Jordan Taylor
Oct 2, 2015
Section 2

In order to synchronize my code, I created two classes to help manage the data structures. Each class is thread safe and uses a different method to make it so. The first uses a mutex to lock and unlock a shared data structure while the second uses multiple semaphores to implement a producer consumer model for the incoming clients.

MessageBox.h

-Contains a map<string, vector<pair<string, string>>> as the data structure
-has a mutex surrounding the guts of
store_message(21)
create_list_results(32)
create_read_result(52)
-example for store_message

```
void store_message(string name, string subject, string message){
  message_box_mutex.lock();
  vector<pair<string, string>> vector_of_messages;
  if (message_box.count(name) > 0){
    vector_of_messages = message_box[name];
  }
  vector_of_messages.push_back(make_pair(subject, message));
  message_box[name] = vector_of_messages;
  message_box_mutex.unlock();
}
```

This works by first locking the message_box data structure before modifying anything in it and then immediately unlocking it once i'm done with it.

ClientBuffer.h

-Contains a vector<int> as the data structure
-This uses the producer consumer model to protect the client_buffer
-Most of the file is the semephores
-Source code:

```
class ClientBuffer{
  private:
    vector<int> client_buffer;
    int BUFFER_LIMIT = 100;
    sem_t s,n,e;
```

```
  public:
    ClientBuffer(){
      sem_init(&e, 0, BUFFER_LIMIT);
      sem_init(&s, 0, 1);
      sem_init(&n, 0, 0);
    }
    void append(int client){
      sem_wait(&e);
      sem_wait(&s);
      client_buffer.push_back(client);
      sem_post(&s);
      sem_post(&n);
    }
    int take(){
      sem_wait(&n);
      sem_wait(&s);
      int client = client_buffer[0];
      client_buffer.erase(client_buffer.begin());
      sem_post(&s);
      sem_post(&e);
      return client;
    }
};
```