Synchronisation dans le scheduler

HARROCH Yohanan, EL AMRAOUI Adel

April 2019

1 Introduction

Voici l'algorithme principal du scheduler qui contient trois points de synchronisation. On utilise pour cela la structure std::sync::mpsc composé d'au moins une structure Sender et une structure Receiver. On s'appuiera en particulier des méthodes recv et send qui nous permettrons de simuler respectivement les wait et notify. on s'aidera aussi de trois variables, counter wait, counter write et counter, utilisées en tant que compteur partagés entre les threads. Cette partie s'occupe de l'algorithme en faisant abstraction de l'architecture client/serveur, les key et plain sont donc fourni.

```
let mut cpt = counter.lock().unwrap();
   if *cpt == -1 {
       if size != 1 {
           r.lock().unwrap().recv().unwrap(); // premier wait
       }
       *cpt = 0;
   }
   let index = *cpt;

   *cpt += 1;
   std::mem::drop(cpt);
   assert!(index >= 0 && index <= size as i32);
   let mut buff = buf.lock().unwrap();

   let plain = lock_plain.lock().unwrap();

   let key = lock_key.lock().unwrap();

   buff[index as usize].key = *key;
   buff[index as usize].plain = *plain;
   let local_plain = *plain;
   let local_key = *key;

   std::mem::drop(buff);
```

```
25      std::mem::drop(plain);
26
27      std::mem::drop(key);
28
29      if index == size as i32 - 1 {
30          rd.lock().unwrap().recv().unwrap(); //deuxième wait
31          let mut buff = buf.lock().unwrap();
32          for i in 0..(size) {
33              buff[i].plain ^= buff[i].key;
34              thread::sleep(time::Duration::from_millis(1));
35          }
36          let result = buff[index as usize].plain;
37          std::mem::drop(buff);
38
39          let mut cpt = counter.lock().unwrap();
40          *cpt = -1;
41               sen.lock().unwrap().send(()).unwrap(); //notify du troisième wait
42          }
43          std::mem::drop(cpt);
44      } else {
45          let mut c_wait = counter_wait.lock().unwrap();
46          *c_wait += 1;
47          if *c_wait == (size as i32) - 1 {
48              sd.lock().unwrap().send(()).unwrap(); //notify du deuxième wait
49              *c_wait = 0;
50          }
51          std::mem::drop(c_wait);
52           rec.lock().unwrap().recv().unwrap(); //troisième wait
53          let buff = buf.lock().unwrap();
54          let result = buff[index as usize].plain;
55          assert!(result == local_plain ^ local_key);
56          let mut c = counter_write.lock().unwrap();
57          *c += 1;
58          if *c == (size as i32) - 1 {
59              s.lock().unwrap().send(()).unwrap(); //notify du premier wait
60              *c = 0;
61          }
62          std::mem::drop(buff);
63      }
```

- Le premier point de synchronisation à la ligne 4, accompli par la méthode recv du Receiver r, empêche d'autres threads d'écrire dans le buffer tant que les premières tâches n'ont pas fini de lire le résultat calculé par la dernière thread. L'attente est donc terminée à la ligne 59, lorsque la variable counter write, qui sert de compteur pour le nombre d'écriture, atteint size -1.

- Le deuxième wait, réalisé par la méthode recv du Receiver rd à la ligne 30, met en attente la dernière thread qui est libéré par le send du Sender sd à la ligne 48. Le send est envoyé seulement lorsque la variable counter wait, qui est incrémenté après qu'une thread a écrit dans le buffer son plain et key, atteint size -1. Ce wait permet donc d'attendre que toute les threads terminent d'écrire dans le buffer avant que le dernier commence le calcul.

- Le troisième wait est implementé à la ligne 52 par le recv du Receiver rec. Le send est réalisé après que la derniere thread finisse de faire le calcul. Ce point de synchronisation met donc en attente les thread pendant le temps du calcul.

2 Version avec une Architecture Client-Serveur

2.1 main Serveur

```
1  use std::io::prelude::*;
2  use std::net::TcpStream;
3  use std::net::TcpListener;
4  use web_server::{ThreadPool, Cell};
5  use std::sync::{Arc, Mutex, mpsc};
6  use rand::prelude::*;
7  use std::env;
8  use std::{thread, time};
9
10
11 fn main() {
12     let args: Vec<String> = env::args().collect();
13     let size: usize = args[1].parse().unwrap();
14     let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
15
16     let vec: Vec<Cell> = vec![Cell { plain: 0, key: 0 }; size];
17     let vec: Arc<Mutex<Vec<Cell>>> = Arc::new(Mutex::new(vec));
18
19     let counter = Arc::new(Mutex::new(0));
20     let counter_write = Arc::new(Mutex::new(0));
21     let pool = ThreadPool::new(size);
22     let (sender, receiver) = mpsc::channel();
23     let (s, r) = mpsc::channel();
24     let r = Arc::new(Mutex::new(r));
25     let s = Arc::new(Mutex::new(s.clone()));
26
27
28     let (sd, rd) = mpsc::channel();
29     let rd = Arc::new(Mutex::new(rd));
30     let sd = Arc::new(Mutex::new(sd.clone()));
```

```
    let counter_wait = Arc::new(Mutex::new(0));


    let receiver = Arc::new(Mutex::new(receiver));
    let sender = Arc::new(Mutex::new(sender.clone()));

    for stream in listener.incoming() {
        let stream = stream.unwrap();
        let sender = Arc::clone(&sender);
        let receiver = Arc::clone(&receiver);
        let r = Arc::clone(&r);
        let s = Arc::clone(&s);

        let rd = Arc::clone(&rd);
        let sd = Arc::clone(&sd);
        let counter_wait = Arc::clone(&counter_wait);


        let counter = Arc::clone(&counter);
        let buffer = Arc::clone(&vec);
        let counter_write = Arc::clone(&counter_write);

        pool.execute(move || {
            handle_connection(stream, counter, buffer, sender, receiver, size, counter_write
        });
    }
}


pub fn handle_connection(mut stream: TcpStream, counter: Arc<Mutex<i32>>, buf: Arc<Mutex<Vec
                         sen: Arc<Mutex<mpsc::Sender<()>>>, rec: Arc<Mutex<mpsc::Receiver<()
                         size: usize, counter_write: Arc<Mutex<i32>>, s: Arc<Mutex<mpsc::Sen
                         r: Arc<Mutex<mpsc::Receiver<()>>>, sd: Arc<Mutex<mpsc::Sender<()>>>
                         rd: Arc<Mutex<mpsc::Receiver<()>>>, counter_wait: Arc<Mutex<i32>>)
    let mut buffer = [0; 8];
    stream.read(&mut buffer).unwrap();
    let plain = u64::from_be_bytes(buffer);
    let key = rand::thread_rng().gen();


    let mut counter_thread = counter.lock().unwrap();
    if *counter_thread == -1 {
        let _received = r.lock().unwrap().recv().unwrap();
        *counter_thread = 0;
    }
    let index = *counter_thread;
```

```
77      *counter_thread += 1;
78      std::mem::drop(counter_thread);
79      assert!(index >= 0 && index <= size as i32);
80
81      let mut buff = buf.lock().unwrap();
82      buff[index as usize].key = key;
83      buff[index as usize].plain = plain;
84      std::mem::drop(buff);
85
86
87      if index != (size as i32 - 1) {
88          let mut c_wait = counter_wait.lock().unwrap();
89          *c_wait += 1;
90          if *c_wait == (size as i32) - 1 {
91              sd.lock().unwrap().send(()).unwrap();
92              *c_wait = 0;
93          }
94          std::mem::drop(c_wait);
95          let _received = rec.lock().unwrap().recv().unwrap();
96          let buff = buf.lock().unwrap();
97          let result = buff[index as usize].plain;
98          std::mem::drop(buff);
99          assert!(result == key ^ plain);
100         stream.write(&u64_to_array_of_u8(result)).unwrap();
101         let mut c_write = counter_write.lock().unwrap();
102         *c_write += 1;
103         if *c_write == (size as i32) - 1 {
104             s.lock().unwrap().send(()).unwrap();
105             *c_write = 0;
106         }
107     } else {
108         rd.lock().unwrap().recv().unwrap();
109         let mut buff = buf.lock().unwrap();
110         for i in 0..(size) {
111             buff[i].plain = buff[i].plain ^ buff[i].key;
112             thread::sleep(time::Duration::from_millis(1));
113         }
114         assert!(buff[size - 1].plain == key ^ plain);
115         stream.write(&u64_to_array_of_u8(buff[size - 1].plain)).unwrap();
116         std::mem::drop(buff);
117
118         let mut counter_thread = counter.lock().unwrap();
119         *counter_thread = -1;
120         std::mem::drop(counter_thread);
121
122         for _ in 0..(size - 1) {
```

```
123                 sen.lock().unwrap().send(()).unwrap();
124             }
125         }
126     }
127
128
129     fn u64_to_array_of_u8(x: u64) -> [u8; 8] {
130         let b1: u8 = ((x >> 56) & 0xff) as u8;
131         let b2: u8 = ((x >> 48) & 0xff) as u8;
132         let b3: u8 = ((x >> 40) & 0xff) as u8;
133         let b4: u8 = ((x >> 32) & 0xff) as u8;
134         let b5: u8 = ((x >> 24) & 0xff) as u8;
135         let b6: u8 = ((x >> 16) & 0xff) as u8;
136         let b7: u8 = ((x >> 8) & 0xff) as u8;
137         let b8: u8 = (x & 0xff) as u8;
138         return [b1, b2, b3, b4, b5, b6, b7, b8];
139     }
```

- Dans cette version on commence par récupérer le taille du pool de thread (lignes 12-13) puis on lance l'écoute du serveur en créant une socket TCP sur le port 7878 (ligne 14).
- On crée ensuite un vecteur qui contiendra les plain et key de chaque client (plain et key sont des entiers non signés sur 64 bits)
- Lignes 19 à 35: On met ici en place les dispositifs de synchronisations de manière analogue au scheduler sans architecture Client-Serveur(mutex, mpsc...).
- Les clients vont envoyer un nombre de requêtes égal à la taille du pool de thread(size) modulo size. Ces requêtes correspondent au slice d'un u64 sous la forme d'un tableau de 8 u8.
- Lors d'une connection d'un client le serveur récupère l'entier correspondant au tableau d'octets buffer (ligne 67) dans plain et procède à un ET logique (bit à bit) entre plain et un u64 key(ligne 68) qui est généré aléatoirement. On traite chaque requête d'un client comme avec un thread worker disponible dans le pool de thread, il est donc nécessaire de réutiliser les mécanismes de synchronisation implémentés en première partie pour garantir l'exclusion mutuelle lors de l'accès au buffer plain notamment.