Excercise Sheet 6

Johannes Koch

May 15, 2018

1 Task 1

```c
#define _POSIX_SOURCE
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <ctype.h>
#include <pthread.h> // compile with -pthread
#include <sys/syscall.h>
#include <string.h>

#define gettid() syscall(SYS_gettid)

#define THREADS 10
#define FNAME_SIZE 32

// error handling
#define handle_error(msg) \
  perror(msg); _exit(EXIT_FAILURE)

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

void cleanup_handler(void *fp){
  if (fclose((FILE *) fp) != 0){ handle_error_t("fclose"); }
}

void* writeToFile(void *n){
  sleep(rand()%4);

  // cannot be canceled
  if( pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL) != 0){
    handle_error_t("pthread_setcancelstate");
  }

  // get filename
  char fname[FNAME_SIZE];
```

```
  snprintf(fname, sizeof(fname), "thread%d.txt", *((int *) n));

  // open file
  FILE *fp;
  if( (fp = fopen(fname, "w")) == NULL){
    handle_error_t("fopen");
  }

  // if the thread gets canceled, close the file
  pthread_cleanup_push(cleanup_handler, fp);


  // can be canceled again
  if( pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL) != 0){
    handle_error_t("pthread_setcancelstate");
  }

  // write to file
  fprintf(stdout, "%ld\n", gettid());

  // fprintf(fp, "%lu\n", pthread_self());
  fprintf(fp, "%ld\n", gettid());

  // close file
  pthread_cleanup_pop(1);

  // exit thread
  pthread_exit(NULL);
}

int main (void){
  srand(time(NULL));
  int err, tnum[THREADS];
  pthread_t tid[THREADS];

  // create THREADS threads
  fprintf(stdout, "creating %d threads... ", THREADS);
  for(int i = 0; i < THREADS; i++){
    tnum[i] = i;
    if( (err = pthread_create(&(tid[i]), NULL, &writeToFile, (void *) &
    tnum[i])) != 0){
      handle_error("pthread_create");
    }
  }

  fprintf(stdout, "done\ncanceling threads...\n");

  // for each thread decide if it gets cancelled
  for(int i = 0; i < THREADS; i++){
    if(rand()%2){
      fprintf(stdout, "\tthread %d got canceled\n", i);
      if (pthread_cancel(tid[i]) != 0){
        handle_error("pthread_cancel");
```

```
      }
    }
  }

  fprintf(stdout, "done\nwaiting for threads to finish...\n");

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

  return EXIT_SUCCESS;
}
```

2 Task 2

2.1 myqueue.h

```
#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;
}
```

2.2 std::queue

```
#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 <queue>
#include <iostream>

#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;
std::queue<unsigned> queue;

void* consumer(void *arg){
  int sum = 0, read;
  while(1){
    // mutex
    pthread_mutex_lock(&lock);
    // read
    if(!queue.empty()){
      read = queue.front();
      queue.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 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
    queue.push(1);
    // mutex
    pthread_mutex_unlock(&lock);
  }
  for(int i = 0; i < THREADS; i++){
    // mutex
    pthread_mutex_lock(&lock);
    // write
    queue.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;
}
```