Rapport : SAE - Système

Groupe : Daniel YANG & Michel BUI (Binôme - TP3 tous les deux)

Le projet a été réalisé principalement pendant les vacances.
Daniel YANG était absent lors des deux premières semaines de vacances.

Michel BUI a donc principalement codé et Daniel a suivi la progression, écrit les Makefiles et le rapport.

Sujet

Un client désire que nous créions et lancions un serveur qui gère une base de données contenant des jeux. Il a une exigence forte, le serveur doit pouvoir exécuter des opérations en parallèle tout en limitant la consommation de mémoire et en réduisant au maximum les temps d'exécution.

Ainsi, on développera trois versions différentes et simplifiées afin de répondre à la demande du client.

Architecture du projet (Git)

Notre projet Git est constitué de dossiers correspondants aux principales versions du projet.
Chaque dossier aura des sous-dossiers pour spécifier les différents types de versions (V1.a, V1.b, V2.a, etc.)
Chaque version possède un *main* pour tester les opérations.

Makefiles

Afin de tout compiler facilement, on a un **Makefile** par version. Chaque **Makefile** sera différent puisque toutes les versions utilisent des processus différents ; comme base de Makefile, nous avons utilisé celle de la ressource ***P31*** puis remodifié si nécessaire.

Les Makefiles de la V0 et la V1.a ne sont pas modifiés.
Le Makefile de la V1.b utilise des threads, on ajoute donc le flag –**lpthread.**
Les Makefiles des V2 sont particuliers : chaque serveur qui se spécialise dans une opération contient un main. On a alors recréé le Makefile de zéro ; on utilise le principe d'exécution ***all*** pour générer plusieurs fichiers exécutables, chaque serveur et le main sont compilé avec des paramètres spécifiques dans le Makefile.
Le Makefile de la V3 suit un peu le principe des V2 avec un server et un client ; il a aussi le flag –**lpthread**.

Codé par Daniel YANG

Version 0 : Base

La V0 est la base de code qu'on réutilisera pour les autres versions.
Elle comporte les structs principaux ***jeu*** et ***demandeOperation*** dans le fichier *utils.h* et la fonction ***execute_demande(demandeOperation)*** dans un fichier *utils.c.*
C'est la base de code qu'on réutilisera pour les autres versions.

La fonction ***execute_demande*** utilise un *switch case* afin de savoir quel est la demande de l'utilisateur.

Afin de **simuler un temps d'action**, on bloque le serveur pendant un moment à l'aide de la fonction ***sleep()***.

Le fichier *main.c* contient les tests des 6 opérations que l'utilisateur peut demander.

Codé par Michel BUI

Version 1 : Monolithique

Pour la V1.a, le serveur doit être capable de lancer plusieurs opérations en parallèles tant qu'elles ne sont pas bloquantes (opérations 4, 5 & 6).
Pour savoir si une opération est non-bloquante, le "flag" dans le struct demandeOperation sera à 0.

A) Sans partage de mémoire

Nous utilisons des forks pour exécuter les opérations.
Le père possède donc plus de variables globales puisque les fils feront des copies du père pour pouvoir exécuter leur part.

Afin de communiquer les résultats avec le père, les fils utilisent des pipes ; en fonction de l'opération à effectuer, on utilisera potentiellement plusieurs pipes.

Pour éviter tout problème dans la base de données, seul le père pourra exécuter l'operation 4 (supprimer un jeu) ; on vérifie d'abord si l'utilisateur demande l'opération 4 avant de faire le switch case.
Si une opération est bloquante, le père utilisera ***waitpid*** pour attendre la fin d'exécution de son fils.

Codé par Michel BUI

B) Avec partage de mémoire

Nous utilisons des threads pour exécuter les opérations.
Les threads partagent le même espace mémoire que leur père, on aura donc plus de variables globales qui seront accessible par tous.

On a créé un struct *threadResult* afin de gérer les threads non-bloquants. Elles possèdent un booléen *estFini* si le thread est terminé, un int *result* qui correspond au bon du thread afin et *tid* afin de garder le pid du thread en cours d'execution.

Si un thread est bloquant, on utilise directement un *pthread_join* pour attendre la fin de l'exécution du fils.

Codé par Michel BUI

Version 2 : Ensemble de serveurs spécialisés

Pour cette version, on a créé six fichiers supplémentaire correspondant à un serveur qui se spécialisera dans une seule opération.
Ces fichiers/serveurs contiennent un *main* qui effectue l'opération demandé.

A) En utilisant execv

Execv nécessite le chemin vers le fichier ainsi que des arguments (optionnel, mais important dans notre cas).
Les arguments nécessaires à l'exécution de la fonction sont stockés dans un tableau puis passé en paramètre de la fonction execv.
Pour savoir quelles informations, on remplit le tableau à l'aide un switch case.

Pour les opérations bloquantes, on utilise des *waitpid* comme pour la version v1.a.

Codé par Michel BUI

B) En utilisant des pipes (nommés)

Les pipes nommées sont créés lors du lancement des servers ou du main.
On utilise un #include <sys/stat.h> pour utiliser ***mkfifo*** qui crée les pipes nommés; on devra ajouter un *#define _POSIX_C_SOURCE 200809L* suivi d'un #include <signal.h> pour des soucis de compatibilité avec les signaux (200809L signifie qu'on utilise la version de 2008 ; le server phoenix de l'iut n'est pas à jour ?).

Les signaux sont utilisés par les fils pour prévenir le père qu'ils ont terminé leur exécution.

Codé par Michel BUI

Version 3 : Utilisation de machines distantes

Fonctionnement client/server

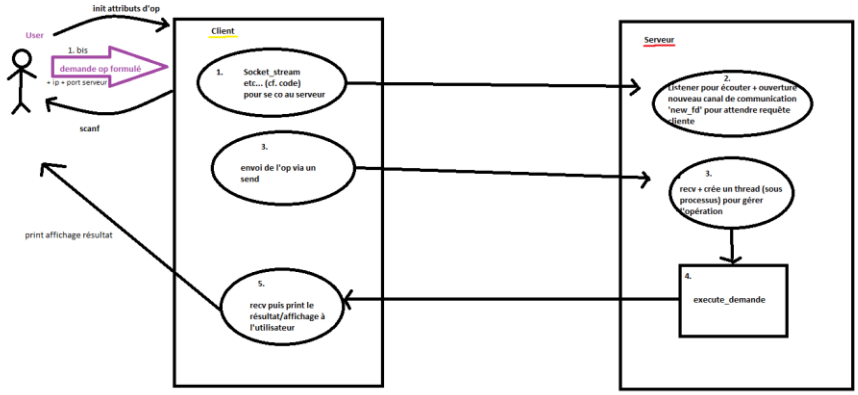
La connexion entre le client et le server se fait par des sockets ; l'utilisateur entrera obligatoirement une IP et un port de server afin de pouvoir communiquer avec ce dernier.

A sa création, le client possède un thread permanent qui écoutera les résultats des opérations non-bloquantes ; les opérations bloquantes sont directement lues par le client lui-même à travers un *recv()*.

Le server est en écoute permanente comme le thread du client, cependant il créera un thread afin d'effectuer l'opération demandé par le client via un *recv()* aussi et l'envoyer depuis ce thread

Codé par Michel BUI

Schéma du fonctionnement



PS: l'image sera déposée sur le Moodle pour plus de lisibilité.

Table des matières