Rapport Systèmes d'exploitation: Multi-threading webserver

Akram BLAL : akram.blal@imt-atlantique.net

Skander CHAYOUKHI : skander.chayoukhi@imt-atlantique.net

Année universitaire : 2024-2025

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

I- Rappel des objectifs:

Le but de ce projet est de modifier un serveur web basique pour le rendre concurrent. Initialement, le serveur fourni fonctionne avec un seul thread, ce qui signifie qu'il ne peut traiter qu'une requête HTTP à la fois. L'objectif est donc d'ajouter un modèle multi-threadé pour permettre au serveur de gérer plusieurs requêtes simultanément.

Ce projet a plusieurs objectifs, dont principalement:

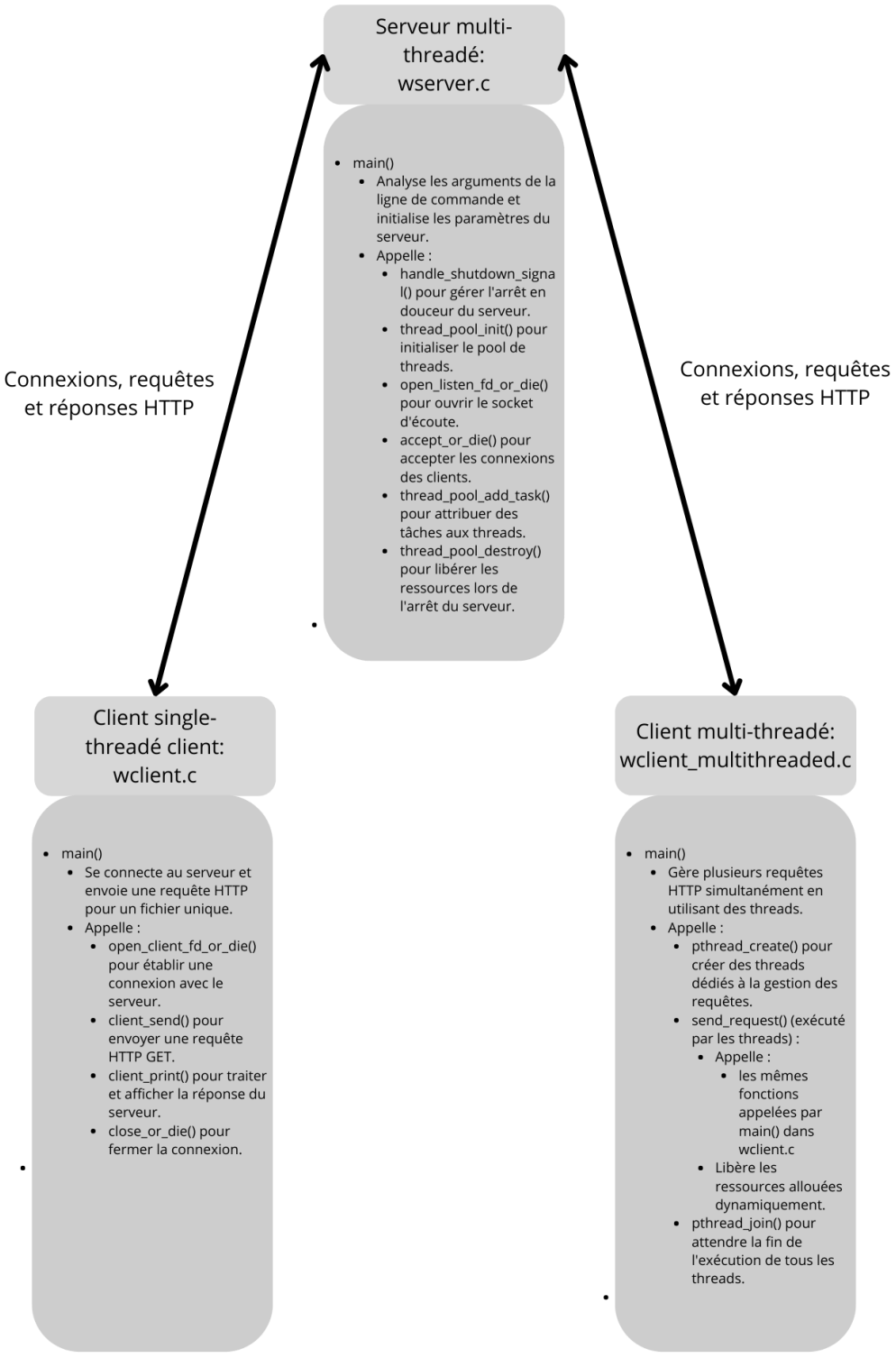
- Comprendre comment fonctionne l'architecture d'un serveur web simple.
- Apprendre à intégrer la concurrence dans un programme séquentiel en utilisant des concepts comme les threads, les mutex et les variables de condition.
- Développer ses compétences sur la lecture, la compréhension et la modification d'un code existant.

Le serveur devra gérer à la fois des contenus statiques (comme des fichiers HTML) et dynamiques (via des scripts CGI). Il devra également inclure un système de synchronisation efficace pour permettre aux threads de consommer les requêtes stockées dans un buffer partagé.

Des paramètres comme le nombre de threads, la taille du buffer, le port et le répertoire racine devront être configurables via des arguments de ligne de commande. Enfin, des contraintes de sécurité seront mises en place, notamment pour éviter que les utilisateurs ne puissent accéder à des fichiers en dehors du répertoire défini.

II- Diagramme des Différents Composants:

Le diagramme suivant représente les principaux composants de l'application et leurs relations. Il est conçu pour illustrer comment les fichiers interagissent et quelles sont les fonctions essentielles utilisées dans chaque module.



1. Serveur Multithread:

Le composant principal est le **serveur multithread**, qui gère les connexions entrantes et répartit les requêtes clients entre plusieurs threads pour un traitement efficace.

- **Fonctionnalités principales :**
  - `main()` : Point d'entrée du serveur. Gère l'initialisation, l'écoute des connexions et le traitement des requêtes.
  - `handle_shutdown()` : Permet d'arrêter le serveur proprement lorsqu'un signal (ex : `SIGINT`) est reçu.
  - `thread_pool_init()` : Initialise un pool de threads pour gérer les requêtes simultanément.
  - `open_listen_fd()` : Ouvre un socket en mode écoute sur le port spécifié.
  - `accept_or_die()` : Accepte les connexions entrantes des clients.
  - `thread_pool_add()` : Ajoute une tâche (connexion client) au pool de threads pour traitement.
  - `thread_pool_destroy()` : Libère les ressources du pool de threads lors de l'arrêt.

Le serveur est donc conçu pour être scalable grâce à son architecture multithreadée qui permet de gérer plusieurs clients simultanément.

2. Client Monothread:

Le **client monothread** est déjà implémenté dans le code initial. Il s'agit d'une version simplifiée qui gère une seule requête HTTP à la fois, ce qui le rend adapté pour des tests simples.

Ce client sert principalement à vérifier les fonctionnalités de base du serveur.

3. Client Multithread:

Le **client multithread** est une version plus avancée, capable d'envoyer plusieurs requêtes HTTP simultanément.

- **Fonctionnalités principales :**
  - `main()` : Point d'entrée du client. Crée plusieurs threads pour gérer différentes requêtes en parallèle.
  - `pthread_create()` : Lance un thread pour chaque requête.
  - `send_request()` : Fonction exécutée dans chaque thread, qui :
    - Appelle `client_send()` pour envoyer une requête.
    - Utilise `client_print()` pour lire et afficher la réponse.
    - Appelle `close_or_die()` pour fermer la connexion.
  - `pthread_join()` : Attend la fin de tous les threads avant de quitter.

Ce client est conçu pour tester la robustesse du serveur sous une charge importante en simulant plusieurs utilisateurs.

III- Diagrammes en pseudo-code :

```
Démarrer Serveur
   |
   v
Initialiser signaux (SIGINT, SIGTERM)
   |
   v
Analyser arguments de la ligne de commande
   |
   v
   Si arguments fournis, mettre à jour root_dir, port, num_threads, buffer_size
   |
   v
Changer répertoire (chdir_or_die)
   |
   v
Initialiser pool de threads (thread_pool_init)
   |
   v
Ouvrir socket d'écoute (open_listen_fd_or_die)
   |
   v
Attente connexions clients
   |
   v
   Tant que !graceful_shutdown
      |
      v
      Accepter connexion entrante (accept_or_die)
      |
      v
      Si graceful_shutdown, fermer connexion
      |
      v
      Ajouter tâche au pool de threads (thread_pool_add_task)
   Fin

Arrêter Serveur
   |
   v
Détruire pool de threads (thread_pool_destroy)
   |
   v
Fermer socket d'écoute (close_or_die)
   |
   v
Afficher "Serveur arrêté"
```

```
Démarrer Client
   |
   v
Analyser arguments (host, port, filename)
   |
   v
Ouvrir connexion au serveur (open_client_fd_or_die)
   |
   v
Envoyer requête HTTP (client_send)
   |
   v
Lire et afficher réponse (client_print)
   |
   v
Fermer connexion (close_or_die)
```

```
Démarrer Client Multi-Threadé
   |
   v
Analyser arguments (host, port, filenames)
   |
   v
   Pour chaque fichier dans filenames
      |
      v
      Ouvrir connexion (open_client_fd_or_die)
      |
      v
      Créer structure client_request_t
      |
      v
      Créer thread (pthread_create)
   Fin

Attendre fin threads (pthread_join)
```

Explication des Diagrammes:

Le diagramme illustre les processus clés du serveur (à gauche) et des clients (à droite) dans un système où le serveur est multi-threadé et capable de gérer plusieurs connexions simultanées, tandis que les clients peuvent être soit simples (un seul fil d'exécution) soit multi-threadés (gérant plusieurs requêtes en parallèle).

Serveur Multi-Threadé

Le serveur commence par initialiser les signaux d'arrêt (SIGINT, SIGTERM) pour permettre une fermeture propre lors de l'interruption du processus. Ensuite, il analyse les arguments passés en ligne de commande (par exemple, le répertoire racine, le port d'écoute, le nombre de threads et la taille des buffers). Ces paramètres permettent de personnaliser le comportement du serveur, notamment le nombre de threads qu'il utilisera pour traiter les requêtes des clients.

Une fois les arguments traités, le serveur change son répertoire de travail (à l'aide de `chdir_or_die`) pour correspondre à l'emplacement des fichiers qu'il devra servir. Il initialise ensuite le pool de threads (`thread_pool_init`) et ouvre un socket d'écoute sur le port spécifié (`open_listen_fd_or_die`).

Après l'initialisation, le serveur passe dans une boucle principale où il attend les connexions des clients. Lorsqu'une connexion entrante est détectée, il accepte la connexion et attribue cette tâche à un thread du pool. Chaque connexion est gérée par un thread distinct qui traitera la requête du client. Si un signal d'arrêt est reçu, le serveur ferme les connexions en cours, arrête le pool de threads (`thread_pool_destroy`), puis ferme le socket d'écoute et affiche un message confirmant la fermeture correcte du serveur.

Client Single-Threadé

Le client simple (single-threaded) fonctionne de manière linéaire : il commence par analyser les arguments de ligne de commande pour obtenir l'hôte, le port et le fichier à demander. Ensuite, il établit une connexion avec le serveur via `open_client_fd_or_die`. Une fois la connexion établie, il envoie une requête HTTP pour le fichier spécifié en appelant `client_send`. Après l'envoi de la requête, le client attend la réponse du serveur, qu'il lit et affiche à l'aide de `client_print`. Une fois que la réponse est affichée, il ferme la connexion.

Client Multi-Threadé

Le client multi-threadé fonctionne de manière similaire au client single-threadé, mais il est capable de traiter plusieurs fichiers en parallèle. Pour chaque fichier demandé, un nouveau thread est créé. Chaque thread ouvre une connexion avec le serveur, envoie une requête HTTP pour le fichier, puis affiche la réponse. Les threads sont gérés à l'aide de `pthread_create`, et le programme attend que tous les threads terminent leur exécution via `pthread_join`. Une fois tous les threads terminés, le programme peut fermer les connexions ouvertes.

IV-Tests effectués et état du code

Nous avons effectué plusieurs tests pour valider les fonctionnalités de notre serveur, notamment en utilisant des scripts conçus pour envoyer des requêtes simultanées et tester les réponses du serveur dans des scénarios variés.

Test avec les script `test_requests.sh`et `testing_multiple_clients.sh`
Les scripts `test_requests.sh` et `testing_multiple_clients.sh` visent à tester un serveur en envoyant des requêtes HTTP (en alternant entre des fichiers statiques et des scripts dynamiques). Le premier envoie une seule requête avec des paramètres définis par l'utilisateur, tandis que le second permet de simuler plusieurs clients envoyant des requêtes multiples, alternant entre différentes pages.
Résultat :
Les deux scripts affichent les en-têtes HTTP et les réponses, permettant ainsi d'évaluer les performances du serveur dans différentes situations.

Test avec le script `testing_multiple_multithreaded_clients.sh`
Ce script a permis de tester la capacité du serveur à gérer plusieurs clients multithreadés. Le script lance plusieurs clients simultanés en envoyant des requêtes pour plusieurs fichiers, y compris des fichiers statiques et dynamiques. L'objectif de ce test était de vérifier le comportement du serveur sous une charge de requêtes simultanées et de valider la fonctionnalité multithreadée.
Résultat :
Le serveur a correctement géré jusqu'à 4 requêtes simultanées, mais certains fichiers n'ont pas été reçus. Cela pourrait être lié à un problème dans la gestion du tampon et des threads. Le serveur semble fonctionner avec des charges modérées, mais des améliorations sont nécessaires pour mieux gérer des scénarios à plus grande échelle.

Test de l'arrêt de serveur (CTRL +C):
Lors du test de l'arrêt du serveur, nous avons simulé une situation où le serveur continue de traiter des connexions tout en recevant un signal d'arrêt (via Ctrl+C).
Résultat :
Lorsque le signal `SIGINT` a été envoyé, le serveur a initié une fermeture propre en fermant le descripteur de fichier d'écoute et en signalant aux threads du pool de se terminer. Le test a montré que le serveur a bien interrompu l'acceptation de nouvelles connexions tout en permettant aux threads existants de terminer le traitement des connexions en cours.

Vérification des tentatives de traversée de répertoire
Nous avons mis en place une vérification de la sécurité pour empêcher les tentatives de traversée de répertoire.
Résultat :
Le serveur a correctement détecté et rejeté les requêtes avec des tentatives de traversée de répertoire, renvoyant un message d'erreur approprié pour les fichiers en dehors du répertoire autorisé.

Etat du code :

Serveur multi-threadé : opérationnel

Client multi-threadé : opérationnel
Fonctionnalité d'arrêt de serveur :

- La gestion des signaux d'arrêt (`SIGINT`, `SIGTERM`) : opérationnel
- Fermeture propre des ressources (descripteurs de fichiers, threads): opérationnel
- Gestion des erreurs lors de l'arrêt : partiel (des améliorations peuvent être apportées pour la gestion d'erreurs imprévues)

Empêcher les tentatives de traversée de répertoire : opérationnel

V-Conclusion :

Apprentissage et difficultés rencontrées

Ce projet a permis d'acquérir des compétences solides dans la conception d'un serveur web multi threadé, notamment la gestion de connexions simultanées, la gestion des ressources et la sécurisation des accès.

La principale difficulté a été de permettre au serveur de se fermer proprement tout en traitant les connexions en cours. La fonction `accept_or_die`, étant bloquante, empêchait de répondre immédiatement aux signaux d'arrêt.

Pour résoudre ce problème, nous avons intégré une gestion des signaux, permettant d'arrêter le serveur tout en terminant les connexions en cours et en améliorant la gestion des erreurs dans la fonction `accept_or_die` déjà définie dans le code de base. Cela a assuré un arrêt ordonné du serveur, avec la fermeture correcte des threads et des descripteurs de fichiers.

Perspectives d'améliorations

Malgré les bonnes performances, plusieurs axes d'amélioration existent :

1. **Gestion des requêtes simultanées** : Les tests ont montré des ralentissements avec un grand nombre de clients simultanés. Une optimisation du mécanisme de mise en file d'attente et de répartition des tâches entre les threads pourrait améliorer la performance.

2. **Sécurité des fichiers** : Bien que les traversées de répertoires soient bloquées via la détection de `../` dans l'URI, des protections supplémentaires (par exemple, restrictions sur le répertoire racine) renforceraient la sécurité.

3. **Gestion des erreurs** : Une gestion d'erreurs plus détaillée, avec des journaux plus explicites, pourrait améliorer la stabilité du serveur.

4. **Tests sous forte charge** : Des tests de stress supplémentaires permettraient d'identifier des points faibles dans l'implémentation actuelle.