DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Rapport du Projet : Système de réservation distribué

HMIN105M — Principes de la programmation concurrente et répartie

Référent: Hinde Bouziane

2020

Table des matières

Introduction

La présente introduction est tirée du sujet donné.
Consigne importante pour le développement :

« Ce projet est à réaliser en binôme. Lire attentivement l'énoncé avant de commencer le travail. Pour la mise en place des connexions et communications distantes, vous utiliserez le protocole TCP/IP et les fonctions C utilisées en cours. Le plagiat est strictement interdit. »

Nous voulons mettre en place un système de réservation distribué développé en C sous un système UNIX.

« L'idée de ce projet s'inspire d'un système de réservation de ressources de calcul et/ou de stockage sur une plate-forme de grille ou de cloud. »

« Il a pour objectif de permettre à des clients de louer des puissances de calcul ou des espaces de stockage distants répondant à des besoins spécifiques (exemple : exécuter une simulation scientifique sur une architecture distribuée de processeurs et de mémoire pour stocker (le temps de la location) les données traitées et produites). »

« Dans le système à mettre en oeuvre, un client aura la possibilité de louer des ressources, soit en mode exclusif (les ressources louées sont utilisées par un seul client pendant toute la durée de la réservation), soit en mode partagé (les ressources louées peuvent être utilisées en même temps par plusieurs clients). »

Avant de définir l'architecture de notre application, nous avons repris les éléments du sujet pour définir le cahier des charges suivant.

1 Cahier des charges

Voici un aperçu général des tâches qui ont été repérées dans le sujet :

→ **Le rôle du serveur**
- mettre en place un espace partagé représentant l'ensemble des ressources disponibles et louées.
- gérer les accès à cet espace.
- maintenir cet espace dans un état cohérent.
- diffuser toute modification à tous les clients.

→ **Le rôle des clients**
- est d'échanger avec le serveur pour effectuer des réservations et libérations de ressources.
- avoir en continu une vue de l'état global de toutes les ressources offertes par le système de réservation.

Ainsi nous pouvons préciser les **caractéristiques techniques** ainsi que les contraintes importantes imposées par le sujet sur le serveur et sur les clients :

On souhaite alors pouvoir disposer d'un :
- **Serveur** concurrent lourd, qui va donc utiliser l'appel système **fork**()
  - **Le parent** est alors :
    - responsable de la mise en place de l'état initial du système de réservation.
    - en attente des connexions des clients et délègue ensuite leur traitement a un fils.
  - **Un fils**, met en oeuvre des **threads** pour paralléliser le traitement de son client. Les traitements parallèles à effectuer sont les suivants :
    - maintien et diffusion en continu de l'état des ressources cohérentes avec son client.
    - traitement et communication exclusive avec un client. La communication peut correspondre à une :
      - réservation de ressources.
      - libération de ressources.
- **Client** qui met en oeuvre un parallélisme léger, c'est-à-dire en mettant en place des **threads** pour paralléliser les traitements suivants :
  - recevoir et sauvegarder l'état du système de réservation dès sa connexion au serveur.
  - en boucle, affichage des ressources et saisie au clavier des demandes de réservation de ressources ainsi que, la nécessité de bloquer jusqu'à satisfaction de la demande de réservation de ressource. Évidemment nous avons besoin de pouvoir libérer des ressources.

Enfin, nous apprenons dans l'énoncé qu'un client peut réserver :

- de manière exclusive ou non une ressource.
- plusieurs ressources différentes à la fois.

Nous verrons par la suite que certains points du cahier des charges n'ont pas pu être implémentés dans les temps impartis. Nous nous attarderons maintenant

à la définition et la présentation de l'architecture de notre application dans le chapitre suivant.

2 Définition et présentation de l'architecture mise en place

Maintenant que nous avons toutes les informations mentionnées précédemment, nous savons ce que nous avons besoin de modéliser grâce à une structure du langage C dans notre système. Avant cela, notons qu'un site est le composant fondamental de notre modèle.

2.1 Les structures C

Voici donc la représentation, sous forme d'une structure C, de la définition d'un *site* :

```
struct site {
    char label[255];
    int nbProcesseur;
    int nbCapaciteStockage;
    int nbProcesseurExclusif;
    int nbCapaciteStockageExclusif;
    int nbProcesseurPartage;
    int nbCapaciteStockagePartage;
};
```

Premièrement, on remarque que le nom de la ville est limité à 255 caractères.

En effet, pour des raisons intrinsèques aux systèmes d'exploitation, 255 est le plus grand entier écrit sur un octet, l'octet étant l'unité par défaut de plusieurs mécanismes qui sont présents à certains endroits critiques du fonctionnement d'Unix. Par exemple, la limite du nom d'un fichier dans l'arborescence Unix est de 255 caractères, mais c'est également la taille des tampons sous les couches les plus basses du modèle TCP/IP.
Pour de telles raisons, nous avons fait ce choix-là.
Grâce à cette structure nous pouvons donc définir l'état global du système par la structure suivante :

```
struct memoirePartager {
    int taille;
    struct site sitesDispo[1024];
};
```

Nous avons choisi le nom *memoirePartager* pour nommer la structure, en effet, cette modélisation se trouve en mémoire partagée du serveur. Nous retrouvons bien les caractéristiques suivantes dans notre structure qui sont essentielles à la bonne gestion des clusters de calculs.

Le système possède :

- plusieurs sites.
- un accès aux quantités louées par site de façons exclusives et partagées.
- un accès aux quantités totales et restantes (via un calcul simple) par sites.

Par la suite, nous avons écrit en dur la quantité maximale de sites que notre application pourra utiliser via deux critères importants du projet.

Sachant que l'utilisation de la mémoire partagée au sein du serveur était nécessaire et que tel qu'il est défini par System V ne permet pas d'y injecter des pointeurs ni structures dynamiques, nous avons dû fixer une taille limite.

Nous avons ici fixé le nombre maximal de sites à 1024, la mémoire partagée étant initialisée en global, avant de recevoir le fichier de configuration. Un retour du client pendant le cycle de développement avant livraison du produit permettrait de l'adapter à la hausse ou à la baisse selon ses besoins.

Cependant, une fois le logiciel terminé, bien qu'un fichier de configuration indique le nombre de sites à effectivement initialiser, la taille en mémoire de cette structure ne peut plus varier.

De plus, nous avons une seconde raison plus pragmatique, parce qu'il faut savoir, que le segment mémoire est limité par la taille maximum d'un

$$\textit{unsigned long} - 2^{24}$$

Voir le manuel depuis Linux 3.16.

Étant donné l'intervalle d'un *unsigned long* qui est de [0, 4.294.967.295], si on y retire 2^{24} il nous reste quand même plusieurs milliers de megabytes, nous ne connaissons pas la mémoire disponible utilisable par le système au moment de l'exécution du programme ni son évolution dans le temps. On estime que notre application ne consommera que très peu d'espace en mémoire partagée, de l'ordre de quelques kilobytes dans le pire des cas et que donc elle n'est pas une nuisance pour le système d'exploitation.

Jusque-là, nous avons présenté les structures minimales nécessaires au bon fonctionnement de notre programme serveur.

Nous utilisons aussi la structure *union senum* pour assurer le bon fonctionnement de notre ensemble de sémaphores, mais cette structure est définie dans la bibliothèque *<sys/sem.h>* et non par nous-même, nous ne la détaillerons pas.

Nous allons conclure cette section sur les structures avec les deux suivantes. On y retrouve en premier la structure nécessaire au bon fonctionnement d'une file de message :

```
struct mesg_buffer {
    long mesg_type;
  int nbProcesseur;
  int nbCapaciteStockage;
};
```

Les messages déposés dans une file de message possèdent deux attributs entiers en plus de leur identifiant de type *long* spécifique à leur utilisation selon le manuel Linux.

En effet, nous utiliserons cette structure pour pouvoir nous sortir d'une situation délicate, qui est décrite plus loin dans ce rapport. Elle consiste à pouvoir réveiller un processus frère, côté serveur, afin de tenter à nouveau une prise de ressource qui n'avait pas pu être attribuée à la réception.

La deuxième structure définie par nos soins consiste à pouvoir échanger simplement les requêtes de réservation ou de libération de ressource :

```
struct reservation {
  int label;
  int nbProcesseur;
  int nbCapaciteStockage;
  int estPartage;      // 0 exclusif,     1 partage
  int estLiberation;  // 0 reservation, 1 liberation
} __attribute__((packed));
```

A posteriori, le nom est mal choisi, on aurait pu la nommer *requete* ce qui était le cas pendant quelque temps mais qui a été modifié au cours du débogage.

Nous faisons remarquer que le label n'est plus une chaîne de caractères mais est devenu un entier. La propriété invariante appliquée est la suivante :

L'ordre des sites ne change pas lors de son exécution.

Ce changement apporte quelques avantages non négligeables tels qu'une réduction du débit réseau consommé sur le long terme. Cette astuce amène aussi des simplifications techniques dans le code pour n'avoir à manipuler qu'un simple entier pour identifier le site, celui-ci indiquant le sémaphore et la case en mémoire partagée à modifier .

Et enfin on remarque l'utilisation d'un qualificatif pour la structure *___attribute___((packed))* qui est apparue après notre définition, lors de nos recherches dans le manuel et sur Internet, qui recommandait l'utilisation de cet attribut

pour "compacter" notre structure, ce qui a pour contrainte de ne pas autoriser certaines expressions C [1].

2.2 Les objets IPC, principes et utilisations

Notre développement nous amène enfin à aborder les objets IPC mis en place pour réaliser notre application, côté serveur ainsi que client.

Nous les présentons dans l'ordre d'instanciation dans le code.

File de messages

Étrangement nous commençons par les files de messages quand celles-ci ont été les dernières à être implémentées.

En effet, nous avons eu à utiliser les files de messages qu'en fin de développement. Nous nous sommes confrontés à un problème côté serveur : réveiller un processus frère qui s'était bloqué après une tentative de réservation de ressource qui n'a pas pu être satisfaite.

Comme nous l'avons indiqué précédemment sur la définition d'une structure d'un message, on y retrouve deux entiers, l'un qui indique la quantité de processeurs libérés, le second qui correspond à la quantité de stockage libéré.

Elle est définie globalement pour facilement être héritée lors d'un appel système à *fork()*. Elle est accessible via la variable *msgid*.

Enfin nous ne distinguerons qu'un seul type de message à travers notre application d'étiquette 1. Son utilité c'est de permettre de réveiller un autre processus qui s'est endormi à cause d'un manque de ressources à un instant T, à l'instant T + X, avec X un temps quelconque non fixé mais dépendant de l'activité des autres clients pouvant aller de quelques millisecondes à plusieurs minutes, voire des heures.

Mémoire partagée

La variable *ptrShm* la définissant est définie globalement. Les sites dont l'indice se situe aux delà de la valeur de l'attribut *taille* ne seront pas accessibles, leurs sémaphores n'existant pas.

Elle est consultée en écriture et/ou lecture par tous les processus lourds ou légers créés par le serveur.

Elle offre un aperçu en temps réel sur l'état des sites. C'est grâce à cette variable qu'on met à jour et stocke l'information sur les quantités totales, utilisées (exclusif et partagé) et qu'on calcule le nombre de ressources libres.

1. Voir le manuel pour plus d'explications et son utilité si jamais nous devions sérialiser cette structure pour un envoi via un socket.

Sémaphores

On possède un ensemble de sémaphores déclarés *semId*, avec tous les sémaphores initialisés à 1.

Les sémaphores protègent l'accès à un site en particulier dans la mémoire partagée pour s'assurer qu'il n'y ai qu'un seul accès à un instant donné pour garantir un état cohérent.

Nous avons donc implémenté nos fonctions P et V qui prennent en paramètres le jeu de sémaphores et l'identifiant du site ; P diminue de 1 et V augmente de 1 l'attribut *semval* associé à l'identifiant du site. Cela afin de ne pas bloquer l'intégralité de la mémoire lorsqu'on décide d'opérer sur un seul site, deux opérations sur des sites distincts ne posant pas de problème de cohérence.

2.3 Les processus, threads et leurs rôles

À présent on peut expliciter les détails techniques concernant le programme serveur et client.

Commençons par cette image inspirée du livre FreeBSD Developers' Handbook, qui schématise notre architecture pour la mise en place du serveur concurrent.

Vous trouverez en annexe, Figure 1 notre propre schéma, adapté à nos besoins qui illustre simplement le fonctionnement de notre serveur.

Nous pouvons commenter et compléter cette illustration via l'explication suivante.

En premier lieu, le serveur met en place des processus lourds, donc un appel système à Fork() pour chacun des clients. Par la suite, nous avons fait le choix du protocole de communication via TCP. Cela nécessite d'indiquer une taille maximale quant à la file d'attente de connexion sur le serveur. Nous avons donc limité via une macro C nommée *NB_CLIENT*, le nombre de connexions en attente d'acceptation en simultané à 50 clients. Cela n'impacte pas la quantité de serveurs fils créés.

Puis chacun des serveurs fils, met en place exactement deux threads pour son propre traitement et servir son client. Deux threads paraissent raisonnables sachant qu'un CPU moderne possède au moins 2 threads par coeur si ce n'est plus.

Le premier *thread* a pour rôle d'une part d'envoyer en premier lieu l'information complète concernant les sites stockés, puis d'autre part d'inspecter en continu les changements dans la mémoire partagée de manière cohérente avec la prise et la libération du sémaphore associé ; il envoie uniquement les modifications aperçues.

Le second *thread* joue le rôle du caissier, qui écoute le client en continu pour

satisfaire si possible les demandes de réservation et de libération, de façon également à ne pas rendre la mémoire incohérente.

D'autre part et du même genre que le serveur, le client met en place seulement 2 threads dont l'un reçoit les mises à jour de son serveur fils et l'autre dialogue unilatéralement avec son caissier.

Serveur

La toute première action du serveur est d'initialiser la liste des sites, il passera ensuite à l'écoute des clients en continu.

Pour cette initialisation, nous avons fait le choix de proposer deux solutions. La première consiste en une liste codée en dur pour permettre le test du programme par une personne n'ayant pas de fichier de configuration. Il est cependant possible de fournir un fichier pour respecter le cahier des charges.

Pour cette solution nous avons fait le choix de lire via l'entrée standard, qu'il faut donc rediriger via un :

```
$ cat <fichier> | ServeurConcurrent <port> <cleIPC> 1
```

Le dernier paramètre *<1>*, indique que nous fournissons les informations des sites soit via un fichier de configuration, soit via une saisie clavier selon ce qui a été dit juste avant.

Ce choix réfléchi n'est aucunement dû à une méconnaissance de la librairie de lecture sur le disque, mais ce programme ayant aussi un accès réseau, nous voulions qu'une personne n'ayant pas lu tout le code mais uniquement les *includes* ne refuse pas de l'exécuter.

Un programme avec un accès disque et réseau, peut voler des fichiers sensibles sur l'ordinateur de l'enseignant ou de l'étudiant effectuant la correction par les pairs. Ces pratiques qui consistent à utiliser des -lectures sur l'entrée standard- et indiquer en première ligne le nombre de lignes qui suivent plutôt que d'attendre un *EOF*, suivent les spécifications des différents services compilant un code soumis par un utilisateur et l'exécutant sur leur plate-forme tel que https://battledev.blogdumoderateur.com et https://onlinejudge.org.

Le serveur fils traite les requêtes qu'il reçoit en faisant les modifications nécessaires en mémoire partagée.

Avant chaque accès à cette mémoire, une demande d'accès exclusif au thread du serveur fils est fait via le sémaphore correspondant au site impacté. Si la demande est exclusive, le programme soustrait le nombre de ressources demandées, s'il y en a suffisamment de disponibles, ou alors il attend une libération pour le faire via un envoi de messages par file de messages à tous les processus en attente.

Si un processus en attente vient à lire ce message et qu'il ne permet pas de

satisfaire sa demande, il le remet dans la file de message et retourne attendre un autre message. Éventuellement, le programme garantit que sa demande se verra satisfaite quand la quantité de clients ou de ressources utilisées tend vers zéro grâce à une vérification préalable côté serveur de la quantité demandée.

En effet, il est évident qu'un client demandant plus de ressources que ce qu'un site ne dispose ne verra jamais sa demande traitée.

Les demandes de libération peuvent être traitées dès la réception, une fois un accès à la mémoire partagée obtenu.

Les demandes en ressources partagées n'affectent que la différence nécessaire entre ce qui est déjà partagé et ce qui est demandé.

Client

Le client se connecte au serveur avec une adresse IP et un port passé en paramètre.

Suite à ça, un thread est responsable de recevoir en continu les mises à jour des données du serveur, cependant, elles ne sont pas affichées en continu, mais à la demande du client. Le client étant sur un terminal et non sur une interface graphique permettant de séparer en deux zones distinctes l'entrée et la sortie, nous souhaitons ainsi éviter de perturber une saisie au clavier de l'utilisateur.

Un deuxième thread permet de demander si l'on veut afficher les dernières données reçues ou saisir une requête et l'envoyer. S'il n'y a pas suffisamment de ressources disponibles, c'est le thread dédié au client qui est bloqué côté serveur, temporisant donc ses requêtes suivantes dans les limites imposées par TCP du modèle TCP/IP.

2.4 Protocole de communication mis en place

Le serveur est en écoute sur un port connu de tous. Lorsqu'un client se connecte, le serveur crée un descripteur de fichier dédié au client via la fonction *accept()*, puis il se divise via un appel système Fork() alors il ferme la socket nouvellement créée pour ne pas surcharger la table des fichiers ouverts, comme il a été schématisé en Figure 1, par la suite le serveur fils aura deux threads créés, le premier sera responsable de diffuser les mise à jour, le deuxième de recevoir les requêtes.

Nous avons ajouté en annexe, Figure 2 le diagramme de séquence qui décrit très simplement le protocole de communication mis en place qui met en œuvre un exemple de communication.

Voici une explication de ce diagramme.

Threads

Le serveur fils possède deux rectangles, un hachuré rouge et un rectangle vide jaune symbolisant respectivement le thread de mise à jour et le thread de requêtes de réservations/libérations.

Réception des requêtes clientes

En boucle, le thread s'attend à recevoir une requête de taille *struct reservation*, on reçoit la *reservation* qui aura été correctement construite côté client (c'est-à-dire après la saisie au clavier des valeurs nécessaires par ce dernier, tel que l'id de la ville, le nombre de processeurs à réserver, le nombre de stockages à réserver et le mode et le type de la réservation, respectivement exclusif/partagé et prendre/libérer).

Le traitement continue comme nous l'avons explicité précédemment lors de l'explication des threads.

Dans le protocole de communication, nous avons défini un message de déconnexion. Pour cela, il faut formater côté client la valeur *-1* sur l'identifiant de la ville indiquant alors une déconnexion.

Mise à jour des données

Le deuxième thread scrute une première fois la mémoire partagée, il envoie ensuite toutes les informations, concernant les sites. Premièrement le nombre de modifications, puis en boucle, l'indice du site modifié suivi de son contenu.

Du façon similaire, le client répond et traité de l'information de façon appropriée.

Processus

Le deuxième protocole de communication intervient entre deux fils du serveur, soit deux processus lourds indépendants, mais appartenant au même *group id*.

Nous l'avions brièvement évoqué dans la section sur les objets IPC. Nous disposons d'une file de messages qui implémente le protocole de communication suivant.

Lorsque le serveur a reçu une demande de réservation qui n'est pas satisfaite dès sa réception le thread du serveur fils qui est responsable de cette requête se met en attente d'une libération de ressource. Par la suite, si un second thread d'un second client libère les ressources, il génère un message de type 1 avec la quantité de processeurs et de stockages libérés. Puis le thread élu pour recevoir le précédent message, compare ses besoins à la quantité libérée.

Si cette quantité n'est pas suffisante, il remet le message à l'identique dans la file de message et se remet en attente.

Sinon il consomme le message et recommence son itération depuis l'étape pour ajouter sa réservation au système.

3 Conclusion

Pour conclure ce rapport sur le travail effectué lors de ce projet, nous présentons les fonctionnalités manquantes et les améliorations possibles.

Parmi les fonctionnalités non implémentées, nous retrouvons :

- la possibilité de faire plusieurs réservations de sites avec une seule requête.
- le support de l'affichage en continu des mises à jour sans casser la saisi clavier.
- un historique des réservations faites par un client connecté, qui a pour conséquence directe de ne pas permettre de libérer des ressources partagées.

Par ailleurs, il n'était pas indiqué dans le sujet, mais nous avons implémenté la gestion du signal émis par les combinaisons $CTRL + C$, correspondant au signal *SIGINT* qui permet non plus de tuer sauvagement le serveur, mais de libérer toutes ressources prises par son exécution avant de se tuer.

4 Annexe

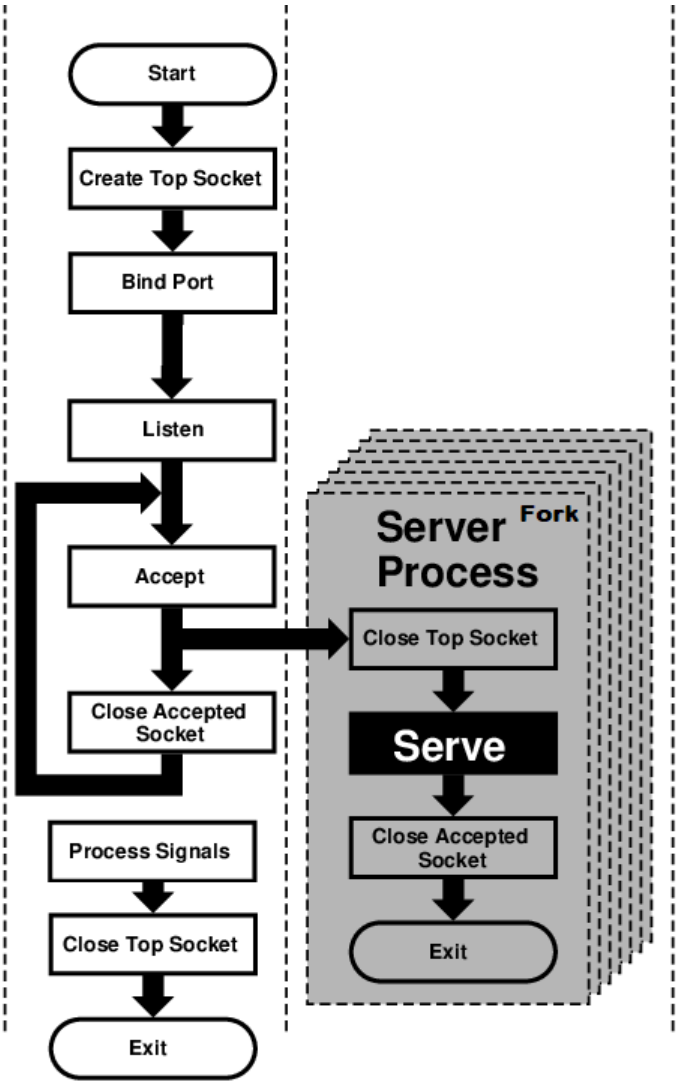

FIGURE 1 – Schéma d'exécution du serveur concurrent

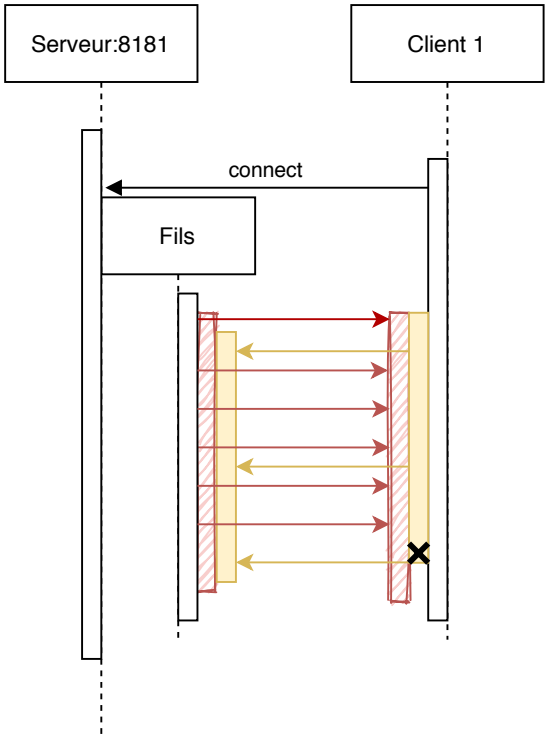

FIGURE 2 – Diagramme de séquence du serveur concurrent et d'un client