Alyssa Kelley

DUCK ID: alyssak
STUDENT ID: 951480571
Project 2 - Parts 1-5 - CIS 415 - Spring 2019

I did this project entirely myself, and all the code is my own.

I did discuss this project with Anne Glickenhaus, Miguel Hernandez, and Zack Bower.

Each part of this project is in its own subdirectory with all of the test files needed. There is a Makefile for each of the parts, so just typing "make" will compile it, then the execution lines are in the README.md in each of the part's directories, and here they all are in one location. It goes over what I think the part of the project wanted, the functions I used, and how I tested it, as well as how to run it. There are also sample outputs of each part in it's own directory.

Disclaimer: Each part of the project takes anywhere between 10 seconds - about a minute to run. It is not stuck in a loop, it just has a few sleeps in there.

PART 1 -

For this part of the project, my understanding was to implement Enqueue, Dequeue and Getentry.

```
make
./part1.exe
```

Here are my descriptions for these functions:

Enqueue(new topic entry, topic number) -
This function takes the new topic entry and places it in the Topic[topic number] queue which is holding all of the topic entries for each individual topic. When there is a new entry, enqueue will take that entry and put it in the back of the queue. This is done by keeping track of the entrynum to show what entry it was, and then it increments the back because this entry was added to the back so now the back available space would be the one behind this newly added entry. Then number of elements and all_entries_num are also incremented in enqueue since another entry was added. num_elements keeps track of how many entries are currently avaialble in the topic queue, and the all_entries_num keeps track of all the entries that have ever been added to this queue, even if they have been removed (dequeued). If an entry is trying to be added, but the queue is full, then it will not be able to be enqueued and nothing is incremented.

Dequeue(topic number) -
This function removes the oldest entry which will be the entry that is at the front of the queue. It first checks if the queue is empty, and if it is, there is nothing for it to remove, so it returns. If it is not empty, then it will increment the front since when this oldest one is "removed" then the second oldest one now becomes the oldest. This function also decrements the number of elements that are currently in the queue. Note: all_entries_num is not decremented because it keeps track of the number of entries that have ever been in the queue, not what is currently in there.

Getentry(last read entry, topic number, topicentry placeholder) -
This function has been harder to determine exactly what it wants from the spec's explanation versus Professor/TA's. However, my understand especially with the help from Matt is that this function is supposed to take in the last entry that was read. The first thing this function does is see if the queue is empty. If it is empty, there is no entry that you can read so it returns -1 to indicate it is empty (Note: the project spec said to return 0, but my first entry index starts at 0 so I return -1). Then the function starts at the front entry number and loops through until all_entries_num. This allows it to go to the very end of the current queue. If the entry number for that last entry is = to the last read entry, then it is a valid entry to read, and returns 1 to indicate it's success. However, if all the entry numbers are larger than this last read entry number, then this means the entry you are trying to read has been dequeued so you cannot view that entry. Instead, it returns the next entry number you can view. There is also a variable in the Topic struct that keeps track of the last read entry, and this is updated if the last entry is found or when a different entry is found for you to read. However, this function also checks to see if the last entry is even a valid entry number. If it is not, then -100 is returned, and the last read entry variable in the struct is not updated, (also not updated if the queue is empty). It the entry is not valid, this means that entry has not been enqueued yet.

I have other helper functions in part 1 such as:

is_empty -
This function is checking is all of the topic queues are empty for all of the current topics. If they are all empty, it returns 1, if they are not all empty, return 0. cleanup_thread() calls this function, and this function calls is_empty()

is_full -
Checks to see if the queue for a specific topic is full or not

count_file_lines -
This function should go through the file, and create a line counter (which is the number of commands for the program) and returns this int.

init_topics -
Initializes everything to be 0, and the lock to be NULL for the Topic queue

enqueue_the_topic -
Helper function for part 1 to help act as a mock publisher() function to fill up entries in the topic queue. Calls enqueue()

print_the_entry -
This function will print most of the information for a Topic entry number. It will tell you which topic number the entry is for, and the entry number, URL, caption and timestamp from when it was enqueued. This function also updated the last read entry number that is saved in the Topic struct. Getentry uses this last read entry number.

print_the_entire_topic -
This function loops over all of the entries in a specific topic, and prints all of there information by calling print_the_entry() function.

fill_the_entire_topic_queue -
This function is a helper function for part 1. It will call the enqueue_the_topic() function to completely fill up a specific Topic. This is helpful to test that enqueue() is working properly.

In this function, I pass in fake URL and captions that have the iteration number attached to the end so they are enqueue which further helps with making sure the strings are being saved properly in the Topic/Topicentry structs.

fill_all_topics -
This function loops through all of the maxtopics that we can have (global variable) and will call fill_the_entire_topic_queue() for each topic, and then it will print that entire topic by calling print_the_entire_topic(). This is a helper function for part 1, and useful to make sure that all of the topics can be implemented correctly.

dequeue_entire_topic_queue -
This function is a helper function for part 1. It acts as a mock cleanup_thread() to completely removed all the entires from the queue by calling Dequeue() until the number of elements is = 0. This is helpful to ensure that Dequeue is working properly.

empty_all_topics -
This function will loop over all of the topics that are possibly created, and call dequeue_entire_topic_queue() to ensure that everything is completely empty.

sub -
This function calls the getentry and keeps track of what it returns. If it returns 1, then that means it is a valid entry and the last read entry is still in the topic queue and you can read/view it. If 1 is returned, then the last read entry is incremented so it can try to read the next one next. If found is -1, the topic queue is empty, and if it is -100 then that means the last read entry is not valid because that number is not enqueued yet. In those two cases, you cannot view an entry. If it is anything other than -1, -100, or 1, this means that you cannot read the last read entry, but you can read the next available entry in the topic queue and that is the number that is returned.

check_if_all_emtpy -
This function is checking is all of the topic queues are empty for all of the current topics. If they are all empty, it returns 1, if they are not all empty, return 0. cleanup_thread() calls this function, and this function calls is_empty()

This is how I tested PART 1 -

1) I will be having three topic queues by setting MAXTOPICS = 3

2) I make sure that all the queues for each topic are empty

3) I fill all the Queues for each Topic

4) I then check to see if the queues are empty or full to ensure the enqueing was done correctly, and that they are all saying they are full with 5 entries in there since I set each queue to hold a max of 5 entires for testing

5) I then read all of the entries for Topic 1, this means the last entry read will be saved at the index 4 position (since I start the index for entries numbers at 0), and I also have a print statement confirming this last number that was saved.

6) I then dequeue the entire Topic 1, so everything is "removed"

7) I then re-fill the Topic 1 queue

8) I dequeue two entries from the newly filled Topic 1 queue

9) I know try to read the last read entry, but this is entrynum 4, which has been dequeued, so a print message will inform you that the entry you are trying to read has been dequeued and it will return the next available entry to view (entrynum 7)

10) I then call getentry a few more times continuing to try to read the next entry and you can see the last read entry number continues to increase until it is trying to read an entry number that is not valid because it hasn't been enqueued yet.

I have saved my output as part1_testoutput.txt in this folder as well so you can the exact output I am referring to. This testing is also hard coded in main of part 1 so when you run it, you will see the same output.

To compile, you can just type "make" and the Makefile will compile the part1.c into the part1.exe and then you can run this part with:

```
make
./part1.exe
```

This part was tested successfully in the virtual machine.

PART 2 -

For this part of the project, my understanding was to implement publisher, subscriber, and cleanup, and to also initialize threads to execute these functions.

```
make
./part2.exe animal_pub.txt animal_sub.txt
```

Here are my descriptions for these functions:

Publisher() -
This function is supposed to read in the publisher_file which is formatted like:
topic number
url
caption
time to sleep

The publisher then saves this information into a topic entry and enqueues it. This function is very similar to my enqueue_the_topic function from part 1.

Subscriber() -
This function is supposed to read in the subscriber_file which is just a topic number on each line, and calls getentry to read the next available entry for that topic. The way the entries will be read is in order of oldest to newest. I know there are a few different descriptions on if you should get the newest or the oldest from getentry, but I am implementing it as reading the next oldest entry from the last read entry.

Cleanup() -
Cleanup checks the timestamp for the entry, and if (the current time - the timestamp) is greater than the DELTA global value, then it is dequeued because this entry is too old and has "expired". Note: I am not implementing this cleanup function to remove all the entires, it just continue to loop until the EXIT_VAR changes and publisher and subscriber are done, and then it stops. It is possible that there are entires left in the queue when cleanup is done. It is my understanding that this is how it should be implemented.

This is how I tested PART 2 -

1) I create every thread and I pass in the publisher file and the subscriber file that it will be parsing through. These two files are modified input files from those posted on canvas.

2) Each thread function (publisher and subscriber) wait for their RUN_VAR to be updated to allow them to continue on.

3) I then join all the publisher and subscriber threads.

4) I then change EXIT_VAR so that cleanup does not finish before publisher() and subscriber() finish. The cleanup function will continue to loop until this EXIT_VAR is changed.

5) I then join the cleanup thread so it can continue running.

I have saved my output as part2_testoutput.txt in this folder as well so you can the exact output I am referring to. This testing is also hard coded in main of part 2 so when you run it, you will see the same output.

To compile, you can just type "make" and the Makefile will compile the part2.c into the part2.exe and then you can run this part with:

```
make
./part2.exe animal_pub.txt animal_sub.txt
```

This part was tested successfully in the virtual machine.

PART 3 -

For this part of the project, my understanding was to implement a function that is parsing a mainfile which has key words in it indicating for the program to create a topic, add a thread, or query a topic or thread for more information on what is currently saved. This mainfile then starts everything running using the "start" keyword which will call another function I wrote to create the thread_pool.

```
make
./part3.exe animal_mainfile.txt
```

Here are my descriptions for these functions:

read_in_mainfile() -
This function reads in the mainfile that will be argv[1] when running part3. It will parse the file and look for the following keywords: create, add, query, Delta, start.

Here is what happens when it finds each keyword:

Create - This keyword is only used when creating a topic. A topic is created with the topic number, the name of the topic, and the max_size that this topic can be. This information is saved in the static global array I have created which is Topic Topics[MAXTOPICS]. I also am keeping track of how many actual topics there are so my code can be more efficient down the line so when I am looping over my topics, I do not have to use MAXTOPICS which may be more than the number of actual topics there are.

Add - This keyword is only used when adding a thread. The thread will be either type publisher or subscriber. The information read would be the thread_type and then file it needs to parse. This information is saved in thread_pool_elements which is a static global struct of type thread_pool Thread_Pools and inside that struct is the element array. As I did with topics, I am also keeping track of the number of actual threads so I do not have to loop over NUMPROXIES which may be a larger number than the actual number of threads created.

Query - This can be used on topics, publishers, and subscribers. It will print the information for all the topics, or all the publishers, or all the subscribers. If it is a query for topics, then it will print the topic number, the topic max_size, and the topic name. If it is a query for the publishers/subscribers, it will print the thread_type (so only subscriber or only publisher depending on the query), and then the number thread this is, and then the filename this thread's function will be reading from.

Delta - This will update the global DELTA value to reflect how old an entry can be before it is expired (this happens in the cleanup_thread function)

Start - This triggers the function call to start_the_thread_pool() to get everything going.

read_stdin() -

This function is exactly like read_in_main() but instead of looping through a file, it takes in the standard input line by line and determines the keyword in just one line, and this is why this function is called in a while. The user needs to type CONTROL-D in order to stop the standard input stream and still allow the program functionalities to continue. If you click CONTROL-C instead, the program terminates and nothing happens.

start_the_thread_pool() -

This function is called when the keyword "start" is found in the mainfile read in in parts 3, 4, and 5.

Once start is found, this means to start at the threads, and to do so, we are going to loop through ACTUAL_PROXY_AMT (global) which is incremented when "add" keyword is found in the mainfile in parts 3, 4, and 5. If the thread name for that thread position is a "publisher", then it will call pthread_create for the publisher function while passing in the thread number to the publisher function so it can look in the Thread_Pools elements array to file the filename to loop through.

The same logic is used if the thread name is "subscriber", but in this case, the subscriber thread function is passed into pthread_create, and the create_html_page() function is called to start the HTML files for each subscriber and each topic for each subscriber.

After all the publisher and subscriber threads have been created, then the 1 single cleanup thread is created. Up until this point all of the EXIT_P/S/C variables are 0 so the thread functions are being hung up on a waiting while loop so they so not go and finish before intended. Before we join all the publisher and subscriber threads (by calling the join_all_threads() function) we need to change the EXIT_P/S/C variables (all globals) to 1 so those while loops are not waiting to be finished anymore in the thread functions.

After the publisher and subscriber threads are all joined, we change the EXIT_VAR to 1 so that the cleanup_thread can finish and then we join the cleanup_thread(). Once the cleanup_thread() finishes, this function is done.

join_all_threads() -

This function is called after the publisher and subscriber threads are created, and after the cleanup thread is created and right after all the EXIT_P/S/C variables are changed to 1, so all the thread functions can get past their waiting while loops. This function is then called to join all of the publisher and subscriber threads. This is done by looping through the ACTUAL_PROXY_AMT (global) which is incremented when "add" keyword is found in the mainfile in parts 3, 4, and 5.

Then it will pull into the Thread_Pools array and go to the thread_elements struct and full out the thread name (aka either publisher or subscriber) and then the filename it reads from to print out a statement saying which thread is being joined and what file it is reading from, and it calls pthread_join on the tid that is saved in the struct.

Note: The tid is saved when the thread is created in start_the_thread_pool() function.

There is a slight sleep after each thread is joined so you can see the threads starting/joining/continuing better.

I also am using two new structs for a thread pool which consists of an array of thread elements. I then have the thread_pool_elements struct which keeps track of the tid for the thread, the filename it will need to read from, and the thread_type_name. This information is read in from read_in_mainfile and used when query is requested and when looping through the NUMPROXIES global to create all the threads and join them.

This is how I tested PART 3 -

1) Created a mainfile that uses all keywords.

2) The file is parsed and the topics are all created, as well as the threads. I use the keyword query a few times in my mainfile to ensure everything is being correctly added, which it is.

3) I then use the same publisher and subscriber files that I used in part 2 in this part as well.

4) At the end of this part, I print out the front index, back index, and the number of elements that are left in the queue and you can see that each change from the beginning which shows that things are being enqueued and dequeue by the publisher and subscriber threads, and this is how I am determining it is being run correctly.

Note: There are still elements left in the queue at the end of this part, and that is intended because the cleanup thread is finished before those last few elements have "expired".

To compile, you can just type "make" and the Makefile will compile the part3.c into the part3.exe and then you can run this part with:

```
make
./part3.exe animal_mainfile.txt
```

This part was tested successfully in the virtual machine.

PART 4 -

For this part of the project, my understanding is to have the publisher() and subscriber() thread functions loop through their own files (publisher.txt and subscriber.txt). However, I have already been doing this since part 2 so my part 4 is very similar/identical to my part 2.

```
make
./part4.exe animal_mainfile.txt
```

As a re-cap, this is what the **subscriber and publisher** do:

The publisher and subscriber are thread functions. They both pass in their individual thread_id for this specific thread. Note: I do not have seperate structs for publishers and subscriber. They are all kept in the thread_pool struct, and you can tell the difference based on the thread_name which is a string in the struct. This may means that you have 3 publishers and 2 subscribers, but your publisher id's may not be 1, 2, 3, it all depends on the order the thread is added in based on the mainfile.

The first part of each function is a while loop that doesn't stop until the EXIT_P or EXIT_S (globals) depending on which function you are in (depending on if it is the publisher and subscriber) and these globals are changed to 1, and this happens after you create the publisher thread and before you join it, and same this with the subscriber. This is in order to not have the publisher function start right away and go to fast. You want to create the publisher and subscriber and then change their EXIT_P/S variables so they start closer to the same time.

After the global EXIT_P/S variable is changed, the function will reach into the thread_elements struct and full out the filename that it is supposed to read from. In parts 3, 4 and 5, this filename is inside the mainfile, and in part 2 this filename is argv[1], and this function is not implemented in part 1.

Once you open the publisher file, this function will loop through the entries 4 lines, and pull out the topic number (line 1), topic url (line 2), topic caption (line 3), and the time it should sleep (line 4) and will continue this until the publisher file has been completely looped through.

It will save the information for this newly read topic in a temporary topicentry TQ, and then passes this struct, with this information in it, into Enqueue to have it be added to the Topic queue (at the end of the queue). Note: The last line for the publisher is the time it should sleep and this is for seeing when it expires. This is why before we enqueue, we get the time of day to keep the timestamp for this entry, and then sleep for however long line 4 for the topic says, and this way this topic will be at least that many seconds old when cleanup_thread checks to see if it is expired.

Once you open the subscriber file, this function will loop through each line, and each line is indicating which topic it wants you to pull the last read entry out of. The subscriber is the viewer and in charge of making the HTML files in part 5. The subscriber function will call sub() which calls getentry() to see if the subscriber can see this last read entry and post its image.

Once the publisher is done looping through the publisher file, it will increment P_EXITED (global) to keep track of all the publisher threads that have finished and this is good to compare to NUM_P which is the number of total publisher threads. This P_EXITED is not actively used, but it was to test the code while writing it, the same this happens when the subscriber is done but instead of P in all those variables, it is S (S_EXITED, NUM_S).

This is how I tested PART 4 -

1) Created a mainfile that uses all keywords.

2) The file is parsed and the topics are all created, as well as the threads. I use the keyword query a few times in my mainfile to ensure everything is being correctly added, which it is.

3) I then use the same publisher and subscriber files that I used in part 2 in this part as well.

4) At the end of this part, I print out all of the active entries for that topic and I also print how many entries have ever been part of that topic. This allows me to see that the publisher/subscriber/cleanup threads are all working. I also added more print statements to my getentry/sub that the subscriber calls to ensure that it is looping through correctly. For this part, I use fake URL's and fake captions that are easy to see what is going on. For example, topic 0 = dog, and the first entry for dog has the URL DOG0_URL and the caption is DOG0_CAPTION to see where everything is and how it was all read in.

To compile, you can just type "make" and the Makefile will compile the part4.c into the part4.exe and then you can run this part with:

```
make
./part4.exe animal_mainfile.txt
```

This part was tested successfully in the virtual machine.

PART 5 -

For this part of the project, my understanding is to have each subscriber thread create its own HTML page for every single topic. The images and captions that get added to this HTML page are coming from the subscriber thread function calling getentry and seeing if it is a valid entry, and if it is, post the entry to the HTML. I create an array of filenames in the thread_elements struct so that each subscriber thread has its own array of HTML files for each topic.

There are 3 functions for writing to the HTML file:
1) create each file
2) fills in the topic image url and the image caption
3) Closes off the HTML file.

```
make
./part5.exe animal_mainfile.txt
```

Here are the HTML functions created in part 5:

create_html_page-
This function create a HTML file for each subscriber thread for each topic, and it saves this file in the threads_elements struct (which is an array of filenames). It then will create the initial part of the HTML while and input the subscriber number, and then the topic name.

Note: the subscriber number is the number of thread in the order of which it is "add"ed. This means that the subscriber number will NOT necessarily be in order depending on how everything is added.
In my animal_mainfile.txt I add a publisher, then subscriber 1 and subscriber 2 so they are in order, but if I did add subscriber, then publisher, then subscriber, the subscriber numbers would be 0 and 2.

fill_in_html_body -
This function is called in sub() function when it is able to read a valid last entry, then it will call this function to print that topic in the HTML file for that subscriber.

end_html_body -
This function is called in main, and everything is ended by looping through all of the topics and all of the threads in the thread pool, and if that thread elements name is "subscriber" then it will call this function with that subscriber thread_id, and the topic.

This is how I tested PART 5 -

1) Create a publisher file that uses real images of dogs, cats, and fish, and make two different subscriber files.

2) Create a main file that will create 4 topics. Topic 0 (a topic that is not meant to ever be filled), Topic 1 (dogs),
Topic 2 (cats), and Topic 3 (fish).

3) I then have print statements updated what the publisher and subscriber threads are doing and what information is being added to the HTML files, and when they are being created and ended. This allows me to make sure the creation, edit, and completion of the HTML files are all in the correct order and location during execution.

I name all of the files in this format: Topic_<topic name>_from_<sub filename>.html. I can then open the HTML files using Chrome (or Firefox on the VM) and the images that the subscriber was able to access are printed with the correct caption. I do repeat the image entries in the publisher files, so that is why there are multiple copies of the same image, but most of them have a different caption to indicate their difference.

To compile, you can just type "make" and the Makefile will compile the part4.c into the part5.exe and then you can run this part with:

```
make
./part5.exe animal_mainfile.txt
```

I also added the line specifically to this Makefile to remove all the .html files after you view them. You can do so by:

```
make remove_html
```

All of the Makefiles also have the remove all executables command with:

```
make clean
```

This part was tested successfully in the virtual machine.