Machine Problem Four:

# Dealing with a Reluctant Data Server

Daniel Frazee and Edgardo Angel

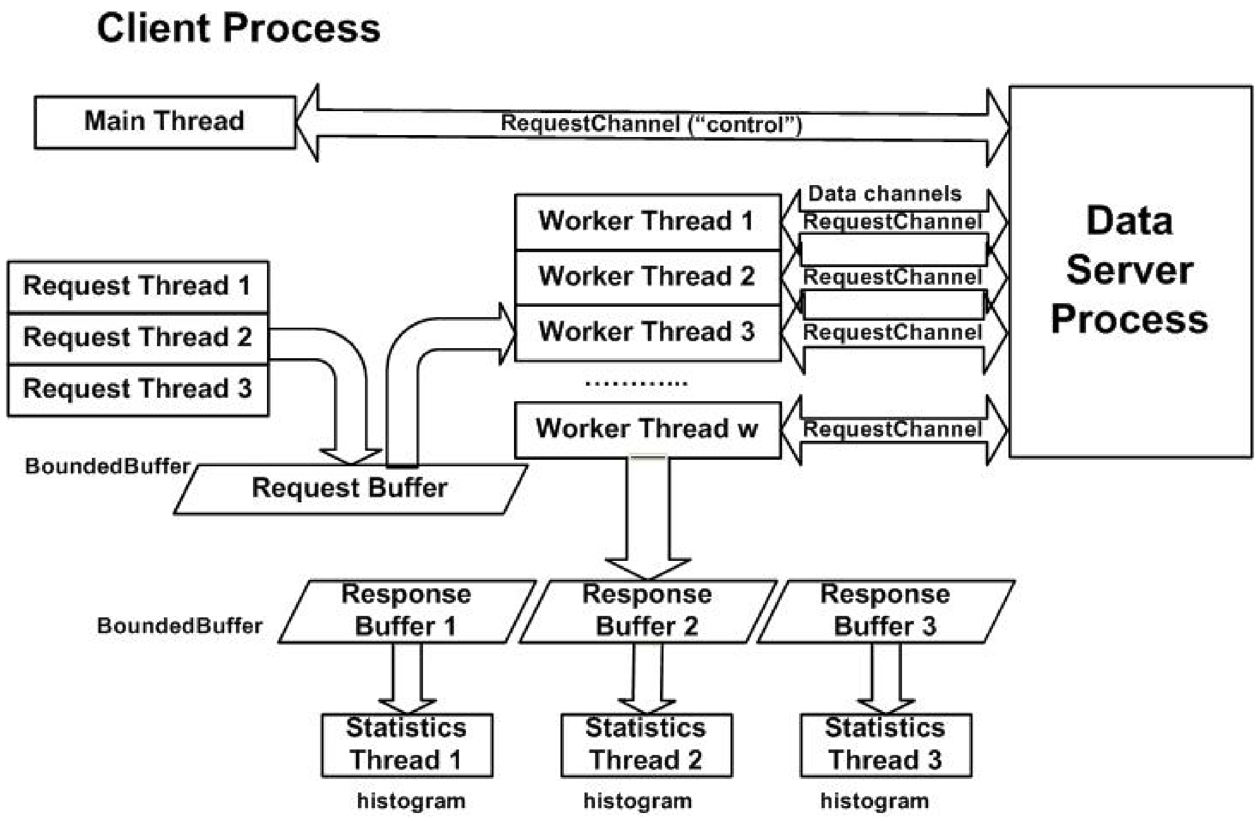
Due: April. 18, 2015

**Introduction**

In Machine Problem Four we created two classes and completed a third: boundedbuffer, item and semaphore. The objective for this machine problem was to implement something similar to machine problem two, but with a little bit more complexity. We had to work with concurrency to talk to a server, from a three clients, and keep statistics for each client based on what information the server returned. This was handled by a buffer that allowed for us to keep track of the threads, and a semaphore that acted as our lock to prevent our threads from interfering with each other. These were the main processes that composed the entirety of Machine Problem four.

**Procedures**

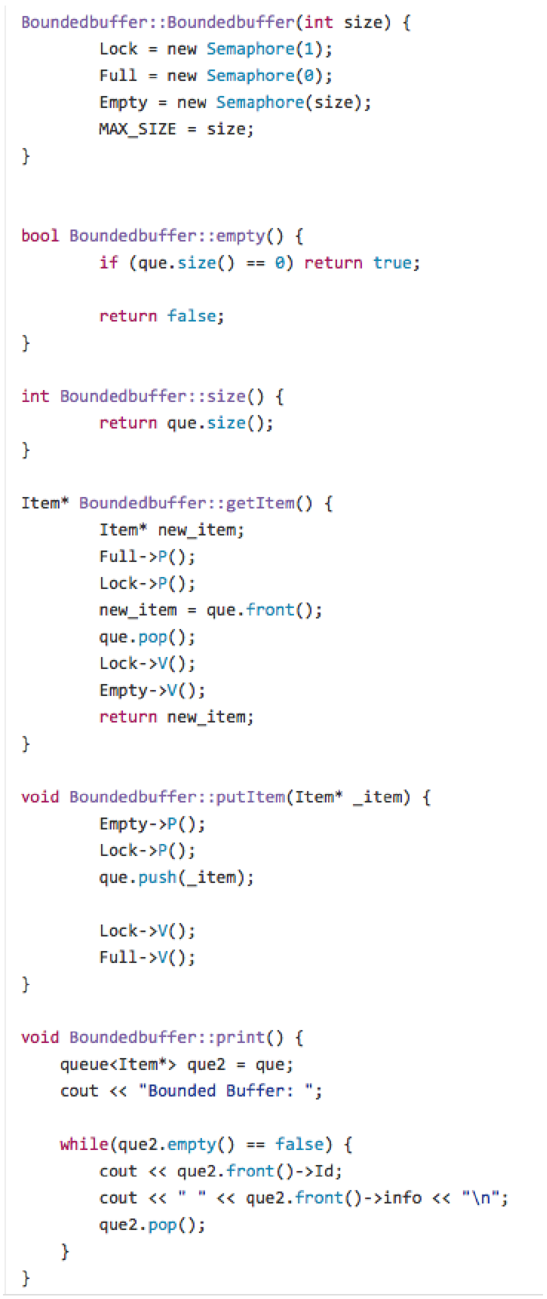
As a high level overview, this picture gives us an idea of exactly what this program is accomplishing. The three request threads are John, Jane, and Joe. These clients are requesting data from the server on the right, and that data is being stored in the statistics threads. Buffers are used to handle these threads. Our procedures for this program are explained below.



The main files in this project we had to work on were `client.C`, `boundedbuffer.C`, `boundedbuffer.H`, `semaphore.C`, and `item.H`. These files made up the majority of our functionality. `Client.c` ran our main.

`Boundedbuffer.H` and `Boundedbuffer.C`

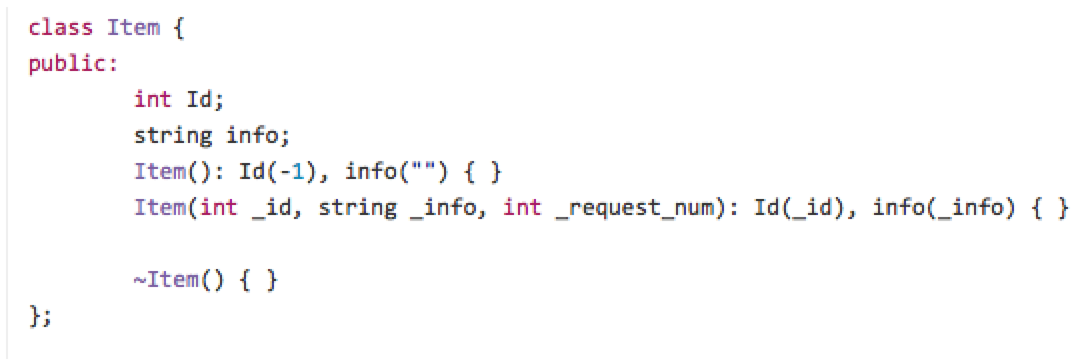
The bounded buffer was our way of regulating how items were taken up by the threads. We used two main functions: `getItem()` and `putItem()`. The buffers were essentially queues. Our buffers would be filled with items, and the threads would take an Item from the buffer and place Items into the buffer.



```
Boundedbuffer::Boundedbuffer(int size) {
        Lock = new Semaphore(1);
        Full = new Semaphore(0);
        Empty = new Semaphore(size);
        MAX_SIZE = size;
}


bool Boundedbuffer::empty() {
        if (que.size() == 0) return true;

        return false;
}

int Boundedbuffer::size() {
        return que.size();
}

Item* Boundedbuffer::getItem() {
        Item* new_item;
        Full->P();
        Lock->P();
        new_item = que.front();
        que.pop();
        Lock->V();
        Empty->V();
        return new_item;
}

void Boundedbuffer::putItem(Item* _item) {
        Empty->P();
        Lock->P();
        que.push(_item);

        Lock->V();
        Full->V();
}

void Boundedbuffer::print() {
    queue<Item*> que2 = que;
    cout << "Bounded Buffer: ";

    while(que2.empty() == false) {
        cout << que2.front()->Id;
        cout << " " << que2.front()->info << "\n";
        que2.pop();
    }
}
```

`Item.H`

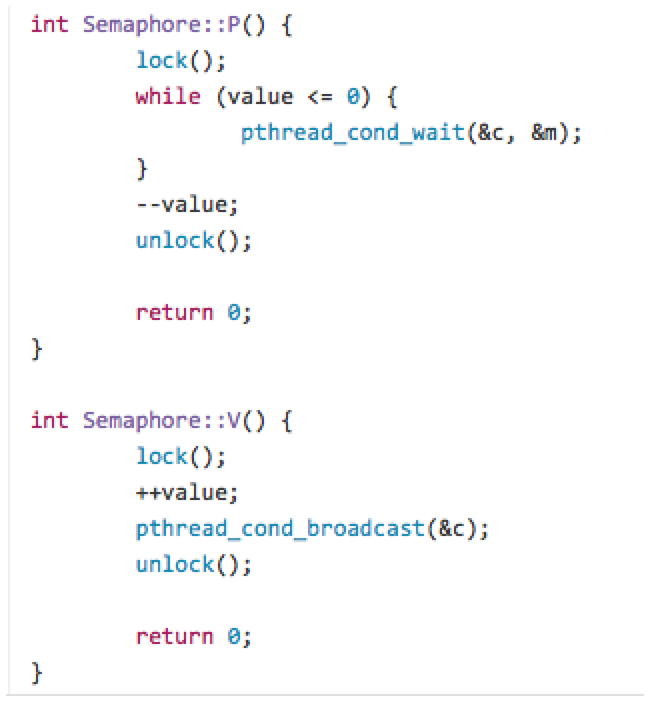
The item class simply created the object we wanted to store into our buffer. It consisted of the information we wanted to store into the item in the form of a string and an ID in the form of an integer.



```
class Item {
public:
        int Id;
        string info;
        Item(): Id(-1), info("") { }
        Item(int _id, string _info, int _request_num): Id(_id), info(_info) { }

        ~Item() { }
};
```

`semaphore.C`

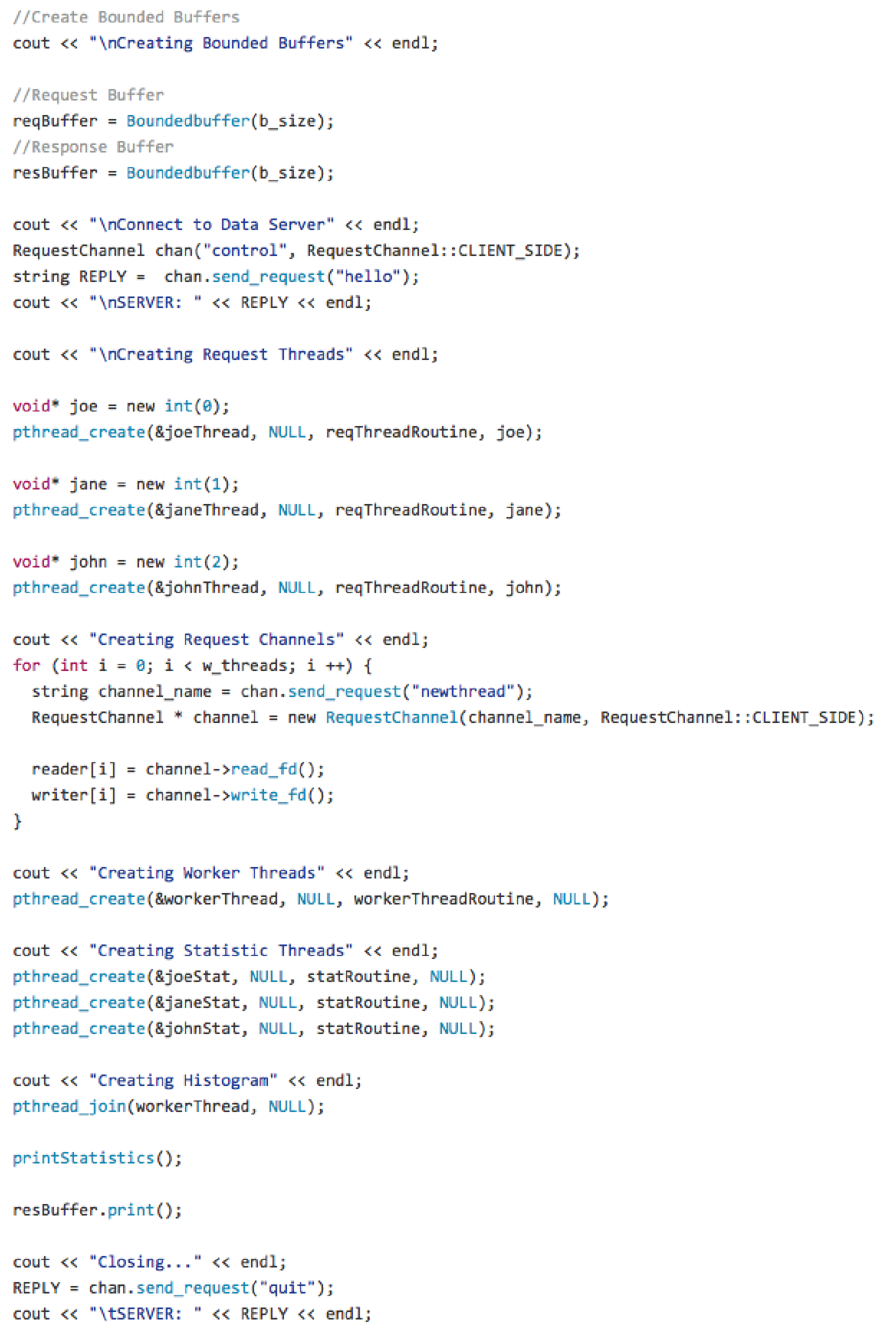
The semaphore acted as our locking mechanism to prevent threads from accessing data we did not want them to access. Semaphore essentially just waits or doesn't wait depending on the value of the semaphore. If the value is less than or equal to zero, then the thread must wait.



```
int Semaphore::P() {
        lock();
        while (value <= 0) {
                pthread_cond_wait(&c, &m);
        }
        --value;
        unlock();

        return 0;
}

int Semaphore::V() {
        lock();
        ++value;
        pthread_cond_broadcast(&c);
        unlock();

        return 0;
}
```

client.C

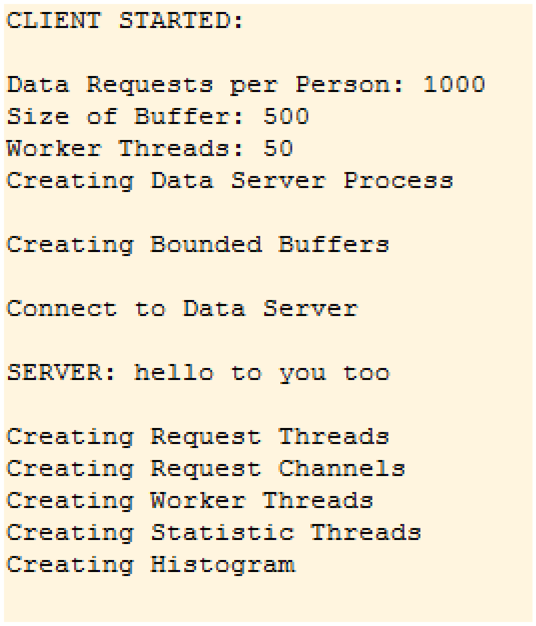
This is the main for our program, as well as a large portion of our functions that run the core of our program. Client creates our threads for the entire program. This is where the functionality for the bounded buffer, semaphore, and item are used. We create the threads in reverse or they are used to prevent them from completing before they are used. This means we start with the stat threads and end with the client threads. We then run the threads with help of the bounded buffers in communication with the server. The statistics are saved and at the end we use our print functions to output the data cleanly on the screen as seen below.



```
//Create Bounded Buffers
cout << "\nCreating Bounded Buffers" << endl;

//Request Buffer
reqBuffer = Boundedbuffer(b_size);
//Response Buffer
resBuffer = Boundedbuffer(b_size);

cout << "\nConnect to Data Server" << endl;
RequestChannel chan("control", RequestChannel::CLIENT_SIDE);
string REPLY =  chan.send_request("hello");
cout << "\nSERVER: " << REPLY << endl;

cout << "\nCreating Request Threads" << endl;

void* joe = new int(0);
pthread_create(&joeThread, NULL, reqThreadRoutine, joe);

void* jane = new int(1);
pthread_create(&janeThread, NULL, reqThreadRoutine, jane);

void* john = new int(2);
pthread_create(&johnThread, NULL, reqThreadRoutine, john);

cout << "Creating Request Channels" << endl;
for (int i = 0; i < w_threads; i ++) {
  string channel_name = chan.send_request("newthread");
  RequestChannel * channel = new RequestChannel(channel_name, RequestChannel::CLIENT_SIDE);

  reader[i] = channel->read_fd();
  writer[i] = channel->write_fd();
}

cout << "Creating Worker Threads" << endl;
pthread_create(&workerThread, NULL, workerThreadRoutine, NULL);

cout << "Creating Statistic Threads" << endl;
pthread_create(&joeStat, NULL, statRoutine, NULL);
pthread_create(&janeStat, NULL, statRoutine, NULL);
pthread_create(&johnStat, NULL, statRoutine, NULL);

cout << "Creating Histogram" << endl;
pthread_join(workerThread, NULL);

printStatistics();

resBuffer.print();

cout << "Closing..." << endl;
REPLY = chan.send_request("quit");
cout << "\tSERVER: " << REPLY << endl;
```

## Results

Our results were positive, and we managed to get output that correctly displayed. First, we started the client and took the input from the command line and displayed it to the user.



```
CLIENT STARTED:

Data Requests per Person: 1000
Size of Buffer: 500
Worker Threads: 50
Creating Data Server Process

Creating Bounded Buffers

Connect to Data Server

SERVER: hello to you too

Creating Request Threads
Creating Request Channels
Creating Worker Threads
Creating Statistic Threads
Creating Histogram
```

In our results we had varying ranges of statistics.

```
------------------------
Statistics for Joe Smith
------------------------
0 - 9           136 times
10 - 19         115 times
20 - 29         71 times
30 - 39         78 times
40 - 49         64 times
50 - 59         40 times
60 - 69         78 times
70 - 79         107 times
80 - 89         116 times
90 - 99         38 times
------------------------
Total requests: 1000
------------------------

-------------------------
Statistics for Jane Smith
-------------------------
0 - 9           121 times
10 - 19         71 times
20 - 29         114 times
30 - 39         108 times
40 - 49         116 times
50 - 59         114 times
60 - 69         136 times
70 - 79         67 times
80 - 89         11 times
90 - 99         53 times
-------------------------
Total requests: 1000
-------------------------

-----------------------
Statistics for John Doe
-----------------------
0 - 9           45 times
10 - 19         84 times
20 - 29         85 times
30 - 39         84 times
40 - 49         90 times
50 - 59         116 times
60 - 69         56 times
70 - 79         96 times
80 - 89         143 times
90 - 99         128 times
-----------------------
Total requests: 1000
-----------------------

Closing...
        SERVER: bye
close requests channel control
close requests channel control
close IPC mechanisms on server side for channel control
```

## Conclusion

From this we learned that threading has its quirks. Getting the threads to work together was a major challenge. Semaphore and boundedbuffer were rather simple once we were sure the logic was correct; client was the major challenge. Learning which threads to start first and how to manipulate them in the buffers was the real challenge. Storing their data from the statistics threads was also a challenge. Although, threading was challenging, but it gave us a better understanding of operating systems and CPUs.