FT2245 - Systèmes d'exploitation

Travail Pratique 2 (20%)

DIRO - Université de Montréal À faire en équipe de deux.

Disponible : 29/02/2016 - Remise : 21/03/2015 avant minuit (23h59)

Introduction

Pour ce TP, vous devez implémenter en C l'algorithme du banquier, un algorithme qui permet de gérer l'allocation des différents types des ressources, tout en évitant les interblocages (deadlocks). Vous pouvez faire le travail demandé en équipe de deux. Le code fourni implémente un modèle de client-serveur qui utilise les prises (sockets) comme moyen de communication. D'un côté l'application serveur est analogue à un système d'exploitation qui gère l'allocation des ressources, comme par exemple la mémoire, le disque dur ou les imprimantes. Ce serveur reçoit simultanément plusieurs demandes de ressources des différents clients à travers des connexions. Pour chaque connexion/requête, le serveur doit décider si les ressources peuvent être allouées ou non au client, de façon à éviter les interblocages en suivant l'algorithme du banquier. De l'autre côte, l'application client simule l'activité de plusieurs clients dans différents fils d'exécution (threads). Ces clients peuvent demander des ressources si elles sont disponibles ou libérer des ressource qu'ils détiennent à ce moment.

Ce TP vous permettra de mettre en pratique quatre sujets du cours différents :

1. Fils d'exécution multiples (multithreading)
2. Prévention des séquencements critiques (race conditions)
3. Évitement d'interblocages (deadlock avoidance)
4. Communication entre processus via des prises (sockets).

Mise en place

Un ensemble des fichiers de code est fourni pour vous aider à commencer ce TP. On vous demande de travailler à l'intérieur de la structure déjà fournie. Tout code se trouve dans le dossier `src`.

Les deux applications, client et serveur, utilisent les librairies du standard POSIX pour l'implémentation des structures dont vous allez avoir besoin, notamment les sockets, les threads, les semaphores et les mutex. Ces librairies sont par defaut installées sur une installation du système Linux.

Pour faciliter le débogage (et l'évaluation) on utilisera un fichier de configuration (`<program>/conf.c`) auquel les deux applications ont accès pour récupérer les

paramètres de l'algorithme du banquier et des configurations additionnelles. Comme ce fichier de configuration spécifie les constantes il est nécessaire de recompiler les 2 applications après tout changement.

Il est suggéré d'utiliser un éditeur de texte avancé pour éditer votre code et avoir en tout temps deux consoles ouvertes pour compiler et exécuter chaque application séparément. Pour la compilation en console de chaque application, vous pouvez utiliser la commande `make` à partir du dossier `src`.

Makefile

Pour vous faciliter le travail, vous pouvez utiliser le makefile qui se trouve dans le dossier `src` qui offre les fonctionnalité suivantes.

```
all         Compile les deux applications.
client      Compile uniquement le client.
server      Compile uniquement le serveur.
format      Formatte le code correctement pour la remise.
release     Archive le code dans un tar.gz pour la remise.
valgrind    Lance 2 fois l'exécutions des clients et serveur.
run         Lance le client et le serveur.
run_timed   Lance le client et le serveur avec un timeout de 10s.
            Il est possible de spécifier la durée du timeout
            avec la commande
                make run TIMEOUT=<time>
            La valeur <time> doit être une valeur valide pour la
            commande timeout (Voir 'man timeout').
clean       Nettoie le dossier build.
```

Protocole client serveur

Le protocole entre le client et le serveur est dirigé par le client qui fait des requêtes au serveur.

Celui-ci initialise l'échange en annonçant le nombre de ressources dont il a besoin avec la requête `BEGIN`. Le serveur doit alors répondre soit en acceptant la requête (`ACK`), donnant un délai pour rééssayer la requête (`WAIT uint32`) si un autre client a déjà réservé le seurveur ou bien en refusant (`REFUSE`) si le serveur est incapable de supporter le nombre de ressources demandées (si malloc ne peut allouer les ressources demandées par exemple).

Requête du client	Réponse du serveur
BEGIN *num_resources:uint32* num_clients:uint32*	ACK

Requête du client	Réponse du serveur
	WAIT *seconds:uint32* REFUSE
CLOSE	ACK

Chaque thread client s'annonce au serveur en lui donnand le nombre de ressources maximales dont il a besoin. Si la requête est valide, le serveur accepte avec (`ACK`). Si le nombre de ressources dépasse ce que le serveur peut gérer, le serveur refuse le thread client (`REFUSE`).

Ensuite, chaque thread client envoie des requêtes de ressources que le serveur peut accepter si l'algo du banquier le permet (`ACK`) ou mettre en attente (`WAIT`). Si le nombre de ressources dépasse le maximum attendu pour le thread client, le serveur doit refuser la requête (`REFUSE`).

Une fois la requête attendue et acceptées, le serveur "effectue le traitement". Une fois le traitement terminé, il envoie un message `END` au thread.

Le thread peut alors annoncer une autre requête ou bien qu'il a fini avec le message (`END`) que le serveur doit accepter.

Requête des clients	Réponse du serveur
INIT *tid max_ressources:uint32[num_ressources]*	ACK WAIT *seconds* REFUSE
REQ *tid int32[num_ressources]*	ACK WAIT *seconds* REFUSE
	END
END *tid*	ACK

*Note les données dont le type n'est pas spécifié sont de type `uint32`

Une fois tout les threads terminé, le client doit envoyer une requête `CLOSE` que le serveur doit approuver avec la requuquête `ACK`. S'il reste des ressources, le serveur les libère mais ne considère pas que le client s'est correctement terminé. Le serveur effectue alors l'impression du journal à l'intérieur d'un fichier et le client fait de même à la réception de `ACK`.

Le serveur peut alors se fermer ainsi que le client.

Description du projet

Dans le code source fourni, vous trouverez trois fichiers de code pour chaque application, en plus des fichiers 'common.h' et 'conf.h'.

Chaque application a son dossier où se trouve un `main.c` ainsi qu'un fichier de fonction

La plus grande partie de votre travail se concentre dans les fichiers 'server_threads.c' et 'client_thread.c' sur les fonctions délimitées par TODO et TODO END. Ça ne veut pas dire que vous ne devez rien modifier ailleurs, mais c'est plutôt un guide pour commencer. Les prises (sockets) qu'on utilise dans le protocole TCP/IP sont supposées de faire communiquer des applications dans différents ordinateurs sur un réseau, en utilisant la direction IP du serveur, mais pour faciliter votre implémentation on utilise la direction d'IP locale ('localhost'), pour se connecter sur le même ordinateur et un port arbitraire disponible (normalement plus grand que 2000) qui est spécifié dans le fichier de configuration dans `conf.h`. Le nombre de ressources différentes disponibles dans le serveur est défini dans le fichier de configuration dans numResources, cette valeur aide le serveur et le client à allouer la mémoire nécessaire pour les structures des données, vous n'avez pas à vous inquiéter pour l'allocation et la libération de cette mémoire.

Application serveur

La structure `server_thread` (fichiers : `server_threads.c` et `server_threads.h`) implémente le serveur, et elle inclut déjà les structures des données de base pour implémenter l'algorithme du banquier, dont on a parlé au cours de la démo. Le serveur fourni utilise plusieurs threads pour recevoir les requêtes. Le nombre de threads ainsi que la taille de la file d'attente de chaque thread sont spécifiés dans le fichier de configuration avec les champs : `server_threads` et `server_backlog_size`. Essayez votre application avec différentes configurations de ces valeurs pour modifier la charge que votre serveur peut recevoir.

server_threads.c

La fichier `server_threads` est le principal à compléter pour le serveur.

Vous trouverez à l'intérieur les structures nécessaires pour gérer les threads et les sockets.

Vous trouverez ainsi les structures de données que vous avez à remplir et à modifier pour implémenter l'algorithme du banquier.

Le code donne des commentaires pertinents pour expliquer chaque fonction. Quand un client fait une requête au serveur, la fonction process_request() est appelée.

Les clients devraient faire un certain nombre (spécifié par `num_request_per_client`) de demandes et de libérations de ressources de façon pseudo-aléatoire (d'ailleurs, ils ne devront jamais libérer des ressources qu'ils n'ont pas).

À la fin, ils doivent toujours libérer toutes les ressources qu'ils ont prises. C'est-à-dire, si par exemple les clients doivent faire N requêtes, la première (0) est toujours une demande de ressource, et la dernière (N - 1) est toujours une libération de toutes les ressources prises par ce client.

Entre ces deux, les autres peuvent être des libérations ou des demandes. Le serveur doit accepter la requête seulement si elle est valide et si elle laisse le système dans un état sécuritaire.

Il peut y avoir trois types des requêtes :

- Accepted : Les requêtes acceptées doivent retourner au client la valeur 0, pour lui confirmer que les ressources demandées ont été allouées. Après vous devez coder les instructions nécessaires pour mettre à jour les valeurs des ressources dans le système.
- OnWait : Les requêtes qui sont valides, mais qui laissent le système dans un état non-sécuritaire, ou dans le cas de manque de ressources, doivent retourner une valeur plus grande que zéro pour indiquer au client qu'il doit attendre avant d'essayer encore la même requête. Cette valeur correspond au temps d'attente, en millisecondes.
- Invalid : Les requêtes invalides (par example, qui dépassent le maximum permis pour ce client) doivent être refusées par le serveur en retournant au client la valeur (REFUSE).

Pendant l'exécution de l'application serveur et aussi de l'application client vous devez compter chacune des types des requêtes reçues et envoyées, ainsi que le nombre des clients correctement traités, utilisez les variables `count_accepted`, `count_on_wait`, `count_invalid`, `count_clients_dispatched` à cet effet. Notez qu'à la fin d'exécution, ces valeurs doivent être les mêmes pour les deux applications. Faites attention au fait que plusieurs données sont modifiées par différents clients (threads) "en même temps". Vous devez donc également implémenter le contrôle d'accès à ces données avec des verrous mutex ou semaphores pour éviter la bataille des ressources. Aussi, vu que les connexions sont asynchrones, c'est fort probable que le client finisse avant le serveur, mais il ne peut certainement pas finir son exécution avant que toutes les requêtes soient traitées par le serveur. Pour le moment un `sleep(2)` est mis dans les fonctions `st_signal` et `ct_wait_server`, mais ce comportement doit être changé.

Application client

L'application client doit vous permettre de tester votre application serveur avec des plusieurs fils d'exécution (threads) qui font des requêtes de façon pseudo-aléatoire. Vous avez aussi à compléter la fonctionnalité de base de ces clients, en gardant le format des message qui sont passés entre le serveur et le client. Vous serez évalués avec une application client qui fait plusieurs requêtes spécifiques (qui pourraient être même invalides) et votre application serveur doit répondre également à ces requêtes. Cette application lit automatiquement du fichier configuration les paramètres suivants : le nombre de clients à simuler (`num_clients`), le nombre de requêtes par client à créer et à lancer au serveur (`num_request_per_client`) et le nombre de différentes ressources disponibles dans le ser- veur (`num_resources`). Les valeurs maximale de chaque ressources par client sont définie statiquement dans le fichier de configuration du client (`max_resources_per_client`).

ClientThread

Le fichier `client_thread` doit implémenter dans les fonctions `ct_code` et `send_request` le comportement de chaque client. L'indice (commençant à 0) de chaque client est une propriété de chaque thread client. Par défaut, l'application client crée `num_clients` threads et les exécute avec la fonction `ct_code`.

Après, chaque client doit faire un ensemble de `num_requests` requêtes aléatoires tel que spécifié dans la section précédente. Les requêtes ont aussi un indice passé comme paramètre à la fonction `send_request`.

Les messages envoyés par le client devront être sous la forme indiquée précedemment. Pour le message `REQ`, une valeur négative indique une quantité de ressources demandée, alors qu'une valeur positive indique une quantité de ressources à libérer.

Par exemple, le message `REQ 2 {-5, -1, 0}` (au format binaire) représente une requête du client 2 pour demander 5 instances de la première ressource et 1 instance de la deuxième, tandis que le message `REQ 0 {0, 1, -2}` représente une requête du client 0, pour libérer 1 instance de la deuxième ressource et demander 2 de la troisième.

Évaluation

Votre code sera évalué de trois façons différentes:

- votre serveur avec votre client, votre serveur
- avec un client standard
- votre client avec un serveur standard.

Dans tous les cas l'exécution doit se faire correctement sans problèmes pour différentes configurations. Au moins un gros test va se faire pour les trois formes d'évaluation.

On considère comme gros, celui avec un grand nombre des requêtes totales (`num_requests_per_client*num_clients`) au maximum de 5000, et une grosse quantité des ressources à suivre (`num_resources*num_clients`) au maximum de 1000.

Travail à faire

Code à compléter 5%

Vous devez compléter des fichiers `server_threads` et `client_thread` (voir les commentaires du code pour plus de détails). Pour bien réussir cette partie, assurez vous que votre code compile, et vos applications échangent des messages entre elles; que les messages, ainsi que les réponses, suivent la syntaxe demandée, et que toutes les requêtes selon le fichier de configuration se font des deux côtés.

Fonctionnement correct 10%

En plus, vous devez vérifier que les requêtes de l'application client ne mettent pas le système en deadlock, et vous devez garantir que l'intégrité des ressources dans le système est préservée. À la fin de l'exécution de l'application client, le serveur doit revenir à son état initial. Finalement, peu importe les configurations utilisées, les deux applications doivent finir correctement leur exécution.

Rapport (deux pages maximum) 5%

Décrivez brièvement le fonctionnement de votre solution, et ensuite répondez aux questions suivantes. N'oubliez pas d'écrire les noms des deux membres de l'équipe.

1. Comment vous pouvez garantir que votre système n'arrivera jamais à être en deadlock?
2. Donnez un exemple d'une situation où les données pourraient être corrompues. Comment vous avez réglé ce problème ?
3. Quelle solution vous avez implémentée pour bien synchroniser la fin d'exécution des deux programmes?, pourquoi c'est important? et pourquoi un simple `sleep()` ne suffit pas dans tous les cas?

Format des documents à remettre

Remettez sur StudiUM l'archive générée par la commande `make release` (soumettre seule- ment une archive par équipe).

Liens utiles (en anglais)

- (Tutoriel sur sockets)[http://www.linuxhowtos.org/C_C++/socket.htm]
- (Documentation sur les threads et mutex du standard POSIX)[https://computing.llnl.gov/tutorials/pthread
- (Pipes examples) [http://tuxthink.blogspot.ca/2012/02/inter-process-communication-using-named.html]