Rapport BE - Programmation Système et Concurrente

ANNEXE

API de communication entre threads

Guilhem Buisan & Léopold Désert-Legendre

Table des matières

II-Étude technique

II-1 Manuel d'utilisation

Errata :

int recvMsg(int flag, int id, char * message) : Le char ** a été remplacé en char * car étant donné que cette fonction est appelé par le thread utilisateur, il n'y a pas besoin de pointeur vers la chaîne de caractères.

IV-Résultats des tests

IV-a Tests unitaires

Initialisation :

1. L'initialisation fonctionne correctement, le thread gestionnaire se lance et son identifiant est renseigné dans id_thread_gest. OK
2. Le premier thread prend le mutex, les autres attendent alors, puis initialise le service. Une fois le mutex libérés les autres threads s'aperçoivent que l'id_thread_gest n'est pas égal à 0 et renvoie donc -1. OK
3. Tous les appels d'initialisation supplémentaire renvoient -1. OK

Abonnement :

1. La fonction retourne -1. OK
2. Le thread gestionnaire traite les requêtes une part une et abonne chaque thread avec l'id demandé, les fonctions retournent 0 (jusqu'au nombre threadmax atteint). OK
3. Le thread gestionnaire abonne le même thread avec des id différents, les fonctions retournent 0 (jusqu'au nombre threadmax atteint). OK
4. Au premier abonnement avec l'id, l'abonnement se passe bien et la fonction retourne 0, mais pour tous les autres, l'abonnement ne fonctionne pas et la fonction retourne -2. OK
5. Le premier thread à s'abonner avec cet identifiant est bien abonné et à le code retour 0. Les autres ne sont pas abonnés et on le code retour -2. OK
6. Les trois premiers threads s'abonnent et ont le code retour 0, les deux d'après ne sont pas abonnés et ont le code retour -3. OK

Envoi :

1. Les fonctions retournent le code retour -1. OK
2. Le message n'est pas envoyé et on a le code retour -2. OK
3. Le message n'est pas envoyé et on a le code retour -4. OK
4. Les messages ne sont pas envoyés et on a le code retour -2. OK
5. Les messages ne sont pas envoyés et l'envoyeur à le code retour -4 jusqu'à ce que le destinataire s'abonne. Tous les envois pris en compte après l'abonnement fonctionnent et renvoie le code retour 0. OK
6. Le message n'est pas envoyé et on a le code retour -3. OK
7. Le message est délivré et l'envoyeur à le code retour 0. OK

8. Le thread d'écriture reste en attente et l'envoyeur à le code retour 0. OK
9. Le temps de traitement est un plus long, mais tous les messages sont envoyés et la fonction renvoie 0. OK

Réception

1. Si l'argument est valide (0 ou 1) on a le code retour -1, sinon -4. OK
2. Dans le cas de réception de messages ou de consultation du nombre on a le code retour -3. OK
3. On ne reçoit pas de messages et on a le code retour -2. OK
4. Aucun message n'est renvoyé, les boites à lettre ne changent pas d'état et on a le code retour -3. OK
5. Le flux d'exécution s'arrête sur cette fonction jusqu'à ce qu'il y est un message à lire. OK
6. Le code retour de la fonction est 0 et le flux d'exécution continue. OK
7. Le code retour est 0, et dans la variable *message* passé en argument on reçoit le plus ancien message message non lu. OK
8. Le code retour est égal au nombre de messages non lus présents dans la boite aux lettres. OK

Désabonnement

1. On a le code retour -1. OK
2. L'annuaire du gestionnaire ne change pas et on a le code retour -2. OK
3. L'annuaire ne change pas et on a le code retour -3. OK

Lorsqu'un thread est abonné et se désabonne, on a le code retour 0 et sa boite à lettre ainsi que son entrée dans l'annuaire sont supprimées, de plus les threads écriture/lecture en attente de la boite à lettre se terminent. OK

Terminaison

1. Que l'on demande une terminaison douce ou brutale, le code retour est -1. Si l'argument *force* est différent de 0 ou 1, le code retour est -3. OK
2. Le thread gestionnaire reste actif et on a un code retour -2. OK
3. En cas de terminaison brutale, les threads d'écriture/lecture et le gestionnaire se terminent et on a le code retour 0. En cas de terminaison douce, s'il n'y a plus aucun abonné (donc plus de threads écriture/lecture) le thread gestionnaire se terminent et on a le code retour 0. Sinon, les threads en attente et le thread gestionnaire restent dans le même état et on a le code retour -2. OK

IV-b- Tests par scénarii

Scénario 1 : *scenario1.c*

Pour cette application, nous avons voulu conserver le thread de supervision, permettant de suspendre et de relancer l'affichage (en appuyant sur « entrée »). De plus une fois suspendu, on peut demander à l'application de quitter (en appuyant sur « q » puis « entrée »), dans ce cas on affiche tous les messages restants en attente et l'application s'éteint. Pour ce faire il faut se passer du caractère bloquant de la fonction de lecture de message, on vérifie donc le nombre de messages avant d'essayer d'en lire un (pour toujours tester le flag de fin de temps en temps). Ensuite, lors de la suspension, le plus simple est de faire une attente active sur la variable de suspension.

Pour les résultats, les messages s'affichent bel et bien dans l'ordre. Le 1er message (Message n°0) n'est parfois pas envoyé, car le thread de lecture n'est pas forcément abonné, on pourrait tester le code retour de l'envoi et envoyer le même message tant que le code retour est différent de 0. En mettant un temps entre les envois plus grand qu'un temps entre les lectures, les affichages sont cadencés par la vitesse d'envoi, ce qui montre bien le caractère bloquant de cette fonction. Ensuite, en cas de suspension, les messages ne sont plus affichés mais continue d'être envoyés, il s'affichent alors lors de la reprise à la cadence de la lecture, jusqu'à ce que la file soit vide. Lors de l'arrêt, la file se vide, mais seul les 10 premiers messages sont dans l'ordre, si d'autres messages étaient en attente il s'affichent aussi mais dans un ordre aléatoire. OK

Scénario 2 : *scenario2.c*

Les messages s'affichent bel et bien dans l'ordre des envois, par contre, une fois la boite à lettres remplie, les threads en attente d'écriture dans la boite à lettre s'accumulent et le processus devient vite gourmand en mémoire. Une implémentation future peut être une limite de messages en attentes afin d'éviter que le processus ne devienne trop grand en cas d'erreur de programmation par exemple.

Il est à noter qu'une telle application se code maintenant très rapidement et facilement en faisant appel à l'API. OK

Scénario 3 : *scenario3.c*

Chaque thread affiche tour à tour le message envoyé par l'autre, ainsi, un thread peut bel et bien jouer le rôle d’émetteur et de recepteur. De plus, le comportement de la fonction d'envoi en broadcast est bien celui qui a été décrit.

Encore une fois, le gain d'expressivité et de temps pour coder une telle application est indéniable. OK

V-Conclusion

Comme prévu, l'API est fonctionnelle, grâce à elle, la communication entre threads est grandement simplifiée. En effet, l'API permet de gagner un niveau d'abstraction, nous sommes plus proche dans la façon de coder de la solution du problème initial plutôt que des adaptations de la solution pour cause de difficultés techniques.

Le projet a donc été mené à terme, dans les délais imposés. Certes, certains points peuvent être améliorés tels que la fuite mémoire non résolu due à un besoin de garder un contrôle sur les threads d'écriture et de lecture sans avoir à attendre absolument leur terminaison. Une autre amélioration possible est la possibilité d'envoyer autre chose qu'une chaîne de caractères, l'idéal aurait été un paramètre de la fonction initMsg, désignant le type de message.

Malgré ce défaut, l'API fonctionne et respecte le cahier des charges.