DANCOISNE Sébastien
ETEVENARD Thibaud
16/12/2021

Rapport de projet systèmes d’exploitation

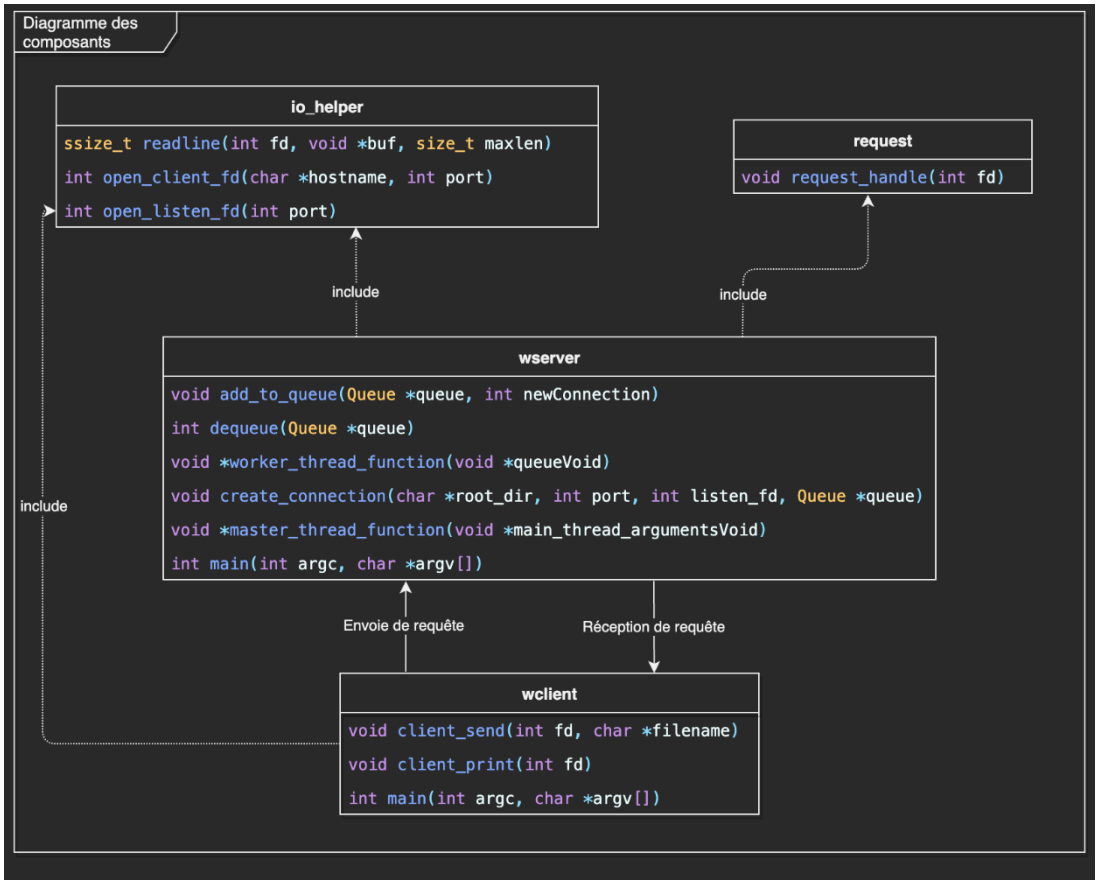
IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

I) RAPPEL DES OBJECTIFS ET DE L'EXISTANT

L'objectif de ce projet était de transformer un serveur single thread en serveur multi threads. Pour réaliser cette transition, le fonctionnement du serveur nous était imposé, nous avons dû créer un master thread dans lequel une pool de worker thread est initialisée avec une taille donnée en paramètre. De plus, ce master thread doit récupérer les différentes connections et les stocker dans un buffer lui aussi ayant une taille donnée en paramètre. Le rôle des worker threads est de récupérer ces connexions stockées dans un buffer et de les gérer, cependant une des contraintes qui nous était fixée était l'utilisation des variables de conditions afin de ne pas effectuer d'attente inutile. Pour finir, nous avions aussi à prendre en compte la sécurité du serveur en s'assurant que le client ne pouvait pas obtenir des fichiers en dehors du répertoire dans lequel il se trouve.

Afin de réaliser ce projet, nous avions initialement à notre disposition le code d'un serveur single thread. Nous avions donc un client fonctionnel capable d'envoyer des request à notre serveur, ces requests nous ont, elles aussi, été fournis dans le code initial. Pour nous permettre de facilement effectuer des tests, un programme spin.cgi nous a aussi été fourni.

II) DIAGRAMME DES DIFFERENTS COMPOSANTS DU CODE



Dans notre serveur multi thread, quatre fichiers différents sont utilisés lors de son fonctionnement : les fichiers request, io_helper, wserver et enfin wclient. De plus, nous avons à notre disposition un fichier spin.cgi nous permettant d'effectuer des tests sur notre serveur. Enfin nous avons aussi un fichier Makefile nous permettant de compiler le code nécessaire au bon fonctionnement du serveur. Dans cette partie, nous allons expliciter le fonctionnement de chacun de ces composants ainsi que de leurs interactions.

Tout d'abord, le fichier request.c qui a pour objectif de traiter la requête préalablement envoyée par le client. La fonction principale que l'on retrouve au sein de ce fichier est request_handle, cette dernière est appelée par les worker threads au sein du serveur.

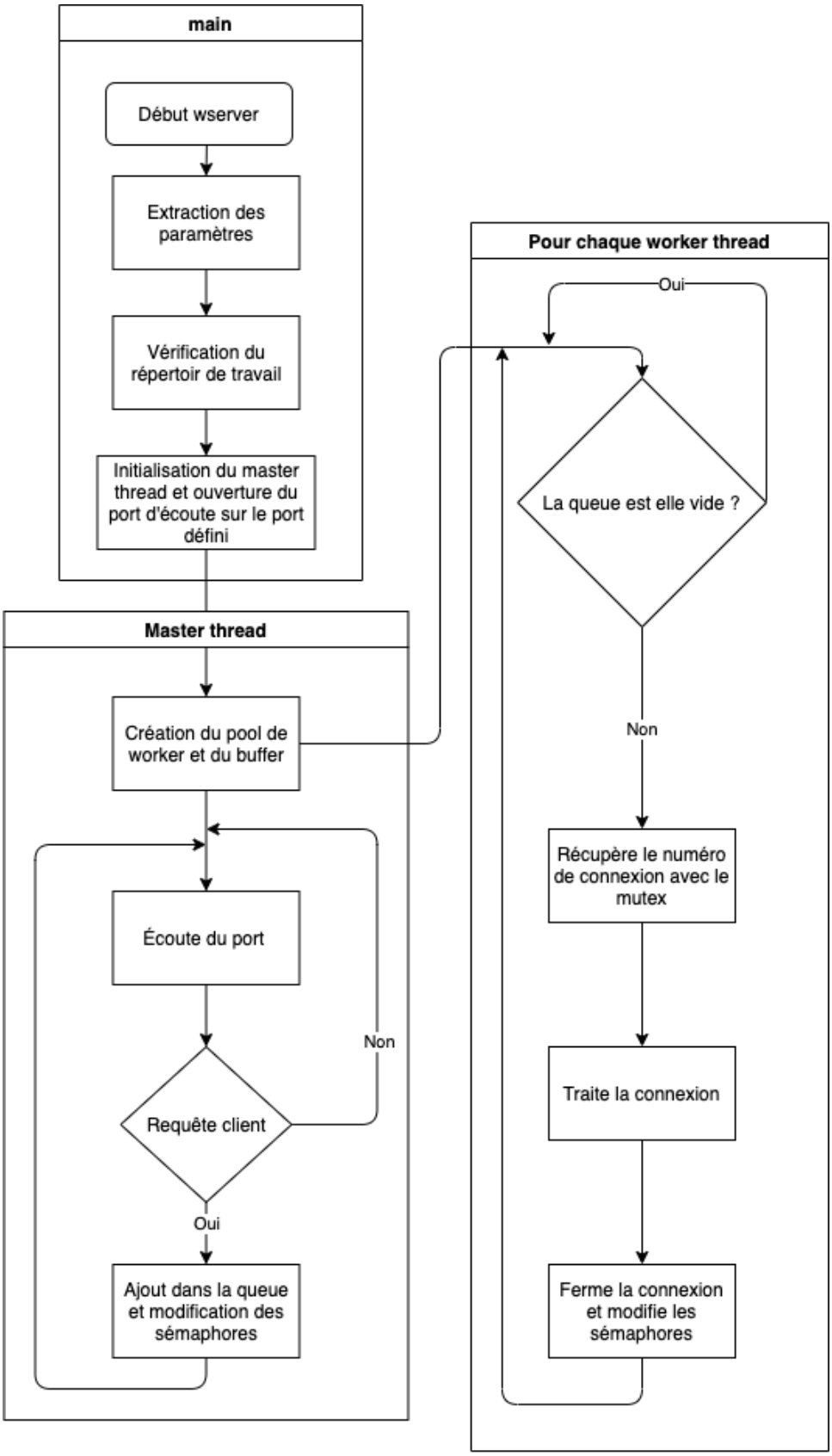
Concernant le fichier io_helper.c, ce dernier est responsable de tous ce qui touche à la création et à la connexion ainsi qu'à sa récupération via une écoute sur un port. Cette dernière étant préalablement générée grâce à la fonction open_listen_fd de ce fichier, selon le port donné en paramètre lors de la création du serveur.

En outre, nous avons aussi à notre disposition le fichier wclient.c, il est responsable de la création de requête vers le serveur. Pour ce faire, il utilise deux fonctions principales : client_send et client_print. La première servant à envoyer ladite requête vers le serveur et la seconde sert à afficher le contenu de la demande.

Finalement, le fichier le plus important et au sein duquel nous avons effectué les modifications afin de rendre le serveur multi thread est wserver.c. Ce fichier contient la fonction main permettant de récupérer les paramètres donnés en arguments lors de l'initialisation du serveur. Puis, elle initialise le master thread qui est la fonction chargée de créer le pool de worker thread et d'écouter sur le port désigné afin de récupérer les connexions client. Ces workers threads récupèrent la connexion dans le buffer, ledit buffer est préalablement initialisé par le master thread. Une fois qu'une connexion est ajoutée au buffer, le worker thread traite la connexion grâce à la fonction request_handle de request.c. Pour finir, le worker ferme la connexion et attend une nouvelle connexion. De plus, pour éviter le biais lié à l'ajout de connexion en direct, qui repousserait potentiellement le traitement de la connexion, nous avons mis en place une liste FIFO, permettant de conserver l'ordre d'ajout des connexions.

Enfin, le projet possède des fichiers en .h. Ces fichiers sont utilisés pour importer les fichiers request.c et io_helper.c dans les fichiers wclient.c et wserver.c. En effet, les fichiers .h permettent de donner le prototype des fonctions utiles aux deux autres fichiers, afin que le compilateur arrive à vérifier le type des objets de manière statique.

III) PRESENTATION EN PSEUDO-CODE DES DIFFERENTS ALGORITHMES ESSENTIELS DE L'APPLICATION



Dans cette partie, nous allons présenter les différents algorithmes essentiels de notre application ainsi que son fonctionnement global. On peut découper cela selon trois catégories principales : la fonction main, servant à l'initialisation ; le master thread et les workers threads.

Tout d'abord, lors de la création du serveur, la fonction est invoquée et extrait les différents paramètres nécessaires au bon fonctionnement de l'application. De plus, on s'assure que les paramètres fournis sont en accord, type et nombre, avec les spécifications du serveur. Finalement, cette fonction s'occupe de créer un master thread ainsi qu'ouvrir l'écoute sur le port sélectionné dans les paramètres.

Nous allons maintenant nous attarder sur le fonctionnement du master thread. Ce thread est le point essentiel de l'application car il est à l'origine de la création des différents travailleurs ainsi que de la récupération des requêtes client. Pour ce faire, un buffer est initialisé selon une structure d'une First In First Out (FIFO). Afin de pouvoir manipuler cette structure à notre guise, nous avons créé une fonction permettant d'ajouter des éléments à notre FIFO et une seconde permettant d'en enlever. Une fois ce buffer créé, la pool de workers threads est créé. En effet, chaque travailleur a besoin de ce buffer pour fonctionner correctement. Le second rôle de ce thread est la récupération des différentes requêtes clients, à l'aide d'une boucle infinie. Pour ce faire, il appelle la fonction create_connection qui bloque l'algorithme tant qu'il n'y a pas de nouvelle demande de connexion. Puis, à la suite d'une connexion, l'algorithme attend que le sémaphore empty, représentant la place libre dans le buffer, puis récupère le lock afin d'ajouter le numéro de connexion dans la FIFO sans risque d'écriture en simultané dans le cas où l'application serait améliorée avec plusieurs threads pouvant écrire au sein du buffer. Finalement, il incrémente un second sémaphore : full, qui représente le nombre d'éléments dans le buffer.

Concernant les workers threads, ces derniers servent à traiter les requêtes client stockées dans le buffer. Pour remplir cet objectif, ils attendent qu'un élément soit situé dans le buffer, cela se faisant par le biais du sémaphore full. Puis, il récupère le lock afin d'empêcher un autre worker d'interagir avec le buffer en même tant que lui, ce qui pourrait entraîner une erreur, car ils pourraient se retrouver tout deux à traiter la même requête. Ensuite, le travailleur libère le lock afin de permettre à un autre worker de prendre sa relève puis traite la requête. Finalement, le worker s'occupe de fermer la connexion et d'incrémenter le sémaphore empty afin de signaler qu'une nouvelle place est disponible dans le buffer et qu'il est alors possible d'ajouter un nouvel élément. Une fois cette opération finie, le worker retourne à l'étape initiale et reprend la boucle.

Pour conclure, l'harmonie entre les différents blocs de l'algorithme permettent son bon fonctionnement. À l'aide des différents tests explicités ci-dessous, nous pouvons dire que notre algorithme semble être opérationnel.

IV) TESTS EFFECTUES ET ETAT DU CODE

Une fois notre serveur devenu multi thread, nous avons effectué différents tests afin de nous assurer de son bon fonctionnement. Nous avons dans un premier temps, écrit un script bash permettant de lancer un nombre N, fourni en paramètre, de client envoyant une requête au serveur. Lors de nos premiers tests, la requête effectuée concernait une simple page « Hello page ! » en html ainsi, au vu du peu de quantité d'informations disponibles sur cette page, nous ne savions pas s'il était possible de tirer des conclusions quant au bon fonctionnement de notre serveur.

Nous avons donc décidé d'utiliser la fonction spin.cgi car il permet de créer une connexion pendant un temps défini et rien d'autre. Ainsi, si le serveur est bien multi threads, le temps d'exécution du script bash est alors égal à la durée d'ouverture de la connexion du spin.cgi multiplié par le minimum entre le reste de la division de la taille du buffer par le nombre de requêtes client ou le reste de la division de la taille de notre pool de thread par le nombre de requêtes client. Nous avons comparé ce temps d'exécution au temps de traitement du même nombre de demandes mais avec cette fois-ci uniquement une taille de pool de 1 afin de simuler un serveur single thread.

Les résultats ont été sans appel, pour 20 requêtes de spin.cgi ?20 avec un buffer de taille 100 et une pool de 20 threads, l'ensemble des requêtes a été traité en 20 secondes comme on pouvait s'y attendre. En reprenant les mêmes paramètres mais avec cette fois uniquement un thread, les 20 clients ont été servis en 400 secondes. Ainsi, en effectuant différents tests de ce type, nous avons pu en conclure que notre serveur semble bien avoir un fonctionnement multi threads.

De plus, nous avons aussi testé la bonne prise en compte des paramètres par défaut lorsque rien n'était précisé lors de la création de wserver ainsi que de la règle de sécurité stipulant que le client ne peut pas effectuer une requête concernant un fichier situé en dehors de son dossier.

Pour conclure, notre code semble actuellement être fonctionnel. Les différentes demandes ont été suivies, on possède bien un master thread capable d'initialiser une pool de worker thread puis de stocker les différentes requêtes qui lui sont soumises dans un buffer partagé avec l'ensemble des worker threads. Ces derniers sont bien capables de s'occuper d'une connexion et d'attendre entre chaque connexion sans consommer de ressources inutiles. Finalement, en matière de sécurité, notre serveur n'est certes pas optimal, cependant le client ne peut pas effectuer de requête concernant un fichier en dehors de son dossier, la contrainte demandée est donc bien respectée.

V) CONCLUSION

Pour conclure, ce projet nous a permis de nous familiariser avec le langage C qui était totalement nouveau pour nous. Nous avons aussi dû apprendre à travailler à partir d'un code déjà fourni et dont nous n'étions pas les auteurs. Nous avons dû faire en sorte de nous l'approprier afin de pouvoir mener à bien ce projet. De plus, l'enseignement principal que nous avons obtenu via ce projet est le fonctionnement d'un serveur multi thread en utilisant des sémaphores et des variables de condition et l'utilisation des sockets pour créer des connexions. De plus, nous avons dû nous renseigner sur les différentes méthodes d'implémentations d'une file d'attente en C.

Concernant les difficultés que nous avons rencontrées, le point qui nous a le plus causé de soucis est la création des sémaphores. En effet, suite à une interruption du programme en cours d'exécution, les sémaphores gardaient en mémoire leur valeur et donc entraînaient une erreur lors de l'exécution suivante. La difficulté suivante concerne le passage de paramètres lors de la création d'un thread, car le type d'entrée doit être void* ainsi, nous avons dû convertir en void* le paramètre à envoyer puis le reconvertir en son type d'origine au sein du thread. Cette difficulté s'est surtout fait ressentir lorsque nous avions plusieurs paramètres à transmettre au thread et que nous avons donc dû utiliser une structure permettant de les regrouper. Néanmoins, nous avons eu un dernier problème concernant le transfert de paramètre dans les workers threads. Cela consistait en l'envoi de la valeur du pointer de la file générée dans le master thread au lieu de son pointer, résultant en la création d'une nouvelle file avec pour valeur initiale 0 et donc causant un blocage dans le serveur.