CIS 415 Operating Systems

Assignment #3 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Claire Kolln

Report

Introduction

In this project, it was tasked for students to build a simple publisher/subscriber server that can handle the work of many publishers and many subscribers running at the same time. Entries were published to circular ring buffers for each topic. Like the photo sharing websites of today, the Quacker server is meant to display the content pushed by publishers that is then accessed by subscribers. To free up space in the topic buffers, a cleaning thread accessed each queue and cleared out old content so more content could be pushed to the queue. Thread synchronization was a big task in this project, because with multiple threads wanting access to the same resource we must avoid race conditions.

Background

Because the C language has no built in data structures, it was required for this project to implement the circular ring buffer using structs and then defining functions to perform actions with the structs such as to enqueue an entry, get an entry from the buffer, or dequeue an old entry. To avoid mallocing in my code, I used a macro to define the topic queues. Although my macro was changed between part 1 of the project and part 5, it performs the same function of defining a new topicQueue. In my implementation of parts 1 and 2, I create my topic queues statically using my macro. In parts 3-5, I create my topic queues dynamically in accordance with the specific command file in use.

While doing background research, I came across a list of functions that are not thread-safe. One important one that I probably would have tried to use had I not read the pthreads manual page was strtok(). It turns out that this method of tokenizing strings was not thread-safe. Instead of using strtok_r(), which is thread safe, I decided to use sscanf() because it required less lines of code and allowed me to get rid of quotation marks, newline characters, and assign the line values to my variables in one line of code.

Implementation

In my project, I tested heavily throughout the entire writing process. In part one, before I even got threads involved I heavily tested my dequeue, enqueue, and getEntry() functions. Because they are the basis of the project, I implemented them very carefully. A problem I had was determining when to loop back around to the front of the buffer. As I tested, I came across many cases where my circular buffer queue didn’t quite circulate for some cases. In my end implementation, I have many if-else blocks that correctly determine when the pointers need to circulate back to the front of the buffer.

I implemented my subscriber threads so that they try to read all the topics from each topic queue all at once before moving on to the next command in its command file. Because of this, my html files for each queue have nice specific tables for each topic. If a subscriber tried to read from a queue that had not been written to yet, it tried to read from it MAXTRIES time before giving up

and moving to the next command. While it was “trying”, it called sched_yield() which often allowed for a publisher to come along and push something into the queue.

```
else if (returned == -1){
    // queue is empty
    tries++;
    if (tries >= TRYLIMIT) {
        printf("Sub #%d: Reached attempt limit for \"get\" command on an empty queue\n", ((sArgs *)args)->threadId);
        fprintf(output,"<tr><td>Empty Topic Queue, no caption found</td><td>Empty Topic Queue, no image found</td></
        pthread_mutex_unlock(&(mutex[store_index]));
        break;
    }
    pthread_mutex_unlock(&(mutex[store_index]));
    sched_yield();
}
```

The changes made between my implementation of part one and my implementation of part four made it very difficult for me to put all my tests for each part into one file. Even though it functions as required for every part, I changed a significant amount of my code from the beginning of the project to the end. I was having trouble thinking about how to display all my tests from one file, so I decided to submit 3 C files: part1.c part2.c and quacker.c. As their names suggest, parts 1 and 2 are implemented in their respective files. Parts 3,4, and 5 (quacker.c) are built from parts 1 and 2, but because of the changes are kept separate. After all files are compiled with Make, execute parts 1 and 2 separately to view their test results. To see the main program implementation of the quacker server, execute quacker.c. **EDIT:** Upon reading a Piazza post I have taken away parts 1 and 2 and put in print statements into my quacker.c so as to demonstrate the functionality of my circular ring buffer. Quacker.c is the only file in my new submission.

One big problem that I faced when testing my code on the VirtualMachine was that the compiler on my Mac assigned 0 to empty struct values by default and so I didn’t initialize the entryNumber of my queues to 0. When I tested it at the end on my VM, it was a pretty confusing bug to unpack. I realized when I went to enqueue just the first topicEntry that it was incrementing the value of head when it wasn’t supposed to. From there, I realized that the entryNum I thought that head had at the very beginning was not what was actually there. From there I was able to fix my problem by initializing that value properly in my macro.

Performance Results and Discussion

There were no specific performance results that were required for this project. In terms of compiling, to see the tests from parts one and two as mentioned above please execute them separately.

Conclusion

I learned a lot about threading through writing this project, especially how they differ from processes that we used in project 3. Honestly I found threads easier to work with, and found that after I did some reading online that I really understood what was happening when I had them wait, and then broadcasted to them. After I figured out my whole compiler fiasco I mentioned

above, it is interesting to know that I need to make sure to initialize my struct values just in terms of good practice.

Code

quacker.c

```
/*
Description: The Quaker Server. Parts 3,4,5 of the project (built from parts 1 and 2).
Author: Claire Kolln
Notes: This file should compile with "make", execute to see the results
 */
/*-------------------------Preprocessor Directives----------------------------*/
#define _DEFAULT_SOURCE
#include <stdio.h>
#include <sys/time.h>           // for gettimeofday()
#include <time.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sched.h>

#define MAXENTRIES 100
#define MAXTOPICS 20
#define URLSIZE 50000
#define CAPSIZE 100
#define MAXNAME 25
#define NUMPROXIES 10
#define MAXFILENAME 200
#define TRYLIMIT 5000

/*-----------------------------------------------------------------------------*/
/*-----------------------------Declarations------------------------------------*/
/* Variables */
double DELTA = 0.0;

/* Structs */
```

```
struct topicEntry;
struct topicQueue;
struct timeval;

/* Functions */

void *publisher(void *args);
void *subscriber(void *args);
void *cleaner(void *args);
ssize_t getline(char **lineptr,size_t *n, FILE *stream);
int nanosleep(const struct timespec *req, struct timespec *rem);
/*-------------------------------------------------------------------------*/
/*---------------------Struct/Enumeration Definitions----------------------*/

typedef struct {
    int entryNum;               // unique numerical identifier for each topic entry
    struct timeval timeStamp;
    int pubID;
    char photoURL[URLSIZE];
    char photoCaption[CAPSIZE];
} topicEntry;

typedef struct {
    char name[MAXNAME];
    int counter;
    topicEntry * buffer;
    int head;
    int tail;
    int length;
    int ID;
} topicQueue;

typedef struct {
    int threadId;
    int free;
    int topicId;
    char file[MAXFILENAME];
```

```
} pArgs;

typedef struct {
    int threadId;
    int free;
    char file[MAXFILENAME];
} sArgs;

typedef struct {
    int threadId;
    int free;
    int queues;
} cArgs;

#define TQ(index,n,len,id)                  \
    strcpy(store[index].name,n);            \
    store[index].buffer = buffers[index];   \
    store[index].buffer[0].entryNum  = 0;   \
    store[index].length = len;              \
    store[index].head = 0;                  \
    store[index].tail = 0;                  \
    store[index].counter = 0;               \
    store[index].ID = id;                   \


topicQueue store[MAXTOPICS];
topicEntry buffers[MAXTOPICS][MAXENTRIES];

pArgs pubargs[NUMPROXIES];
sArgs subargs[NUMPROXIES];
cArgs cleaner_args[1];

pthread_t pubs[NUMPROXIES];
pthread_t subs[NUMPROXIES];

int pub_free[NUMPROXIES];
```

```
pthread_mutex_t mutex[MAXTOPICS];
pthread_mutex_t proxy_mutex = PTHREAD_MUTEX_INITIALIZER;

// condition var
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
/*-------------------------------------------------------------------------*/
/*----------------------Function Definitions-----------------------------*/

int enqueue(topicEntry *entry, int q) {
    // printf("Inside Enqueue accessing queue %d\n",q);
    // do mutex stuff in here
    if(store[q].buffer[(store[q].head +1)].entryNum == -1) {
        // printf("ENQUEUE: queue full\n");
        // the queue is full, make the thread wait until there is a dequeue
        return 0;
    }
    else {
        // set the topic number
        int c = (store[q].counter) + 1;
        // printf("Topic Number is: %d\n",c);
        store[q].counter++;
        entry->entryNum = c;

        // printf("Entry entryNum is: %d\n",entry->entryNum);
        // set the timestamp
        struct timeval stamp;
        gettimeofday(&stamp,NULL);
        entry->timeStamp = stamp;

        // printf("Timestamp seconds of enqueue: %ld\n",stamp.tv_sec);
        // printf("Timestamp seconds of timeStamp entry: %ld\n",entry->timeStamp.tv_sec);

        // if the entry isn't going to be the first in the buffer
        if (!((store[q].head == store[q].tail) && (store[q].buffer[store[q].head].entryNum == 0))){
            // printf("INSIDE RIGHT THING\n");
            if (store[q].head == (store[q].length)) {
                // if head is currently at the top of the buffer, new head val = 0
```

```
            store[q].head = 0;
        }
        else {
            store[q].head++;
        }
    }
    // printf("ENQUEUE: added new entry with topicNumber %d\n",c);
    store[q].buffer[store[q].head] = *entry;
    // printf("Timestamp seconds of timeStamp after added to the buffer:
%ld\n",store[q].buffer[store[q].head].timeStamp.tv_sec);
    return 1;
  }
}

int getEntry(int lastEntry, topicEntry *empty, int q) {

  if ((store[q].head == store[q].tail) && (store[q].buffer[store[q].head].entryNum == 0)) {
    // the queue is empty, nothing to get
    return -1;
  }

  int y = store[q].tail;
  while (store[q].buffer[y].entryNum != -1) {
    if (store[q].buffer[y].entryNum == (lastEntry + 1)) {
      empty->entryNum = store[q].buffer[y].entryNum;
      empty->timeStamp = store[q].buffer[y].timeStamp;
      empty->pubID = store[q].buffer[y].pubID;
      strcpy(empty->photoURL,store[q].buffer[y].photoURL);
      strcpy(empty->photoCaption, store[q].buffer[y].photoCaption);
      return 1;
    }
    else if (store[q].buffer[y].entryNum > (lastEntry + 1)) {
      empty->entryNum = store[q].buffer[y].entryNum;
      empty->timeStamp = store[q].buffer[y].timeStamp;
      empty->pubID = store[q].buffer[y].pubID;
      strcpy(empty->photoURL,store[q].buffer[y].photoURL);
      strcpy(empty->photoCaption,store[q].buffer[y].photoCaption);
```

```
            return store[q].buffer[y].entryNum;
        }

        if (y == store[q].length) {
            y = 0;
        }
        else {
            y++;
        }
    }
    // if we have gotten  here, we looked through the entire queue, didn't find the topic, and didn't find an entryNum >
than the topic
    return 0;
}

int dequeue(int q) {
    // printf("Inside Dequeue accessing queue %d\n",q);
    if ((store[q].head == store[q].tail) && (store[q].buffer[store[q].head].entryNum == 0)) {
        // the queue is empty, nothing to dequeue
        return -1;
    }

    // printf("Timestamp seconds of oldest PRE: %ld\n",store[q].buffer[store[q].tail].timeStamp.tv_sec);

    // get current time value
    struct timeval now;
    gettimeofday(&now,NULL);
    struct timeval result;

    timersub(&now,&(store[q].buffer[store[q].tail].timeStamp),&result);
    // printf("Timestamp seconds of oldest POST: %ld\n",store[q].buffer[store[q].tail].timeStamp.tv_sec);
    // printf("Timestamp seconds of current stamp: %ld\n",now.tv_sec);

    // printf("%ld\n",result.tv_sec);
    int removed = 0;
    topicEntry null_entry = { .entryNum = -1 };
    topicEntry non_null_entry = { .entryNum = 0 };
```

```
// printf("Oldest entry has been in queue for %ld seconds\n",result.tv_sec);
while(result.tv_sec >= DELTA) {

    removed = 1;
    // dequeue what is at tail
    store[q].buffer[store[q].tail] = null_entry;

    // in all three statements, must check to make sure head and tail are both incremented if they are equal
    if (store[q].tail == 0) {
        // the null is at the store[q].length index, tail is incremented by one
        store[q].buffer[store[q].length] = non_null_entry;
        if (store[q].tail == store[q].head){
            store[q].head++;
        }
        store[q].tail++;
    }
    else if (store[q].tail == store[q].length){
        // null is at tail - 1, but tail is incremented to front of buffer (index 0)
         store[q].buffer[store[q].tail - 1] = non_null_entry;
        if (store[q].tail == store[q].head){
            store[q].head = 0;
        }
         store[q].tail = 0;
    }
    else {
        // then the null is at the store[q].tail - 1 index, and tail is incremented by one normally
        store[q].buffer[store[q].tail - 1] = non_null_entry;
        if (store[q].tail == store[q].head){
            store[q].head++;
        }
        store[q].tail++;
    }

    // check if next oldest item in queue also needs to be removed
    if (store[q].buffer[store[q].tail].entryNum > 0) {
        gettimeofday(&now,NULL);
        timersub(&now,&(store[q].buffer[store[q].tail].timeStamp),&result);
```

```
        }
        else {
            break;
        }
    }
    if (removed) {
        return 1;
    }
    return 0;
}

void *publisher(void *args) {
    // int x = ((pArgs *)args)->topicId;
    int id = ((pArgs *)args)->threadId-1;
    while(1) {
        pthread_mutex_lock(&proxy_mutex);
        while(((pArgs *)args)->free) {
            pthread_cond_wait(&cond, &proxy_mutex);
        }
        pthread_mutex_unlock(&proxy_mutex);
        printf("Proxy thread %d- type: Publisher\n",((pArgs *)args)->threadId);
        sleep(5);

        // at this point the publisher "starts" and starts to read the commands from the command file until it stops
        FILE *commandf;
        commandf = fopen(((pArgs *)args)->file, "r");
        if (commandf == NULL)
        {
            printf("Error! No file with name %s found. Aborting...\n", ((pArgs *)args)->file);
            exit(2);
        }

        char *fline = NULL;
        fline = (char *)malloc(32 * sizeof(char));
        if (fline == NULL) {
            printf("Error allocating line buffer\nExiting...\n");
            exit(1);
```

```
}
// get the first line
size_t numchars;
size_t buffersize = 32;
numchars = getline(&fline, &buffersize, commandf);

while(numchars != -1) {
    char cmd_words[1][50] = {0};
    int character_number = 0;
    // get the first word of the line
    for (int i = 0; i < numchars;i++) {
        if(fline[i]!= ' '){
            // if line[i] is a character, add it to the word array
            cmd_words[0][character_number] = fline[i];
            character_number++;
        }
        else {
            // if it isnt a character then a whole word has been allocated and we do something w it
            break;
        }
    }
    if (!strcmp(cmd_words[0],"put")) {
        char dump[4] = {'\0'};
        int id = 0;
        char url[URLSIZE]= {'\0'};
        char caption[CAPSIZE]= {'\0'};
        int r = sscanf(fline,"%s %d \"%[^\"]\" \"%[^\n\"]\"",dump,&id,url,caption);
        // printf("Command: %s, Queue: %d, URL: %s, Caption: %s\n",dump,id,url,caption);

        // make a topicEntry
        topicEntry enter;
        enter.pubID = ((pArgs *)args)->threadId;
        strcpy(enter.photoURL,url);
        strcpy(enter.photoCaption,caption);

        // find index of topic with ID = id
        int store_index = -1;
```

```
            for (int q = 0; q < MAXTOPICS ; q++) {
                if (store[q].ID == id) {
                    store_index = q;
                    break;
                }
            }
            if (store_index == -1) {
                printf("PUB: caption: %s,URL %s,id %d\n",caption,url,id);
                printf("PUB: No topic with ID %d speciifed. Aborting...\n",id);
                exit(2);
            }

            // get mutex
            while(1) {
                pthread_mutex_lock(&(mutex[store_index]));
                // printf("Thread %d acquired mutex\n",((pArgs *)args)->threadId);
                int y = enqueue(&enter,store_index);
                if (!y) {
                    // printf("Waiting for cleaner\n");
                    pthread_mutex_unlock(&(mutex[store_index]));
                    sched_yield();
                }
                else {
                    printf("Proxy thread %d- type: Publisher - Executed command: put\n",((pArgs *)args)->threadId);
                    pthread_mutex_unlock(&(mutex[store_index]));
                    break;
                }
            }
        }
        else if (!strcmp(cmd_words[0],"sleep")) {
            char dump2[6] = {'\0'};
            int time;
            int r = sscanf(fline,"%s %d\n",dump2,&time);
            struct timespec t;
            t.tv_sec = time / 1000;
            t.tv_nsec = (time % 1000) * 1000000;
            int slept = nanosleep(&t,NULL);
```

```
            if (!slept) {
                printf("Proxy thread %d- type: Publisher - Executed command: sleep\n",((pArgs *)args)->threadId);
            }
        }
        else if (!strcmp(cmd_words[0],"stop\n") ||!strcmp(cmd_words[0],"stop")) {
            // Free the thread
            ((pArgs *)args)->free = 1;
            // close the file and free the line
            fclose(commandf);
            free(fline);
            fline = NULL;
            printf("Proxy thread %d- type: Publisher - Executed command: stop\n",((pArgs *)args)->threadId);
            // stop should be the last command in the file, but break just in case
            break;
        }
        numchars = getline(&fline, &buffersize, commandf);
    }
  }
}

void *subscriber(void *args) {
  int returned;
  int x, q;
  int last;
  // where you are going to call getEntry with mutex locks
  // if the thread is not free: it should be running until there are no more entries for it to publish
  while(1) {
    pthread_mutex_lock(&proxy_mutex);
    while(((sArgs *)args)->free) {
      // printf("Subscriber proxy thread is waiting\n");
      pthread_cond_wait(&cond,&proxy_mutex);
    }
    pthread_mutex_unlock(&proxy_mutex);

    printf("Proxy thread: %d - type: Subscriber\n",((sArgs *)args)->threadId);
    sleep(5);
```

```
    //subscriber "starts" and read the commands from the command file until it "stops"
    FILE *commandf;
    commandf = fopen(((sArgs *)args)->file, "r");
    if (commandf == NULL)
    {
        printf("Error! No file with name %s found. Aborting...\n", ((sArgs *)args)->file);
        exit(2);
    }

    char *fline = NULL;
    fline = (char *)malloc(32 * sizeof(char));
    if (fline == NULL) {
        printf("Error allocating line buffer\nExiting...\n");
        exit(1);
    }

    // get the first line
    size_t numchars;
    size_t buffersize = 32;
    numchars = getline(&fline, &buffersize, commandf);

    // get subscriber file name
    char fname[20];
    sprintf(fname,"sub%d.html",((sArgs *)args)->threadId);

    // open subscriber file
    FILE *output;
    output = fopen(fname, "w");
    if (output == NULL)
    {
        printf("Error! No file with name %s found. Aborting...\n", fname);
        exit(2);
    }

    fprintf(output,"<!DOCTYPE html> <html><head><title>Subscriber %d </title><style>table, th, td {border: 1px
solid black;border-collapse: collapse;}th, td {padding: 5px;}th {text-align: left;</style></head><body>",((sArgs *)args)-
>threadId);
```

```
while(numchars != -1) {
    char cmd_words[1][50] = {0};
    int character_number = 0;

    // get the first word of the line to get command
    for (int i = 0; i < numchars;i++) {
        if(fline[i]!= ' '){
            // if line[i] is a character, add it to the word array
            cmd_words[0][character_number] = fline[i];
            character_number++;
        }
        else {
            // if it isnt a character then a whole word has been allocated and we do something w it
            break;
        }
    }

    if (!strcmp(cmd_words[0],"get")) {
        char dump[4] = {'\0'};
        int id = 0;
        int r = sscanf(fline,"%s %d\n",dump,&id);

        // find topic queue index that has topic ID = id
        int store_index = -1;
        for (int q = 0; q < MAXTOPICS ; q++) {
            if (store[q].ID == id) {
                store_index = q;
                break;
            }
        }
        if (store_index == -1) {
            printf("SUB: No topic with ID %d speciifed. Aborting...\n",id);
            exit(2);
        }

        // print beginning of table
```

```
            fprintf(output,"<h1>Subscriber: %d </h1><h2>Topic Name: %s</h2><table style=\"width:100%%\"
align=\"middle\"><tr><th>CAPTION</th><th>PHOTO-URL</th></tr>",((sArgs *)args)-
>threadId,store[store_index].name);

            last = 0;
            int tries = 0;
            while(1) {
                pthread_mutex_lock(&(mutex[store_index]));

                // Empty topicEntry
                topicEntry empty;
                returned = getEntry(last,&empty,store_index);
                if (returned > 1) {
                    last = returned;
                    fprintf(output,"<tr><td>%s</td><td><img src=%s></td></tr>",empty.photoCaption,empty.photoURL);
                    pthread_mutex_unlock(&(mutex[store_index]));
                }
                else if (returned == 0) {
                    // queue has no more entries to read from this topic
                    printf("Proxy thread %d- type: Subscriber - Executed command: get\n",((sArgs *)args)->threadId);
                    fprintf(output,"</table>");
                    // unlocks mutex for q, and breaks
                    pthread_mutex_unlock(&(mutex[store_index]));
                    break;
                }
                else if (returned == -1){
                    // queue is empty
                    tries++;
                    if (tries >= TRYLIMIT) {
                        printf("Sub #%d: Reached attempt limit for \"get\" command on an empty queue\n", ((sArgs *)args)-
>threadId);
                        fprintf(output,"<tr><td>Empty Topic Queue, no caption found</td><td>Empty Topic Queue, no
image found</td></tr></table>");
                        pthread_mutex_unlock(&(mutex[store_index]));
                        break;
                    }
                    pthread_mutex_unlock(&(mutex[store_index]));
```

```
            sched_yield();
        }
        else {
            // found the next topicEntry, increment last
            fprintf(output,"<tr><td>%s</td><td><img src=%s></td></tr>",empty.photoCaption,empty.photoURL);
            last++;
            pthread_mutex_unlock(&(mutex[store_index]));
        }
    }
}
else if (!strcmp(cmd_words[0],"sleep")) {
    // convert time and call nanosleep
    char dump2[6] = {'\0'};
    int time;
    int r = sscanf(fline,"%s %d\n",dump2,&time);

    struct timespec t;
    t.tv_sec = time / 1000;
    t.tv_nsec = (time % 1000) * 1000000;
    int slept = nanosleep(&t,NULL);
    if (!slept) {
        printf("Proxy thread %d- type: Subscriber - Executed command: sleep\n",((sArgs *)args)->threadId);
    }
}
else if (!strcmp(cmd_words[0],"stop\n") ||!strcmp(cmd_words[0],"stop")) {
    ((sArgs *)args)->free = 1;
    // close the file and free the line
    fclose(commandf);
    free(fline);
    fline = NULL;
    fprintf(output,"</body></html>");
    fclose(output);
    printf("Proxy thread %d- type: Subscriber - Executed command: stop\n",((sArgs *)args)->threadId);
    // stop should be the last command in the file, but break just in case
    break;
}
numchars = getline(&fline, &buffersize, commandf);
```

```
        }
    }
}

void *cleaner(void *args) {
    int v;
    int first = 1;
    while(1) {
        pthread_mutex_lock(&proxy_mutex);
        while(((cArgs *)args)->free) {
            pthread_cond_wait(&cond,&proxy_mutex);
        }
        pthread_mutex_unlock(&proxy_mutex);

        if (first) {
            printf("Starting cleaner thread\n");
            sleep(5);
            first = 0;
        }
        for (int q = 0; q < (((cArgs *)args)->queues); q++) {
            pthread_mutex_lock(&(mutex[q]));
            v = dequeue(q);
            if (v == 1) {
                printf("Dequeued from queue %d\n",q);
            }
            // else if (v == -1) {
            //     printf("DEQUEUE: found empty queue\n");
            // }
            // else {
            //     printf("DEQUEUE: found nothing old enough to dequeue\n");
            // }
            pthread_mutex_unlock(&(mutex[q]));
            sched_yield();
        }
    }
}
```

```
/*------------------------------------------------------------------------*/
/*----------------------------Program Main-------------------------------*/

int main(int argc, char *argv[]) {
    int c;

    pthread_attr_t attr;                    // thread attributes
    pthread_attr_init(&attr);               //initializes the thread attributes object pointed to by attr with default attribute
values.

    printf("-------------------------------- Part 3 starts --------------------------------\n");
    //initialize mutex locks for all topicQueues
    for (int k = 0; k < MAXTOPICS; k++) {
        pthread_mutex_init(&mutex[k],NULL);
    }

    //create publisher proxy threads
    for (int i = 0; i < NUMPROXIES; i++) {
        pubargs[i].threadId = i + 1;                                    // assign each thread in the pool an ID
        pubargs[i].free = 1;                                            // each thread in the pool is initially free
        c = pthread_create(&pubs[i],&attr,publisher, (void*) &pubargs[i]);
        if (c) {
            printf("Problem creating publisher proxy\n");
        }
        // printf("PUB ID: %d, Free: %d\n",pubargs[i].threadId,pubargs[i].free);
    }

    //create subscriber proxy threads
    for (int i = 0; i < NUMPROXIES; i++) {
        subargs[i].threadId = i + 1;                                    // assign each thread in the pool an ID
        subargs[i].free = 1;                                            // each thread in the pool is initially free
        c = pthread_create(&subs[i],&attr,subscriber, (void*) &subargs[i]);
        if (c) {
            printf("Problem creating subscriber proxy\n");
        }
        // printf("SUB ID: %d, Free: %d\n",subargs[i].threadId,subargs[i].free);
    }
```

```
//create cleaning thread
pthread_t clean;
cleaner_args[0].threadId = 1;
cleaner_args[0].free = 1;
c = pthread_create(&clean, &attr,cleaner, (void*) &cleaner_args[0] );
if (c) {
    printf("Problem creating cleaner thread\n");
}

// counter for topicQueues
int num_queues = 0;

// created flags, can only query (queue,pub threads,sub threads) if theyve been created
int pub_created = 0;
int sub_created = 0;

// null entry to set to last index in queues
topicEntry last = { .entryNum = -1, };

// get file name from from argv and open
FILE *input;
input = fopen(argv[1], "r");
if (input == NULL)
{
    printf("Error! No file with name %s found. Aborting...\n", argv[2]);
    exit(2);
}

// allocate memory for line
char *line = NULL;
line = (char *)malloc(32 * sizeof(char));
if (line == NULL) {
    printf("Error allocating line buffer\nExiting...\n");
    exit(1);
}
```

```
// get the first line
size_t numchars;
size_t buffersize = 32;
numchars = getline(&line, &buffersize, input);

// get the first word, then call sscanf on the whole string based on first word
while(numchars != -1) {

    char cmd_words[1][10] = {0};
    int character_number = 0;
    // get the first word of the line
    for (int i = 0; i < numchars;i++) {
        if(line[i]!= ' '){
            // if line[i] is a character, add it to the word array
            cmd_words[0][character_number] = line[i];
            character_number++;
        }
        else {
            // if it isnt a character then a whole word has been allocated and we do something w it
            break;
        }
    }

    if (!strcmp(cmd_words[0],"create")) {
        char dump[10] = {'\0'};
        char dump2[10] = {'\0'};
        int id, len;
        char name[20] = {'\0'};

        int create = sscanf(line,"%s %s %d \"%[^\n\"]\" %d", dump, dump2,&id,name,&len);
        if (create == EOF) {
            printf("Incorrect command format for create command\n");
        }
        // create topic queue
        TQ(num_queues,name,len,id);

        // set last value to null value
```

```
        store[num_queues].buffer[len] = last;

        // increment num_queues
        num_queues++;
    }
    else if (!strcmp(cmd_words[0],"add")) {
        char dump[10] = {'\0'};
        char type[10] = {'\0'};
        char f[200] = {'\0'};

        int add = sscanf(line,"%s %s \"%[^\n\"]\"", dump, type, f);
        if (add == EOF) {
            printf("Incorrect command format for add command\n");
        }

        //find a free thread of correct type
        for (int i = 0; i < NUMPROXIES; i++) {
            if (!strcmp(type,"publisher") && pubargs[i].free) {
                // Found a free publisher thread, assign file string to value and negate free value
                // printf("Adding Publisher w/ command file %s\n",f);
                strcpy(pubargs[i].file,f);
                pubargs[i].free = 0;
                pub_created = 1;
                break;
            }
            else if (!strcmp(type,"subscriber") && subargs[i].free) {
                // Found a free publisher thread, assign file string to value and negate free value
                strcpy(subargs[i].file,f);
                subargs[i].free = 0;
                sub_created = 1;
                break;
            }
        }
    }
    else if (!strcmp(cmd_words[0],"query")) {
        char dump[10] = {0};
        char type[15] = {0};
```

```
        int query = sscanf(line,"%s %[^\n]",dump,type);
        if (query == EOF) {
            printf("Incorrect command format for query command\n");
        }
        if (strcmp(type,"publishers") && strcmp(type,"topics") && strcmp(type,"subscribers")) {
            printf("Incorrect field, cannot query\n");
        }
        else {
            if (!strcmp(type,"publishers") && pub_created) {
                // query publishers: print out current (nonfree) publishers & their command file names
                for (int i = 0; i < NUMPROXIES;i++) {
                    if (!pubargs[i].free) {
                        printf("Publisher proxy thread ID: %d, file name: %s\n",pubargs[i].threadId, pubargs[i].file);
                    }
                }
            }
            else if (!strcmp(type,"subscribers")&& sub_created){
                // query subscribers: print out current (nonfree) subscribers & their command file names
                for (int i = 0; i < NUMPROXIES;i++) {
                    if (!subargs[i].free) {
                        printf("Subscriber proxy thread ID: %d, file name: %s\n",subargs[i].threadId,subargs[i].file);
                    }
                }
            }
            else if(!strcmp(type,"topics")&&num_queues>0) {
                for (int i = 0; i < num_queues; i++) {
                    printf("Topic ID: %d, Name: %s, Topic Length: %d\n",store[i].ID,store[i].name,store[i].length);
                }
            }
            else {
                printf("Cannot query %s because none of that type have been added yet\n",type);
            }
        }
    }
    else if(!strcmp(cmd_words[0],"delta")) {
        char dump[10];
        double d;
```

```
        int delta = sscanf(line,"%[delta] %lf",dump,&d);
        DELTA = d;
    }
    else if (!strcmp(cmd_words[0],"start\n") || !strcmp(cmd_words[0],"start")) {
        // tell the Publishers and Subscribers to start
        sleep(5);
        printf("Starting Threads\n");
        cleaner_args[0].free = 0;                    // so the cleaner will start at "start"
        cleaner_args[0].queues = num_queues;
        pthread_mutex_lock(&proxy_mutex);
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&proxy_mutex);
        sleep(2);
        printf("--------------------------------- Part 3 ends ---------------------------------\n");
        printf("--------------------------------- Part 4 starts ---------------------------------\n");
        break;                                       // just in case. Last line should be start anyways
    }
    else {
        printf("Invalid command\n");
    }
    numchars = getline(&line, &buffersize, input);
}

// check to see if all proxies are stopped
while(1) {
    int stopped = 1;
    for (int i = 0; i<NUMPROXIES;i++) {
            // if either pub or sub is not free, stopped will negate
            if (!pubargs[i].free || !subargs[i].free ){
                stopped = 0;
            }
    }
    if (stopped) {
        // all threads have stopped, so stopping cleaner thread
        cleaner_args[0].free = 1;
        break;
    }
```

```
    }

    sleep(1);
    printf("All threads have stopped\n");
    printf("------------------------------- Part 4 ends -------------------------------\n");

    // free memory from line
    fclose(input);
    free(line);
    line = NULL;
    exit(0);
    return 1;

}
/*----------------------------Program End---------------------------------*/
```