Christopher Wycoff (951078512)

CIS 415 Operationg Systems

Project 3 Report

Submitted to :
Prof. Allen Malony, Jared Hall, Grayson Guan

Author:
Christopher Wycoff

Christopher Wycoff (951078512)

First and foremost, TO RUN:

./server < <file_name>

Example:

./server < main.txt

(if testing parts 1/2 simply)

./server

In my submission I am including some main.txt files and subscriber.txt/publisher.txt files that work well to demonstrate functionality.

I have included parts 1,2,3,4 and 5 in separate directories (just in case). Part 5 though contains the majority of the logic of 1,2,3,4. In my opinion part 5 could be used for the majority if not entirety of assessment. Parts 1-2 have tests hardcoded.

In directories part1_2, part3_4 and part5 : typing “make” in the command line will build “server”

Introduction

Network servers often use multiple threads to handle incoming requests. This project seeks to simulate the multithreaded processing of get and put requests. The “InstaQuack” server receives requests from either publishers and subscribers, assigns threads to the requests and then executes the specifics of the requests. The subscribers activity can be visualized by the associated “<subscriber_name>.html” files that are created when the “server” is run.

Implementation

I chose to implement a more realistic version of a thread pool. My threads are started up then draw commands from a queue. After executing the command file the threads can theoretically be recycled. The main function in quaker.c instead of assigning commands directly to threads to be created, simply places them in a linked list queue to be dequeued by threads waiting for commands. Dr. Malony suggested that this would be the way he would do it.

The html pages written by the c file from part 5 are refreshed automatically after opening them, so no need to hit refresh.

The current state of the main macros are:

```
#define URLSIZE 100
#define CAPSIZE 100
#define MAXENTRIES 15
#define MAXTOPICS 10
#define NUMPROXIES 10
#define TEST_DELTA 4
#define UNUSED(x) (void)(x)

#define NUMCOMMANDS 16
```

Here are the 2 required data structures and 2 that I used to implement my “more realistic” thread pool..

```
struct node{
    struct node* next;
    char command_file[100];
};


struct pub_sub_queue{
    struct node* head;
    struct node* tail;
    int count;
};

struct topicEntry{
  int entryNum;
  struct timeval timestamp;
  int pubId;
  char photoURL[URLSIZE];
  char photoCaption[CAPSIZE];
};

struct topic_queue{
    int topic_id;
    char name_of_topic[100];
    int exists;
    int entry_number;
    int max;
    int count, head, tail;
    struct topicEntry *entries;
};
```

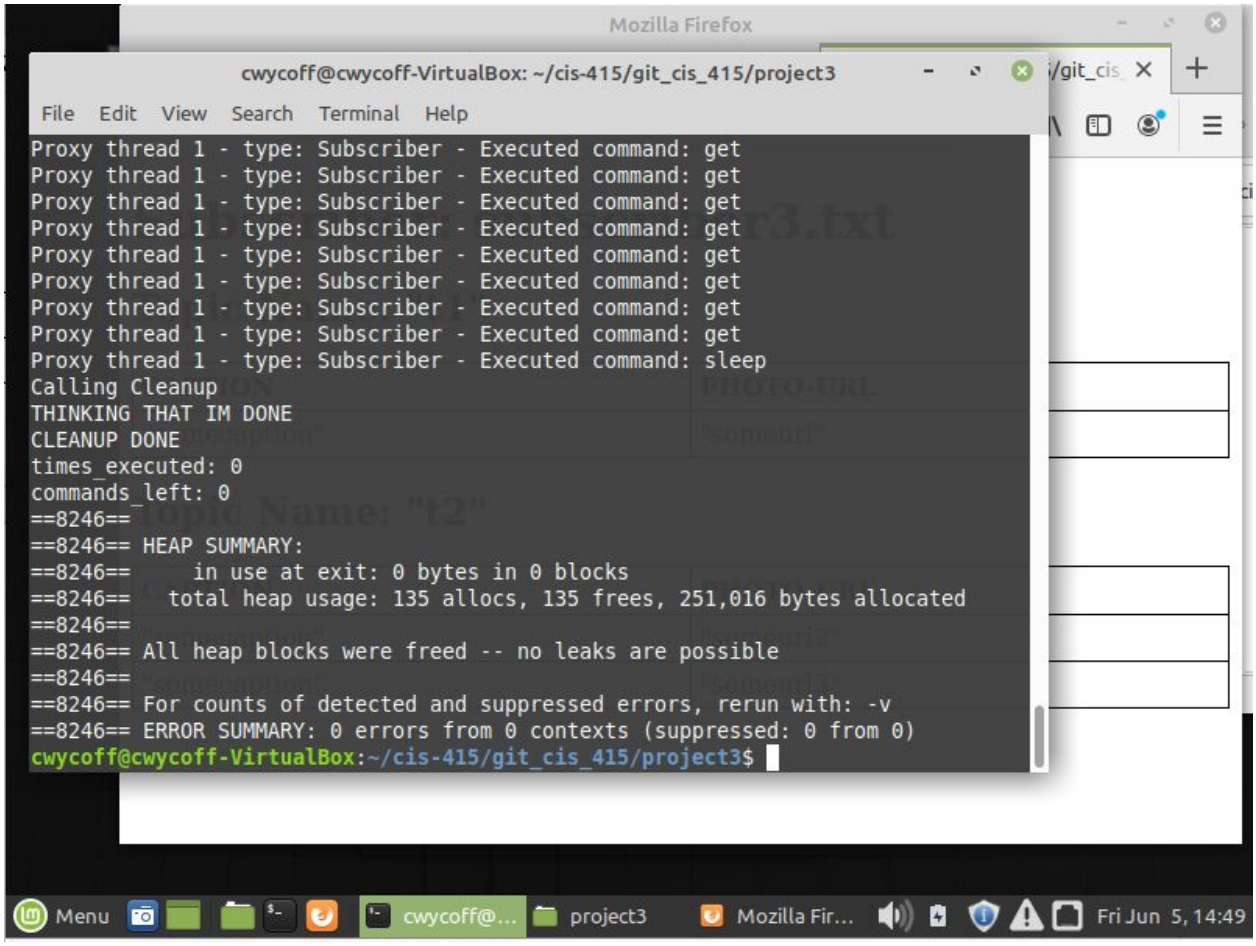
Performance Results and Discussion

Seems to work well. After completing part 5 it was fun to see the html update in real time.
This is the main logic where I add the html code to the appropriate <subscriber>.html file

```
/// end header ///
                int first = 1;
                for(int  i = 0; i<=got_counter; i++){
                    first = 1;
                    for(int  j = 0; j<=got_counter; j++){
                        if (got_topics_ids[j] == i){
                            if (first){
                                //printf("HELLO!!!! : %d %d \n", i, got_topics_ids[j]);
                                first = 0;
                                strcpy(dest, first_part_of_table1);
                                strcpy(src, first_part_of_table2);
                                strcat(dest,got_topics[j]);
                                strcat(dest,src);
                                fputs(dest, htmlfp);
                                fputs("\n", htmlfp);
                            }

                            strcpy(dest, entry_part_of_table1);
                            strcat(dest,got_captions[j]);
                            //char entry_part_of_table1[200] = "<tr><th>"       //CAPTION
                            strcat(dest,entry_part_of_table2);
                            //char entry_part_of_table2[200] = "</th><th>"  //PHOTO-URL
                            strcat(dest,got_urls[j]);
                            strcat(dest,entry_part_of_table3);
                            fputs(dest, htmlfp);
                            fputs("\n", htmlfp);
                            //char entry_part_of_table2[200] = "</th></tr>";
```

I decided to put it in all at once every time, I kept the “got” information in 4 arrays.



The programs outputs some exit prints, including how many times it executed its escape routine.

There is a failsafe 100 seconds shut off.

```
// logic to finish the program
for(int i = 0; i < 100; i++){
    if (number_of_file_issued_commands <= 0){
        printf("THINKING THAT IM DONE\n");
    DONE = 1;
    }
    sleep(1);
    sched_yield();
    if (DONE){
        break;
    }
}
```

Conclusions

What I learned:

- Threads are used in real servers
- Thread signaling and different ways to handle a “thread pool”
- How to avoid deadlock in multithreaded programming
- Pthreads library syntax and function
- How to write to files in a multithreaded program
- More about the differences between operating systems like MACOS and LINUX