Concurrence

Problème de la section critique

Tout programme a deux sections :

- une section non critique qui peut être exécutée en parallèle de n'importe quel thread,
- une section critique qui ne peut être exécutée que par un seul thread à la fois.

Dijkstra expose ce problème dans son article « *Solution of a Problem in Concurrent Programming Control*[1] » et liste les contraintes à satisfaire si on veut construire une solution au problème des sections critiques.

1. https://dl.acm.org/doi/pdf/10.1145/365559.365617

- La solution doit considérer tous les threads de la même façon et ne peut faire aucune hypothèse sur la priorité relative des différents threads.
- La solution ne peut faire aucune hypothèse sur la vitesse relative ou absolue d'exécution des différents threads. Elle doit rester valide quelle que soit la vitesse d'exécution non nulle de chaque thread.
- La solution doit permettre à un thread de s'arrêter en dehors de sa section critique sans bloquer l'accès à la section critique pour les autres thread.
- Si un ou plusieurs threads souhaitent entamer leur section critique, aucun de ces threads ne doit pouvoir être empêché indéfiniment d'accéder à sa section critique.

En C, on va utiliser des verrous ou **mutex** (abréviation de mutual exclusion) pour résoudre le problème de la section critique.

On peut schématiquement représenter un mutex comme étant une structure de données qui contient deux informations :

- la valeur actuelle du mutex (locked ou unlocked),
- une queue contenant l'ensemble des threads qui sont bloqués en attente du mutex.

Les mutex sont fréquemment utilisés pour protéger l'accès à une zone de mémoire partagée. Ils sont partagés entre les threads.

Le principe est le suivant :

- un thread qui veut accéder à des données partagées demande le verrou.
- Si celui-ci est libre, il l'obtient et continue son exécution,
- sinon, il bloque jusqu'à ce que le verrou soit libéré.
- Lorsque le thread a terminé avec les données partagées, il libère le verrou.

En pratique, on commence par déclarer et initialiser un verrou :

```
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
```

Puis, lorsqu'on veut protéger une section de code, on fait appel à

- `int pthread_mutex_lock(pthread_mutex_t *mutex);` pour prendre un verrou,
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);` pour libérer un verrou.

Ce qui donne

```
pthread_mutex_lock(&verrou);
// section critique...
pthread_mutex_unlock(&verrou);
```

En pratique, une section critique étant une zone de mémoire partagée peut correspondre à :

- une variable globale,
- une variable sur le tas (pointeur),
- un fichier.

Attention, on utilise des verrous différents pour protéger les accès à différentes zones de mémoire.

Concurrence

Les mutex

```
#define LENTAB 15
int var = 0;
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;

void *serve(void *arg) {
  pthread_mutex_lock(&verrou);
  var += 1;
  pthread_mutex_unlock(&verrou);
  return NULL;
}

int main(void){
  int compt = 0;
  pthread_t tpthread[LENTAB];

  while(compt < LENTAB){
    pthread_t thread;

    if (pthread_create(&tpthread[compt], NULL, serve, NULL) == -1)
      perror("pthread_create");
      continue;
    }
    compt++;
  }

  for(int i=0; i<LENTAB; i++)
    pthread_join(tpthread[i], NULL);

  return 0;
}
```

VI - Les options de socket

Les options de socket

setsockopt

La fonction

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

permet de modifier les options de socket.

- `sockfd` : descripteur de la socket à modifier
- `level` : indique le protocole de la suite auquel s'adresse l'appel. Valeurs possibles : `IPPROTO_IP`, `IPPROTO_TCP`, `IPPROTO_UDP` ou, pour les options plus générales, `SOL_SOCKET`
- `optname` : nom de l'option à modifier
- `optval` : pointeur sur la nouvelle valeur de l'option de longueur `optlen`

Les options de socket

getsockopt

La valeur courante d'une option peut être consultée avec la fonction

```
int getsockopt(int sockfd, int level, int optname,
                           void *val, socklen_t *len);
```

L'Internet est en phase de transition de l'IPv4 vers l'IPv6.

Une application qui ne communique qu'en IPv4 peut devenir obsolète.

Une application qui ne communique qu'en IPv6 n'est pas encore très utile.

On veut donc écrire des applications dites *double-stack*, c'est-à-dire qui communiquent en IPv4 ou en IPv6 selon la situation. Pour cela, on peut programmer une application qui :

- soit utilise deux sockets séparées, une pour IPv4 et une pour IPv6,
- soit utilise une socket polymorphe.

Les options de socket

Socket polymorphe

Une socket IPv6 écoutant sur un port peut, sur certains systèmes, accepter les connexions IPv4. On parle alors de **socket polymorphe**.

Un socket polymorphe acceptant une connexion IPv4, traduit l'adresse IPv4 en une adresse IPv6 dite ***IPv4-mapped*** :

- L'adresse IPv4 `w.x.y.z` se traduit en l'adresse IPv4-mapped `::ffff:w.x.y.z`
  `127.2.3.4` → `::ffff:127.2.3.4` (ou `::ffff:ff02:304`)
- Ces adresses sont des représentations pour un usage interne de l'hôte. Elles ne correspondent à rien sur le réseau.
- Si on a une socket polymorphe liée à un port et qu'un client IPv4 se connecte, alors `accept` retourne une adresse IPv4-mapped.

Les options de socket

Socket polymorphe

Pour qu'une socket IPv6 devienne polymorphe, Il faut désactiver l'option `IPV6_V6ONLY` au niveau `IPPROTO_IPV6`

```
int no = 0;
int r = setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, &no, sizeof(no));
if(r < 0)
    fprintf(stderr, "échec de setsockopt() : (%d)\n", errno);
```

Cela doit être fait **avant** le `bind` pour un serveur.

Mais peut-on écrire un client qui se connecte en IPv4 ou IPv6 ?

Oui, en utilisant `getaddrinfo` dans le client IPv6 avec les bons *flags* pour la variable `hints`.

```
hints.ai_flags = AI_V4MAPPED | AI_ALL;
```

Alors si un serveur accepte uniquement les connexions IPv4 et tourne en local sur le port 7777, on obtient :

```
Cours5$ ./client6 localhost 7777
adresse : IP: ::1 port: 7777
adresse : IP: ::ffff:127.0.0.1 port: 7777

adresse creee !
IP: ::ffff:127.0.0.1 port: 7777
```

```
Cours5$ ./client6 127.0.0.1 7777
adresse : IP: ::ffff:127.0.0.1 port: 7777

adresse creee !
IP: ::ffff:127.0.0.1 port: 7777
```

Les options de socket

Réutilisation d'un numéro de port

Lorsqu'une application ferme sa socket avec `close` ou si elle plante, la socket passe dans un état `TIME-WAIT` et le système la maintient dans cet état pendant quelques secondes, voire quelques minutes.

Il est alors impossible de lancer une application qui fait un `bind` sur la même adresse et le même port pendant ce temps. L'état `TIME-WAIT` représente le temps qu'il faut attendre pour être sûr que l'application distante a bien reçu la demande de fin de connexion.

Pour parer à l'échec du `bind`, il faut ajouter l'option `SO_REUSEADDR` à la socket.

```
int yes = 1;
int r = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
if(r < 0)
    fprintf(stderr, "échec de setsockopt() : (%d)\n", errno);
```

Cela doit être fait **avant** le `bind`.

VII - La lecture sur socket

La lecture sur socket

TCP est un protocole par flux

$\Rightarrow$ on ne peut considérer que les données envoyées par un `send` pourront être entièrement lues par un unique `recv`.

Si deux applications A1 et A2 communiquent et

- A1 envoie 100 octets à A2 avec un unique `send`,
- alors on ne peut pas savoir avec combien de `recv`, A2 pourra lire les 100 octets.

$\Rightarrow$ on ne peut considérer que les données envoyées par k `send` pourront être entièrement lues par k `recv` lisant la quantité de données envoyée par le send correspondant.

Si deux applications A1 et A2 communiquent et

- A1 envoie avec 3 `send`, 100 octets par `send` à A2,
- alors on ne peut pas savoir avec combien de `recv`, A2 pourra lire les 300 octets.

La lecture sur socket

C'est pourquoi on a besoin de protocoles très précis décrivant les formats des messages échangés.

- Si l'application sait exactement combien d'octets elle doit lire ⇒ boucle sur le `recv` tant que tous les octets n'ont pas été lus.
- Si l'application sait par quelle suite de caractères termine le message qu'elle doit lire ⇒ boucle sur le `recv` tant que la suite de caractères n'a pas été trouvée.
  Attention si par exemple le message termine par les deux caractères `\r\n`, alors il est possible que ces deux caractères ne soient pas récupérés par le même `recv`. Le message d'un premier `recv` peut terminer par `\r` et le message suivant commencer par `\n`.
- Si l'application sait que l'autre application fermera la connexion juste après l'envoi, elle peut faire une boucle sur `recv` tant que ce dernier ne retourne pas 0.

La lecture sur socket

→ Pour le cas où une suite de caractères annoncent la fin de la transmission, il faut pouvoir rechercher cette suite dans le buffer.

→ Il peut être nécessaire d'accumuler dans un buffer les messages reçus (ou une partie des messages reçus) jusqu'à obtenir le bon nombre d'octets ou la suite de caractères attendue ou un retour nul de `recv`.

Pour cela, vous avez à votre disposition les fonctions :

- `str...` : si le buffer est une chaîne de caractères (termine par le caractère \0).
  voir `strchr`, `strstr`, `strcpy`...

- `mem...`
  voir `memchr`, `memmem` (non POSIX), `memmove`...

En mode connecté, des **situations de blocage** peuvent arriver dès qu'une des applications communicantes ne respecte pas le protocole.

Si l'application A1 est en attente d'un message d'une application A2 et que cette dernière n'envoie pas la totalité du message attendu, alors A1 peut bloquer indéfiniment.

Pour un serveur, cela peut être problématique car cela permet des **attaques**.