Project #2
InstaQuack: A Better Instagram for Ducks!

CIS 415 - Operating Systems
Spring 2019 - Prof. Malony

Due date: June 9, midnight

Introduction

Today, social networking is where it is at. Puddles, the Oregon Duck and a social animal himself, was browsing the University of Oregon's **Around*the*O** webpage and got a thought about how to take it to the next level. Puddles's 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 goal is to have a way for the UO student community to communicate what they are doing with realtime photos. A cross between Twitter and Instagram, InstaQuack will have proprietary server technology called *Quacker* that can connect photo bombers (like Puddles) with the photosphere (where is his Fan Club hangs out), but with the twist that new photos will always be more accessible than old photos. Now, all he needs is to recruit OS developers from Prof. Malony's CIS 415 class to make it happen.

At the heart of many systems that share information between users is the *publish/subscribe* model. The idea is that *publishers* of information on different *topics* want to share that information (articles, pictures, ...) with *subscribers* to those topics. The publish/susbscribe (*pub/sub*) model makes this possible by allowing publishers and subscribers to be created and operate in a simple manner: publishers send the pub/sub system information for certain topics and subscribers receive information from the pub/sub system for those topics that they are subscribed to. 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.

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, you will implement the heart of InstaQuack – the Quacker pub/sub server architecture – shown in Figure 1.

There are 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.

Part 1 – Quacker Topic Store

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

- Create a circular, FIFO topic queue capable of holding MAXENTRIES topic entries, where each topic entry consists of:

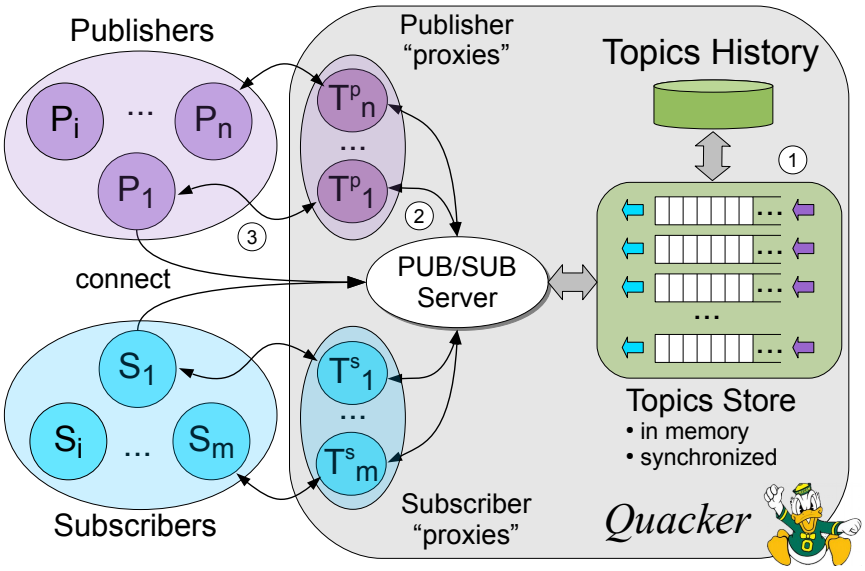

Figure 1: InstaQuack and the Quacker pub/sub server architecture.

```
struct topicentry {
  int entrynum;
  struct timeval timestamp;
  int pubID;
  char photoUrl[QUACKSIZE];         /* URL to photo  */
  char photoCaption[CAPTIONSIZE];   /* photo caption */
}
```

There will be $MAXTOPICS$ total topic queues.

- 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.
- 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 a entry counter. Because multiple threads are access 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.
- Write an `getentry()` routine to get a topic entry from the topic queue. The routine will take an argument `int lastentry` which is the number of the last entry read by the calling thread on this topic. The routine will attempt to get the `lastentry+1` entry, if it is in the topic queue. There are 4 cases to consider:
  - **Case 1: topic queue is empty** – `getentry()` will return 0.
  - **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 `topicentry` structure pointed to by the `struct topicentry *t` argument and return 1.

– **Case 3: topic queue is not empty and `lastentry+1` entry is not the queue** – There are 2 sub-cases:

  * a. `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.)
  * b. `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 the `topicentry` structure pointed to by the `struct topicentry *t` argument 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.)

Because multiple (subscriber) threads can call `getentry()`, it must gain synchronized access to the topic queue.

- 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 only 1 thread call the *topic cleanup* thread. It should periodically call `dequeue()` on every topic queue. After checking each queue, it should yield[1] Because there are multiple threads trying to access to the topic queue, `dequeue()` must synchronize its access.

- Test the Quacker topic queue implementation to show that it is working. Start with just a single publisher thread, a single subscriber thread, and the cleanup thread. Then add more threads. When it is working, replicate the queues and do more testing.

Figure 2 give 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 bounded buffer code that you are working on in the lab, or write your own from scratch.

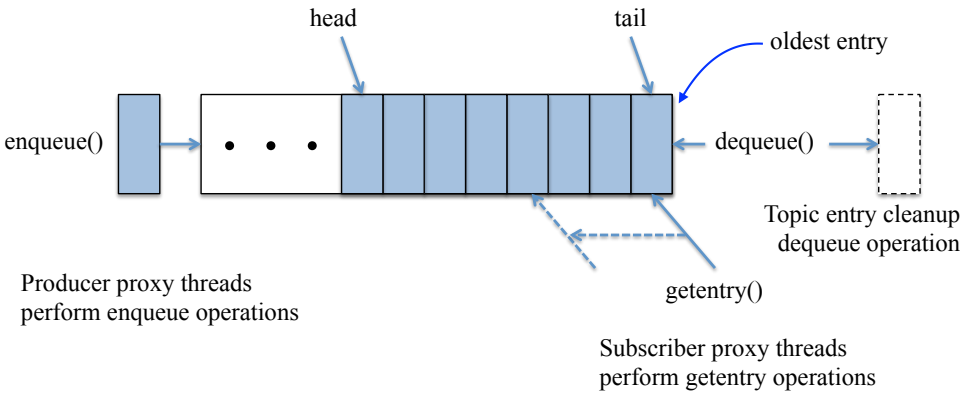

Figure 2: Topic entry queue and operations.

Part 4 is trickier than it seems. Let's start with the publishers. Any topic can have multiple publishers. 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

[1] See `sched_yield(2)`.

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 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.

Part 2 – Constructing the Quacker Server

Now that you have the Quacker topic store working, Part 2 looks to build the the rest of the Quacker server. We first will make it multithread. The idea is that when an InstaQuack producer 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 a pool $NUMPROXIES$ producer proxy threads and a pool of $NUMPROXIES$ subscriber proxy threads. A "free" proxy thread (of the appropriate type) is used by a producer or consumer and then returned to the pool when they complete. Part 3 will implement a program that creates publisher and subscriber proxy threads, connects them to the Quacker topic store, and tests their functionality as follows:

- Create a program that will be the skeleton of the Quacker 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.
- 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. Each table entry has a flag to indicate whether the thread is free and the thread's id. Each thread ($T_i^p,\ 1 \leq i \leq NUMPROXIES$ is initially free.
- Create $NUMPROXIES$ subscriber proxy threads, initialize them, and put them in a table representing the subscribr proxy thread pool. Each proxy thread is assigned a table entry. Each table entry has a flag to indicate whether the thread is free and the thread's id. Each thread ($T_i^s,\ 1 \leq i \leq NUMPROXIES$ is initially free.
- 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.
- 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.
- 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.

You should use Pthreads to implement the threading. Be careful to make the code you develop *thread safe*. Completing this part should give a functional core Quacker server.

Part 3 – Creating InstaQuack (Pseudo) Publishers/Subscribers

In Part 3 we want to make it possible for InstaQuack publishers and subscribers to connect to Quacker. 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 server as follows:

- `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 threads start to read from their command files.

Part 4 – Execute Publisher/Subscriber Commands

Once the publisher and subscribers are started, they read commands from the files and process them 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.

Once the publishers and subscribers are up and running, things proceed until they all are stopped. Make sure to check that the topic ID are 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 instance, 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.

Part 5 – InstaQuack Topic Web Pages

Part 4 did not say what the subscriber does with the topic entry data it gets. One idea is for every subscriber and for all its topics to create a simple HTML file that looks something like the following:

```
<html>
<head>
<title>
</head>
<body>
<!--
Fill in <topic name> in the following.
-->
<table>
  <tr>
  <td align="left">
    <img "SRC=puddles.gif" WIDTH="140" HEIGHT="140">     </a>
  </td>
  <td align="center">
    <h1>InstaQuack</h1>
    <h1>Subscriber <subscriber I> : Topic <topic name></h1>
  </td>
  <td align="right">
         <img "SRC=puddles.gif" WIDTH="140" HEIGHT="140"></a>
  </td>
</tr>
</table>
<!--
For all topic entries seen by THIS subscriber for THIS topic, insert the
following, where <photo URL> and <photo caption> come from the entry:
-->
<hr>
<img "SRC=<photo URL>"></a><br>
<photo caption>
<hr>
<!--
Continue inserting until the subscriber is done.
-->
</body>
</html>
```

In this way, you will be able to open each of these files in a web browser and see that 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. You will be provided a set of routines that can be used to produce the file content from topic entry information.

Developing Your Code

The best way to develop your code is in Linux running inside the virtual machine image provided to you. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state. You may use the room 100 machines to run the Linux VM image within a Virtualbox environment, or run natively or within a VM on your own personal machine. Importantly, do not use `ix` for this assignment.

Individual Work and Helping Classmates

This is an individual assignment. You all should be reading the manuals, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project's end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. It is understood that students have different levels of programming skills. If you can not get help from the TA, it is possible that a classmate can be of assistance.

Test Files

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 experiment, there is the command file to create publishers/-subscribers that the master thread reads, and then the command files for publishers and subscribers. We will arrange for students to contribute their command files for others to use.

Grading and Due Date

The grading will be based on the project as a whole. Credit will be given for parts completed, but the score will be weighted more towards full solutions. You should make sure to be able to demonstrate what you have working. The project is due on June 9, at midnight. That gives 3+ weeks to complete the work. No late projects are allowed.