Operateur Channel en Go

Ensimag 2A

1 Opérateur Channel en Go

Le language de programmation **go** (cf. `http://golang.org`) inclut un outil de synchronisation original : les *channels*. Le principe de fonctionnement est celui d'un schéma producteur-consommateur avec un tampon borné de taille programmable.

```
package main
import (
        "fmt"
)

func a(c0 chan int, c1 chan int) {
        c1 <- 1
        c1 <- 2 // bloquant tant que b n'a pas retirer le premier
        c0 <- 3 // bloquant tant que b ne retire pas le 3
}

func b(c0 chan int, c1 chan int, fini chan int) {
        i := <-c1
        j := <-c1
        k := <-c0
        fmt.Printf("%d %d %d\n", i ,j, k)
        fini <- 1
}

func main() {
        c0 := make(chan int, 0) // cree un channel sans tampon, synchrone
        c1 := make(chan int, 1) // cree un channel avec un tampon de 1 int
        fini := make(chan int) // cree un channel synchrone
        go a(c0, c1) // lance le thread 1
        go b(c0, c1, fini) // lance le thread 2
        <-fini // attendre que b soit fini
}
```

2 Réimplanter le channel de Go en C

Le but est d'implanter un module C définissant une structure similaire, défini par les fonctions suivantes :

```
/* Retourne un channel alloué et initialisé. */
struct chan *chan_init(int nb_cases, int taille_case);

/* Envoi d'un message dans le channel. */
void chan_send(struct chan *c, void *mesg)

/* Réception d'un message depuis le channel. */
void chan_recv(struct chan *c, void *dest);

/* Libération d'un channel. */
void chan_destroy(struct chan *c);
```

La sémantique des channels est la suivante :

— le nombre de cases du tampon, et leur taille, est choisi à l'initialisation de la structure `struct channel` par un appel à `chan_init` ;
— l'opération `chan_send` copie le message dans une case du tampon. Elle est bloquante si le tampon est plein ;
— l'opération `chan_recv` retire un message du tampon et le copie à l'adresse `dest`. Elle est bloquante si le tampon est vide ;
— l'opération `chan_destroy` libère la mémoire allouée par `chan_init` ;
— Par défaut, le tampon est de taille 0. Le channel synchronise alors un dépot et un retrait : il n'y a pas de copie intermédiaire du message, juste une copie directe depuis l'adresse `mesg` du producteur vers l'adresse `dest` du consommateur.

Pour manipuler la mémoire vous utiliserez les fonctions classiques de la libc suivantes :

```
/* Alloue _taille_ octets en mémoire. */
void *malloc(size_t taille);

/* Libère la zone mémoire pointée par _ptr_. */
void free(void *ptr);

/* Copie la zone mémoire pointée par _source_ à l'adresse pointée */
/* par _destination_. Les zones mémoire peuvent se chevaucher.    */
void *memmove(void *destination, const void *source, size_t taille);
```

Pour le code réalisé avec les moniteurs dans ce problème, vous utiliserez la sémantique des PThread (Java, etc.) pour la fonction signal, c'est-à-dire qu'elle ne donne pas la main immédiatement au processus réveillé.

3 Channel de taille 0

1. Proposez une implantation des channels qui fonctionne uniquement pour des tampons de taille 0. Les fonctions seront nommées `chan_init_0`, `chan_send_0`, `chan_recv_0`, `chan_destroy_0`. Elles ont les mêmes paramètres que les fonctions complètes. Vous préciserez le contenu de la structure `struct channel`.

Solution:

```
struct channel {
  int taille_case;
  mutex_t m;
  cond_t fprod;
  cond_t prod1;
  cond_t fconso;
  cond_t conso1;
  void *msg;
  void *dest;
}


int chan_init_0(chan *ch, int nb_cases, int taille_case) {
  ch->taille_case = taille_case; return 0;
}

int chan_destroy_0(chan *ch, int nb_cases, int taille_case) { }

int chan_send_0(chan *c, void *msg) {
  mutex_lock(& ch->m);
  while(sas_prod) wait(&ch->fprod, &ch->m);
```

```
    ch->sas_prod = true; ch->sas_msg = msg;

    if (! sas_conso) {
      wait(&ch->prod1, &ch->m);
    } else {
      memmove(ch->sas_msg, ch->sas_dest, ch->taille_case);
      signal(ch->conso1);
    }

    ch->sas_prod = false;

    mutex_unlock(& ch->m);
  }

  int chan_recv_0(chan *c, void *recv) {
    mutex_lock(& ch->m);
    while(sas_conso) wait(&ch->fprod, &ch->m);

    ch->sas_conso = true; ch->sas_dest = dest;

    if (! sas_prod) {
      wait(&ch->conso1, &ch->m);
    } else {
      memmove(ch->sas_msg, ch->sas_dest, ch->taille_case);
      signal(ch->prod1);
    }

    ch->sas_prod = false;

    mutex_unlock(& ch->m); }
```

4 Channel avec un tampon de taille arbitraire

1. Proposez une implantation complète des channels qui fonctionne pour toutes les tailles de tampon, y compris 0.

Solution: Brute force : reprendre la solution 0 et faire un prod-conso classique. Il est possible de le faire en un coup, mais c'est plus technique.

```
int chan_send(chan *c, void *msg) {
  if (ch->nb_cases == 0)
    chan_send_0(c, msg);
  else { // prod/conso classiques
    mutex_lock(& ch->m);
    while(ch->nb_msg == ch->nb_cases)
      wait(&ch->fprod, &ch->m);

    memmove(msg, ch->tampon + ie * ch->taille_case, ch->taille_case);

    ie = (ie + 1) % ch->nb_cases;
    ch->nb_msg ++;

    signal(ch->fconso);
```

```
    mutex_unlock(& ch->m);
  }
}

int chan_recv(chan *c, void *dest) {
  if (ch->nb_cases == 0)
    chan_send_0(c, msg);
  else { // prod/conso classiques
    mutex_lock(& ch->m);
    while(ch->nb_msg == 0)
      wait(&ch->fconso, &ch->m);

    memmove(ch->tampon + il * ch->taille_case, dest, ch->taille_case);

    il = (il + 1) % ch->nb_cases;
    ch->nb_msg --;

    signal(ch->fprod);
    mutex_unlock(& ch->m);
  }

}
```