Projet de programmation système

Juliusz Chroboczek et Dominique Poulalhon

24 mars 2017

1 Introduction

Le but de ce projet est d'implémenter, à l'aide d'une zone de mémoire partagée, une primitive de communication entre processus (ou *threads*) par flots d'octets, permettant les accès multiples avec des propriétés d'atomicité bien définies. Informellement, on vous demande d'implémenter une structure semblable à un tube (nommé ou anonyme) mais avec des propriétés d'atomicité différentes.

2 Interface

Un *conduit* sera représenté par un pointeur sur une `struct conduct` :

```
struct conduct;
```

Du point de vue du client de la bibliothèque, la `struct conduct` est opaque. Le client n'en connaît pas la structure, il sait juste comment manipuler des pointeurs dessus — de manière analogue à la structure `FILE` de `stdio`. Ce n'est qu'à l'intérieur de la bibliothèque que la définition de la structure est visible.

Il existe deux fonctions qui retournent une `struct conduct` : `conduct_create` et `conduct_open`.

```
struct conduct *conduct_create(const char *name,
                               size_t a, size_t c);
struct conduct *conduct_open(const char *name);
```

La fonction `conduct_create` crée un nouveau conduit de capacité `c` et ayant une taille atomique `a` (voir paragraphe 3), puis retourne une structure référant à ce conduit ; si `name` vaut `NULL`, le conduit est anonyme, sinon il est stocké dans le fichier identifié par `name`. La fonction `conduct_open` ouvre un conduit existant, et retourne une structure qui y réfère.

De même, il existe deux fonctions qui libèrent une structure `conduct`.

```
void conduct_close(struct conduct *);
void conduct_destroy(struct conduct *);
```

La fonction `conduct_close` libère la structure sans détruire le conduit ; la fonction `conduct_destroy` détruit le conduit (et libère la structure) : il ne peut alors plus être utilisé, et il est de la responsabilité du programmeur de s'assurer que d'autres processus n'accèdent plus à ce conduit. Si

n processus (ou *threads*) ont des références à un même conduit, les $n-1$ premiers doivent appeler `conduct_close` et le dernier doit appeler `conduct_destroy`.

Trois fonctions permettent de réaliser des entrées-sorties sur les conduits :

```
ssize_t conduct_read(struct conduct *c,
                     void *buf, size_t count);
ssize_t conduct_write(struct conduct *c,
                      const void *buf, size_t count);
int conduct_write_eof(struct conduct *c);
```

Si un conduit contient $n > 0$ octets, la fonction `conduct_read` extrait $\min(\texttt{count}, n)$ octets du conduit, les stocke à l'adresse `buf`, puis retourne ce nombre à l'appelant. Si le conduit est vide et sans marque de fin de fichier, elle bloque jusqu'à ce qu'il ne soit plus vide ou qu'une marque de fin de fichier y soit insérée. Enfin, si le conduit est vide et contient une marque de fin de fichier, elle retourne 0.

Inversement, la fonction `conduct_write` insère dans le conduit au plus `count` octets (voir précisions ci-dessous) stockés à l'adresse `buf` puis retourne le nombre d'octets stockés à l'appelant. Si une marque de fin de fichier a été insérée dans le conduit, alors la fonction `conduct_write` retourne -1 et `errno` vaut `EPIPE`.

Enfin, la fonction `conduct_write_eof` insère une marque de fin de fichier dans le conduit, ce qui change le comportement des lectures sur un conduit vide et des écritures comme décrit ci-dessus. Une fois qu'une marque de fin de fichier a été insérée, elle ne peut plus jamais être enlevée. Cette opération est idempotente — insérer une nouvelle marque de fin de fichier sur un conduit qui contient déjà une marque de fin de fichier ne fait rien.

3 Détails du comportement

Un conduit est paramétré par deux entiers : sa *capacité* c, et la taille a en dessous de laquelle il offre une garantie d'*atomicité* ($a \leq c$). Le comportement exact des écritures dans le conduit dépend des valeurs de a et c.

3.1 Caractère bloquant ou non bloquant

Une lecture depuis un conduit vide (et sans marque de fin de fichier) bloque jusqu'à ce que le conduit ne soit plus vide (ou qu'une marque de fin de fichier y soit insérée). Une lecture depuis un conduit non vide ne bloque jamais, les lectures partielles sont possibles.

Une écriture dans un conduit plein bloque toujours, et une écriture de $n \leq a$ octets bloque jusqu'à ce qu'il reste au moins n octets libres ; les n octets sont alors écrits. Au contraire, une écriture de $n > a$ octets dans un conduit où il reste de la place ne bloque jamais — une écriture partielle (d'au moins un octet) est possible dans ce cas.

3.2 Atomicité

Une écriture dans un conduit est toujours atomique : si un appel à `conduct_write` retourne m, alors les m octets écrits sont contigus dans le conduit. Bien sûr, cette propriété d'atomicité est bornée par la sémantique des écritures partielles, définie au paragraphe précédent.

3.3 Conduits nommés

Si le paramètre `name` n'est pas nul lors de la création d'un conduit, alors le conduit est persistant et représenté par le fichier dont le nom est donné par `name` (créé à l'aide de `open`, pas à l'aide de `shm_open`). Le conduit peut alors être ouvert par un autre processus (ou le même) à l'aide de `conduct_open`.

Un conduit nommé peut être fermé à l'aide de `conduct_destroy` ou à l'aide de `conduct_close`. Cette dernière fonction détruit le *mapping* en mémoire du conduit, mais ne détruit pas le fichier sur disque. Quant à `conduct_destroy`, cette fonction détruit le *mapping* et aussi le fichier sur disque.

Typiquement, si n processus ouvrent le même conduit, $n - 1$ le ferment à l'aide de `conduct_close`, et le dernier le détruit à l'aide de `conduct_destroy`. Cependant, il est aussi possible que tous utilisent `conduct_close`, dans quel cas il est possible de reouvrir le même conduit plus tard.

3.4 Héritage à travers `fork`

Un conduit, nommé ou anonyme, peut être hérité à travers `fork`. Si un conduit est dupliqué par `fork`, un seul des fils devra appeler `conduct_destroy`.

4 Code fourni

Nous vous fournissons le fichier `conduct.h` qui contient les prototypes des fonctions qu'il est obligatoire d'implémenter. Vous n'avez pas le droit de changer les interfaces, mais vous pouvez ajouter des fonctions supplémentaires.

Nous vous fournissons aussi le programme `julia.c` qui utilise des conduits pour calculer des ensembles de Julia. Vous pouvez le compiler à l'aide de :

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

Rien que ça.

5 Sujet minimal

Le sujet minimal consiste à implémenter les sept fonctions ci-dessus en respectant obligatoirement la sémantique définie au paragraphe 3.

Les conduits devront obligatoirement être implémentés à l'aide de `mmap` dans une zone de mémoire partagée, soit anonyme, soit nommée comme expliqué au paragraphe 3. Nous vous laissons libres d'utiliser les primitives de synchronisation que vous désirez ; nous suggérons cependant d'utiliser soit les sémaphores POSIX [1], soit les mutex et les variables de condition *pthread* [2].

1. Faisable selon Dominique.
2. Plus facile selon Juliusz.

Nous vous laissons libres du choix des structures de données — vous pouvez utiliser un tampon simple, un tampon circulaire, ou autre chose. Le choix des structures de données sera naturellement pris en compte lors de l'évaluation.

Vous devrez nous fournir du code portable au moins entre différents systèmes Linux. Nous testerons au moins sur Linux/x86 et Linux/ARM [3], faites donc attention aux problèmes d'alignement en mémoire.

6 Extensions

Le sujet minimal ci-dessus est facile. 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.

Détection des erreurs Le comportement de votre code en cas d'utilisation incorrecte sera pris en compte lors de l'évaluation. Par exemple, une création de conduit qui échoue devrait retourner `NULL`, avec `errno` ayant une valeur raisonnable. Il serait désirable de retourner une erreur lors d'un accès à un conduit détruit, mais c'est plus difficile.

Lecture et écriture par lots Pour certaines applications, il est utile d'avoir une fonction

```
ssize_t conduct_writev(struct conduct *c,
                       const struct iovec *iov, int iovcnt);
```

Le tableau `iov`, de taille `iovcnt`, contient une suite de tampons. Un appel à `conduct_writev` est équivalent à un seul appel à `writev` ayant pour paramètre la concaténation de tous les éléments de `iov`, et le comportement a les mêmes propriétés d'atomicité — un appel à `conduct_writev` n'est donc pas équivalent à `iovcnt` appels à `conduct_write`.

De même, un appel à

```
ssize_t conduct_readv(struct conduct *c,
                      struct iovec *iov, int iovcnt);
```

est équivalent à un appel à `conduct_read` appliquée à la concaténation des éléments de `iov`.

Si vous implémentez les entrées-sorties par lots, vous devrez nous fournir un exemple d'utilisation.

Lectures et écritures non-bloquantes Toutes les opérations définies dans le sujet de base sont bloquantes. Il pourrait être intéressant d'étendre l'interface avec des opérations non-bloquantes. Cependant, comme `select` n'agit pas sur les conduits, comment utiliser une telle interface ne semble pas évident.

Nous apprécierons une extension non-bloquante à l'interface, surtout si elle est accompagnée d'un exemple convaincant d'utilisation.

3. Sur un Raspberry Pi ou un BeagleBone, par exemple.

Comparaison avec les tubes et les *sockets* Une opération sur un conduit manipule une zone de mémoire partagée, ce qui ne recquiert pas d'appel systèmes (sauf dans le cas bloquant). On s'attendrait donc à ce que les conduits soient plus efficaces que les tubes ou les *sockets*, qui demandent un appel système pour chaque opération. Cependant, les appels systèmes sont implémentés de façon efficace, surtout sur Linux. Nous apprécierons que vous nous fournissiez un jeu de *benchmarks* ainsi que des données obtenues expérimentalement qui comparent les conduits aux tubes ou aux *sockets*.

Exemples d'utilisation Nous apprécierons que vous nous fournissiez un ou plusieurs exemples d'utilisation des conduits.

Autres extensions Toutes les autres extensions au sujet seront examinées avec intérêt et bienveillance.

7 Modalités de soumission

Le projet est à réaliser par groupes de 2 ou 3 étudiants. Vous nous remettrez avant la soutenance une archive *nom1-nom2-nom3*`.tar.gz` contenant les sources complètes de votre programme, accompagnées d'un fichier `README` indiquant comment le compiler [4] et s'en servir, ainsi qu'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-nom3* du répertoire courant.

Par exemple, si vous vous appelez Paul Stookey, Mary Travers et Peter Yarrow, votre archive devra s'appeler `stookey-travers-yarrow.tar.gz` et son extraction devra créer un répertoire `stookey-travers-yarrow` contenant tous les fichiers que vous nous soumettez.

Cette archive devra être déposée sur le site Moodle du cours.

4. À l'aide de `make`, bien sûr.
5. On adore les graphiques.