Excercise Sheet 7

Johannes Koch

May 22, 2018

1 Excercise Sheet 6 Task 2

```c
#define _POSIX_SOURCE
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <ctype.h>
#include <pthread.h>
#include "myqueue.h"

#define gettid() syscall(SYS_gettid)

#define err_t(msg) \
  perror(msg); pthread_exit(NULL)

#define err(msg) \
  perror(msg); _exit(EXIT_FAILURE)

#define THREADS 5
#define NUM_ENTRYS 10000

pthread_mutex_t lock;

void* consumer(void *arg){
  int sum = 0, read;
  while(1){
    // mutex
    pthread_mutex_lock(&lock);
    // read
    if(!empty()){
      read = front();
      pop();
      // check wheter to terminate
      if(read){
        sum += read;
      } else {
        pthread_mutex_unlock(&lock);
```

```
        break;
      }
    }
    // mutex
    pthread_mutex_unlock(&lock);
  }
  fprintf(stdout, "%ld: %d\n", gettid(), sum);
  pthread_exit(NULL);
}

int main (void){
  pthread_t tid[THREADS];

  fprintf(stdout, "creating queue... ");

  create();

  // create THREADS threads
  fprintf(stdout, "done\ncreating %d threads... ", THREADS);
  for(int i = 0; i < THREADS; i++){
    // TODO queue and mutex as arg -> struct
    if( pthread_create(&(tid[i]), NULL, &consumer, NULL) != 0){
      err("pthread_create");
    }
  }

  fprintf(stdout, "done\nwriting to queue...\n");

  for(int i = 0; i < NUM_ENTRYS; i++){
    // mutex
    pthread_mutex_lock(&lock);
    // write
    push(1);
    // mutex
    pthread_mutex_unlock(&lock);
  }
  for(int i = 0; i < THREADS; i++){
    // mutex
    pthread_mutex_lock(&lock);
    // write
    push(0);
    // mutex
    pthread_mutex_unlock(&lock);
  }

  fprintf(stdout, "done\nwaiting for threads to finish...\n");
  // wait for threads
  for(int i = 0; i < THREADS; i++){
    pthread_join(tid[i], NULL);
  }
  fprintf(stdout, "done\ndestroying mutex...");
  // destroy mutex
  pthread_mutex_destroy(&lock);
```

```
  fprintf(stdout, "done\n");
  return EXIT_SUCCESS;
}
```

2 Task 1

In theory when a mutex unsuccessfully tries to lock a mutex it will go to sleep, allowing other processes to run immediately and it will retry when it is woken by another thread which unlocked the mutex.
When a thread unsuccessfully tries to lock a spinlock it will continue to retry until it successfully locks the spinlock or the threads CPU runtime quantum has been exceeded. This results in busy waiting.
If the critical region is only locked for a short time, putting a thread to sleep and waking it again creates a lot of overhead. A spinlock on the other hand would create less overhead but would use 100% of the CPU for a short amount of time. If the critical region is locked for a considerable amount of time or if the System only has a single CPU-Core spinlocks will needlessly occupy recources another thread could have used if mutexes were used.

```
#define _POSIX_SOURCE
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <ctype.h>
#include <pthread.h>
#include "myqueue.h"

#define gettid() syscall(SYS_gettid)

#define err_t(msg) \
  perror(msg); pthread_exit(NULL)

#define err(msg) \
  perror(msg); _exit(EXIT_FAILURE)

#define THREADS 5
#define NUM_ENTRYS 10000

pthread_spinlock_t spinlock;

void* consumer(void *arg){
  int sum = 0, read;
  while(1){
    pthread_spin_lock(&spinlock);
    // read
    if(!empty()){
      read = front();
      pop();
```

```
      // check wheter to terminate
      if(read){
        sum += read;
      } else {
        pthread_spin_unlock(&spinlock);
        break;
      }
    }
    pthread_spin_unlock(&spinlock);
  }
  fprintf(stdout, "%ld: %d\n", gettid(), sum);
  pthread_exit(NULL);
}

int main (void){
  pthread_t tid[THREADS];

  if( pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE) != 0){
    err("pthread_spin_init");
  }

  fprintf(stdout, "creating queue... ");

  create();

  // create THREADS threads
  fprintf(stdout, "done\ncreating %d threads... ", THREADS);
  for(int i = 0; i < THREADS; i++){
    if( pthread_create(&(tid[i]), NULL, &consumer, NULL) != 0){
      err("pthread_create");
    }
  }

  fprintf(stdout, "done\nwriting to queue...\n");

  for(int i = 0; i < NUM_ENTRYS; i++){
    pthread_spin_lock(&spinlock);
    // write
    push(1);
    pthread_spin_unlock(&spinlock);
  }
  for(int i = 0; i < THREADS; i++){
    pthread_spin_lock(&spinlock);
    // write
    push(0);
    pthread_spin_unlock(&spinlock);
  }

  fprintf(stdout, "done\nwaiting for threads to finish...\n");
  // wait for threads
  for(int i = 0; i < THREADS; i++){
    pthread_join(tid[i], NULL);
  }
```

```
  fprintf(stdout, "done\ndestroying spinlock...");
  pthread_spin_destroy(&spinlock);
  fprintf(stdout, "done\n");
  return EXIT_SUCCESS;
}
```

2.1 /usr/bin/time

3 Task 1

Using condition variables and mutexes naturally creates more overhead than simply using mutexes or spinlocks, but on the upside a lot more control is gained, for example in a reader-writer/consumer-producer scenarion it can be controlled which side gets access to the critical region.

```
#define _POSIX_SOURCE
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <ctype.h>
#include <pthread.h>
#include "myqueue.h"

#define gettid() syscall(SYS_gettid)

#define err_t(msg) \
  perror(msg); pthread_exit(NULL)

#define err(msg) \
  perror(msg); _exit(EXIT_FAILURE)

#define THREADS 5
#define NUM_ENTRYS 10000

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int queueIsEmpty = 1;

void* consumer(void *arg){
  int sum = 0, read;
  while(1){
    // mutex
    pthread_mutex_lock(&lock);
    // pause as long as the queue is empty
    while(queueIsEmpty == 1){
      pthread_cond_wait(&cond, &lock);
    }
    // read
```

```
    read = front();
    pop();
    queueIsEmpty--;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&lock);
    // check whether to terminate
    if(read){
      sum += read;
    } else {
      break;
    }
  }
  fprintf(stdout, "%ld: %d\n", gettid(), sum);
  pthread_exit(NULL);
}

int main (void){
  pthread_t tid[THREADS];

  fprintf(stdout, "creating queue... ");

  create();

  // create THREADS threads
  fprintf(stdout, "done\ncreating %d threads... ", THREADS);
  for(int i = 0; i < THREADS; i++){
    if( pthread_create(&(tid[i]), NULL, &consumer, NULL) != 0){
      err("pthread_create");
    }
  }

  fprintf(stdout, "done\nwriting to queue...\n");

  for(int i = 0; i < NUM_ENTRYS; i++){
    // mutex
    pthread_mutex_lock(&lock);
    // write
    while(queueIsEmpty == 0){
      pthread_cond_wait(&cond, &lock);
    }
    push(1);
    queueIsEmpty++;
    pthread_cond_broadcast(&cond);
    // mutex
    pthread_mutex_unlock(&lock);
  }
  for(int i = 0; i < THREADS; i++){
    // mutex
    pthread_mutex_lock(&lock);
    // write
    while(queueIsEmpty == 0){
      pthread_cond_wait(&cond, &lock);
    }
```

```
    push(0);
    queueIsEmpty++;
    pthread_cond_broadcast(&cond);
    // mutex
    pthread_mutex_unlock(&lock);
  }

  fprintf(stdout, "done\nwaiting for threads to finish...\n");
  // wait for threads
  for(int i = 0; i < THREADS; i++){
    pthread_join(tid[i], NULL);
  }
  fprintf(stdout, "done\n");
  return EXIT_SUCCESS;
}
```

3.1 /usr/bin/time