Western Engineering

Electrical and Computer Engineering
Software Engineering

SE 3313 – Operating Systems

Lab 3: Multi-Threaded Socket Programming

Table of Contents

1. Lab 3: Multi-Threaded Socket Programming

1.1. Objective

By now, I believe that in your networking course, you are familiar with sockets and socket programming. What I don't know is exactly how your socket programs have been structured. It is very difficult to talk about socket programming with any complexity, without also using a multi-threaded approach. So, this lab is an exercise in how to do exactly that.

The objectives for this lab are as follows:

1. Give you a chance to work with an extensible Thread class.
2. Give you a chance to design, and implement, a multi-threaded application
3. Give you a chance to work with sockets and socket servers in a multi-threaded context

1.2. Introduction

This document contains a lot of information. This section describes the tasks for Lab 3 while section 2 "Lab 3 Resources" provides additional information for completing this lab. I STRONGLY recommend that you read it all the way through and refer to it often as you work on the problem.

For lab 3, begin by cloning the repository:
https://github.com/uwoece-km/se3313-2017-Lab4.git

In lab 3 you should create two applications, and we will call them **Server** and **Client**.
Server, as the name implies, opens a socket server on a particular port and waits for connections.
Client, immediately when executed, attempts to open a socket connection on the appropriate port.
Server must be designed in such a way that an *arbitrary* number of **Client**s can be run at the same time. Each one should successfully connect to a socket on the port, and display the same outward behavior.
The user interface of **Client** is quite simple. It should just sit there waiting for user input. Once the user has typed a string, it sends the string to the **Server** over the socket connection. Once the **Server** receives the transmitted string, it does something to it, and sends it back. I don't care what it does to the string: transform it to upper case, reverse the order of the letters, make every third letter an 'X', use it to index into a hash table containing the complete works of Shakespeare…You can do whatever you like, as long as the transformation carried out by **Server** means that the string sent from **Server** to **Client** received in reply is:

1) Not exactly the same as the string transmitted TO the **Server** AND
2) In some way dependent on the string transmitted TO the **Server**

At the **Client** side, when the user enters "done", the **Client** should terminate gracefully. In other words, it should close the socket, clean up after itself, and terminate, **without core dumps**, **unhandled exceptions**, and so on.
The **Server** must also terminate gracefully. In this case, you can decide how to do it, either by typing something at the **Server**, or by sending a special command from a **Client**, or some other technique. I

should warn you, though, that I don't consider entering Ctrl-C at the **Server** to be a "graceful" form of termination.
I have provided some resources that I believe will help support your work in this lab. They are described in what follows.

1.3. Deliverables

1. ZIP file with the code for lab3
    - Name your submission: se3313a-username-lab3.zip

If the student did not demo the solution to TA during the schedule lab time due to illness or other special circumstances (with documentation as per the course outline), the student must contact his/her assigned TA as soon as possible. ***For no demo, 50% of the available mark will be deducted.***

2. Lab 3 Resources

2.1. Outline

The UNIX synchronization primitives are…primitive. Accordingly, I have wrapped them in some useful classes. This document explains those classes (and some auxiliaries), which I have wrapped into a namespace called Sync.

This guide contains both a very useful set of objects for synchronization (the **Blockables**) and some classes for socket communication (Socket and **SocketServer**) that are useful in Lab 4 and the project.

2.2. Meet the Blockables

Unix has a very interesting, if sometimes maddening, approach to blocking calls. Most (but unfortunately not all) of the things that can block are based around an element called a **file descriptor**. This is just what it sounds like. It’s a number you give to Unix to identify the file you want to read from or write to. However, it has been generalized so that (among other things) pipes, sockets, stdin, stdout and stderr are all described by **file descriptors**.

Unix also has a function called **select()** that has a rather opaque interface, but the gist of it is as follows. Let us suppose that you have a set of **file descriptors**. Each of them has a corresponding call that might block (ie: **read(), recv(), accept(), cin**…). You are unsure which one will need servicing next. You can’t afford to block on any one of them, because then the others will go unserviced. Then what you do is pass the whole set to **select()**. This call tells Unix “Here are a set of objects that each have a blocking call. Please tell me when one of them is in a state such that I can make that call without blocking.” THEN you block on ***select()*** that is unlike all other blocking calls, allows you to specify a timeout. (ie: wait this many milliseconds, then give up if nothing good happens.)

Therefore, there is a class called ***Blockable*** and an associated class called ***FlexWait***, which is described later in the document.

2.2.1. Blockable base class

```
class Blockable
{
protected:
    int fd;
public:
    Blockable(int f=0):fd(f){;}
    Blockable(Blockable const & b) : fd(dup(b.fd)){;}
    virtual ~Blockable(void){;}
    operator int(void)const {return fd;}
    void SetFD(int f){fd =f;}
    int GetFD(void) const {return fd;}
};
extern Blockable cinWatcher;
```

Figure 1: Blockable Interface

As you can see, ***Blockable*** wraps a ***file descriptor***. You can use the base class object to wrap simple file descriptors if you want to. In particular, I assume you will want to wrap ***stdin (cin)*** in order to wait for user input without blocking, so I have done that for you with the externally defined object ***cinWaiter***.

Blockable is primarily useful as the base class for inputs to ***FlexWait***, which is described last.

2.2.2. ThreadSem Class

The named ***Semaphore*** class is still available to you, if you need to share semaphores between processes. However, there is a lightweight class derived from ***Blockable*** that you may prefer to use to synchronize between threads ***within the same process***. This class is not useful to synchronize between processes, so don't try.

The named ***Semaphore*** class is not ***Blockable*** (per this definition) because it does not contain a file descriptor. As a result, you can block on a ***Semaphore*** if you want to (as in Lab 3), but you cannot simultaneously block on something else as well.

However, ***ThreadSem*** is a ***Blockable***. That means it can be combined with other ***Blockables*** to make multiple blocking calls using ***FlexWait***. Other than that, ***ThreadSem*** acts like a semaphore, as you will see below:

```
class ThreadSem : public PipeUser
{
public:
    ThreadSem(int initialState = 0);
    ThreadSem(ThreadSem const &);
    ~ThreadSem() { ; }
    ThreadSem &operator=(ThreadSem const &);
    void Wait(void);
    void Signal(void);
};
```

Note that there are two ways to work with this object. If you call ***ThreadSem::Wait***, it acts like a typical semaphore wait and can block. If you pass it as part of a multiple wait using ***FlexWait***, you will STILL have to call ***ThreadSem::Wait***, with the expectation that it will not block (somebody could still steal the signal from you, though).

2.2.3. Event class

The ***Blockable*** family also includes a class called ***Event***. Its declaration is here:

```
class Event : public PipeUser
{
public:
    Event(void) { ; }
    ~Event() { ; }
    Event(Event const &);
    Event &operator=(Event const &);
    void Trigger(void);
    void Wait(void);
    void Reset(void);
};
```

An ***Event*** is essentially a binary semaphore. It can be in two states (signaled and not-signaled). You can block on it. You can use it to synchronize two threads.

What makes it special is that it must be *manually reset*. If you call ***Event::Wait***, the calling thread will block until the ***Event*** is signaled by ***Event::Trigger***. However, unlike a semaphore, when you return from ***Event::Wait***, the ***Event*** is *still in the signaled state*. You have to manually call ***Event::Reset*** to put it back to not-signaled. As a result, if you pass it to a ***FlexWait*** you DO NOT need to then call ***Event::Wait***. You know it is signaled just because you returned.

The functionality of ***Event*** is not very good for mutual exclusion, but it is very useful for implementing the "I'm going to wait here until you tell me it's OK to proceed" style of inter-thread communication. This is VERY useful in thread termination.

2.2.4. FlexWait Class

At the heart of the concept is the ***FlexWait*** class. The interface (which is very simple) is below:

```
class FlexWait
{
public:
    static const int FOREVER; //==-1
    static const int POLL;    // == 0
private:
    std::vector<Blockable *> v;

public:
    FlexWait(int n, ...);
    Blockable *Wait(int timeout = -1);
};
```

As you can see, the constructor takes a variable argument list. The first parameter is an integer, saying how many things you would like to wait on. It can be any number greater than or equal to 1. After that, you pass a set of pointers to ***Blockable***. Because all ***Blockables*** are related by inheritance, that means you can pass pointers to ***Blockable, ThreadSem, Event, Socket*** or ***SocketServer***. As long as you pass the same number of pointers as the number you claimed in the first parameter, all will be well.

Then we have ***Wait***. It will block until one of the ***Blockables*** you passed to the object is ready for whatever thing it does. (That is, if it's an ***Event,*** it's triggered. If it's a ***SocketServer,*** there is an incoming socket request. If it's a ***Socket***, there is data waiting to be read. And so on.) ***Wait*** takes only one parameter, which is optional and can be used to specify a timeout. If you do not specify a timeout, the function will block forever (until something triggers).

2.2.5. Socket objects as Blockables

As you will see below, the classes ***Socket*** and ***SocketServer*** are both ***Blockable***:
You can call the ***SocketServer::Accept*** and use ***SocketServer::Shutdown*** to interrupt it. Or you can pass the ***SocketServer*** to a ***FlexWait*** (maybe along with something else) and call ***FlexWait::Wait*** to tell you when there is an incoming connection. Similarly, you can still call ***Socket::Read*** to block until you have

data and call ***Socket::Close*** to break out of it. Or, you can pass the ***Socket*** to a ***FlexWait*** with something else and call ***FlexWait::Wait*** to tell you when you have data to read.

2.3. SocketServer Class

I have implemented for you a very simple (VERY simple) ***SocketServer*** class. It should be familiar to any of you who have experience with socket programming in an object-oriented environment. It's just very, very limited in what it can do. The definition is like this:

```
class SocketServer : public Blockable
{
private:
    int pipeFD[2];
    Event terminator;
    sockaddr_in socketDescriptor;

public:
    SocketServer(int port);
    ~SocketServer();
    Socket Accept(void);
    void Shutdown(void);
};
```

It has the following capabilities:

1. The constructor ***SocketServer*** takes only one parameter, which is the integer number of the ***port*** you want to listen on. My understanding is that UNIX reserves all ports below 1000 to pre-defined network services, but as long as you use a number greater than that (I've been using 2000), you should have no trouble. The ***SocketServer*** will accept incoming requests for stream sockets from any IP address. Up to five requests can be queued.
2. The function that does most of the work is ***Accept***. ***Accept*** is a blocking call. If you call it, it will not return until some remote client attempts to open a ***Socket*** on the chosen port. If you call it from within a thread (for example, a thread implemented using ***Thread***) and it blocks, the thread can't be **join**ed until you interrupt the block.
   The return value from ***Accept*** is a ***Socket*** object, which is already open and ready for use. ***Sockets*** have value semantics, so you can freely copy or assign them.

   After a successful call to ***Accept***, the ***SocketServer*** is still active and can be used for a subsequent call to ***Accept***.
3. To handle the problem identified in (2), we have ***Shutdown***. Calling ***Shutdown*** will terminate an active ***Accept*** and make the server stop listening for socket requests on the port. Remember, the thread that called ***Accept*** is BLOCKED. Should you need to call ***Shutdown***, you will need to do so from a different thread.
4. The class throws two types of exceptions. For general, run-of-the-mill errors, it will throw a ***std::string*** with some text indicating what kind of error it was. I would recommend always using ***SocketServer*** inside a ***try…catch…*** block so that you have meaningful information to debug.
5. The one meaningful, non-error exception thrown by ***SocketServer*** is the ***TerminationException***. This is just an integer, the value of which is unimportant. However, ***Accept*** throws this type of

exception if a blocked ***Accept*** is interrupted by ***Shutdown***. You will probably find this functionality useful when it comes time to implement a graceful termination of the server.

2.4. Socket class

I have also implemented a very simple ***Socket*** class. It should also be familiar and is also very, very limited. The definition is like this:

```
class Socket : public Blockable
{
private:
    sockaddr_in socketDescriptor;
    bool open;
    Event terminator;

public:
    Socket(std::string const &ipAddress, unsigned int port);
    Socket(int socketFD);
    Socket(Socket const &s);
    Socket &operator=(Socket const &s);
    ~Socket(void);

    int Open(void);
    int Write(ByteArray const &buffer);
    int Read(ByteArray &buffer);
    void Close(void);
};
```

It has the following capabilities:

1. A ***Socket*** constructed using the first constructor is intended for use by a **Client**. An IP address is specified as a ***std::string*** (like “127.0.0.1”), and the port number is provided as an integer. Calling the constructor DOES NOT OPEN THE PORT. It only creates the necessary conditions for a call to ***Open***.
2. ***Open*** should be used by **Client**s after constructing the ***Socket*** object. It will attempt to connect to the IP address and port as specified.
3. A ***Socket*** created by a ***SocketServer*** in response to a connection request uses the second constructor. **Client**s should not use this form of the constructor. The ***Socket*** returned by ***SocketServer::Accept*** is already open, so **Server**s do not need explicit calls to ***Socket::Open***.
4. ***Write*** takes a single ***ByteArray*** parameter, which contains the data to send. (See below for the definition of ***ByteArray***). ***Write*** never blocks. The return value is the number of bytes successfully written. If the return value is different from the number of elements in ***ByteArray***, the most likely explanation is that the ***Socket*** has been closed at the other end.
5. ***Read*** takes a single ***ByteArray*** parameter and populates the ***ByteArray*** with data received. ***Read*** is a BLOCKING call. It will block until data is received, the connection is closed or there is an error. The return value is the number of bytes received. A return value of 0 means the ***Socket*** was gracefully closed at the other end. A return value less than 0 indicates an error.
6. ***Close*** closes the connection. It can be called either by the client or the server side.

2.5. ByteArray class

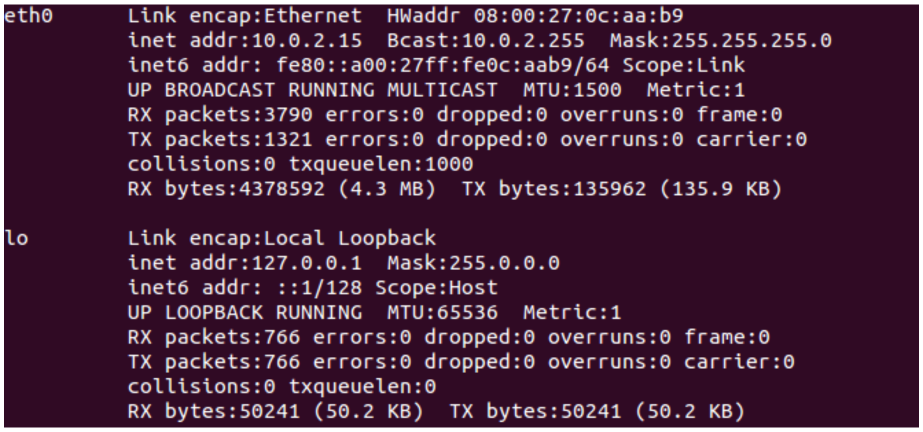
I have implemented a rudimentary class called ***ByteArray*** to simplify the task of writing and reading over a socket. Basically, UNIX sockets expect a raw pointer to byte data and a length. ***ByteArray*** just lets us encapsulate this data into a class. The definition is as follows:

```
class ByteArray
{
public:
    std::vector<char> v;
    std::string ToString(void) const
    {
        std::string returnValue;
        for (int i = 0; i < v.size(); i++)
            returnValue.push_back(v[i]);
        return returnValue;
    }
    ByteArray(void) {}
    ByteArray(std::string const &in)
    {
        for (int i = 0; i < in.size(); i++)
            v.push_back(in[i]);
    }
    ByteArray(void *p, int s)
    {
        char *temp = (char *)p;
        for (int i = 0; i < s; i++)
            v.push_back(temp[i]);
    }
};
```

As you can see, this is not much of a class. If you have a ***std::string*** to transmit, you can just pass it to the constructor and it will turn your string into raw data. If you have raw data received, ***ToString*** makes it into a ***std::string***. If you have some other type you want to send, you can pass a pointer to it (cast to ***void ****) and its size, and it will make your data into raw data.

2.6. Networking in UNIX

It is virtually certain that you know more about networking than I do. However, I do know this much. If you type the command ***ifconfig*** at the UNIX command line, you will get an output that looks like this:



```
eth0      Link encap:Ethernet  HWaddr 08:00:27:0c:aa:b9
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe0c:aab9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3790 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1321 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4378592 (4.3 MB)  TX bytes:135962 (135.9 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:766 errors:0 dropped:0 overruns:0 frame:0
          TX packets:766 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:50241 (50.2 KB)  TX bytes:50241 (50.2 KB)
```

As you can see, this output can be used to find out the IP address that has been assigned to your machine, in this case “10.0.2.15”. You can use this value in a string in your **Client** applications. It is fine to hard code this quantity: don’t worry about trying to determine it at run time. Or of course, you can use the address “127.0.0.1” which I believe is “home” and should always work.

2.7. A Warning About UNIX Networking

Here is something interesting I have discovered about Unix sockets. A socket can be closed at either end (ie: by client or server). However, it seems that under certain circumstances, after being closed, sockets placed into a “timed wait” state. This is NOT an error state; it is defined to prevent unauthorized users from accessing your server. However, it will make the associated port unusable for a significant period of time (measured in minutes). This is kind of annoying while debugging.
Some resources suggest that “timed wait” does not occur provided the client closes the socket. As a result, you may wish to design your implementation such that only the client end ever actually closes a socket. However, this functionality appears to be platform dependent, so you might need to live with the wait.

2.8. Exporting ports from Virtual Box

Many of you are running Unix in a Virtual Box environment. Since this environment is virtual, its ports are “not real”, in the sense that it is not possible to access them externally. For the purposes of Lab 3, this is not a problem. If both client and server live in the same virtual environment, then they can communicate quite easily. However, if you want your client to run outside the virtual environment (in a browser, on your phone, whatever…) you will need to host your service on a real port on the real internet. You can set up Virtual Box to accept incoming connections BUT…

IMPORTANT NOTE: This probably won't work for you. It requires that your computer have a permanent internet connection. If you are running on a laptop over WiFi (especially if you are running on a laptop over **uwosecure-v2**), your computer can't accept incoming socket connections. For that, we have Amazon Web Services. If you have a permanent internet connection, then you can use the "port forwarding" feature in Virtual Box to turn the port on the virtual machine into a port on the host machine.

2.9. A new Makefile

We are starting now to have something more like a C++ application, involving a large number of compilation units that must be linked together. Accordingly, I have made a new Makefile for you. The Makefile looks like this:

```
all: Client Server

Client : Client.o socket.o Blockable.o
    g++ -o Client Client.o socket.o Blockable.o -pthread -l rt

Client.o : Client.cpp socket.h
    g++ -c Client.cpp -std=c++11

Server : Server.o thread.o socket.o socketserver.o Blockable.o
    g++ -o Server Server.o thread.o socket.o socketserver.o Blockable.o -pthread -l rt

Blockable.o : Blockable.h Blockable.cpp
    g++ -c Blockable.cpp -std=c++11

Server.o : Server.cpp thread.h socketserver.h
    g++ -c Server.cpp -std=c++11

thread.o : thread.cpp thread.h
    g++ -c thread.cpp -std=c++11

socket.o : socket.cpp socket.h
    g++ -c socket.cpp -std=c++11

socketserver.o : socketserver.cpp socket.h socketserver.h
    g++ -c socketserver.cpp -std=c++11
```

As you can see, the Makefile is becoming non-trivial. Basically, though, what it does is individually compile each source file (compiles it only to object). Then, in a separate step, it links all relevant object files into the executables. The rules for the object files include dependencies on all relevant header files as well as the source file. The rules for the executables include dependencies on all relevant object files. You can ***make all*** to get both ***Server*** and ***Client***, or you can ***make Client*** or ***make Server*** to get just one.

2.10. Discussion

Finally, as always, please discuss the experience of the lab. Was it worthwhile? Did you learn anything? Can you see ways to apply what you have learned? Was it too easy? Too difficult? There is not a wrong answer, except no answer at all. If you would like to be critical, that's OK, too, but please use professional language.