Implémentation des canaux en C

Eric Uzenat - Christ Kankolongo Musenga - Adrian Thibaud

2016

Table des matières

1 Utilisation

Avant d'expliquer comment nous avons implémenter nos canaux, il convient d'abord de bien expliquer comment les utiliser et présenter les différentes fonctions que propose notre interface.

1.1 Création d'un canal

Pour créer un canal, on utilisera la fonction

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

Pour une simple communication asynchrone entre threads, `size` devra être strictement supérieure à `0`, le flags devra lui être mis à `0`. Ceci créera un channel de taille `eltsize` × `size`, qui supporte donc une communication asynchrone entre threads. On peut vouloir que des processus communiquent entre eux et non des threads. Pour cela, on positionnera `flags` à la valeur

```
CHANNEL_PROCESS_SHARED
```

pour ainsi partager la mémoire. Pour que la communication entre le lecteur et l'écrivain se fasse de manière synchrone, `size` devra prendre la valeur `0`, cette communication ne fonctionnera que entre les threads. Pour utiliser l'optimisation des canaux asynchrones à une copie, on positionnera le flag à :

```
CHANNEL_PROCESS_ONECPY
```

Il faudra simplement s'assurer que la donnée à écrire soit alloué dans un espace mémoire dynamique et non statique. Enfin, pour faire communiquer des processus non apparenté (à l'exterieur d'un `fork`) il faudra nommer notre fichier. On utilisera la fonction

```
struct channel *channel_unrelated_create(int eltsize,int size,char *path)
```

où `path` sera le nom du canal. Pour ouvrir le canal non apparenté, on utilisera la fonction

```
struct channel *channel_unrelated_open(int eltsize,int size,char *path)
```

Le fonction retourne `NULL` en cas d'erreur, et `errno` est positionnée.

1.2 Envoi d'une donnée

Pour envoyer une donnée au canal, on utilisera la fonction

```
struct channel *channel_send(struct channel *channel,void *data)
```

`data` étant la donnée à envoyer dans le canal, qui doit obligatoirement être de taille `eltsize` sous peine d'être tronquée.
Cette fonction aura des comportements différents si le canal est synchrone ou asynchrone. Si il est synchrone et si aucun lecteur ne bloque pour lire une donnée, alors l'écrivain bloque en attendant un tel lecteur. Lorsqu'un lecteur est présent (bloque pour une lecture), alors l'écrivain écrit sa donnée et termine. En revanche, si il est asynchrone, on peut écrire jusqu'à ce que le tampon soit plein. Dans ce cas, la fonction bloquera, et écrira la donnée lorsqu'une place se libérera dans le tampon (lorsqu'un lecteur lira une donnée).

1.3 Reception d'une donnée

Pour recevoir une donnée, on utilisera la fonction

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

`data` étant la variable qui contiendra la valeur reçue par le canal.
De la même manière que channel_send, cette fonction aura des comportements différents dans le cas où le canal est synchrone ou asynchrone. Si il est synchrone, le lecteur bloque jusqu'à ce que l'écrivain se connecte et écrive la donnée dans le canal. En revanche, si il est asynchrone, le lecteur peut lire des données jusqu'à ce que le tampon soit vide, dans ce cas, il bloquera en attendant une nouvelle donnée de l'écrivain. Pour finir, si le canal est partagée, il faudra que la variable data allouée avec une mémoire partagée.

1.4 Fermerture d'un canal

Pour fermer le canal, on utilisera la fonction

```
struct channel *channel_close(struct channel *channel)
```

Une fois fermé, il sera impossible d'écrire de nouvelles données dans le tampon, On pourra en revanche en lire jusqu'à ce qu'il soit vide. Dans ce cas, le lecteur ne sera plus bloquant. Si le canal est synchrone, on ne pourra ni lire ni écrire.

1.5 Destruction d'un canal

Pour vider la mémoire, on utilisera la fonction

```
struct channel *channel_destroy(struct channel *channel)
```

Cette fonction ne fait pas de synchronisation, il faudra donc être sûr que le canal ne sert plus avant de l'utiliser.

2 Implémentation

Venons-en maintenant à la partie intéressante de ce rapport : la manière dont nous avons conçu nos canaux, ainsi que les algorithmes utilisés pour la synchronisation.

2.1 La structure struct channel

La structure struct channel est definie comme suit :

```
struct channel { void *mem; };
```

Elle ne contient qu'un unique pointeur qui correspondra à l'adresse de la zone mémoire vers laquelle pointe le canal.
Un canal sera représenté par des données d'entête :

```
 struct header {
int pos_r;
int pos_w;
int pos_deb;
int size;
int eltsize;
int incr;
int waiter_r;
int waiter_w;
sem_t lock;
sem_t sem_a;
sem_t sem_c;
sem_t sem_d;
int close;
int flags;
void *addr_sync;
const void **buffer;
};
```

suivi d'un bloc de taille `eltsize` × `size`. Schématiquement on la représentera comme suit :

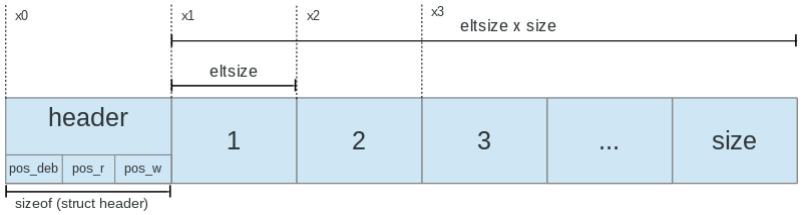


FIGURE 1 – Représentation d'un canal en mémoire

`pos_r` contient la position de la prochaine donnée à lire. `pos_w` de la prochaine donnée à écrire, et `pos_deb` contiendra toujours la même valeur, soit la position du début de la file. Ainsi, il sera facile de connaitre l'adresse d'une zone mémoire, si `x0` est l'adresse de notre plage, alors `x0 + ptr_deb` coresspond à l'adresse du

début de la file. `eltsize` et `size` contiennent respectiviement la taille d'un bloc et le nombre de blocs que contient la mémoire. `waiter_w` contiendra le nombre de processus endormis sur le sémaphore `sem_c` et `waiter_r` contiendra le nombre de processus endormis sur le sémaphore `sem_d`. On a ensuite trois sémaphores (`sem_a`, `lock` et `sem_c`) qui seront utilisés pour la synchronisation des processus communicants. Enfin, `addr_sync` sera utilisé seulement en mode synchrone et contiendra l'adresse dans laquelle il faut écrire la donnée. Nous expliqueront l'utilité de la variable `buffer` dans la section sur l'explication des canaux à une copie.

2.2 Les canaux asynchrones

Maintenant que l'on a bien présenté comment un canal était représenté en mémoire, on peut continuer sur l'explication de l'algorithme pour la communication asynchrone. Rappelons qu'en mode asynchrone, les données sont bufferisées. Lorsque le tampon est vide, la lecture est bloquante et lorsque le tampon est plein, l'écriture est bloquante.
Commençons par l'envoi des données, l'algorithme est le suivant :

Listing 1 – Envoi d'une donnée en asynchrone

```
sem_wait (&chan->lock);

if (chan -> close == 1)
{ sem_post (&chan->lock); return 0; }

while (chan->pos_w == -1)
{ chan->waiter_w ++;
  sem_post (&chan->lock);
  sem_wait (&chan->sem_c);
  sem_wait (&chan->lock);

  if (chan -> close == 1)
  { sem_post (&chan->lock); return 0; }

}

/* envoi de la donnee */

chan->pos_w = chan->pos_w + chan-> incr;

if(chan->pos_r == -1) {
  chan->pos_r=chan->pos_w - chan-> incr;
  while (chan->waiter_r > 0)
  { chan->waiter_r --; sem_post (&chan->sem_d); }
}
if (chan->pos_deb+chan->size == chan->pos_w)
chan->pos_w = chan->pos_deb;
if (chan->pos_w == chan->pos_r)
chan->pos_w = -1;

sem_post (&chan->lock);
```

le sémaphore `lock` représente la zone de la section critique. En décrémentant `lock`, on s'assure qu'un seul processus pourra rentrer dans cette section. La première chose que nous allons faire est de tester si le tampon n'est pas plein (`pos_w == -1`). Si c'est le cas, alors on rentre dans la boucle, on s'enregistre comme dormeur en incrémentant la variable `waiter_w`, on libère `lock` puis on s'endord sur `sem_c`. On se réveillera au moment ou un lecteur aura lu une donnée et ainsi libérer de la place dans le tampon. Ce procédé est exactement celui des variables de condition. Nous avons préféré utiliser des sémaphores et simuler le fonctionnement du `pthread_cond_mutex` plutôt que de l'utiliser car pour le passage aux canaux globaux, il suffit de déclarer les sémaphores comme partagés à leur initialisation. Une fois sortie de la boucle, on est donc certain qu'une place est disponible pour être écrite, on vérifie simplement . On écrit donc la donnée dans le canal. Viens ensuite différents cas à traiter : on teste si il n'y avait plus rien à lire (`pos_r == -1`). Dans ce cas là, on écrit dans `pos_r` la position à laquelle on vient d'écrire la donnée, puis on réveille tous les dormeurs bloquants sur le semaphore `sem_d`. On teste ensuite si la donnée que l'on vient d'écrire a rempli le buffer, si c'est le cas alors on retourne au début en donnant à `pos_w` la position du début de la file. Pour finir, on teste si la position de l'écrivain est la même que celui du lecteur, si c'est le cas alors on ne peut plus écrire car le tampon est plein, `pos_w` vaut alors -1 pour spécifier qu'aucune donnée ne peut être écrite.

Pour la lecture d'une donnée, l'algorithme est similaire, à la différence qu'au lieu d'écrire une donnée on la lit, et on teste non pas la valeur de `pos_w`, mais celle de `pos_r`. De la même manière, on utilise `waiter_r` et on s'endort sur le sémaphore `sem_d`.

Listing 2 – Réception d'une donnée en asynchrone

```
sem_wait (&chan->lock);

if (chan -> close == 1 && chan -> pos_r == -1)
{ sem_post (&chan->lock); return 0; }

while (chan -> pos_r == -1)
{
  chan -> waiter_r ++;
  sem_post (&chan->lock);
  sem_wait (&chan->sem_d);
  sem_wait (&chan->lock);

  if (chan -> close == 1 && chan -> pos_r == -1)
  { sem_post (&chan->lock); return 0; }

}

/* lecture */

chan->pos_r = chan->pos_r + chan -> incr;

if(chan->pos_w == -1) {
  chan->pos_w=chan->pos_r - chan->incr;
  while (chan->waiter_w > 0)
```

```
    { chan->waiter_w --; sem_post (&chan->sem_c); }
  }
  if (chan->pos_deb+chan->size == chan->pos_r)
  chan->pos_r = chan->pos_deb;
  if (chan->pos_w == chan->pos_r)
  chan->pos_r = -1;

  sem_post (&chan->lock);
```

2.3 Les canaux synchrones

Maintenant que l'on a bien détaillé l'implémentation des canaux asynchrones, on peut passer au canaux synchrones. L'algorithme pour rendre un canal synchrone est extrêmement court, trois lignes pour l'envoi et cinq lignes pour la réception.

Listing 3 – Envoie d'une donnée en synchrone

```
sem_wait(&a);

//ecriture de la donnee

sem_post (&c);
```

Listing 4 – Récéption d'une donnée en synchrone

```
sem_wait(&b);

//lecture de la donnee

sem_post(&a);
sem_wait(&c);
sem_post(&b);
```

Pour l'implémentation des canaux synchrones, on aura a disposition trois sémaphores (`a`, `b` et `c`), `a` et `c` étant initialisé a `0` et `b` a `1`. Commencont par l'écrivain, l'écrivain bloque sur le semaphore `a`, si le processus passe le sémaphore, cela veut dire qu'un lecteur est présent et attend de pouvoir recevoir une donnée. Une fois passé, le sémaphore `a`, l'écrivain envoie la donnée. Puis, il incrémente le sémaphore `c` pour reveiller le lecteur et termine. Le fonctionnement du lecteur n'est pas plus compliqué, il rentre dans la section de lecture en prenant le sémaphore `b`, donne son adresse de `data`, et indique en incrémentant le sémaphore `a` qu'un lecteur est présent et attend une donnée, puis s'endort sur le sémaphore `c`. Une fois reveillé, cela signifie que l'écrivain a écrit la donnée, le lecteur peut donc terminer et libère le sémaphore `b` pour éventuellement laisser la ressource à un autre lecteur.

2.4 Les canaux globaux apparentés

L'avantage d'avoir utilisé des sémaphores c'est qu'il suffit de les initialiser de la manière suivante pour les partager entre processus apparentés

```
sem_init (&sem, 1, n)
```

De la même manière, la création de la mémoire se fera par un mmap initialisé avec les arguments `MAP_SHARED | MAP_ANONYMOUS` pour créer un espace mémoire partagé et anonyme (la mémoire ne sera pas dupliquée après un `fork`).

2.5 Les canaux globaux non apparentés

L'implémentation des canaux non apparentés se fait par l'intermédiaire d'un fichier crée à partir de `shm_open`. La création du channel se fait de la même manière que les canaux simples à la seule différence que la mémoire correspond à ce fichier crée, projeté en mémoire à l'aide de mmap. L'envoi et la récéption des données se fait exactement de la même manière que les canaux simples.

2.6 Les canaux à une copie

Dans notre implémentation classique des canaux asynchrones, on effectuait deux copies : une pour l'ecrivain qui copiait sa donnée dans le tampon et une autre pour le lecteur qui copiait la donnée du tampon vers sa mémoire locale. Dans cette section, nous allons expliquer comment nous avons modifié notre structure pour pouvoir supporter une seule copie. Et surtout, nous expliquerons le bug rencontré lors de l'utilisation.
La structure du header est pratiquement similaire à la précédente à la seule différence que le header ne fait plus partie du tampon mais c'est une zone mémoire qui contient un pointeur vers le tampon qui correspond à un tableau de `void *` (un schéma est toujours plus représentatif que des mots) :

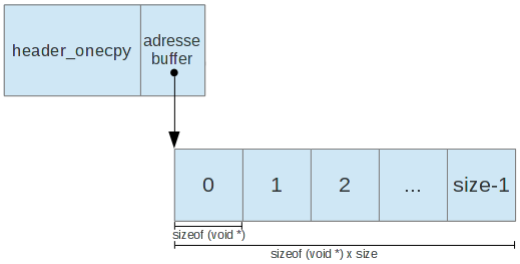


FIGURE 2 – Représentation d'un canal à une copie en mémoire

Maintenant que nous avons bien expliqué comment les canaux à une copie fonctionnaient, il convient d'expliquer dans quel cas il ne fonctionne pas. La copie s'effectuant pour le lecteur, c'est l'écrivain qui écrit l'adresse de sa mémoire locale dans le tampon. Comme il n'est pas synchronisé avec le lecteur, si l'écrivain termine avant que le l'écrivain lise la donnée, alors l'adresse est détruite en même temps que le processus et la donnée est corrompue. Une solution pour pallier ce problème et de créer un espace mémoire (avec malloc) pour la donnée que l'on veut écrire, et donc donner son adresse à l'écrivain.

2.7 Fermeture d'un canal

Comme nous l'avons expliqué plus haut, la fermeture du canal empêche les écrivains d'écrire de nouvelles données. Quand aux lecteurs, ils peuvent lire des données jusqu'à que le tampon soit vide.

Pour les canaux asynchrones, le problème est que si des écrivains sont endormis sur le sémaphore `sem_c` alors ils pourront être réveillé uniquement par un lecteur et si des lecteurs sont endormis sur le sémaphore `d` alors ils pourront être uniquement réveillé par des écrivains. A la fermeture du canal, si le tampon est plein, des écrivains peuvent être restés endormis sur le sémaphore `sem_c` et donc lorsqu'un lecteur viendra lire une donnée, il réveillera les écrivains endormis, qui pourront enfin écrire leurs données, ce qui est en contradiction avec la fermeture du canal. Ainsi lorsqu'on ferme le canal, on doit réveiller tous les écrivains endormis pour y mettre fin. De la même manière pour les processus lecteurs qui restent endormis sur le sémaphore `sem_d` (le tampon est donc vide), aucun écrivain ne pourra écrire de nouvelles données et puis réveiller les lecteurs endormis. Ainsi, la fermeture du canal réveillera tous les lecteurs endormis.

De la même manière pour les canaux synchrones, si un lecteur est endormi sur le sémaphore `sem_c` ou si un écrivain est endormi sur le sémaphore `sem_a`, la fermeture du canal entraîne le réveil de ce lecteur endormi ou de cet écrivain endormi.

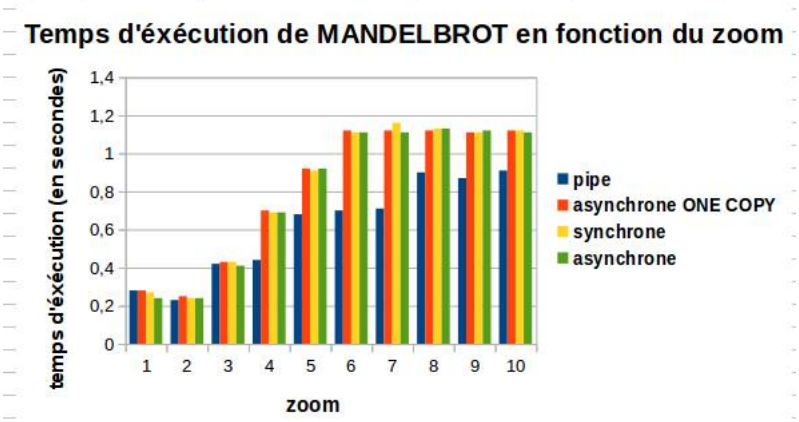
3 Benchmark

Test sur un Lenovo ThinkPad T440 ayant les propriétés :
- 8 Go de RAM
- Intel Core i5-4300U CPU @ 1,90GHz x 4
- Type d'OS : Ubuntu 32 bits

3.1 Mandelbrot

Notre premier exemple de benchmark a été réalisé a partir de l'exemple mandelbrot.c, dans lequel nous avons modifié certaines parties pour pouvoir l'éxécuter avec des pipes et les différents canaux que nous avons implémentés. Pour les tester, nous avons défini une maccro MODE qui lorsqu'elle vaut 0 on compile le programme avec les canaux simples, lorsqu'elle vaut 1 avec les canaux non-apparenté, lorsqu'elle vaut 2 avec les canaux à une copie, et lorsqu'elle vaut 3 avec les pipes.

pipe	asynchrone ONE COPY	synchrone	asynchrone
0,28	0,28	0,27	0,24
0,23	0,25	0,24	0,24
0,42	0,43	0,43	0,41
0,44	0,7	0,69	0,69
0,68	0,92	0,91	0,92
0,7	1,12	1,11	1,11
0,71	1,12	1,16	1,11
0,9	1,12	1,13	1,13
0,87	1,11	1,11	1,12
0,91	1,12	1,12	1,11

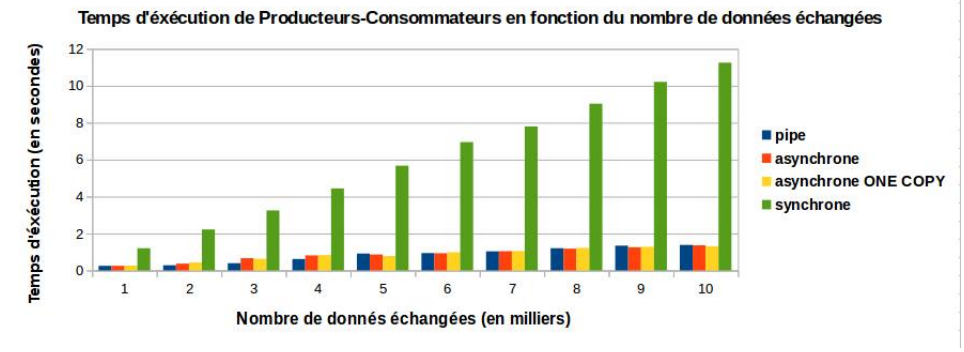


Dans ce graphique, on observe que les éxécutions du mode synchrone, asynchrone, asynchrone ONE COPY sont similaires mais que les pipes sont plus rapides.

3.2 Producteur-Consommateur

Notre second test à été fait sur un programme de type producteur-consommateur. On remarque que les canaux synchrones sont nettement plus lents que les trois autres structures.

pipe	asynchrone	asynchrone ONE COPY	synchrone
0,25	0,25	0,26	1,2
0,28	0,37	0,43	2,22
0,39	0,66	0,63	3,25
0,62	0,81	0,83	4,44
0,91	0,86	0,78	5,67
0,94	0,93	0,98	6,95
1,03	1,04	1,05	7,8
1,2	1,18	1,22	9,03
1,34	1,25	1,28	10,21
1,38	1,36	1,31	11,25



Dans ce deuxième graphique, on observe que les pipes sont légèrement plus rapides que le mode asynchrone et le mode asynchrone ONE COPY. Cependant plus on augmente le nombre de données échangées et plus l'éxécution des pipes, du mode asynchrone et du mode asynchrone ONE COPY sont similaires. Par contre le mode est largement plus lent que ces derniers.