Project 3

(InstaQuack: A Better Instagram for Ducks!)

CIS 415 - Operating systems

Fall 2019 - Prof. Allen Malony

Due date: **11:59 pm, Sunday, June 7th, 2020**

Introduction:

Today, social networking is the name of the game if you want to reach out and interact with others online. Puddles, the Oregon Duck and quite the avid socialite, if he does say so himself, was browsing the University of Oregon's **Around*the*O** webpage and got a thought about how to make his interactions with fans hit that next level. Puddles' idea was to create a social networking platform that would appeal to the UO undergraduates that make up his fan base. After consulting his "branding" manager, Puddles decided to call his new social media service **InstaQuack!** The ultimate goal is to give the UO student community a way to communicate what they are doing with realtime photos, sort of like a cross between Twitter and Instagram. After pitching his idea to the university, Puddles settled on a proprietary server technology for InstaQuack! called *Quacker* that can connect photo bombers (like Puddles) with the photosphere (where is his Fan Club hangs out), but with the added twist that new photos will always be more accessible than old photos. Now, all he needs is to recruit skilled OS developers from Prof. Malony's CIS 415 class to build app.

Project Details:

At the heart of many systems that share information between users is the publish/subscribe (Pub/Sub) model. The central idea of the Pub/Sub model is that publishers of information on different topics want to share that information (articles, pictures, etc.) with subscribers to those topics. The Pub/Sub model makes this possible by allowing publishers and subscribers to be created and operate in the following manner: publishers send their data to a *broker* which stores it under a given *topic* (specified by the publisher). From here, any number of subscribers can receive data from the broker for those topics that they are subscribed to. The relationship between publishers and subscribers is many-to-many (i.e. there can be multiple publishers and subscribers and they all may be interested in different topics).

In the case of InstaQuack!, what is being published are only photos with a very brief caption. Puddles, being the photophile that he is, wants the topics to be associated whatever best describes the photo, like birds, mountains, parties, people, and so on. Because he has a limited

attention span, Puddles also wants to see only the most recent photos. Thus InstaQuack! must be responsive, scalable, and rival anything that can be found at other Pac-12 schools. With the extraordinary skillset that you are learning in CIS 415, for this project you will be implementing the heart of InstaQuack! - the Quacker pub/sub server architecture (shown in Fig. 1 below).

There 5 parts to the project, each building on the other. The objective is to get experience with a combination of OS techniques in your solution, mainly threading, synchronization, and file I/O.

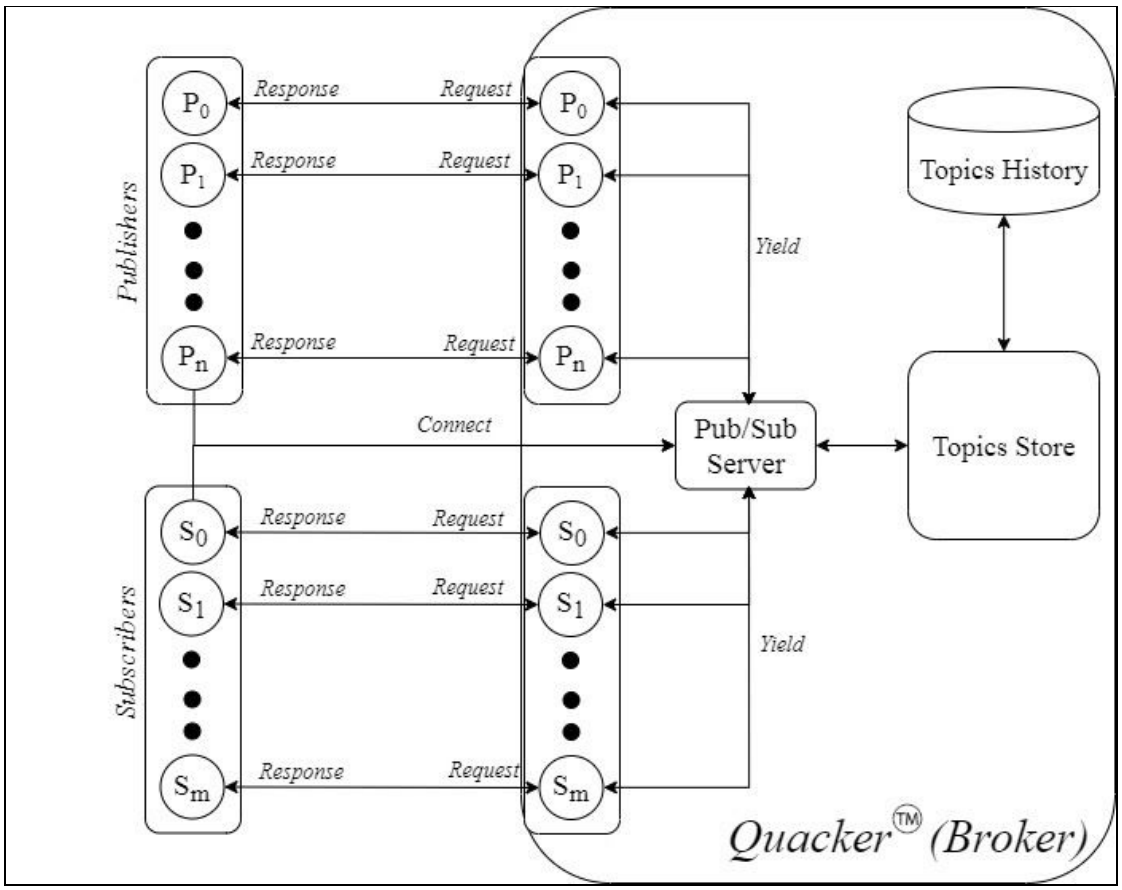

Fig. 1: InstaQuack! Architectural Diagram

Part 1: The Quacker Topic Store

The Quacker Topic Store is at the heart of the InstaQuack! Software stack. It is where recently published photos are stored. Each topic has a bounded queue (buffer) where publishers enqueue and subscribers dequeue. The central objective of Part 1 is to build a bounded queue that can be used by multiple publisher threads and multiple subscriber threads. If this is implemented successfully, then all that is needed to create the Quacker topic store is to replicate the queues.

Program Requirements:

1. Implement a circular ring buffer capable of holding `MAXENTRIES` topic entries, where each topic entry consists of the struct shown in Fig. 2 below. There will be MAXTOPICS total topic queues.

```
struct topicEntry {
    int entryNum;
    struct timeval timeStamp;
    int pubID;
    char photoURL[URLSIZE]; //URL to photo
    char photoCaption[CAPSIZE] //photo caption
}
```

Fig.2: Topic entry C structure

2. Each topic queue needs to have a head and a tail pointer. The head of the topic queue points to the newest (most recent) entry in the topic queue. The tail of the topic queue points to the oldest entry put in the queue.
3. Write an `enqueue()` routine to enqueue a topic entry. Each topic entry enqueued will be assigned a monotonically increasing entry number starting at 1. The topic queue itself will have an entry counter. Because multiple threads are accessing the topic queue, `enqueue()` must synchronize its access to the topic queue. Once a thread has gained access, the `enqueue()` routine will read the counter, increment it, and save it back in the topic entry. A timestamp will be also taken using `gettimeofday()` and saved in the topic entry. See: http://man7.org/linux/man-pages/man2/gettimeofday.2.html
4. Write a `getEntry()` routine to get a topic entry from the topic queue. The routine will take two arguments: A) An integer argument `lastEntry` which is the number of the last entry read by the calling thread on this topic, and B) A reference to an empty `topicEntry` struct. The routine will attempt to get the `lastEntry+1` entry if it is in the topic queue. (**Note:** Because multiple (subscriber) threads can call `getentry()`, it must gain synchronized access to the topic queue.) There are possible 3 cases to consider:
    a. **Case 1**: *topic queue is empty* - `getEntry()` will return 0.
    b. **Case 2**: `lastEntry+1` *entry is in the queue* - `getEntry()` will scan the queue entries, starting with the oldest entry in the queue, until it finds the `lastentry+1` entry, then it copies the entry into the empty `topicEntry` structure passed to our routine and return 1.
    c. **Case 3**: *topic queue is not empty and* `lastentry+1` *entry is not the queue* - For this case, there are 2 possible sub-cases:

i. `getEntry()` will scan the queue entries, starting with the oldest entry in the queue. If all entries in the queue are less than `lastentry+1`, that entry has yet to be put into the queue and `getEntry()` will return 0. (Note, this is like the queue is empty.)

ii. `getEntry()` will scan the queue entries, starting with the oldest entry in the queue. If it encounters an entry greater than `lastEntry+1`, copy that entry into our empty `topicEntry` struct and return the `entryNum` of that entry. (**Note**: This case occurs because the `lastEntry+1` entry was dequeued by the cleanup thread (see below). The calling thread should update its `lastEntry` to the `entryNum`. If you think about this case, the first entry that is greater than `lastentry+1` will be the oldest entry in the queue.)

5. Topic entries can get too old to keep in the topic queues. If a topic entry ages *DELTA* beyond when it was inserted into the queue, it should be dequeued. Write a `dequeue()` routine that dequeues old topic entries. This routine will be executed by a special thread called the *topic cleanup thread*. It should periodically call `dequeue()` on every topic queue. After checking each queue, it should yield. Because there are multiple threads trying to access the topic queue, `dequeue()` must synchronize its access.

Remarks:

In Fig. 3 below, we show a high-level view of the topic entry queue. There is one of these queues for each topic. It has a fixed size and should be implemented as a circular ring buffer. You should be able to retrofit the simple topic queue code that you will be working on in lab 8, or write your own from scratch.

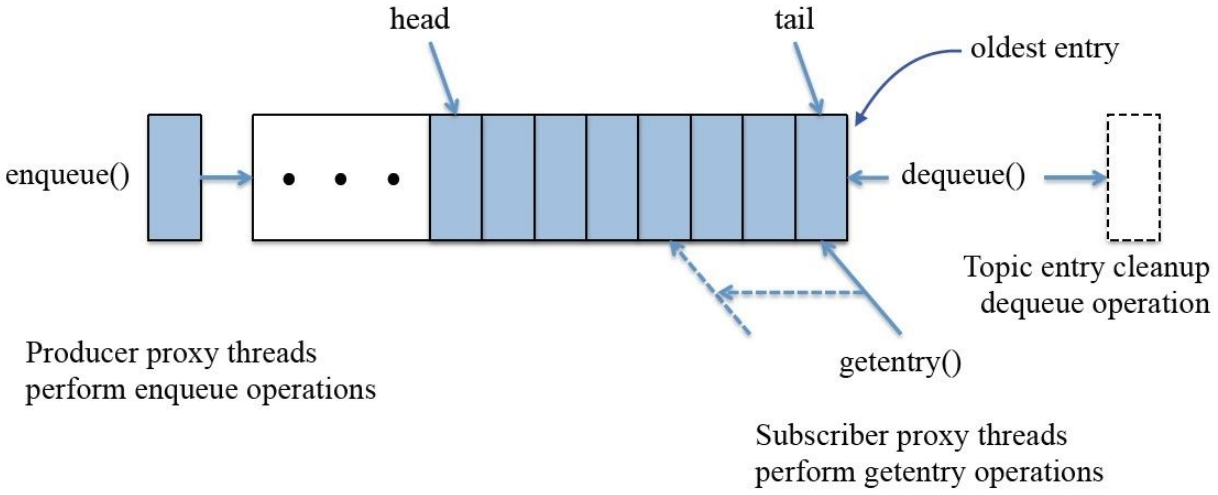

Fig. 3: Topic entry queue and operations

Part 1 is trickier than it seems. Let's start with the publishers. Any topic can have multiple publishers. As such, enqueueing must be synchronized, but it is possible for a topic queue to become full. If a publisher wants to enqueue an entry to a full topic queue, it has to wait until there is space to do so. The simplest way to do this is to just yield the CPU and test again when the thread is re-scheduled. (If you want to get fancier, you could consider using a blocking semaphore to implement this.) Eventually, a dequeue operation will come along and free up queue space.

Now consider the subscribers. All subscribers for a topic must read topic entries in order, using a monotonically increasing message number. However, once a topic entry has reached a certain age, it might be dequeued instead of being read by a subscriber. Who does the dequeuing? The `getentry()` routine will only indicate whether the next entry is available or, if not, whether there is a newer entry available. The problem is that either there are no newer entries for this particular subscriber thread or the queue is empty. In either case, the subscriber thread should try again. The simplest way to do this is to just yield the CPU and try again when the thread is re-scheduled. (If you want to get fancier, you could consider using a blocking semaphore to implement this.) Eventually, an `enqueue()` operation will come along.

Be sure to test your implementation of the topic queue to see that it is working before moving on to the next part of the project.

1. Start with a single queue and spawn the following:
    a. A single thread to push entries to the queue (i.e. a publisher thread).
    b. A single thread to get entries from the queue (ie. a subscriber thread).
    c. A single thread to dequeue old entries from the queue (i.e. a cleanup thread).
2. Try to test out different scenarios, for instance:
    a. Have the publisher fill up the queue before letting the subscriber get entries.
    b. Subscriber attempts to get an entry from the queue when it is empty.
    c. The subscriber attempts to get an entry that was dequeued.
    d. Etc.

Add more publisher/subscriber threads, then redo the tests. When it is working, create more queues and do more testing. Doing comprehensive testing here will save a lot of debugging later. **Note:** you should be setting `MAXENTRIES` to different sizes to test things out.

Your program must use Pthreads to implement the threading. Be careful to make the code you develop thread-safe. You can use *sched_yield()* to have a thread yield the CPU and have itself placed at the end of the ready-to-run scheduler queue (see sched_yield(2): http://man7.org/linux/man-pages/man2/sched_yield.2.html)

Part 2: Constructing the Quacker Server

Now that you have the Quacker topic store working, Part 2 looks to build the rest of our Pub/Sub broker (Quacker), the Pub/Sub Server and proxy pools. First, we will make it multithreaded. The idea is that when an InstaQuack! publisher or subscriber “connects" to the Quacker server, a “proxy" thread (of the appropriate type) is assigned to do their requested actions in the server. Because publishers and subscribers may come and go, instead of creating a new proxy thread each time, the Quacker server is initialized with two thread pools: one containing `NUMPROXIES` publisher proxy threads and another with `NUMPROXIES` subscriber proxy threads. A “free" proxy thread (of the appropriate type) is used by either a publisher or a subscriber and then returned to the pool when they complete. In part 2, you will implement a module for InstaQuack! that creates publisher and subscriber proxy thread pools, connects them to the Quacker topic store, and tests their functionality.

Program Requirements:

1. Create a program that will be the skeleton of the Pub/Sub server. It will create the Quacker topic store and initialize it, it will create the proxy thread pools (see below), and then it will run experiments to test that things are working.
2. Create `NUMPROXIES` publisher proxy threads, initialize them, and put them in a table representing the publisher proxy thread pool. Each proxy thread is assigned a table entry, where each table entry has a flag to indicate whether the thread is free and the thread's id. (**Note:** Each thread in the pool is initially free.)
3. Create `NUMPROXIES` subscriber proxy threads, initialize them, and put them in a table representing the subscriber proxy thread pool. Each proxy thread is assigned a table entry, where each table entry has a flag to indicate whether the thread is free and the thread's id. (**Note:** Each thread in the pool is initially free.)
4. Write tests that allocate publisher proxy threads and provide the threads with topic entries (could be a list of entries) to publish. When there are no more entries for a particular publisher proxy thread, it is returned to the pool.
5. Write tests that allocate subscriber proxy threads and provide the threads with topics to read entries from (could be a list of topics). When there are no more topics to read for a particular subscriber proxy thread, it is returned to the pool.
6. Run the tests concurrently as best you can. You should try to mix things up to get the various proxy threads to overlap in their execution.

Remarks:

Some of the testing harness and scenarios that you used in part 1 might be useful to you here. Your program **must** use Pthreads to implement the threading. Be careful to make the code you develop thread-safe.

Part 3: Creating InstaQuack (Pseudo) Publishers/Subscribers

For part 3 of this project, we want to make it possible for InstaQuack publishers and subscribers to "connect" to our Pub/Sub broker (Quacker). To do this we need to create an interface for this to happen. To not make it overly complicated, the idea is to represent the publisher and subscriber behaviors in a set of files that the Quacker server reads after initialization. Each file is a set of commands that a publisher or subscriber wants to do. To begin, we need to have a set of commands to create the publishers and subscribers. These will come in on standard input and be interpreted by the Quacker broker using the following methods:

- `create topic <topic ID> "<topic name>" <queue length>`
  Create a topic with ID (integer) and length. This allocates a topic queue.
- `query topics`
  Print out all topic IDs and their lengths.
- `add publisher "<publisher command file>"`
  Create a publisher. A free thread is allocated to be the "proxy" for the publisher. When the publisher is started (see below), the thread reads its commands from the file.
- `query publishers`
  Print out current publishers and their command file names.
- `add subscriber "<subscriber command file>"`
  Create a subscriber. A free thread is allocated to be the "proxy" for the subscriber. When the subscriber is started (see below), the thread reads its commands from the file.
- `query subscribers`
  Print out subscribers and their command file names.
- `delta <DELTA>`
  Set DELTA to the value specified.
- `start`
  Start all of the publishers and subscribers. Just before this happens, the cleanup thread is started.

After these commands have been processed by the main program (master thread), the publisher and subscriber proxy threads start to read from their command files in part 4. For part 3, they should just start up and print the following before exiting:
"**Proxy thread <thread id> - type: <Publisher/Subscriber>**"

Note: the query command (for either the topics, publishers, or subscribers) should come **after** the topics, publishers, and subscribers have been created and configured. Also, the configuration file is something that you are writing and should be used in your tests that things are working correctly. We will provide some sample command files as a part of lab 8, but you can also share command files with your classmates via the Piazza post described in the project remarks section.

Part 4: Publisher/Subscriber Command Files

As mentioned in part 3, for this project, we will emulate our publishers and subscribers using text files (i.e. command file) containing their behaviours. For part 4 of the project, you will implement a method for our proxy threads to execute the commands contained in these files. This occurs once the publishers and subscribers are started. They will first read the commands contained in their respective command file and process them. The commands are as follows:

- `put <topic ID> "<photo URL>" "<caption>"`
  The publisher thread will attempt to put a topic entry with this information into the topic ID queue.
- `get <topic ID>`
  The subscriber will attempt to get a topic entry from the topic ID queue.
- `sleep <milliseconds>`
  The publisher or subscriber will sleep for this number of milliseconds.
- `stop`
  The publisher or subscriber thread stops reading commands and the thread is returned to the respective pool.

You will need to implement methods to execute these commands. **Note:** The "stop" command will always be last. We won't be testing the corner cases where the stop command is not last or not present.

Remarks:
Once the publishers and subscribers are up and running, they start reading their command files until they stop. Make sure to check that the topic ID is correct. When all publishers and subscribers have stopped, the cleanup thread is also stopped. **Note:** how to handle the subscriber is a little tricky. For example, you need to decide what to do when a subscriber tries to get an entry for a topic and there is nothing there. You could decide to try a certain number of times before giving up. Do not keep trying indefinitely! Also, when there are multiple entries, you might decide to just get them all. When a publisher/subscriber executes a command it should print the following:
"**Proxy thread <thread id> - type: <Publisher/Subscriber> - Executed command: <cmd>**"

Part 5: InstaQuack Topic Web Pages

Part 4 did not say what the subscriber does with the topic entry data it gets. For part 5 of the project, you will adapt the provided HTML file (See Canvas attached files, provided after lab 7) to be used by our subscribers. This file creates tables for every topic the user is subscribed to with entries consisting of the photo and caption.

In this way, you will be able to open each of these files in a web browser and see what each subscriber was able to get from the topics. The main program can be responsible for setting up the files before starting the publishers and subscribers, or they can do it themselves.

Project Remarks:

One of the more creative aspects of projects of this sort is coming up with the test files that you can use to evaluate your implementation. For any test run of the system, you need command files to create publishers/subscribers that the master thread reads, and then more command files for publishers and subscribers. Of course, there are many different ways to come up with these files for testing edge cases. Another thing regards the actual photos and captions to be used. With this in mind, we are allowing students to share their command files photos on the pinned post (**Share your command files and pictures here**) on Canvas.

Project Structure Requirements:

For a project to be accepted, the project must contain the following files and meet the following requirements: (The naming conventions listed below **must** be followed. Additionally you must use the C programming language with pthread library for this assignment. No projects written in another programming language will be accepted.)

quacker.c: This is the main program. **The quacker server is implemented here.**

Makefile: Your project must include a standard make file. It must produce exe's with the following names: **server**

Report: Write a 1-2 page report on your project using the sample report collection format given. Feel free to go over the limit if you wish. Report format and content suggestions are given in the report collection template.

Note: Additionally, you are allowed to add any other *.h and *.c files you wish. However, when we run your code we will only be running the server file. Make sure your code runs in the VM before submission.

Submission Requirements:

Once your project is done, do the following:

1. Open a terminal and navigate to the project folder. Compile your code in the VM with the -g flag.
2. Run your code and take screenshots of the output as necessary (of each part).
3. Create valgrind logs of each respective part:
    a. "**valgrind --leak-check=full --tool=memcheck ./a.out > log*.txt 2>&1** "
4. Tar the project folder and submit it onto Canvas.
5. Submit a .pdf of your project report separately onto canvas. (i.e. you should upload two things: your report and the tar.gz)

Valgrind can help you spot memory leaks in your code. As a general rule any time you allocate memory you must free it. Points will be deducted in both the labs and the project for memory leaks so it is important that you learn how to use and read Valgrind's output. See (https://valgrind.org/) for more details.

Grading Rubric:

Points	Description
25	Completed Part 1 as described above
20	Completed Part 2 as described above
20	Completed Part 3 as described above.
25	Completed Part 4 as described above.
5	Complete Part 5 as described above.
5	Report collection in proper format and filled out. (1-2 pages)

Late Homework Policy:

- 10% penalty (1 day late)
- 20% penalty (2 days late)
- 30% penalty (3 days late)
- 100% penalty (>3 days late) (i.e. no points will be given to homework received after 3 days)