CIS 415 Operating Systems

Assignment 3 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Noah Tigner

Report

Introduction

For this project, we implemented the Quacker server technology for InstaQuack, which is similar to Twitter or Instagram. This project uses the publisher/subscriber model. Publishers store their entries (photo URLs and captions) in the Topic Store, where entries are stored by topic. These entries can then be accessed by multiple subscribers, who get entries by requesting from a given topic. The storage capacity of any topic must be finite, which can lead to the bounded buffer problem. To fix this, a thread runs in the background and periodically removes the oldest entries in order to free up space. This ensures that the newest entries to be published are more accessible than the older entries.

Background

The topic queues were implemented using a Circular Ring Buffer, a powerful data structure with methods enqueue, dequeue, and getEntry.

The main problem that arises with these topic queues (buffers of a fixed size) is the Bounded Buffer Problem. This problem is addressed in a number of ways. As stated above, the 'TopicCleanupThread' routinely dequeues the oldest entries in order to free up space. In addition to this, the enqueue method will not enqueue entries when a queue is full, and the dequeue method will not attempt to dequeue anything if the queue is empty. Aging is also implemented. When dequeueing, an entry's age is checked against a given time delta to determine if it should be dequeued.

An important feature in this implementation is the ability for multiple publishers to publish entries, and for multiple subscribers to subscribe at a given time. To allow this, the project requires multithreading. I used pthreads, along with the pthread_create and pthread_join methods.

The use of threads requires synchronization. To ensure this, I associated a mutex lock with each topic queue in the topic store. When an attempt is made to enqueue, dequeue, or getEntry on a topic queue, the method must first successfully lock the mutex. Upon the completion of the operation, the mutex is unlocked. This ensures that only one operation can access a queue at a time, securing the critical section.

Implementation

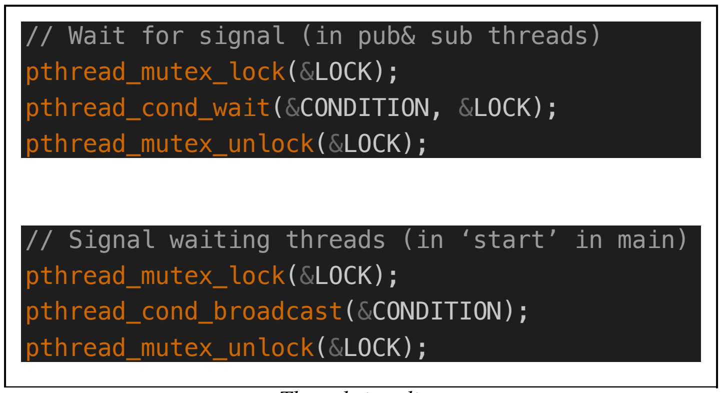
Part 1 of the project involved implementing a Circular Ring Buffer, multiple of which make up the topicStore.

Part 2 made the topicStore thread safe, which, as stated above, was done by securing each queue with its own mutex lock. In addition, 2 thread pools were created for the publishers and subscribers.

Part 3 involved reading commands from standard input. Topics can be created, publishers and subscribers can be created and associated with a command file, the delta can be set, the publisher and subscriber pools can be queried, the topics can be queried, and the execution of the publisher and subscriber threads can begin.

For part 4, the publisher and subscriber threads read from the command file that they were associated with (in part 3). The publisher can put an entry (photo URL and caption) into a topic queue (enqueue) based on topicID, sleep for a given amount of milliseconds, and stop when the commands are completed. The subscriber can get an entry (getEntry) from a given topic queue, sleep and stop. When the stop command is received, the thread frees up its allocated memory, closes the command file, and returns itself to the thread pool. When the start command is received (in main), all proxy threads should begin execution. This is done by signaling each thread *(pthread_cond_broadcast), which are all waiting (pthread_cond_wait) for the signal before being able to execute. Before the other threads are signaled, the cleanup thread is created and begins its execution. This thread is joined when all of the other threads have completed their execution.*

Part 5 brings the other parts together and produces html output. When a subscriber thread begins, it creates its own html file. Then, each entry it gets is written to the table of photos and captions in that file. When the stop command is received, the html closing tags are written to the file, the file is closed, and the thread is returned to the pool. These html files can then be opened in the browser to show off the cool photos the subscriber was able to get!



```
// Wait for signal (in pub& sub threads)
pthread_mutex_lock(&LOCK);
pthread_cond_wait(&CONDITION, &LOCK);
pthread_mutex_unlock(&LOCK);
```

```
// Signal waiting threads (in ‘start’ in main)
pthread_mutex_lock(&LOCK);
pthread_cond_broadcast(&CONDITION);
pthread_mutex_unlock(&LOCK);
```

Thread signaling

Performance Results and Discussion

I believe that this implementation of the project satisfies all of the given criteria. I have tested each part individually, testing for expected behavior as well as edge cases (trying to enqueue to a full queue, trying to dequeue from an empty queue, etc). Due to the nature of the cleanup thread, not all published entries will always be able to be accessed by subscribers, but this is to be expected, since the queues have a fixed capacity. Parts 1-5 should work well for a reasonable amount of topics, proxy threads, and entries. In order to make it easier to grade, I uploaded all 5 parts as separate executables, which can be compiled via the make file with ‘ make‘, and then executed as ‘./part<part>.c’, or ‘./server’ for the finished product. The only thing in my implementation that is not exactly to specifications is my html files for part 5, which display photos in the order that were received by the subscriber, not sorted by topic. This is simply because I was short on time and wanted to focus my efforts on ensuring that thread safety, synchronization, etc. were all working as expected.

Conclusion

I learned a great deal about CRBs, data structure implementation in C, threads (specifically pthreads) and multithreading, mutexes, race conditions and critical sections, synchronization, and pub/sub architecture. Overall I am very satisfied with this project. Being able to open the html files and see all of the photos and captions is a very satisfying end result.

Code

I uploaded .c files for all parts, but in the interest of this report not being 80 pages long, I only attached the code for my part5 (quacker.c), which is the complete version of the project and the most revised and concise.

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <sys/time.h>
#include <pthread.h>
#include <unistd.h>
#include <sched.h>
#include <errno.h>
#include <stdbool.h>

#define URLSIZE 240
#define CAPSIZE 240
#define MAXENTRIES 100
#define MAXTOPICS 20
#define NUMPROXIES 10

int ENTRYNUM = 1;
int TOPICNUM = 0;
double DELTA = 1.0;
char topicNames[MAXTOPICS][32];
bool CLEANING = true;

typedef struct {
    int entryNum;
    struct timeval timeStamp;
    int pubID;
    char photoURL[URLSIZE];
    char photoCaption[CAPSIZE];
} topicEntry;

typedef struct {
    char *name[32];
    int topicID;
    topicEntry *buffer;
    int head;
    int tail;
    int length;
    pthread_mutex_t lock;
} CRB;

CRB topicStore[MAXTOPICS] = {[0 ... MAXTOPICS-1].lock = PTHREAD_MUTEX_INITIALIZER,};
topicEntry buffers[MAXTOPICS][MAXENTRIES+1] = {};
```

```
typedef struct {
    char *topic[20];
    int length;
    int threadID;
    char commandFileName[64];
} arguments;

typedef struct {
    bool free;
    pthread_t thread;
    arguments args;
} threadPool;

threadPool pubThreadPool[NUMPROXIES] = {[0 ... NUMPROXIES-1].free = true, [0 ... NUMPROXIES-
1].args.commandFileName = {0}};
threadPool subThreadPool[NUMPROXIES] = {[0 ... NUMPROXIES-1].free = true, [0 ... NUMPROXIES-
1].args.commandFileName = {0}};

pthread_cond_t CONDITION = PTHREAD_COND_INITIALIZER;
pthread_mutex_t LOCK = PTHREAD_MUTEX_INITIALIZER;


void viewBufferContents() {
    // Print out contents of a CRB
    int i = 0;
    while(i < MAXTOPICS && *topicStore[i].name != NULL) {
        printf("Buffer %d: ", i);
        for(int j = 0; j < topicStore[i].length + 1; j++) {
            printf("%d ", topicStore[i].buffer[j].entryNum);
        }
        printf("    Head: %d Tail: %d", topicStore[i].head, topicStore[i].tail);
        printf("\n");
        i++;
    }
}

int enqueue(int topicID, topicEntry *Entry) {

    for(int i = 0; i < TOPICNUM + 1; i++) {
        if(topicID == topicStore[i].topicID) {

            if(topicStore[i].buffer[topicStore[i].tail].entryNum == -1) {
                return 0;
            }

            struct timeval time;
            gettimeofday(&time, 0);
            Entry->timeStamp = time;

            Entry->entryNum = ENTRYNUM;
            ENTRYNUM++;

            topicStore[i].buffer[topicStore[i].tail] = *Entry;

            int next = (topicStore[i].tail + 1) % (topicStore[i].length + 1);
            topicStore[i].tail = next;

            return 1;
        }
    }
    return 0;
}

int getEntry(int topicID, int lastEntry, topicEntry *emptyEntry) {

    for(int i = 0; i < TOPICNUM + 1; i++) {
        if(topicID == topicStore[i].topicID) {

            if(topicStore[i].head == topicStore[i].tail) {
                return 0;
            }

            if(topicStore[i].buffer[topicStore[i].head].entryNum == lastEntry + 1) {
                *emptyEntry = topicStore[i].buffer[topicStore[i].head];
                return 1;
            }
            if(topicStore[i].buffer[topicStore[i].head].entryNum > lastEntry + 1) {
                *emptyEntry = topicStore[i].buffer[topicStore[i].head];
                return topicStore[i].buffer[topicStore[i].head].entryNum;
            }

            int j = (topicStore[i].head + 1) % (topicStore[i].length + 1);
            while(j != topicStore[i].head) {
                if(topicStore[i].buffer[j].entryNum == lastEntry + 1 && j != topicStore[i].tail) {
                    *emptyEntry = topicStore[i].buffer[j];
                    return 1;
                }
                if(topicStore[i].buffer[j].entryNum > lastEntry + 1 && j != topicStore[i].tail) {
                    *emptyEntry = topicStore[i].buffer[j];
                    return topicStore[i].buffer[j].entryNum;
                }
                j = (j + 1) % (topicStore[i].length + 1);
            }
            return 0;
        }
    }
    return 0;
}

int dequeue(int topicID) {
    // The dequeue only pops 1 entry at a time
    // and return 1 or 0.
    // The cleanup thread will call dequeue on a topic
    // until it gets a 0.

    for(int i = 0; i < TOPICNUM + 1; i++) {
        if(topicID == topicStore[i].topicID) {

            if(topicStore[i].tail == topicStore[i].head) {
                return 0;
            }

            struct timeval time_now;
            gettimeofday(&time_now, 0);

            double elapsed = (time_now.tv_sec -
topicStore[i].buffer[topicStore[i].head].timeStamp.tv_sec) * 1e6;
            elapsed = (elapsed + (time_now.tv_usec -
topicStore[i].buffer[topicStore[i].head].timeStamp.tv_usec)) * 1e-6;

            if(elapsed < DELTA) {
                return 0;
            }

            if(topicStore[i].buffer[topicStore[i].tail].entryNum == -1) {
                int prev = topicStore[i].head - 1;
                if(prev == -1) {
                    // prev = MAXENTRIES;
                    prev = topicStore[i].length;
                }

                topicStore[i].buffer[prev].entryNum = 0;

                // Actual dequeueing operation
                topicStore[i].buffer[topicStore[i].head].entryNum = -1;
                strcpy(topicStore[i].buffer[topicStore[i].head].photoURL, "\0");
                strcpy(topicStore[i].buffer[topicStore[i].head].photoCaption, "\0");

                int next = (topicStore[i].head + 1) % (topicStore[i].length + 1);
                topicStore[i].head = next;
            }
            else {
                int prev = topicStore[i].head - 1;
                if(prev == -1) {
                    prev = topicStore[i].length;
                }

                topicStore[i].buffer[prev].entryNum = 0;

                // Actual dequeueing operation
                topicStore[i].buffer[topicStore[i].head].entryNum = -1;
                strcpy(topicStore[i].buffer[topicStore[i].head].photoURL, "\0");
```

```c
                strcpy(topicStore[i].buffer[topicStore[i].head].photoCaption, "\0");

                int next = (topicStore[i].head + 1) % (topicStore[i].length + 1);
                topicStore[i].head = next;
            }
            return 1;
        }
    }
    return 0;
}

void * subscriberThread(void *arg) {
    arguments *args = arg;
    subThreadPool[args->threadID].free = false;
    printf("Proxy Thread %d - type: Subscriber\n", args->threadID);

    // Wait for signal
    pthread_mutex_lock(&LOCK);
    pthread_cond_wait(&CONDITION, &LOCK);
    pthread_mutex_unlock(&LOCK);

    // Variables
    FILE *infile;

    char fileName[64];

    int c = 0;
    for(int i = 0; i < strlen(args->commandFileName); i++) {
        if(args->commandFileName[i] != '\"') {
            fileName[c++] = args->commandFileName[i];
        }
    }
    fileName[c] = '\0';

    infile = fopen(fileName, "r");
    if(infile == NULL) {
        printf("Input file: %s not found\n", fileName);
        return NULL;
    }

    // Allocate memory for the buffer
    char *buffer;
    size_t bufferSize = 256;
    buffer = (char*)malloc(bufferSize * sizeof(char));
    size_t input;
    char* rest = buffer;

    int last = 0;

    FILE *outfile;
    char outFileName[64] = {0};
    char num[4] = {0};
    sprintf(num, "%d", args->threadID);
    strcpy(outFileName, "subscriber");
    strcat(outFileName, num);
    strcat(outFileName, ".html\0");
    outfile = fopen(outFileName, "w");
    fprintf(outfile, "<!DOCTYPE html><html><head><title>Subscriber %d Filename:
%s</title>Subscriber %d Filename: %s</head><body>", args->threadID, fileName, args->threadID,
fileName);

    while((input = getline(&buffer, &bufferSize, infile)) != -1) {
        int argNum = 1;

        for(int i = 0; i < strlen(buffer); i++) {
            if(buffer[i] ==  ' ') {
                argNum++;
            }
        }

        char *argsList[argNum];
        char *arg = strtok_r(buffer, " ", &rest);

        int i = 0;
        while (arg != NULL && i < argNum)  {
            if(arg[strlen(arg)-1] == '\n') {
                arg[strlen(arg)-1] = '\0';
            }

            argsList[i] = arg;  // strdup or stdcpy??

            arg = strtok_r(NULL, " ", &rest);

            i++;
        }

        if(strcmp(argsList[0], "get") == 0) {
            if(argNum < 2) {
                printf("Error: missing arguments\n");
                continue;
            }

            printf("Proxy thread %d - type: Subscriber - Executed command: %s %s\n", args-
>threadID, argsList[0], argsList[1]);

            for(int i = 0; i < TOPICNUM; i++) {
                if(atoi(argsList[1]) == topicStore[i].topicID) {

                    struct timeval tv;
                    gettimeofday(&tv, 0);

                    topicEntry nullEntry;
                    // Critical Section
                    pthread_mutex_lock(&topicStore[i].lock);
                    int get = getEntry(atoi(argsList[1]), last, &nullEntry);
                    pthread_mutex_unlock(&topicStore[i].lock);

                    printf("    got return value %d\n", get);

                    if(get == 0) {
                        sched_yield();
                    }
                    else {
                        if(get == 1) {
                            last++;
                        }
                        else {
                            last = get;
                        }
                        // printf("        getEntry on topic queue \"%d\" returned %d\n
thread's lastEntry is now %d\n", atoi(argsList[1]), get, last);
                        fprintf(outfile, "\n<br><hr><img src=%s width=\'140\'
height=\'140\'><br>%s<hr>", nullEntry.photoURL, nullEntry.photoCaption);
                    }
                }
            }
        }
        else if(strcmp(argsList[0], "sleep") == 0) {
            if(argNum < 2) {
                printf("Error: missing arguments\n");
                continue;
            }
            int milli = atoi(argsList[1]);
            if(milli > 120000) { milli = 120000; printf("sleeping capped at 2 minutes\n"); };
            printf("Proxy thread %d - type: Subscriber - Executed command: %s %s\n", args-
>threadID, argsList[0], argsList[1]);
            nanosleep((const struct timespec[]){{(milli / 1000), ((milli % 1000) * 1000000)}},
NULL);
        }
        else if(strcmp(argsList[0], "stop") == 0) {
            printf("Proxy thread %d - type: Subscriber - Executed command: %s\n", args->threadID,
argsList[0]);
            break;
        }
        else if(strcmp(argsList[0], "\n") != 0) {
            printf("Error: Unrecognized command!\n");
        }
    }

    fprintf(outfile, "</body></html>");
    free(buffer);
    fclose(infile);
    fclose(outfile);
    subThreadPool[args->threadID].free = true;
    return NULL;
}

void * publisherThread(void *arg) {
    arguments *args = arg;
    pubThreadPool[args->threadID].free = false;
    printf("Proxy Thread %d - type: Publisher\n", args->threadID);

    // Wait for signal
    pthread_mutex_lock(&LOCK);
    pthread_cond_wait(&CONDITION, &LOCK);
    pthread_mutex_unlock(&LOCK);

    // Variables
    FILE *infile;

    char fileName[64] = {0};

    int c = 0;
    for(int i = 0; i < strlen(args->commandFileName); i++) {
        if(args->commandFileName[i] != '\"') {
            fileName[c++] = args->commandFileName[i];
            // c++;
        }
    }
    fileName[c] = '\0';

    infile = fopen(fileName, "r");
    if(infile == NULL) {
        printf("Input file: %s not found\n", fileName);
        return NULL;
    }

    // Allocate memory for the buffer
    char *buffer;
    size_t bufferSize = 256;
    buffer = (char*)malloc(bufferSize * sizeof(char));
    size_t input;
    char* rest = buffer;


    while((input = getline(&buffer, &bufferSize, infile)) != -1) {
        int argNum = 1;

        for(int i = 0; i < strlen(buffer); i++) {
            if(buffer[i] ==  ' ') {
                argNum++;
            }
        }
        char *argsList[argNum];
        char *arg = strtok_r(buffer, " ", &rest);

        int i = 0;
        while (arg != NULL && i < argNum)  {
            if(arg[strlen(arg)-1] == '\n') {
                arg[strlen(arg)-1] = '\0';
            }

            argsList[i] = arg;  // strdup or stdcpy??

            if(strcmp(argsList[0], "put") == 0 && i >= 2) { // split caption
                arg = strtok_r(NULL, "\"", &rest);
            }
            else {
                arg = strtok_r(NULL, " ", &rest);
            }
            i++;
        }

        if(strcmp(argsList[0], "put") == 0) {
            if(argNum < 4) {
                printf("Error: missing arguments\n");
                continue;
            }

            printf("Proxy thread %d - type: Publisher - Executed command: %s %s\n", args->threadID,
argsList[0], argsList[1]);

            for(int i = 0; i < TOPICNUM; i++) {
                if(atoi(argsList[1]) == topicStore[i].topicID) {

                    topicEntry entry = {};
                    strcpy(entry.photoURL, argsList[2]);
                    strcpy(entry.photoCaption, argsList[3]);
                    entry.pubID = args->threadID;

                    // Critical Section
                    pthread_mutex_lock(&topicStore[i].lock);
                    int enq = enqueue(atoi(argsList[1]), &entry);
                    pthread_mutex_unlock(&topicStore[i].lock);

                    printf("    got return value %d\n", enq);

                    if(enq == 0) {
                        sched_yield();
                    }
                }
            }
        }
```

```c
        else if(strcmp(argsList[0], "sleep") == 0) {
            if(argNum < 2) {
                printf("Error: missing arguments\n");
                continue;
            }
            int milli = atoi(argsList[1]);
            if(milli > 120000) { milli = 120000; printf("sleeping capped at 2 minutes\n"); };
            printf("Proxy thread %d - type: Publisher - Executed command: %s %s\n", args->threadID,
argsList[0], argsList[1]);
            nanosleep((const struct timespec[]){{(milli / 1000), ((milli % 1000) * 1000000)}},
NULL);
        }
        else if(strcmp(argsList[0], "stop") == 0) {
            printf("Proxy thread %d - type: Publisher - Executed command: %s\n", args->threadID,
argsList[0]);
            break;
        }
        else if(strcmp(argsList[0], "\n") != 0) {
            printf("Error: Unrecognized command!\n");
        }
    }

    free(buffer);
    fclose(infile);
    pubThreadPool[args->threadID].free = true;
    return NULL;
}

void * cleanupThread(void *arg) {
    printf("Cleanup Thread Assigned\n");

    while(CLEANING){
        int i = 0;
        while(i < MAXTOPICS && *topicStore[i].name != NULL) {

            pthread_mutex_lock(&topicStore[i].lock);
            int deq = dequeue(topicStore[i].topicID);
            pthread_mutex_unlock(&topicStore[i].lock);

            // printf("        dequeue on topic queue %d returned %d\n", topicStore[i].topicID,
deq);
            while(deq != 0) {

                // Critical Section
                pthread_mutex_lock(&topicStore[i].lock);
                deq = dequeue(topicStore[i].topicID);
                pthread_mutex_unlock(&topicStore[i].lock);

                // printf("        dequeue on topic queue %d returned %d\n", topicStore[i].topicID,
deq);
            }
            i++;
        }
        // Wait for a bit
        sleep(10);
    }
    return NULL;
}

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

    // Empty Entry
    topicEntry nullEntry = {
        .entryNum = -1,
    };

    pthread_t cleanup;

    // ================================================================
    // Parse Standard Input
    // ================================================================

    // Variables
    FILE *infile;

    // Allocate memory for the buffer
    char *buffer;
    size_t bufferSize = 64;
    buffer = (char*)malloc(bufferSize * sizeof(char));
    size_t input;
    bool command = false;

    // File Input
    if(argc == 2) {
        infile = fopen(argv[1], "r");
        if(infile == NULL) {
            printf("Input file not found\n");
            free(buffer);
            return 1;
        }
    }
    // Console Input
    else {
        infile = stdin;
        command = true;
        printf(">>> ");
    }

    while((input = getline(&buffer, &bufferSize, infile)) != -1) {
        int argNum = 1;

        for(int i = 0; i < strlen(buffer); i++) {
            if(buffer[i] ==  ' ') {
                argNum++;
            }
        }

        char *args[argNum];
        char *arg = strtok(buffer, " ");

        int i = 0;
        while(arg != NULL && i < argNum) {
            if(arg[strlen(arg)-1] == '\n') {
                arg[strlen(arg)-1] = '\0';
            }
            args[i] = arg;

            arg = strtok(NULL, " ");

            i++;
        }

        // ================================================================
        // Handle Commands
        // ================================================================

        if(strcmp(args[0], "create") == 0) {
            if(argNum < 5) {
                printf("Error: missing arguments\n");
                continue;
            }

            if(TOPICNUM == MAXTOPICS) {
                printf("Error: the number of topics cannot exceed %d\n", MAXTOPICS);
                continue;
            }

            strcpy(topicNames[TOPICNUM], args[3]);

            int l = atoi(args[4]);
            if(l > MAXENTRIES || l < 1) {
                l = (unsigned int)MAXENTRIES;
                printf("Max length exceeded. Setting to Maximum: %u", l);
            }
            topicStore[TOPICNUM].length = l;

            topicStore[TOPICNUM].buffer = buffers[TOPICNUM];
            *topicStore[TOPICNUM].name = topicNames[TOPICNUM];
            topicStore[TOPICNUM].topicID = atoi(args[2]);
            topicStore[TOPICNUM].head = 0;
            topicStore[TOPICNUM].tail = 0;
            topicStore[TOPICNUM].buffer[topicStore[TOPICNUM].length] = nullEntry;
            TOPICNUM++;

        }
        else if(strcmp(args[0], "query") == 0) {
            if(argNum < 2) {
                printf("Error: missing arguments\n");
                continue;
            }
            if(strcmp(args[1], "topics") == 0) {
                printf("Topics:\n");
                int i = 0;
                while(i < TOPICNUM && *topicStore[i].name != NULL) {
                    printf("    topicID: %d, length: %d, name: %s\n", topicStore[i].topicID,
topicStore[i].length, *topicStore[i].name);
                    i++;
                }
            }
            else if(strcmp(args[1], "publishers") == 0) {
                printf("Publishers:\n");
                int i = 0;
                while(i < NUMPROXIES && pubThreadPool[i].args.commandFileName[0] != '\0') {
                    printf("    Publisher: %d, Command File: %s\n", i,
pubThreadPool[i].args.commandFileName);
                    i++;
                }
            }


            else if(strcmp(args[1], "subscribers") == 0) {
                printf("Subscribers:\n");
                int i = 0;
                while(i < NUMPROXIES && subThreadPool[i].args.commandFileName[0] != '\0') {
                    printf("    Subscriber: %d, Command File: %s\n", i,
subThreadPool[i].args.commandFileName);
                    i++;
                }
            }
            else {
                printf("Error: Unrecognized command!\n");
                continue;
            }
        }
        else if(strcmp(args[0], "add") == 0) {
            if(argNum < 3) {
                printf("Error: missing arguments\n");
                continue;
            }
            if(strcmp(args[1], "publisher") == 0) {
                bool assigned = false;
                for(int i = 0; i < NUMPROXIES; i++) {
                    if(pubThreadPool[i].free) {
                        assigned = true;

                        pubThreadPool[i].free = false;
                        strcpy(pubThreadPool[i].args.commandFileName, args[2]);
                        pubThreadPool[i].args.threadID = i;
                        pthread_create(&pubThreadPool[i].thread, NULL, publisherThread, (void
*)&pubThreadPool[i].args);

                        break;
                    }
                }
                if(!assigned) {
                    printf("Could not allocate a proxy publisher thread, waiting for one to free
up\n");
                    break;
                }
            }
            else if(strcmp(args[1], "subscriber") == 0) {
                bool assigned = false;
                for(int i = 0; i < NUMPROXIES; i++) {
                    if(subThreadPool[i].free) {
                        assigned = true;

                        subThreadPool[i].free = false;
                        strcpy(subThreadPool[i].args.commandFileName, args[2]);
                        subThreadPool[i].args.threadID = i;
                        pthread_create(&subThreadPool[i].thread, NULL, subscriberThread, (void
*)&subThreadPool[i].args);

                        break;
                    }
                }
                if(!assigned) {
                    printf("Could not allocate a proxy subscriber thread, waiting for one to free
up\n");
                    break;
                }
            }
            else {
                printf("Error: Unrecognized command!\n");
                continue;
            }
        }
        else if(strcmp(args[0], "delta") == 0) {
            if(argNum < 2) {
                printf("Error: missing arguments\n");
                continue;
            }
            DELTA = atof(args[1]);
        }
        else if(strcmp(args[0], "start") == 0) {
```

```c
            // begin thread execution

            pthread_create(&cleanup, NULL, cleanupThread, NULL);
            sleep(1);

            // pthread_cond_signal(&CONDITION);
            pthread_mutex_lock(&LOCK);
            pthread_cond_broadcast(&CONDITION);
            pthread_mutex_unlock(&LOCK);

            break;
        }
        else if(strcmp(args[0], "\n") != 0) {
            printf("Error: Unrecognized command!\n");
            continue;
        }
        if(command) {
            printf(">>> ");
        }
    }

    // close file, free memory
    free(buffer);
    if(!command) { fclose(infile); }

    // =================================================================
    // Thread Joining and Exit
    // =================================================================

    // Wait for Pubs and Subs to complete
    for(int i = 0; i < NUMPROXIES; i++) {
        if(pubThreadPool[i].args.commandFileName[0] != '\0') {
            pthread_join(pubThreadPool[i].thread, NULL);
            printf("Publisher Thread %d Joined\n", pubThreadPool[i].args.threadID);
        }
        if(subThreadPool[i].args.commandFileName[0] != '\0') {
            pthread_join(subThreadPool[i].thread, NULL);
            printf("Subscriber Thread %d Joined\n", subThreadPool[i].args.threadID);
        }
    }

    // Clean the Queues once more and join the Topic Cleanup Thread
    sleep(10);
    CLEANING = false;
    pthread_join(cleanup, NULL);
    printf("Cleanup Thread Joined\n");

    // printf("\n");
    // viewBufferContents();

    return 0;
```

```
}
```