Machine Problem 4: Multi-Threading Across A Data Server

CSCE-313-501

By: Andrew Schlotzhauer and Jaiden Gerig

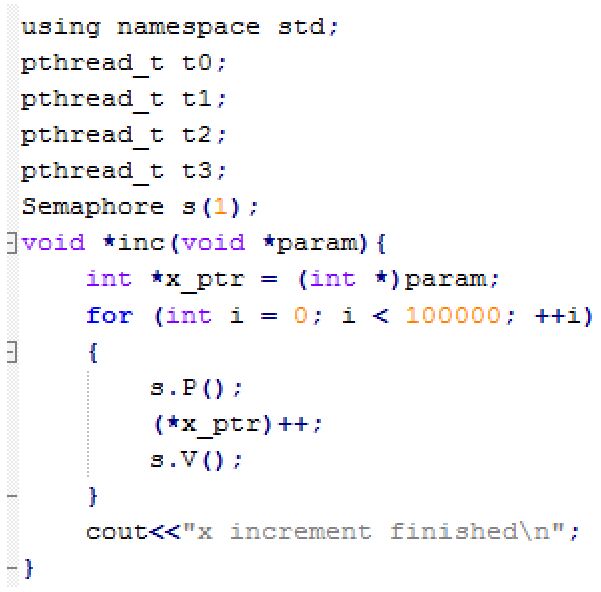
April 17, 2015

Introduction

Concurrent programming is both a very hard and very important topic in Computer Science today because of the many cores a CPU can have and everyone wants things done faster and wants to get the most for their money when buying a CPU. Dealing with threads is difficult because of the possibility of *race conditions*, which happen when 2 parts of the same program are running on separate threads and need to access the same information. If and when these two parts try to access the same data, they may change it in some way that the other part needs to know what the new value is but ends up getting the old value instead and that could be catastrophic in today's computer dependent world. We are to extend upon our MP2 which was dealing with inter-process communication between a client server and a data server. In this machine problem, we need to come up with a way to safely send requests to the server and have them bounce off the server and be collected properly with the associated person that sent the request. This would be easily done, but we are to do so using threads and concurrent programming, so the requests are handled by the server concurrently and don't cause these *race conditions*.

Procedures

The first thing that we did was write the Semaphore class, because we wanted to do the bounded buffer next and that needed the use of the Semaphore in order to be concurrent. After writing the Semaphore class we tested it to make sure it worked properly with a simple check by having multiple threads adding to the same integer value and without our Semaphore the value was always different, but with it, it always was the same value. Below is the function we tested our Semaphore with initially.



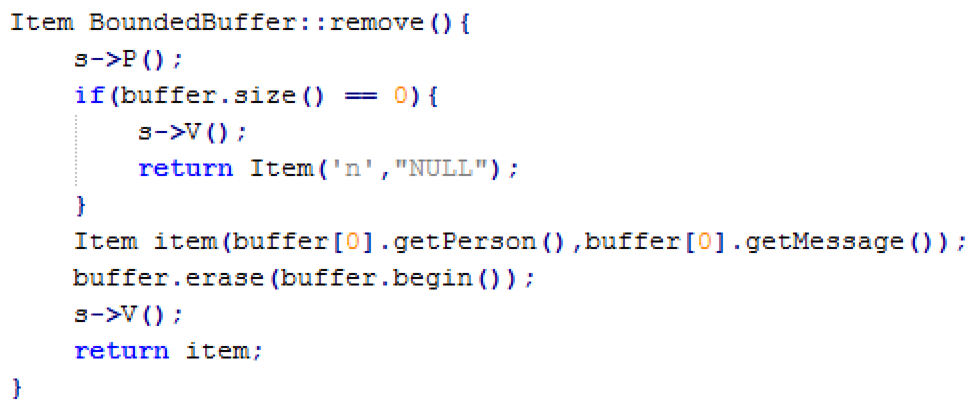
```
using namespace std;
pthread_t t0;
pthread_t t1;
pthread_t t2;
pthread_t t3;
Semaphore s(1);
void *inc(void *param){
    int *x_ptr = (int *)param;
    for (int i = 0; i < 100000; ++i)
    {
        s.P();
        (*x_ptr)++;
        s.V();
    }
    cout<<"x increment finished\n";
}
```

The next thing we did was complete the bounded buffer class. It was fairly simple, we knew we needed a dynamic array with an upper bound as given in the project description. So we implemented it with a vector<Item> in which the Item class was created as well which stores the original string sent to the server, a char id for the identity of the sender and an integer to store the data the server responds with. The Bounded Buffer class also has a variable of maxSize which is an integer value that initializes the maximum size the vector can be.

```
class Item
{
public:
    Item(char p, string m);
    string getData() { return data; }
    void setData(string x) { data = x; }
    char getPerson() { return person; }
    void setPerson(char c) { person = c; }
    string getMessage() { return message; }
    void setMessage(string m) { message = m; }
private:
    string data;
    char person; //identifier for the person who the item belongs to
    string message;
};
```

```
class BoundedBuffer
{
public:
    BoundedBuffer(int size, Semaphore* sem);
    ~BoundedBuffer();
    void add(Item item);
    int getMaxSize() { return maxSize; }
    void setMaxSize(int x) { maxSize = x; }
    Item remove();
    Semaphore* s;
    void finished(){s->P();
                    numFinished++;
                    s->V();}
    int numFinished;
    int getSize() {return buffer.size(); }
private:
    vector<Item> buffer;
    int maxSize;
};
```

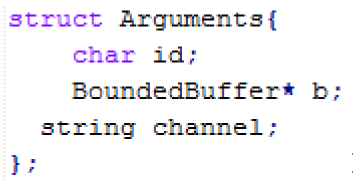
An easy example of how we are using the Semaphore throughout our Bounded Buffer class is the remove() method, where s is our Semaphore* that is pointing to the Semaphore that the Bounded Buffer is initialized with.



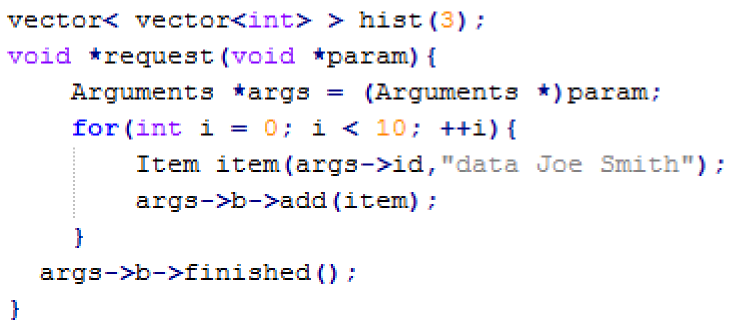
```
Item BoundedBuffer::remove(){
    s->P();
    if(buffer.size() == 0){
        s->V();
        return Item('n',"NULL");
    }
    Item item(buffer[0].getPerson(),buffer[0].getMessage());
    buffer.erase(buffer.begin());
    s->V();
    return item;
}
```

s->P() locks the Semaphore down so others can't change anything while it is locked thus it locks and checks the size, if we didn't lock it then the size could be changing and we wouldn't get an accurate check there every time. If the buffer isn't empty, then it removes the first item and erases the first item so it always gets the oldest value first then returns that item.

After the Bounded Buffer was finished we moved on to finally doing what we originally needed to do which was to send requests through a bounded buffer which makes sure the requests don't get too many at a time. Then worker threads concurrently send requests to the server and get the server's responses and send those into its respective bounded buffer which could be 1 of 3 bounded buffers because there were 3 clients which are identified by the Item class with the char person.

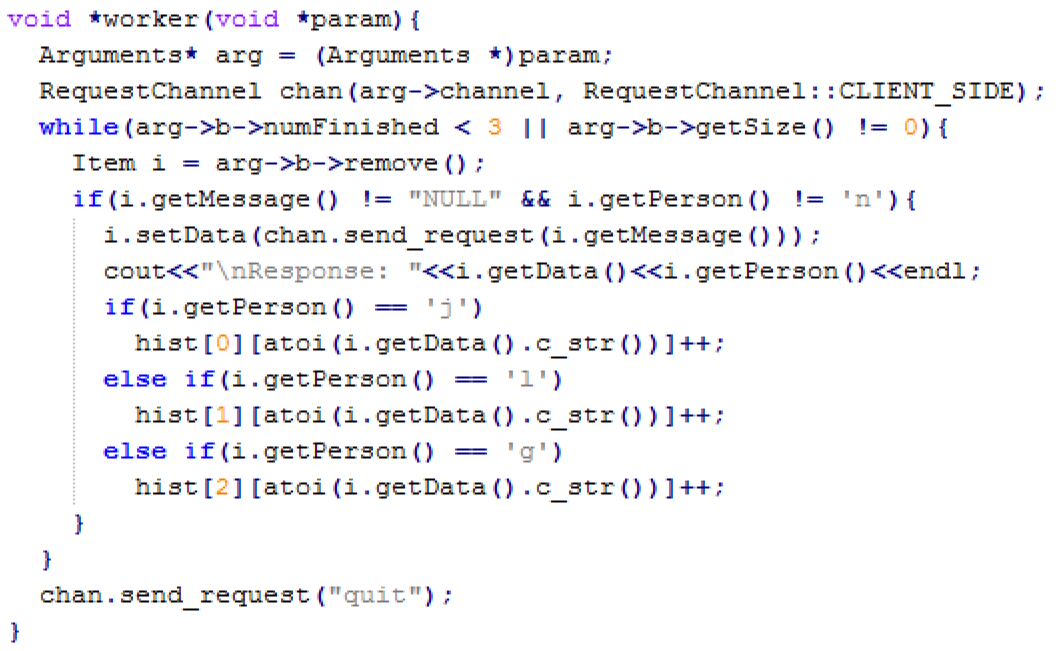


```
struct Arguments{
    char id;
    BoundedBuffer* b;
  string channel;
};
```



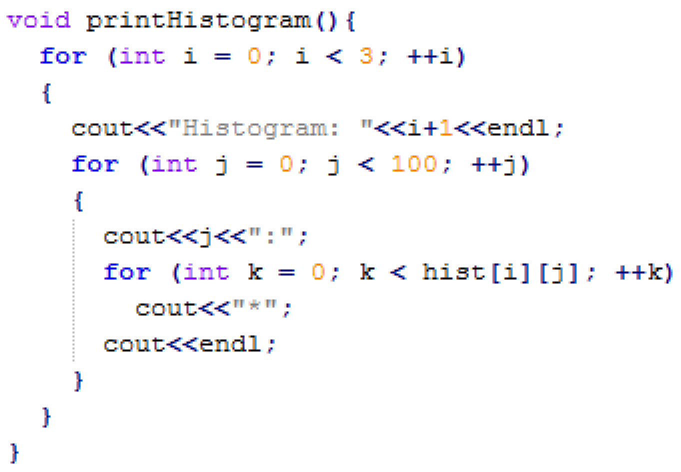
```
vector< vector<int> > hist(3);
void *request(void *param){
    Arguments *args = (Arguments *)param;
    for(int i = 0; i < 10; ++i){
        Item item(args->id,"data Joe Smith");
        args->b->add(item);
    }
  args->b->finished();
}
```

This is our request function which throws requests into the bounded buffer which will be sent to the server with our worker function which is shown next. Above that class it can probably be seen that there is a histogram which is used later.



```
void *worker(void *param){
  Arguments* arg = (Arguments *)param;
  RequestChannel chan(arg->channel, RequestChannel::CLIENT_SIDE);
  while(arg->b->numFinished < 3 || arg->b->getSize() != 0){
    Item i = arg->b->remove();
    if(i.getMessage() != "NULL" && i.getPerson() != 'n'){
      i.setData(chan.send_request(i.getMessage()));
      cout<<"\nResponse: "<<i.getData()<<i.getPerson()<<endl;
      if(i.getPerson() == 'j')
        hist[0][atoi(i.getData().c_str())]++;
      else if(i.getPerson() == 'l')
        hist[1][atoi(i.getData().c_str())]++;
      else if(i.getPerson() == 'g')
        hist[2][atoi(i.getData().c_str())]++;
    }
  }
  chan.send_request("quit");
}
```

Above is our worker function which is for the worker threads. This takes in a parameter which is converted into an Arguments variable which is just a bounded buffer, id, and channel. Then it gets the servers response and then it adds it to the appropriate histogram for the correct person which sent the request. Below is a picture of how we print our histogram data.



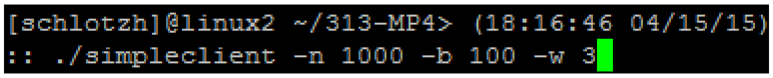
```
void printHistogram(){
  for (int i = 0; i < 3; ++i)
  {
    cout<<"Histogram: "<<i+1<<endl;
    for (int j = 0; j < 100; ++j)
    {
      cout<<j<<":";
      for (int k = 0; k < hist[i][j]; ++k)
        cout<<"*";
      cout<<endl;
    }
  }
}
```

Our main function (shown below) or at least this chunk of our main is just to show how we used the threads for our program and how it should be running concurrently at better pace, but there's no way of knowing with this little of data being sent. Then it prints a histogram of the data after waiting for all the threads to finish which is what the join() method does.

```
vector<Arguments> arr(3);
for (int i = 0; i < 3; ++i)
{
  arr[i].channel = chan.send_request("newthread");
  arr[i].b = &b;
  pthread_create(&workers[i], NULL, worker, &arr[i]);
}
for (int i = 0; i < clients.size(); ++i)
{
  pthread_join(clients[i], NULL);
}
for (int i = 0; i < 1; ++i)
{
  pthread_join(workers[i], NULL);
}
chan.send_request("quit");
usleep(1000000);
printHistogram();
```

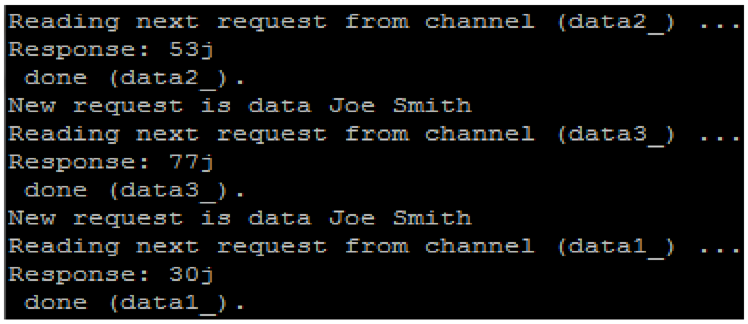
Results

Our results went pretty much as expected, we use the getopt to read in command line arguments otherwise n is initialized to 10, b to 15, and w to 3. But this is an example of how to run our program, after compiling by typing make.



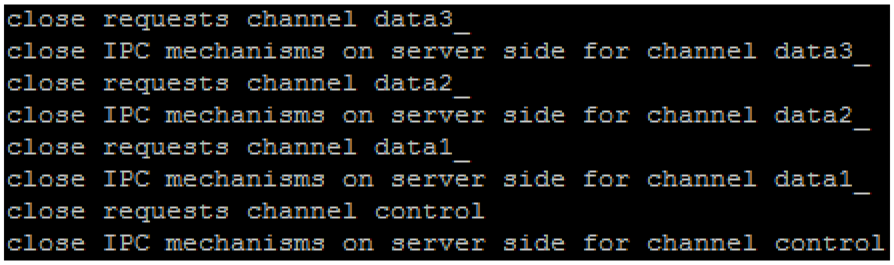
```
[schlotzh]@linux2 ~/313-MP4> (18:16:46 04/15/15)
:: ./simpleclient -n 1000 -b 100 -w 3
```

So what our program does is it first sends the data to the server and stores all the responses and then closes the IPC mechanisms and then prints the histogram. It flows like this:



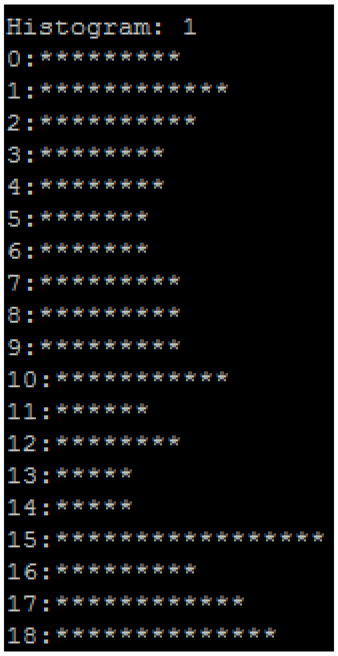
```
Reading next request from channel (data2_) ...
Response: 53j
 done (data2_).
New request is data Joe Smith
Reading next request from channel (data3_) ...
Response: 77j
 done (data3_).
New request is data Joe Smith
Reading next request from channel (data1_) ...
Response: 30j
 done (data1_).
```

And it continues to do this for each request thread for however much n was so for the above program it runs for 3000 times because n is 1000 and there are always 3 clients. Then it closes the IPC mechanisms.



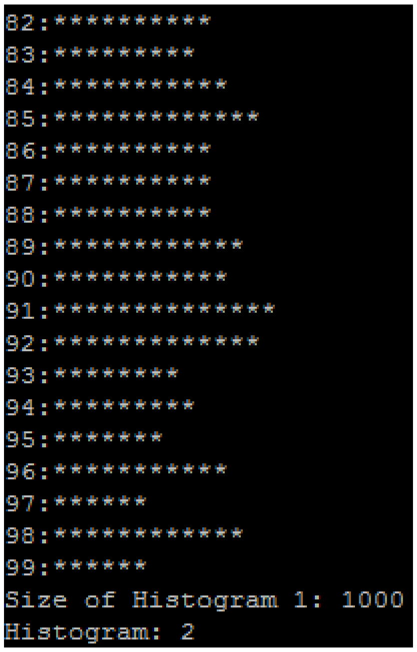
```
close requests channel data3_
close IPC mechanisms on server side for channel data3_
close requests channel data2_
close IPC mechanisms on server side for channel data2_
close requests channel data1_
close IPC mechanisms on server side for channel data1_
close requests channel control
close IPC mechanisms on server side for channel control
```

Then once those are closed, it prints a histogram of the random integers that the server replied with, which looks like this:



```
Histogram: 1
0:*********
1:************
2:**********
3:********
4:********
5:*******
6:*******
7:*********
8:*********
9:*********
10:***********
11:******
12:********
13:*****
14:*****
15:*****************
16:*********
17:************
18:**************
```

… … …



```
82:**********
83:*********
84:***********
85:*************
86:**********
87:**********
88:**********
89:************
90:***********
91:**************
92:*************
93:********
94:*********
95:*******
96:***********
97:******
98:************
99:******
Size of Histogram 1: 1000
Histogram: 2
```

It prints the three histograms for the 3 clients and then the size of the histogram which should be the size of whatever n was, which in this case is indeed 1000. So our program ran as expected and we got good results.

Conclusion

Threads are risky business, too many things can go wrong, and it is very difficult to make sure you have every possibility that could make your program deadlock or have a race condition is handled correctly. But for this machine problem, I believe we handled it well and seem to not have concurrency problems but this was a simple program for threads and should not be mistaken for being easy. Overall I believe our program did what was asked for in the project description and as I said before that it is hard to tell if it is actually speeding things up running in separate threads, but theoretically it should be faster no matter how small of a difference. This was a good way to learn about threads and concurrency, because it was simple yet got the point across about what you need to handle with the locking and unlocking mechanisms of a Semaphore/Lock or whatever may be needed in the future.