INFO-F-201 – Systèmes d'exploitation

Projet de programmation système

SmallDB

TinyDB était une bonne solution, mais les contraintes ont changé.
Il faudrait maintenant une base de données suivant un modèle client-serveur : un programme serveur doit gérer la base de données et traiter des requêtes envoyées par des clients, potentiellement distants.

Ce projet est à réaliser par groupe de deux ou trois étudiants et comporte deux parties :

- le serveur gérant la base de données, celui-ci comprenant un programme principal en C/C++ avec la librairie pthread et un script bash pour gérer et consulter l'état de la base de données,
- un client écrit en bash ou en C/C++.

Contactez Yannick Molinghen avant le 25 novembre 2022 si vous n'avez pas de groupe. Vous serez pénalisé si vous réalisez le projet seul sans en avoir discuté auparavant.

Vous êtes autorisés à utiliser des fonctionnalités de C++ telles que `vector`, `string`, `hashmap`, `new` et `delete`, les classes, etc [1].

1 La base de données (programme serveur)

Cette section détaille le fonctionnement de votre programme de base de données (c.-à-d. le serveur) dont l'exécutable doit s'appeler `smalldb`.

1.1 Contenu

Le contenu de la base de données est le même que pour le premier projet, à savoir : une unique table d'étudiants, chaque étudiant étant représenté par la structure dans la table 1.

student		
Champ	Type	Signification
id	`unsigned int`	Identifiant unique
fname	`char[64]`	Prénom
lname	`char[64]`	Nom
section	`char[64]`	Section (informatique, médecine, …)
birthdate	`struct tm`	Date de naissance

TABLE 1 – Représentation d'un étudiant. Le header `student.h` fourni définit une structure de type `student_t` correspondante.

1. Ce projet évalue les compétences en programmation système, vous ne serez donc pas récompensés pour avoir implémenté une magnifique chaîne d'héritage car ce n'est pas le sujet de ce cours.

1.2 Format des requêtes

Vous devez gérer les mêmes quatre types de requêtes que pour le projet 1, mais non les transactions. Vous trouverez des exemples dans le répertoire `tests/queries`. Vous trouverez dans la table 2 le détail sur les réponses à fournir aux requêtes. Les requêtes mal formées doivent renvoyer un message d'erreur (au client), mais ne doivent pas interrompre le programme. Voyez la figure 1 pour un exemple d'utilisation du client.

1.3 Lancement de la base de données

La base de données doit se lancer via l'exécutable `smalldb` et prend en paramètre obligatoire le chemin vers la base de données. Au lancement, le contenu du fichier donné en paramètre doit être chargé. (Une fonction vous est proposée pour le chargement de la base de données dans les fichiers mis à votre disposition.) Si le fichier n'existe pas, un avertissement est affiché, et la base de données est initialisée vide : le fichier sera créé au moment de la sauvegarde. Le programme doit ensuite attendre des connexions TCP sur le port 28772 de toutes les interfaces réseaux.

1.4 Traitement des requêtes

Chaque client doit être pris en charge par un thread différent. On vous demande donc de créer un nouveau thread à chaque connexion de client. Le thread doit se finir après la déconnexion de celui-ci.

Pour chaque client, le thread doit recevoir les requêtes, les traiter et retourner un résultat sous la forme d'un texte que le client peut afficher. La table 2 indique ce que chaque requête devrait afficher.

Requête	Résultat
`select`	Les étudiants correspondants, un par ligne, suivi du message « n student(s) selected »
`insert`	L'étudiant inséré
`delete`	« n student(s) deleted »
`update`	« n student(s) updated »

TABLE 2 – Résultat attendu pour chaque type de requête.

En cas d'échec, un message commençant par « Error : » et décrivant l'erreur doit être envoyé. Notez que ce n'est pas une erreur qu'un `select` ne retourne aucun étudiant.

1.5 Fin de la base de données

Lorsque la base de données est fermée, le contenu de la base de données doit être sauvegardé sur le disque dans le même format que celui utilisé pour le projet 1. La seule fermeture prévue de la base de données a lieu quand votre processus reçoit le signal `SIGINT` (envoyé par exemple avec `CTRL+C`).

1.6 Sauvegarde sur le disque

La base de données doit intercepter les signaux `SIGUSR1` qui lui sont envoyés et y réagir en sauvegardant le contenu de la base de données sur le disque dans le même format que celui utilisé pour le projet 1.

```
> select fname=Diane
000472905: Diane Vancauwelaert in section pharma, born on the 08/08/1998
1 student(s) selected
> select birthdate=12/12/1994
000111111: Bastien Durandal in section pharma, born on the 12/12/1994
000234556: Corentin Francois in section physics, born on the 12/12/1994
2 student(s) selected
> update section=info set section=truc
3 student(s) updated
> nocommand
Unknown query type
> update
Syntax error in update
> update id=1 set id=2
0 student(s) updated
> update id=1 set id=a
You can not apply id=a
> delete birthdate=30
birthdate=30 is not a valid filter
> select section=true
0 student(s) selected
> select section=truc
000864030: Hadrien Legast in section truc, born on the 30/09/1998
000928453: Fabienne Deschamps in section truc, born on the 06/07/1997
008492742: Anatole Lanegris in section truc, born on the 07/05/1999
3 student(s) selected
>
```

FIGURE 1 – Exemple d'utilisation du client

1.7 Sortie standards du serveur

En plus de l'affichage du nombre d'étudiants dans la base de données au lancement, le serveur devrait utiliser ses flux de sorties standards (stdout et/ou stderr) pour informer des connexions et déconnexions des clients, ainsi que des potentiels problèmes rencontrés (autre que des erreurs dans les requêtes reçues).

1.8 Gestion des erreurs

Il y a deux types d'erreurs. Les erreurs de syntaxe dans les requêtes ne doivent pas provoquer autre chose que l'affichage d'un message d'erreur par le client.

Vous devez aussi gérer les autres erreurs qui pourraient survenir, lors d'un appel système par exemple. Suivant le cas, vous devrez terminer le processus (par exemple, impossible d'écouter sur le port choisi), seulement un thread (par exemple, connexion avec le client perdue) ou continuer si vous êtes capable de gérer l'erreur.

Une erreur dans un thread ne devrait pas interrompre tout le processus.

```
smalldb: DB loaded (/tmp/test_db.bin): 9 students in database
smalldb: Accepted connection (4)
smalldb: Accepted connection (5)
smalldb: Client 5 disconnected (normal). Closing connection and thread
smalldb: Lost connection to client 4
smalldb: Closing connection 4
smalldb: Closing thread for connection 4
smalldb: Accepted connection (7)
smalldb: Accepted connection (4)
smalldb: Client 4 disconnected (normal). Closing connection and thread
smalldb: Client 7 disconnected (normal). Closing connection and thread
```

FIGURE 2 – Exemple d'affichage par le serveur. Ici nous avons choisi d'identifier les connexions dans les messages à l'aide du file descriptor retourné par `accept`. Pour cette raison, le même identifiant peut être utilisé plusieurs fois. Vos messages d'erreurs devraient aussi proposer une façon de différencier les connexions.

1.9 Accès partagé à la base de données

Les threads doivent travailler de façon concurrente sur la base de données. On vous demande que plusieurs lectures (`select`) puissent avoir lieu en même temps. En revanche, lorsqu'une écriture a lieu (`update`, `insert`, `delete`), aucun autre accès à la base de données ne doit avoir lieu, que ce soit en lecture ou en écriture. Vous devez en outre veiller à ne pas provoquer de famine dans le cas où des requêtes en lecture et en écriture arriveraient en continu (par ex. si des requêtes en lectures arrivent en continu, il faut tout de même qu'une requête en écriture puisse s'exécuter). Voyez la section 6.2 pour une solution théorique à ce problème.

2 Monitoring

Vous devez fournir un script bash `smalldbctl` capable de trois actions :

- `smalldbctl list` indique l'adresse IP des différents clients connectés,
- `smalldbctl sync` envoie le signal `SIGUSR1` à smalldb,
- `smalldbctl stop` envoie le signal `SIGINT` à smalldb.

2.1 `smalldbctl list`

La commande `list` doit indiquer l'adresse IP des différents clients connectés. Comme première piste, vous pouvez regarder les outils `netstat` et `ss`, ainsi que partir de la commande `ss --no-header -Ontp4 'sport = :28772'`.

2.2 `smalldbctl sync` et `smalldbctl stop`

Les commandes `sync` et `stop` envoient un signal (`SIGUSR1` et `SIGINT` respectivement) à smalldb.

3 Client

Vous devez écrire un client `sdbsh` (pour *SmallDB SHell*) en bash ou en C/C++ qui permette à un utilisateur d'entrer des requêtes et de recevoir les résultats. Le client doit se lancer avec comme paramètre obligatoire l'IP du serveur.

4 Rapport

Vous devez fournir un rapport qui contiendra :

- les choix d'implémentation (si nécessaire), les difficultés rencontrées et les solutions originales que vous avez fournies,
- une explication détaillée des mécanismes de synchronisations utilisés,
- une comparaison détaillée entre l'utilisation des processus et des threads (projets 1 et 2), avec une discussion sur les avantages et inconvénients des deux méthodes,
- des propositions d'améliorations relatives au fonctionnement et aux performances (non aux fonctionnalités).

5 Ce qui est mis à votre disposition

Vous pouvez télécharger la base du projet sur l'Université Virtuelle afin de ne pas commencer de zéro (**ce n'est pas obligatoire,** et vous pouvez modifier les fonctions qui y sont données). Ce répertoire contient :

- quelques fichiers C++ pour ne pas partir de zéro,
- une base de données de plusieurs milliers d'étudiants dans `students.zip`,
- des tests automatiques réalisés par le script `tests/run_tests.sh`,
- un Makefile à compléter.

6 Pour vous aider

6.1 Signaux et multithreading

Notez que, pour un signal donné, un processus ne peut avoir qu'un seul gestionnaire par signal. Celui-ci est défini par le dernier appel à `signal` ou `sigaction` ayant eu lieu. En revanche, le thread exécutant le signal handler est choisi arbitrairement par l'OS.

Pour vous assurer que `SIGINT` et `SIGUSR1` soient reçus par le thread principal, vous devrez utiliser la fonction `pthread_sigmask` dans le thread principal pour les bloquer avant chaque pthread_create et les débloquer après. Puisque la liste des signaux bloqués (non reçus) est héritée par les threads fils créés par un `pthread_create`, ceci permet de s'assurer que les fils ne reçoivent jamais les signaux qu'ils ne doivent pas gérer.

Si un signal est envoyé alors qu'il a été marqué comme bloqué par tous les threads, il sera mis en attente puis transmis au premier thread qui débloquera ce type de signal. Voyez le listing en figure 3 pour un exemple.

```c
void handler(int signum) {
  printf("Signal %d received", signum);
}
// ...
  // Définit le signal handler
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  sigaction(SIGUSR1, &action, NULL);

  // Bloque le signal (pour le thread courant)
  sigset_t mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGUSR1);
  sigprocmask(SIG_BLOCK, &mask, NULL);

  // Si un pthread_create est fait ici, le signal est bloqué pour les deux threads

  // Débloque le signal (pour le thread courant)
  sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

FIGURE 3 – Exemple : gestion des signaux en multithreading

6.2 Accès partagé

Nous vous donnons dans le pseudo-code ci-dessous les grandes lignes pour réaliser la synchronisation des threads dans votre code. Nous n'expliquons pas pourquoi cette solution fonctionne, ni le but de chacun de ces mécanismes (c.-à-d. pourquoi nous avons besoin de chacun de ces mutex / sémaphore ici). Vous devez fournir cette explication dans votre rapport.

```
mutex new_access = unlocked;
mutex write_access = unlocked;
mutex reader_registration = unlocked;
int readers_c = 0;

// Writer
lock(new_access);
lock(write_access);
unlock(new_access);
// ... WRITE OPERATIONS
unlock(write_access);

// Reader
lock(new_access);
lock(reader_registration);
if readers_c == 0
    lock(write_access);
readers_c++;
unlock(new_access);
unlock(reader_registration);
// ... READ OPERATIONS
lock(reader_registration);
readers_c--;
if readers_c == 0
    unlock(write_access);
unlock(reader_registration);
```

7 Critères d'évaluation

Votre projet doit compiler soit avec g++ version 9.4 (ou ultérieure), soit avec `clang` version 12 (ou ultérieure). Les options de compilations obligatoires sont celles listées ci-dessous (présents dans le Makefile fourni). Si votre programme ne compile pas, vous recevrez une note de 0/20.

```
-std=c++17 -Wall -Wextra -Wpedantic -D_GNU_SOURCE -Werror=all -lpthread
```

7.1 Pondération

- Tests automatiques **/3**
- Ce projet de programmation système va principalement évaluer votre compétence à manier correctement les outils liés aux systèmes d'exploitation (threads, fichiers, sémaphores, ...) dans le langage C. La pertinence des outils utilisés ainsi que la manière dont ils sont utilisés (trop, pas assez, au mauvais endroit, trop longtemps, …) sont évalués. **/5**
- Le client **/2**
- Le script smalldbctl **/1**
- Ce projet doit contenir un rapport dont la longueur attendue est de quatre pages (max. 8). **Reportez-vous à la section 4 pour ce qui y est attendu.** **/7**
  - Orthographe
  - Structure
  - Légende des figures
- Votre code sera aussi évidemment examiné en termes de clarté, de documentation, de commentaires et de structure. Vos fonctions devraient idéalement contenir maximum 30 lignes de code. L'utilisation de fonctions de plus de 50 lignes sera pénalisée. De plus, evitez les répétitions des mêmes lignes de codes dans différentes fonctions. **/2**

8 Remise du projet

Vous devez remettre un projet par groupe contenant un fichier zip contenant
- vos sources
- un Makefile
- votre rapport au format PDF
- les tests que vous auriez écrits

N'incluez ni le fichier de base de données de milliers d'étudiants ni vos logs dans le fichier zip !

Vous devez soumettre votre projet sur l'**Université Virtuelle** pour le **17 décembre 2022 23 h 59** au plus tard.

Retards

Tout retard sera sanctionné d'un point par tranche de 4 h de retard, avec un maximum de 24 h de retard et devra être soumis sur l'université virtuelle.

Questions

Le meilleur moment pour poser vos questions sera pendant les séances de travaux pratiques. Vous pouvez cependant aussi poser vos questions sur l'UV ou via email à alexis.reynouard@ulb.be si nécessaire.