Projet de programmation système

Juliusz Chroboczek

18 février 2016

La programmation avec les *threads* et la mémoire partagée est difficile — il est facile d'omettre de synchroniser les accès aux données partagées. Le langage de programmation *Java* essaie de minimiser le problème en associant un moniteur à chaque objet (méthodes *synchronized*), ce qui rend les objets Java coûteux. Le langage *Go* utilise une autre approche : il intègre au langage une structure de données, le *canal*, qui permet de communiquer des données de taille fixée de façon efficace entre les *threads*.

Le but de ce projet est d'implémenter des canaux à la *Go* en C.

1 Sémantique des canaux

Un canal permet de transférer entre *threads* des éléments de taille `eltsize` fixée, et a une capacité `size` fixée : un tel canal contient un tampon de taille `eltsize` × `size`. La création d'un canal se fait à l'aide de la fonction `channel_create`, et sa destruction à l'aide de la fonction `channel_destroy` :

```
struct channel *
    channel_create(int eltsize, int size, int flags);
void channel_destroy(struct channel *channel);
```

La fonction `channel_create` retourne `NULL` en cas d'erreur, et alors `errno` est positionné. Si vous n'implémentez pas les canaux synchrones (paragraphe 4.4), elle devra retourner `NULL` avec `errno` valant `ENOSYS` si `size` vaut 0. De même, si vous n'implémentez pas les canaux globaux (paragraphe 4.3), elle devra indiquer `ENOSYS` si le bit `CHANNEL_PROCESS_SHARED` est positionné dans le paramètre `flags`. La fonction `channel_destroy` ne fait pas de synchronisation — c'est à l'appelant de garantir que le canal ne sert plus.

La fonction `channel_close` ferme un canal ; il est alors encore possible de lire les éléments qu'il contient, mais pas d'en ajouter de nouveaux, ni de rouvrir le canal fermé.

```
int channel_close(struct channel *channel);
```

Cette fonction retourne 1 en cas de succès, 0 si le canal était déjà fermé, et -1 en cas d'erreur (et alors `errno` est positionné).

L'envoi se fait à l'aide de la fonction `channel_send` :

```
int channel_send(struct channel *channel, const void *data);
```

Si le canal est fermé, cette fonction retourne -1 et `errno` vaut `EPIPE`. Sinon, si le canal n'est pas plein (il contient moins de `size` éléments), cette fonction stocke la valeur sur laquelle pointe `data` dans le canal (la valeur, pas le pointeur), et retourne 1. Si le canal est plein, elle bloque le *thread* appelant jusqu'à ce qu'il y ait de la place.

La réception se fait à l'aide de `channel_recv` :

```
int channel_recv(struct channel *channel, void *data);
```

Si le canal n'est pas vide, cette fonction défile un élément et le stocke à l'endroit où pointe `data`, et retourne 1. Si le canal est vide et fermé, elle retourne 0. Sinon, elle bloque le *thread* appelant jusqu'à ce que le canal ne soit plus vide ou soit fermé. Elle retourne -1 en cas d'erreur (et alors `errno` est positionné).

2 Code fourni

Nous vous fournissons :
- un fichier `channel.h` qui définit l'interface à laquelle doit obéir votre implémentation ;
- un programme `mandelbrot.c` qui est un exemple d'utilisation des canaux. On peut le compiler à l'aide de la commande suivante :

```
gcc -g -O3 -ffast-math -Wall -pthread \
    'pkg-config --cflags gtk+-3.0' \
    mandelbrot.c channel.c \
    'pkg-config --libs gtk+-3.0' -lm
```

3 Sujet minimal

Le but du projet est d'implémenter des canaux utilisant de la mémoire partagée entre *threads* du même processus. Les fonctions `channel_send` et `channel_recv` ne devront faire aucun appel système qui ne soit pas lié à la prise de *mutex* ou la signalisation de variables de condition. En particulier, une implémentation qui communique à travers un tube n'est pas acceptable. (Consultez-nous si vous avez un doute.)

Vous devrez *obligatoirement* implémenter l'interface définie par `channel.h` ; nous vérifierons que le programme `mandelbrot.c` fourni fonctionne avec votre implémentation. Vous devrez aussi effectuer des *benchmarks* et nous fournir des données qui indiquent au minimum les performances de votre implémentation :
- sur de petites données en l'absence de contention ;
- sur de petites données en présence de contention ;

– sur de grosses données.

Nous aprécierons toute donnée empirique supplémentaire, par exemple les performances de votre implémentation en fonction du nombre de processeurs, ou les performances de votre implémentation sur une architecture autre que x86.

4 Extensions

Le sujet minimal ci-dessus est facile — j'ai mis 20 minutes pour faire une première solution (mais je savais déjà comment faire), et 10 minutes de plus pour la faire fonctionner en $O(1)$. Si vous l'implémentez dans son intégralité, de façon parfaite, et vous nous fournissez un rapport magnifique, vous aurez la moyenne, guère plus. Nous attendons donc que vous implémentiez au moins quelques unes des extensions suivantes.

4.1 Comparaison avec les tubes

Nous adorons les *benchmarks*. Tout résultat empirique fourni sera lu avec intérêt, surtout s'il s'agit d'une comparaison. Par exemple, vous pourriez nous fournir des résultats qui comparent l'efficacité de votre implémentation avec une implémentation basée sur les tubes.

4.2 Exemples d'utilisation

L'exemple d'utilisation que nous vous fournissons — un visionneur de l'espace de Mandelbrot — est un exemple d'un problème dit *embarrassingly parallel*, c'est à dire qui est très facile à paralléliser. Cependant, même un exemple aussi simple permet d'illustrer l'intérêt des canaux : en quelques lignes de code, on distribue le calcul sur plusieurs *threads* sans les ordonner manuellement, et sans nous intéresser à la *thread-safety* de notre bibliothèque graphique [1].

Nous vous serions reconnaissants de nous fournir des exemples de programmes plus intéressants qui s'écrivent bien avec les canaux. Notez cependant qu'il s'agit d'un projet de système, ne passez pas trop de temps à soigner une interface graphique ou à faire des boîtes de dialogue.

4.3 Canaux globaux

Dans le sujet minimal, un canal ne permet la communication qu'entre deux *threads* du même processus. Un canal *global* est un canal créé avec le bit `CHANNEL_PROCESS_SHARED` : placé dans une zone de mémoire partagée (par exemple obtenue avec `mmap`), il permet la communication entre processus distincts.

1. À ce sujet, ce n'est pas comme ça qu'il faudrait s'y prendre pour faire un visionneur d'espace de Mandelbrot. Il faudrait utiliser une *backing pixmap* pour éviter de recalculer à chaque *redisplay*, faire une mise à l'échelle purement graphique avant de recalculer, et vectoriser le calcul. Mais ne vous y amusez pas, ce n'est pas le sujet du projet.

Nous vous encourageons à implémenter les canaux globaux, et à nous fournir un exemple d'utilisation.

4.4 Canaux synchrones

Si le paramètre `size` de `channel_create` vaut 0, le canal créé est un canal *synchrone* : il ne contient pas de tampon, et, si aucun lecteur n'est bloqué sur un canal, `channel_send` bloque jusqu'à ce qu'il y en ait un. Si un ou plusieurs *threads* sont bloqués en lecture, `channel_send` en sélectionne un, et copie son paramètre directement dans le *thread* de destination, sans passer par un canal [2].

Nous serions intéressés par une implémentation des canaux synchrones, par des exemples d'utilisation [3], et par des *benchmarks* qui indiquent si les canaux synchrones sont visiblement plus rapides que les canaux asynchrones ordinaires. (*Non* est une réponse acceptable.)

4.5 Communication par lots

L'interface suggérée envoie un message par appel à `channel_send`, et reçoit un message par appel à `channel_recv`. Il serait intéressant de définir une interface qui permet d'envoyer ou de recevoir plusieurs messages d'un coup, soit en définissant une interface par lots *(batched)* qui passe un tableau de messages (à la `writev` et `readv`), soit en scindant les appels en plusieurs fonctions plus primitives (prendre un *lock*, écrire les messages, relâcher le *lock*). Faites attention à la sémantique des appels par lots lorsque le canal est presque plein ou presque vide.

Il faudra nous fournir une description détaillée de la sémantique de la communication par lots, un exemple d'application, et des *benchmarks* qui indiquent si votre solution améliore les performances (une réponse négative est bien sûr acceptable).

4.6 Canaux à une seule copie

Une paire écriture/lecture dans un canal fait deux copies de la donnée : une copie de l'écrivain vers le tampon du canal, et une copie du tampon du canal vers le lecteur. Est-il possible de définir une interface qui permet d'éviter au moins une de ces copies ? Est-elle utilisable par un programmeur normal ? Si vous implémentez la communication à une seule copie, il faudra bien sûr nous fournir un exemple d'utilisation et des *benchmarks*.

4.7 Implémentation *lock-free*

A priori, une implémentation des canaux utilise des *locks* pour synchroniser l'accès au canal. Est-il possible de faire une implémentation qui utilise des opérations atomiques pour éviter de

2. En principe, une écriture sur un canal synchrone devrait aussi passer la main au *thread* de destination, mais ce n'est probablement pas possible à implémenter avec les *threads* POSIX.

3. Le programme `mandelbrot.c` utilise des canaux synchrones lorsqu'il est invoqué avec l'option `-s`.

prendre des *locks* ? Est-elle plus rapide que l'implémentation utilisant des *locks* ?

5 Modalités de soumission

Le projet sera fait par groupes de deux ou trois étudiants [4]. Vous nous remettrez avant la soutenance une archive *nom1-nom2-nom3*`.tar.gz` contenant :

- le source complet de votre programme, accompagné d'un fichier `README` indiquant comment le compiler et s'en servir ;
- un rapport sous format PDF, contenant une description sommaire de votre programme et un résumé des statistiques obtenues [5].

Cette archive devra s'extraire dans un sous-répertoire *nom1-nom2* du répertoire courant [6]. Par exemple, si vous vous appelez *Francis Crick*, *Rosalind Franklin* et *James Watson*, votre archive devra porter le nom `crick-franklin-watson.tar.gz` et son extraction devra créer un répertoire `crick-franklin-watson` contenant tous les fichiers que vous nous soumettez.

Cette archive devra être déposée sur le site DidEL [7] « *PROGSYSM1* » dans la rubrique « *Travaux* ». Vous pouvez bien entendu modifier ensuite votre dépôt initial s'il ne vous satisfait pas, mais évitez de faire une nouvelle soumission [8].

4. Mais pas quatre. Les groupes dégénérés (singletons) seront tolérés, mais traités avec sévérité.
5. Nous adorons les graphiques.
6. Sinon, ça s'extrait dans le mauvais répertoire, ce qui nous énerve, ce qui n'est pas dans votre intérêt.
7. Quelle horreur.
8. Sinon, ça encombre les dépôts, ce qui nous énerve.