le parcours dans un cas de figure du code, une fois lancé.

- De prime abord l'initialisation des variables au lancement des binaires du serveur et du client (adresse, port, fichier, nombre de thread, taille du buffer…)
- ensuite on constate que les deux se lancent simultanément, ainsi est-il conseillé de lancer le serveur avant le client pour lui permettre de mettre en place les threadworkers et threads master. *(1)(2)**[1]
- une fois le(s) client(s) prêt(s), une requête sera effectuée auprès du serveur sur

[1] Ces numéros représentent l'étape sur le pseudo code donc s'y référer pour suivre le processus.

son port d'écoute. *(3)*

- A l'affût de celle-ci, le threadmaster va d'abord la récupérer puis la placer dans le buffer de requête. *(4)* il en sera pour toutes les autres si elles sont plusieurs.
- ensuite les threadworkers entrent en concurrence entre eux pour avoir cette requête suivant l'algorithme FIFO implémenté, il s'agira du prémier à avoir fait la demande au threadmaster, celui a donc le droit sur la requête qu'il va alors traité en l'enlevant du buffer de requête *(5)*
- Le traitement de la requête sera effectué avec le request_handle que le threadworker va appeler. Son traitement dépendra de si la requête est dynamique ou statique. *(6)*
- Enfin le résultat sera renvoyé au client concerné à travers une réponse dans le fichier demandé s'il est disponible ou un code d'erreur sinon. *(7)*



pseudo algorithme

Test de fonctionnement

- Test sur un ensemble de 20 requêtes HTTP, qui dure chacune 10 secondes :
- Création d'un fichier bash "manyclients" pour envoyer les requêtes dynamiques à l'aide du spin.cgi

```
  GNU nano 4.8
#!bin/bash

for j in $(seq 1 20)
do
        ./wclient localhost 8000 /spin.cgi?10 &
        sleep 1
done
```

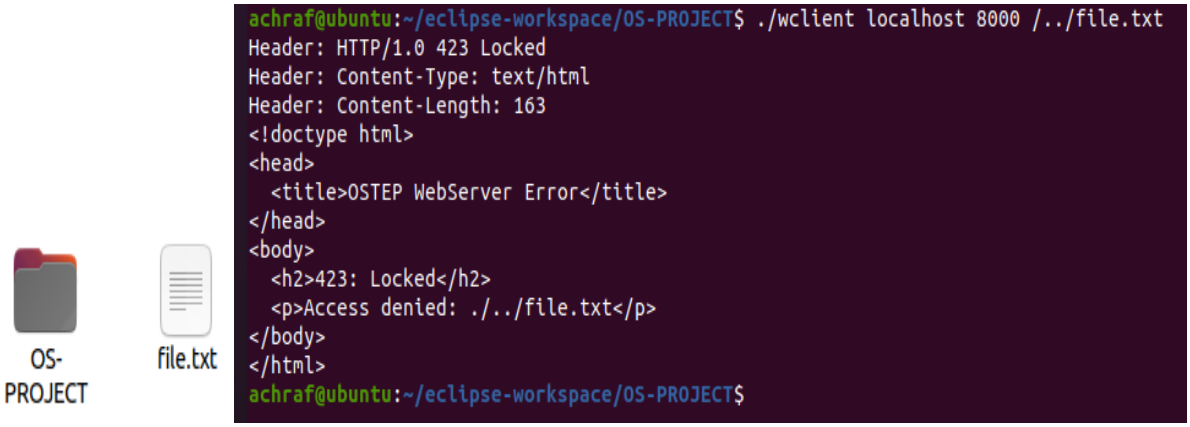
- Exécution du fichier bash "manyclients" :

```
achraf@ubuntu:~/eclipse-workspace/OS-PROJECT$ ./wserver -d . -p 8000 -t 10 -b 15
end of the creation of thred workers
 number of threads :10
 maximum number of requests :15
End of the creation of thread workers
Request for  is added to the buffer.
Request for  is removed from the buffer.
method:GET uri:/spin.cgi?10 version:HTTP/1.1
Request for  is added to the buffer.
Request for  is removed from the buffer.
method:GET uri:/spin.cgi?10 version:HTTP/1.1
Request for  is added to the buffer.
Request for  is removed from the buffer.
method:GET uri:/spin.cgi?10 version:HTTP/1.1
Request for  is added to the buffer.
Request for  is removed from the buffer.
```

```
achraf@ubuntu:~/eclipse-workspace/OS-PROJECT$ bash manyclients.sh
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: HTTP/1.0 200 OK
Header: Server: OSTEP WebServer
Header: Content-Length: 129
Header: Content-Type: text/html
<p>Welcome to the CGI program (10)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spin for 10.01 seconds</p>
```

- Test de sécurité: Assurer que le client ne peut pas entrer un chemin contenant “..”:

- On a créée un fichier “file.txt” dans /../ par rapport au root_dir



⇒ Fonctionnement: Le code fonctionne correctement, tous les fichiers sont presque liés, le makefile est adapté à la nouvelle structure et le but de multi-threading est atteint.

Conclusion :

Le multi-threading a donné une rapidité et efficacité à notre webserver, en permettant une exécution en parallèle des requêtes, ce qui réduit le temps d’attente ou le “spin waiting”. Et c’est grâce à un thread master ou gérant qui joue le rôle d’un orchestrateur, en partageant les ressources et requêtes sur les thread workers pour augmenter la performance du webserver.

Ce projet nous a appris l’optimisation de nos ressources, le concept du multi-threading et la “concurrency” et De plus l’organisation du travail en binôme et les attentes du projet ont nécessité de nous des capacités de travail d’équipe et de synthèse. Nos objectifs de réalisation ont été atteints. En somme, c’était une application concrète de ce qu’on a appris dans le cours du système d’exploitation.