Concurrency-webserver

CE PROJET EST RÉALISÉ PAR() {

Printf(Mohamed Nassim Laadhar;

&

Iheb Kotorsi); **}**

SOUS L'ENCADREMENT DE() {

Printf(Mr. Guillaume Doyen ;

&

Mr. Renzo Navas); **}**

2021-2022

Objectifs de Projet :

Le but de projet c'est de créer un serveur qui peut traiter instantanément les requêtes venant de plusieurs clients en même temps. La solution pour un tel objectif doit être le multithreading. Le processus va partager son espace mémoire en plusieurs threads qui auraient l'impression que chacun d'eux a la totalité de l'espace mémoire et une forte indépendance, or il y aura un schéduleur qui est responsable d'organiser le temps de travail de chaque thread. Globalement, il y aura un thread master et des sous threads ou des thread-workers. Le thread master est celui qui crée les thread-workers, se met à l'écoute d'un port choisi et reçoit les requêtes venant de plusieurs clients. Quant aux thread-workers, ce sont eux qui vont traiter ces requêtes là et chaque thread traite une seule requête à un instant donné.

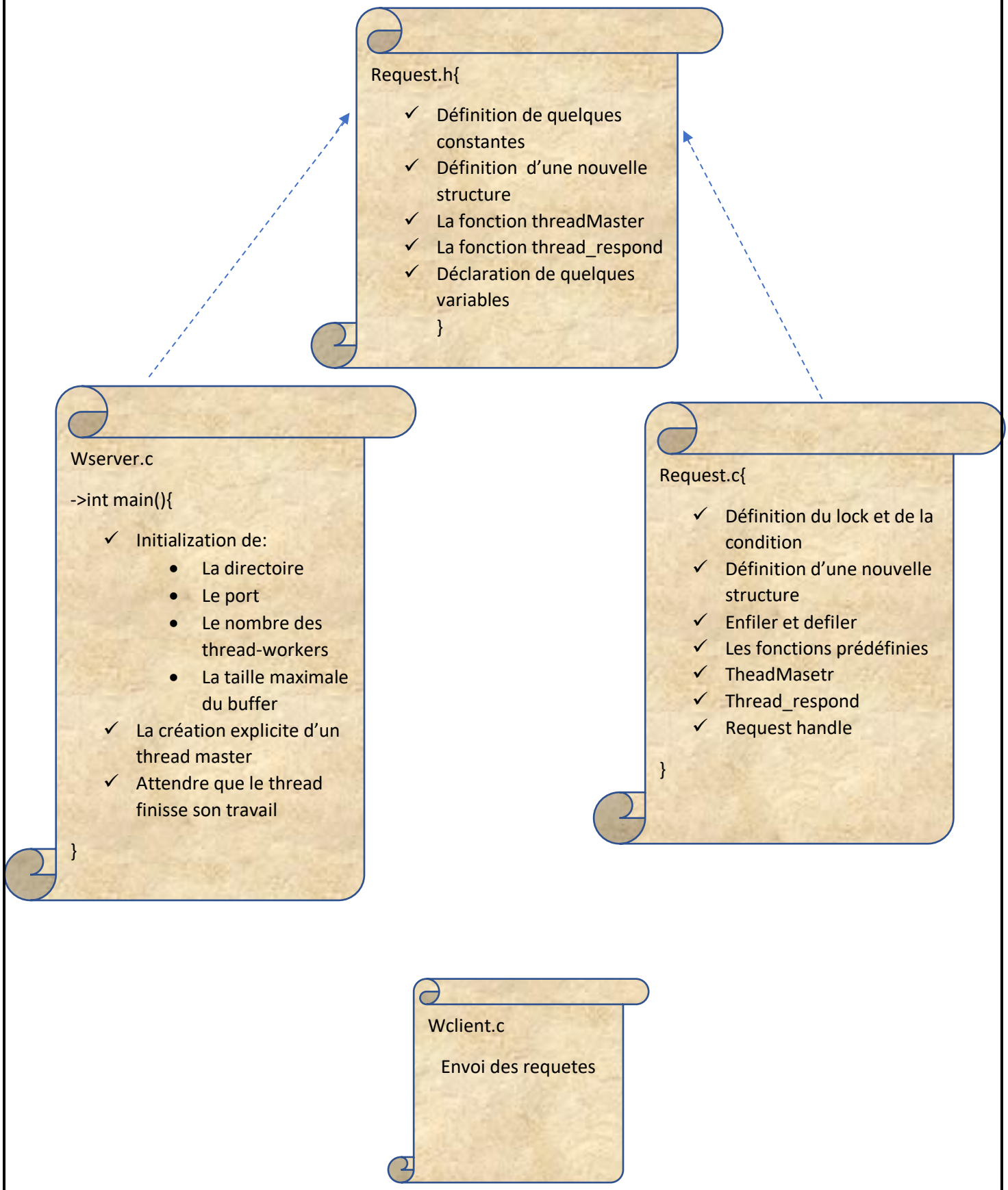
Ainsi, le thread master et ces thread-workers vont avoir une relation de type producer/consummer et l'existence d'une entité « buffer » sera nécessaire pour gérer cette relation.

Rappel de l'existant :

Le code déjà fourni est un code simple ou single-threaded code. C'est un code dans lequel le processus du serveur ne comporte qu'un seul thread et ainsi ne peut gérer qu'une seule requeté à un instant donné.

On a pour travail de rendre ce code un multi-threaded code pour assurer la « concurrency » et avoir un nombre minimum de requête qui peuvent être exécutées en même temps

Diagramme des différents composants :

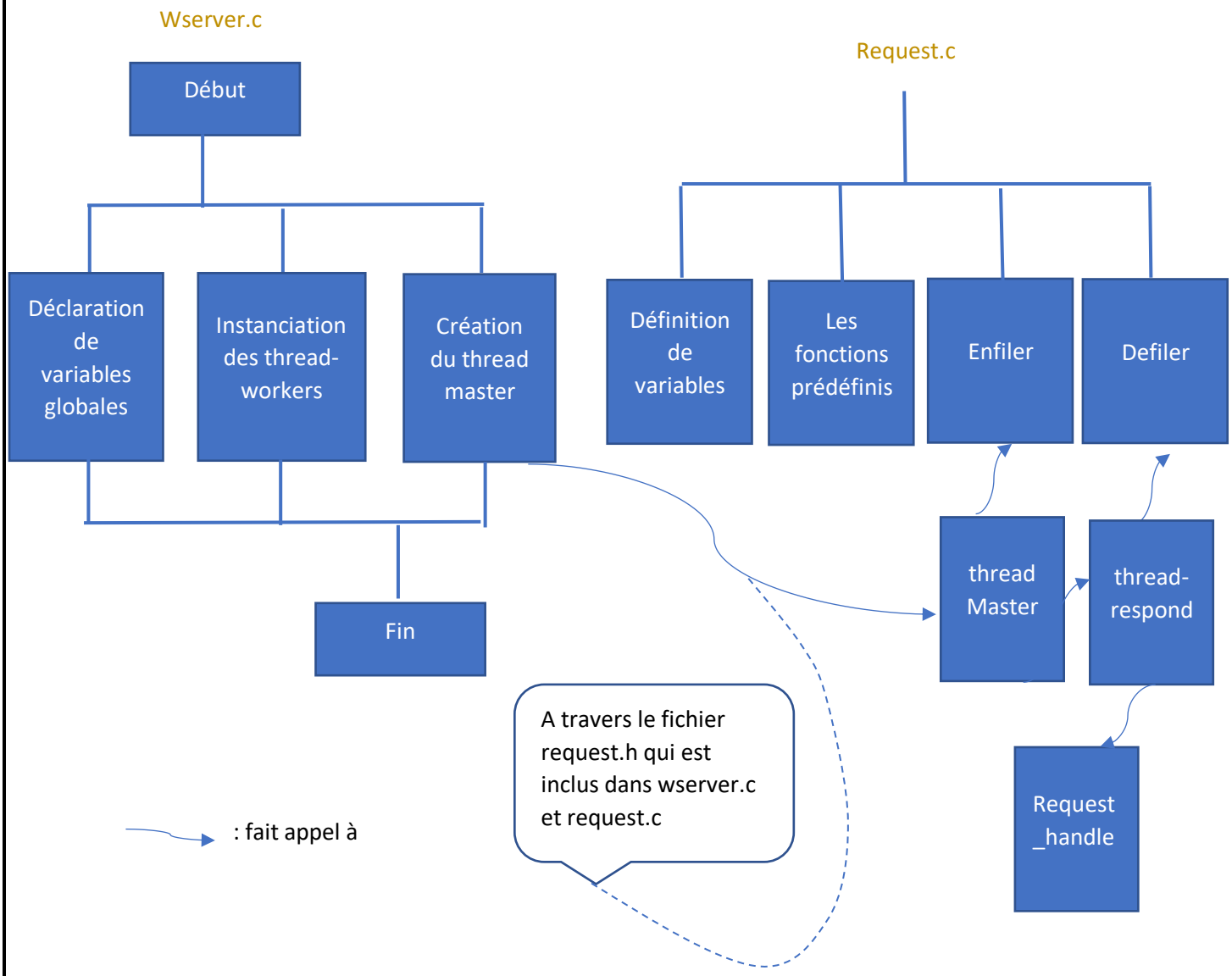


Explication du diagramme :

- Le code est composé de plusieurs fichiers .c et des fichiers .h . Ces fichiers sont wclient.c, wserver.c, request.c , request.h et d'autres fichiers.
- En exécutant le wclient.c , on obtient un fichier « wclient » exécutable et lorsqu'on l'exécute , on va assimiler le fonctionnement d'un client qui va envoyer une requête à un serveur donné sous un port précis.
- Les fichiers wserver.c, request.c et request.h sont interdépendants.
- En exécutant le fichier wserver.c, on crée un serveur qui a pour but de rassurer le bon fonctionnement de 3 opérations qui sont :
    - La création de Thread-Master
    - Rester à l'écoute d'un port précis
    - Ajouter les requêtes à un buffer
- L'exécution de ce fichier nécessite qu'il inclus le fichier request.h qui contient la définition de quelques entités et de quelques fonctions nécessaires au bon fonctionnement du serveur.
- Lorsque le serveur crée le thread-master, ce dernier va lui-même créer les thread-workers via le code écrit sur le request.c. Le request.c permet aussi de définir les fonctions nécessaires pour l'ajout et la suppression des requêtes, leur traitement et l'envoi des résultats ou des erreurs.

Pseudo-code des différents algorithmes essentielles :

- Diagramme :



- Explication en pseudo-code :
  - Pour le fichier wserver.c :

    Début
    - Définition de nombre the thread-workers , de la taille maximale du buffer, le port ,le thread master ainsi que l'ensembe de thread-workers
    - Tant que (il y aura des arguments à entrer dans la commande)

      Faire

      Redéfinition des variables précédentes
    - Fin
    - Créer le threadMaster et attendre son exécution

    Fin
  - Pour le fichier request.c :
    - Lorsque le wserver.c est exécuté, il fait appel à la fonction thradMaster dans request.c, cette fonction crée le threadMaster, La fonction elle-même fait appel à 2 autres fonctions qui sont enfiler et thread_respond
    - La fonction enfiler est utiliser pour que le threadmaster puisse ajouter les requêtes à un buffer
    - Tandis que à la fonction thread_respond, elle permet la création des thread-workers, au cours de cette création , cette fonction fait appel à defiler() qui est une fonction qui retourne une requête du buffer. Le role de chaque thread est de traiter une requête lorsqu'il est réveillé. Une autre fonction appelée est la request_handle, cette fonction est nécessaire pour que le traitement des requêtes soit fonctionnel.
  - Pour le fichier request.h

    C'est un fichier d'extension .h qui sert à lier les fichiers wserver.c et request.c.Il contient les méthodes et les structures qui sont utilisées simultanément dans les 2 fichiers.c

Test effectués et état de code :

- Tester le fonctionnement du serveur après la compilation :

```
user@SYS-Ubuntu20:~/Téléchargements/clientserveraaaaaaa/clientserver$ ./wserver
-d . -p 8000 -t 15 -b 10
Creation des threads workers: begin
15
10
Creation des threads workers: end
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.
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.
```

- Créer un fichier bash qui simule le fonctionnement de plusieurs clients qui veulent envoyer des requêtes http :

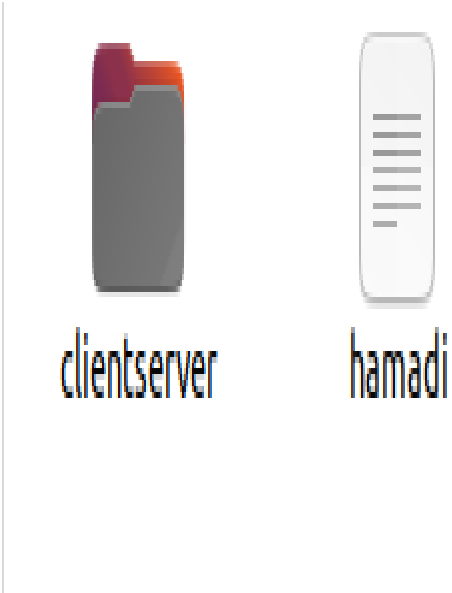
```
#!bin/bash

for j in $(seq 1 30)
do
        ./wclient localhost 8000 /spin.cgi?10 &
        sleep 1
done
```

- Exécuter le fichier pour assurer l'envoi et vérifier si les requêtes sont traitées ou pas :

```
user@SYS-Ubuntu20:~/Téléchargements/clientserveraaaaaaa/clientserver$ 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
```

- Vérifier que le client ne peut pas avoir l'accès à des fichiers inaccessibles en cherchant à demander un fichier qui n'existe pas dans la bonne directoire :




```
user@SYS-Ubuntu20:~/Téléchargements/clientserveraaaaaaa/clientserver$ ./wclient
localhost 8000 /../hamadi
Header: HTTP/1.0 423 Locked
Header: Content-Type: text/html
Header: Content-Length: 161
<!doctype html>
<head>
  <title>OSTEP WebServer Error</title>
</head>
<body>
  <h2>423: Locked</h2>
  <p>Access denied: ./../hamadi</p>
</body>
</html>
user@SYS-Ubuntu20:~/Téléchargements/clientserveraaaaaaa/clientserver$
```

⇨ Le code fonctionne presque parfaitement, il y a quelques de temps en temps mais le but ultime de création d'un serveur qui soit multi-threaded et fonctionnel est déjà atteint.

Conclusion :

Le multi-threading est essentiel pour assurer une exécution en parallèle de plusieurs requêtes, cela réduit le temps d'attente que le client fait pour recevoir le message qui indique la finalisation du traitement de la requête qu'il a envoyé et permet au serveur de partager ces ressources mémoires en des threads et augmenter sa performance.

On a appris ainsi l'importance du concept de « concurrency » et on l'a exploité dans des cas réels pour mieux visualiser son impact .Cela était très éducatif et malgré quelques difficultés le long du projet tels que le time management et la non-simplicité du projet, on a pu réaliser le but dans les bons délais.