Operativsystem ID1206
Gröna trådar

Axel Karlsson

December 16, 2021

1. Inledning

I denna rapport har jag redovisat mitt arbete med seminarieuppgift 3, Green, i kursen ID1206 Operativsystem. Uppgiften var att implementera ett trådbibliotek likt c-biblioteket "pthreads". I del 2 har jag redovisat min implementation och i del 3 har jag jämfört mitt bibliotek med pthreads.

1.2 Vad är ett trådbibliotek?

Ett trådbibliotek är, som man hör på namnet, ett bibliotek som ger oss möjlighet att skapa och köra trådar i ett program. Att programmera med flera trådar är inte ett lätt jobb och därför finns det även funktioner som låter oss (till viss del) styra hur dessa trådar exekverar, och på så sätt kan vi förhoppningsvis skriva ett buggfritt program. Mitt biblioteks api är baserat på pthreadbiblioteket och jag har implementerat följande funktioner:

- pthread_create() & _yield() & _join()
- pthread_mutex_lock() & _unlock() & _init()
- pthread_cond_init() & _wait() & _signal()

2. Implementation

En stor del av koden till denna uppgift var given i instruktionerna och jag har försökt att inte ha med de delarna i rapporten. Detaljerna i min implementation blir som en konsekvens av det svåra att förstå om man själv inte har gjort uppgiften.

2.1 Hantera kontexter

Den första delen av api:n jag implementerade var green_create(), green_yield() och green_join(). De fyller samma funktion som pthreadfunktionerna med liknande namn; create() initierar en ny tråd, yield() flyttar den tråden som körs och lägger den längst bak i exekveringskön och join() suspenderar den tråden som körs tills en given tråd har terminerat.

Trådar som är redo att exekveras läggs i en kö "ready queue" som jag valde att implementera som en länkad lista, med next pekaren i green_t strukturen.

Den största utmaningen med denna del av uppgiften var för mig att implementera den interna funktionen green_thread() på ett korrekt sätt. Funktionens syfte är kort sagt att ta trådar från ready-kön, exekvera dem och sedan terminera dem. När jag först skrev denna funktion förstod jag inte hur den hängde ihop med resten av programmet och jag har därför haft många buggar som härstammar från denna funktion.

```
1  void green\_thread() {
2    ...
3    // Denna if sats var lange ett fragetecken for mig
4    if(this->join != NULL) {
5      queue\_add(this->join, readyqueue);
6    }
7    ...
8
9    green_t* next = queue\_remove(readyqueue);
10   runnign = next;
11   setcontext(next->context);
```

2.2 Conditional variables

I denna del av uppgiften implementerade jag villkorliga variabler. En tråd exekverar tills den hamnar vid en villkorlig variabel, och blir då suspenderad tills variablen signal funktion anropas och tråden väcks igen. Api:n för dessa är cond_init(), cond_wait & cond_signal(). I min implementation finns green_cond_t structen, som skapas med init(). Den innehåller en länkad lista med trådar som väntar på variablen. Vi kan sedan lägga till eller ta bort trådar i denna lista med wait respektive signal, och på så sätt suspendera eller väcka dem. Jag hade ingen stor svårighet med denna del av uppgiften.

```
1  typedef struct green_cond_t {
2    green_t* suspended_list; // Lankad lista som haller koade
     tradar
3    } green_cond_t;
```

2.3 Timer interrupt

I denna del implementerade jag timerinterrupts av trådarna. Varje 100 millisekunder anropas en funktion som liknar yield, i att den sätter tråden som körs längst baj u exekveringskön och startar nästa. Detta möjliggör samtida (concurrent) exekvering av flera trådar utan att de explicit behöver släppa fram varandra (till exempel genom att anropa yield). Att byta tråd på fel ställe i programmet kan orsaka buggar och därför har även delar av koden "skyddats" från timerinterrupts.

I denna del var nästan all kod given i instruktionerna, det enda problemet var att tänka ut vilken kod som skulle skyddas från timerinterrupts. Min tankeprocess var att skydda all kod som ändrar i någon lista eller en global pekare, och det blev i princip all kod.

2.4 Mutex

Det sista jag implementerade var en mutex funktionalitet. Mutex låter oss plac-

era “lås” i koden som trådar kan låsa eller låsa upp, och kort sagt kan vi se till att vissa delar av kod endast exekveras av en tråd i taget.

Även i denna del var majoriteten av koden given i uppgiften. Jag implementerade suspended-kön med en länkad lista lik den i green_cond_t strukturen. Api:n implementerad här är green_mutex_init(), green_mutex_lock() & green_mutex_unlock().

2.5 Förbättring av green_cond_wait()

Denna del var svår. Målet var att implementera en funktion som kan vänta på ett lås tills en signal kommer, och sedan ta det låset. Detta är i princip en kombination av tidigare skrivna funktioner, men det är viktigt att ha dem i samma funktion (som en atomär operation) för att de inte ska bli avbrutna av timerinterrupt. Jag hade till en början svårt att förstå vad syftet med denna funktion var, men när jag förstod att det egentligen är en kombination av funktionaliteten i green_cond_wait() och mutex var det inte så svårt.

```
1       green_cond_wait(cond, mutex) {
2         // Forsta delen av den tidigare cond_wait()
3         if(mutex != NULL) {
4           // I princip mutex_unlock()
5           ...
6         }
7         // Resten av den tidigare cond_wait()
8         ...
9         if(mutex != NULL) {
10          // I princip mutex_lock()
11        }
12        return;
13      }
14
```

3. Benchmarks

Jag har jämfört exekveringstid av de benchmark-funktioner jag har skapat med både pthreadbiblioteket och mitt bibliotek, se kodexempeln nedan. Varje benchmark har körts med två trådar samtidigt och antalet iterationer (variablen i) har testats mellan 100 och 3000 i intervall av 100.

3.1 yield() benchmark

I detta benchmark anropartrådarna yield varje iteration.

```
1       void* bmark_yield(int i) {
2         ...
3         while(i > 0) {
4           fib(10); // Rakna 10:e fibbonaccitalet
5           i--;
6           yield(); // Detta ar pthread eller mitt bibliotek
7         }
8       }
9
```

3.2 Timerbenchmark

I detta benchmark låter vi timerinterrupts byta mellan trådar.

```
1       void* bmark_timer(int i) {
2         ...
```

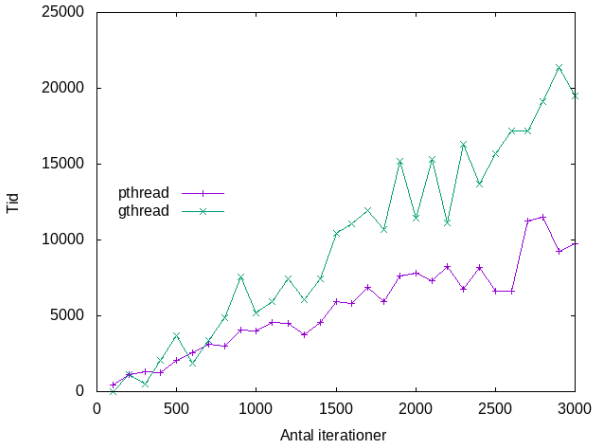

Figure 1: bmark_yield(): Exekveringstid(ticks) / Iterationer

```
        while(i > 0) {
          fib(10); // Rakna 10:e fibbonaccitalet
          i--;
        }
      }
```

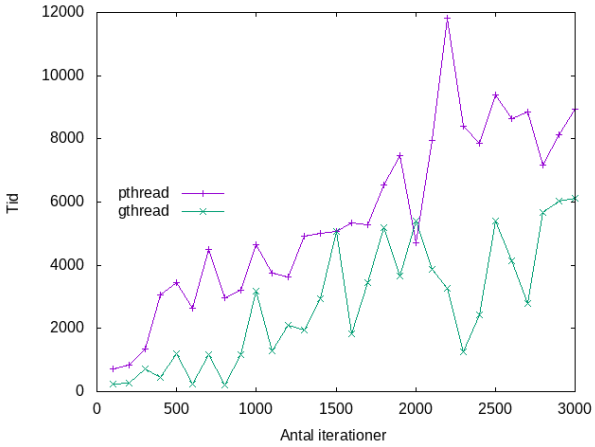

Figure 2: bmark_timer(): Exekveringstid(ticks) / Iterationer

3.3 Mutexbenchmark

I detta benchmarks itereras en global variabel, som är skyddad av ett lås och en flagga. Detta test är samma som det givet i uppgiftinstruktionerna i del 6. Jag har försökt skriva testet i pseudokod, för en mer detaljerad beskrivning hänvisar jag till uppgiftinstruktionerna sida 11.

```
                void* bmark_mutex(int i) {
                  while(i > 0) {
                    mutex_lock(&lock1);
```

```
4                   while(flag != id) {
5                     // En trad ar fast har
6                     cond_wait(&cond1, &lock1);
7                   }
8                   flag = *den andra traden*
9                   counter++;
10                  fib(10); // Rakna 10:e fibonaccitalet
11                  // Den andra traden slapps forbi cond_wait()
12                  cond_signal(&cond1);
13                  mutex_unlock(&lock1);
14                  lim--;
15                }
16              }
17
```

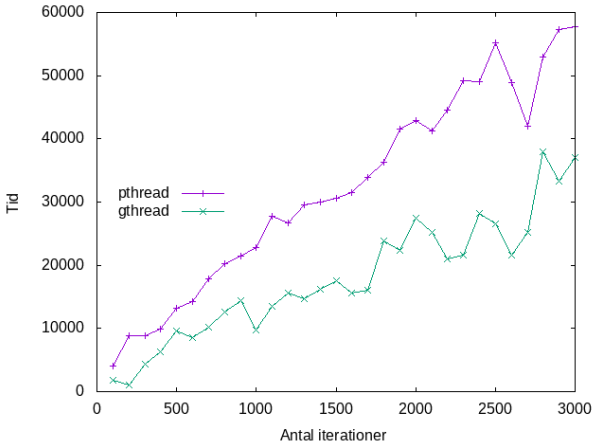

Figure 3: bmark_mutex(): Exekveringstid(ticks) / Iterationer

4. Analys

Resultaten blev inte vad jag förväntade mig, då mitt bibliotek var märkbart snabbare i både timertestet (Figur 2) och mutextestet (Figur 3). I yieldtestet (Figur 1) var pthreadbiblioteket snabbare med stor marginal.

Efter att ha funderat på det så skulle jag gissa att mitt bibliotek har mindre overhead då det är så pass mycket mindre än pthreadbiblioteket. Ett exempel på det är min cond_signal funktion, som innehåller fem rader kod. Jag gissar att den motsvarande pthreadfunktionen är lite längre.
Det påstående säger dock emot resultaten på ett sätt, då mitt bibliotek var snabbare i det test med minst antal funktionsanrop, timertestet, medan pthread-biblioteket var snabbare i yieldtestet. Vad säger det om skillnaderna mellan de olika biblioteken?

5. Reflektion

Överlag är jag nöjd med mitt arbete men det finns fortfarande några buggar kvar som orsakar sporadiska segmenteringsfel. För att besvara frågan i slutet av uppgiftsinstruktionerna så hade jag inte sovit gott om detta bibliotek användes i styrsystemet för ett flygplan.