EE450 Socket Programming Project, Fall 2017

Due Date : Thursday Nov 16th, 2017 11:59 PM (Midnight)

(The deadline is the same for all on-campus and DEN off-campus students)

Hard Deadline (Strictly enforced)

The objective of this assignment is to familiarize you with UNIX socket programming. This assignment is worth **10%** of your overall grade in this course. **It is an individual assignment and no collaborations are allowed.** **Any cheating will result in an automatic F in the course (not just in the assignment).**

If you have any doubts/questions, post your questions on Piazza. **You must discuss all project related issues on Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points.

Problem Statement:

In mathematics, a Taylor Series is a representation of a function as an infinite sum of terms that are calculated from the values of the function's derivatives at a single point.

The Taylor Series of $\frac{1}{1-x}$ is given by $\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n$. The Taylor Series of $log(1-x)$ is given by $log(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n}$. In both cases, the series converge when $|x| < 1$.

In this project, we sum up the terms no higher than x^6 in the Taylor Series. This approximation is valid, since the higher order terms are small enough to be ignored. Therefore, we can rewrite the Taylor Series as:

$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6$, where $|x| < 1$

$log(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5} - \frac{x^6}{6}$, where $|x| < 1$

In this project, you will implement a model of computational offloading where a single client offloads Taylor Series computation to a server which in turn distributes the load over 3 backend servers. The server facing the client then collects the results from the backend servers, performs simple computation on the results, and communicates it to the client in the required format (This is an example of how a cloud-computing service such Amazon Web Services might speed up a large computation task offloaded by the client).

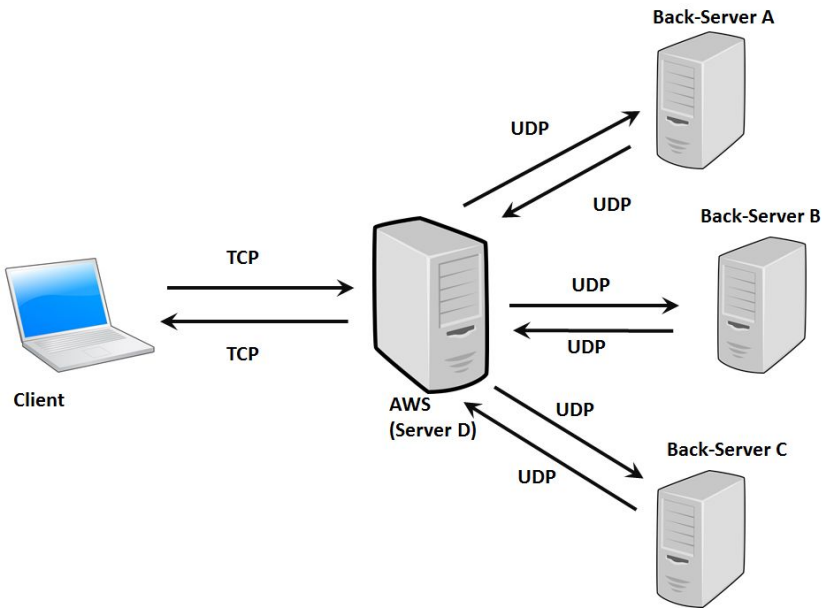

Figure 1. Illustration of the network

The server communicating with the client is called AWS (Amazon Web Server). The three backend servers are named Back-Server A, Back-Server B and Back-Server C. Back-Server A can compute x^2, Back-Server B can compute x^3, and Back-Server C can compute x^5, for any input x. The client and the AWS communicates over a TCP connection while the communication between AWS and the Back-Servers A, B & C is over a UDP connection. In this approximation, a Back-Server can be reused while computing the Taylor Series for one input. This setup is illustrated in Figure 1.

Source Code Files

Your implementation should include the source code files described below, for each component of the system.

1. AWS: You must name your code file: **aws.c** or **aws.cc** or **aws.cpp** (all small letters). Also you must call the corresponding header file (if you have one; it is not mandatory) **aws.h** (all small letters).
2. Back-Server A, B and C: You must use one of these names for this piece of code: **server#.c** or **server#.cc** or **server#.cpp** (all small letters except for **#**). Also you must call the corresponding header file (if you have one; it is not mandatory) **server#.h** (all small letters, except for **#**). The "#" character must be replaced by the server identifier (i.e. A or B or C), depending on the server it corresponds to.

Note: In case you are using one executable for all four servers (i.e. if you choose to make a "fork" based implementation), you should call the file **servers.c** or **servers.cc** or **servers.cpp**. Also you must call the corresponding header file (if you have one; it is not mandatory) **servers.h** (all small letters). *In order to create four servers in your system using one executable, you can use the fork() function inside your server's code to create 4 child processes.* You must follow this naming convention! This piece of code basically handles the server functionalities.

3. Client **:** The name of this piece of code must be **client.c** or **client.cc** or **client.cpp** (all small letters) and the header file (if you have one; it is not mandatory) must be called **client.h** (all small letters).

More Detailed Explanations:

Phase 1A: (25 points for part A and B)

All four server programs (AWS -a.k.a. server-D, Back-Server A, B, & C) boot up in this phase. While booting up, the servers **must** display a boot message on the terminal. The format of the boot message for each server is given in the onscreen messages tables at the end of the document. As the boot message indicates, each server must listen on the appropriate port information for incoming packets/connections.

Once the server programs have booted up, the client program is run. The client displays a boot message as indicated in the onscreen messages table. Note that the client code takes an input argument from the command line, that specifies the computation that is to be run and another argument for the desired input value to compute. The format for running the client code is

```
./client <function> <input>
```

where `<function>` can take a value from `{DIV, LOG}` and `<input>` will be a positive floating point number in the range of (0,1). As an example, to find the log(0.8) using Taylor's expansion up to the 6th order (as described in the problem statement) , the client should be run as follows:

```
./client LOG 0.8
```

After booting up, the client establishes a TCP connection with AWS. After successfully establishing the connection, the client first sends the `<function> <input>` to AWS. Once this is sent, the client should print a message in the format given in the table. This ends Phase 1A and we now proceed to Phase 2.

Phase 2: (75 points)

In Phase 1A, you read the function and input value and sent them to the AWS server over a TCP connection. Now in phase 2, this AWS server will send the value to the 3 back-servers. The value will be sent to their respective back-end server depending on the power to be calculated.

The communication between the AWS server and the back-servers happens over UDP. The AWS server will send the <input> to the server. The port numbers used by back-servers A, B and C are specified in table 2. Since all the servers will run on the same machine in our project, all have the same IP address (the IP address of localhost is usually 127.0.0.1).
The back-end servers are in charge to perform the operations $(x)^2, (x)^3$ *and* $(x)^5$, where n is the input value, for servers A, B and C, respectively. This implies that the backend-servers are only allowed to perform exponential operation and not any other operations such as sum, subtraction or division.

Note that the Taylor series sum for <function> that we obtained from the client will be performed in the AWS server after obtaining all the powers of the <input> value. The Taylor expansion equation used for each operation is detailed in Table 1:

Table 1. Taylor Expansion Equations for AWS Server	
Function	*Taylor Expansion for function with value x*
DIV	$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6$
LOG	$log(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5} - \frac{x^6}{6}$

Each back-server will perform their respective exponential operation and then return their result to AWS (server D) using UDP. AWS is the only server where it is allowed to perform sum, subtraction and division operations, thus this is where the Taylor Expansion formulas are employed.
Finally, for this phase, keep in mind that to calculate the x^4 and x^6 factors needed for the formula, you will have to reuse servers A and B (the one in charge of the power of 2 and power of 3 operations, respectively) by sending the previous x^2 obtained before. In other words, you

will have to request to calculate two times for server A (for x^2, x^4), two times for server B (for x^3, x^6) and one time for server C (x^5).

Phase 1B: (25 points for part A and B)

At the end of Phase 2, all backend-servers have their answers ready and these answer are located now at server AWS. Let's call the value calculate using the Taylor Expansion formula as R. This is to be sent from the AWS server to the client using TCP. Also, AWS should send a list with all the partially obtained values (the powers of x). The format on how to deliver the final solution and the powers of x are explained in the example outputs below.

The ports to be used by the clients and the servers for the exercise are specified in the following table:

Table 3. Static and Dynamic assignments for TCP and UDP ports.		
Process	**Dynamic Ports**	**Static Ports**
Backend-Server (A)	-	1 UDP, 21000+xxx (last three digits of your USC ID)
Backend-Server (B)	-	1 UDP, 22000+xxx (last three digits of your USC ID)
Backend-Server (C)	-	1 UDP, 23000+xxx (last three digits of your USC ID)
AWS (D)	-	1 UDP, 24000+xxx (last three digits of your USC ID) 1 TCP, 25000+xxx (last three digits of your USC ID)
Client	1 TCP	<Dynamic Port assignment>

NOTE: For example, if the last 3 digits of your USC ID are "319", you should use the port: **21000+319 = 21319** for the Backend-Server (A). **It is NOT going to be 21000319.**

ON SCREEN MESSAGES: **Table 4. Backend-Server A (Square Server) on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	"The Server A is up and running using UDP on port <port number>."
Upon Receiving the number:	"The Server A received input <INPUT>"
After calculating the its square:	"The Server A calculated square: <INPUT^2>"

After sending the reduction value to the AWS server (D):	“The Server A finished sending the output to AWS”

ON SCREEN MESSAGES: Table 5. Backend-Server B (Cube Server) on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	“The Server B is up and running using UDP on port <port number>.”
Upon Receiving the number:	“The Server B received input <INPUT>”
After calculating the its cube:	“The Server B calculated cube: <INPUT^3>”
After sending the reduction value to the AWS server (D):	“The Server B finished sending the output to AWS”

ON SCREEN MESSAGES: Table 6. Backend-Server C (5th power server) on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	“The Server C is up and running using UDP on port <port number>.”
Upon Receiving the number:	“The Server C received input <INPUT>”
After calculating the its 5th power:	“The Server C calculated 5th power: <INPUT^5>”
After sending the reduction value to the AWS server (D):	“The Server C finished sending the output to AWS”

ON SCREEN MESSAGES: Table 7. AWS (D) on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (only while starting):	“The AWS is up and running.”
Upon Receiving the number and function from the client:	“The AWS received <INPUT> and function=<FUNCTION> from the client using TCP over port <port number>”

After sending subset of numbers to Backend-Server (i): i is one of A, B, or C	“The AWS sent <INPUT> to Backend-Server <i>”
After receiving result from backend server i): i is one of A, B, or C	“The AWS received <RECD. VALUE> Backend-Server <i> using UDP over port <port number>”
After all values are received	“Values of powers received by AWS: [INPUT^1, INPUT^2, INPUT^3, INPUT^4, INPUT^5, INPUT^6]”
After function calculation by AWS:	“AWS calculated <FUNCTION> on <INPUT>: <CALC. VALUE>”
After sending the calculated value to the client:	“The AWS sent <CALC. VALUE> to client.”

ON SCREEN MESSAGES: **Table 8. Client on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The client is up and running.”
Upon sending the input and function to AWS	"The client sent <INPUT> and <FUNCTION> to AWS.”
After receiving the reduction: output from AWS	“According to AWS <FUNCTION> on <INPUT>: <CALC. VALUE>”

Example Output:

Backend-Server A (Square server) Terminal:

The Server A is up and running using UDP on port 21319.
The Server A received input < 0.5 >
The Server A calculated square: < 0.25 >
The Server A finished sending the output to AWS
The Server A received input < 0.25 >
The Server A calculated square: < 0.0625 >
The Server A finished sending the output to AWS

Backend-Server B (Cube server) Terminal:

The Server B is up and running using UDP on port 22319.
The Server B received input < 0.5 >
The Server B calculated cube: < 0.125 >
The Server B finished sending the output to AWS
The Server B received input < 0.25 >
The Server B calculated cube: < 0.015625 >
The Server B finished sending the output to AWS

Backend-Server C (5th Power server) Terminal:

The Server C is up and running using UDP on port 23319.
The Server C received input < 0.5 >
The Server C calculated 5th power: < 0.03125 >
The Server C finished sending the output to AWS

AWS Terminal:

The AWS is up and running.
The AWS received input < 0.5 > and function=LOG from the client using TCP over port 21319
< 0.5The AWS sent < 0.5 > to Backend-Server A
The AWS sent < 0.5 > to Backend-Server B
The AWS sent < 0.5 > to Backend-Server C
The AWS received < 0.25 > Backend-Server <A> using UDP over port < 24319 >
The AWS received < 0.125 > Backend-Server <B> using UDP over port < 24319 >
The AWS received < 0.03125 > Backend-Server <C> using UDP over port < 24319 >
The AWS sent < 0.25 > to Backend-Server A
The AWS sent < 0.25 > to Backend-Server B
The AWS received < 0.0625 > Backend-Server <A> using UDP over port < 24319 >
The AWS received < 0.015625 > Backend-Server <B> using UDP over port < 24319 >
Values of powers received by AWS: < 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625 >
AWS calculated LOG on < 0.5 >: < 0.691146 >
The AWS sent < 0.691146 > to client

Client Terminal:

The client is up and running.
The client sent < 0.5 > and LOG to AWS
According to AWS, LOG on < 0.5 >: < 0.691146 >

Assumptions:

1. You have to start the processes in this order: **backend-server (A), backend-server (B), backend-server (C), AWS (D), Client.**

2. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and mention them all in your README file.

3. You are allowed to use blocks of code from Beej's socket programming tutorial (Beej's guide to network programming) in your project. However, you need to mark the copied part in your code.

4. When you run your code, if you get the message "port already in use" or "address already in use", **please first check to see if you have a zombie process** (see following). If you do not have such zombie processes or if you still get this message after terminating all zombie processes, try changing the static UDP or TCP port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you have to change the port number, please do mention it in your README file and provide reasons for it.

5. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command: >>ps –aux | grep ee450
   Identify the zombie processes and their process number and kill them by typing at the command-line: >>kill -9 processNumber

Requirements:

1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table 1 to see which ports are statically defined and which ones are dynamically assigned. Use *getsockname()* function to retrieve the locally-bound port number wherever ports are assigned dynamically as shown below:
   /*Retrieve the locally-bound name of the specified socket and store it in the sockaddr structure*/

```
Getsock_check=getsockname(TCP_Connect_Sock,(struct sockaddr *)&my_addr,
(socklen_t *)&addrlen);
//Error checking
if (getsock_check== -1) {
        perror("getsockname");
        exit(1);
}
```

2. The host name must be hardcoded as **localhost (127.0.0.1)** in all codes.
3. Your client should terminate itself after all done. And the client can run multiple times to send requests. However, the backend servers and the AWS should keep be running and be waiting for another request until the TAs terminate them by Ctrl+C. It they terminate before that, you will lose some points for it.
4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.
5. You are not allowed to pass any parameter or value or string or character as a command-line argument except while running the client in Phase 1.
6. All the on-screen messages must conform exactly to the project description. You should not add anymore on-screen messages. If you need to do so for the debugging purposes, you must comment out all of the extra messages before you submit your project.
7. Using fork() or similar system calls are not mandatory if you do not feel comfortable using them to create concurrent processes.
8. Please do remember to close the socket and tear down the connection once you are done using that socket.

Programming platform and environment:

1. All your submitted code **MUST** work well on the provided virtual machine Ubuntu.
2. All submissions will only be graded on the provided Ubuntu. TAs won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code working well on the provided Ubuntu. "It works well on my machine" is not an excuse and we don't care.
3. Your submission MUST have a Makefile. Please follow the requirements in the following ""Submission Rules" section.

Programming languages and compilers:

You must use only C/C++ on UNIX as well as UNIX Socket programming commands and functions. Here are the pointers for Beej's Guide to C Programming and Network Programming (socket programming):

http://www.beej.us/guide/bgnet/

(If you are new to socket programming please do study this tutorial carefully as soon as possible and before starting the project)

http://www.beej.us/guide/bgc/

You can use a unix text editor like emacs to type your code and then use compilers such as g++ (for C++) and gcc (for C) that are already installed on Ubuntu to compile your code. You must use the following commands and switches to compile yourfile.c or yourfile.cpp. It will make an executable by the name of "yourfileoutput".

```
gcc -o  yourfileoutput   yourfile.c
g++ -o  yourfileoutput   yourfile.cpp
```

Do NOT forget the mandatory naming conventions mentioned before!

Also inside your code you need to include these header files in addition to any other header file you think you may need:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h>
```

Submission Rules:

1. Along with your code files, include a **README file and a Makefile**. In the README file write
   a. Your **Full Name** as given in the class list
   b. Your Student ID
   c. What you have done in the assignment
   d. What your code files are and what each one of them does. (Please do not repeat the project description, just name your code files and briefly mention what they do).
   e. The format of all the messages exchanged.
   g. Any idiosyncrasy of your project. It should say under what conditions the project fails, if any.
   h. Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.)

Submissions WITHOUT README AND Makefile WILL NOT BE GRADED.

Makefile tutorial:
https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html

About the Makefile: makefile should support following functions:

make all	Compiles **all** your files and creates executables
make serverA	**Run**s server A
make serverB	**Run**s server B
make serverC	**Run**s server C
make aws	**Run**s AWS
./client <function> <val>	Starts the client

TAs will first compile all codes using **make all**. They will then open five different terminal windows. On 4 terminals they will start servers A, B, C and AWS using commands **make serverA**, **make serverB**, **make serverC** and **make aws**.

Remember that servers should always be on once started. Client can connect again and again with different input values and function. On the fifth terminal they will start the client as **./client LOG 0.5** or **./client DIV 0.5**. Note that input value of 0.5 is just an example. TAs will check the outputs for multiple values of input. The terminals should display the messages shown in table 4, 5, 6, 7 and 8.

2. Compress all your files including the README file into a single "tar ball" and call it: **ee450_yourUSCusername_session#.tar.gz** (all small letters) e.g. my filename would be **ee450_sakulkar_session1.tar.gz**. Please make sure that your name matches the one in the class list. Here are the instructions:

   a. On your VM, go to the directory which has all your project files. Remove all executable and other unnecessary files. **Only include the required source code files, Makefile and the README file**. Now run the following commands:

   b.
   >> tar cvf **ee450_yourUSCusername_session#.tar** *
   >> gzip **ee450_yourUSCusername_session#.tar**
   Now, you will find a file named "ee450_yourUSCusername_session#.tar.gz" in the same directory. Please notice there is a star(*) at the end of first command.
   Any compressed format other than .tar.gz will NOT be graded!

3. Upload "ee450_yourUSCusername_session#.tar.gz" to the Digital Dropbox on the DEN website (DEN -> EE450 -> My Tools -> Assignments -> Socket Project). After the file is uploaded to the dropbox, you must click on the "**send"** button to actually submit it. If you do not click on "**send**", the file will not be submitted.

4. D2L will and keep a history of all your submissions. If you make multiple submission, we will grade your latest valid submission. Submission after deadline is considered as invalid.

5. D2L will send you a "Dropbox submission receipt" to confirm your submission. So please do check your emails to make sure your submission is successfully received. If you don't receive a confirmation email, try again later and contact your TA if it always fails.

6. Please take into account all kinds of possible technical issues and do expect a huge traffic on the DEN website very close to the deadline which may render your submission or even access to DEN unsuccessful.

7. Please DO NOT wait till the last 5 minutes to upload and submit because some technical issues might happen and you will miss the deadline. And a kind suggestion, if you still get some bugs one hour before the deadline, please make a submission first to make sure you will get some points for your hard work!

8. After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. If the outcome is not what you expected, try to resubmit and confirm again. We will only grade what you submitted even though it's corrupted.

9. **You have plenty of time to work on this project and submit it in time hence there is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a zero for the project.**

Grading Criteria:

Notice: We will only grade what is already done by the program instead of what will be done.
For example, the TCP connection is established and data is sent to the AWS. But result is not received by the client because the AWS got some errors.Then you will lose some points for phase 1 even though it might work well.

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, specially the communications through UDP and TCP sockets.

2. Inline comments in your code. This is important as this will help in understanding what you have done.

3. Whether your programs work as you say they would in the README file.

4. Whether your programs print out the appropriate error messages and results.

5. If your submitted codes, do not even compile, you will receive 5 out of 100 for the project.

6. If your submitted codes compile using make but when executed, produce runtime errors without performing any tasks of the project, you will receive 10 out of 100.

7. Each UDP exchange is worth 15 points and TCP exchange is worth 25 points. For example, if you only complete 2 UDP connections and half of the TCP connections, you will receive 42.5 out of 100 for the project. There are 5 UDP connections and one TCP connections in total.

8. If you forget to include the README file or Makefile in the project tar-ball that you submitted, you will lose 15 points for each missing file (plus you need to send the file to the TA in order for your project to be graded.)

9. If your code does not correctly assign the TCP or UDP port numbers (in any phase), you will lose 10 points each.

10. You will lose 5 points for each error or a task that is not done correctly.

11. The minimum grade for an on-time submitted project is 10 out of 100, assuming there are no compilation errors and the submission includes a working Makefile and a README.

12. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend about 2 months on this project and it doesn't even compile, you will receive only 5 out of 100.

13. Using fork() or similar system calls are not mandatory however if you do use fork() or similar system files in your codes to create concurrent processes (or threads) and they function correctly you will receive 10 bonus points.

14. **You must discuss all project related issues on Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points. (If you want to earn the extra credits, do remember to leave your names visible to instructors when answering questions on Piazza.)

15. The maximum points that you can receive for the project with the bonus points is 100. In other words the bonus points will only improve your grade if your grade is less than 100.

16. Your code will not be altered in any ways for grading purposes and however it will be tested with different inputs. Your designated TA runs your project as is, according to the project description and your README file and then check whether it works correctly or not. If your README is not consistent with the description, we will follow the description.

Cautionary Words:

1. Start on this project early!!!

2. In view of what is a recurring complaint near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is *the provided* ***Ubuntu (16.04)***. It is strongly recommended that students develop their code on this virtual machine. In case students wish to develop their programs on their personal machines, possibly running other operating systems, they are expected to deal with technical and incompatibility issues (on their own) to ensure that the final project compiles and runs on the requested virtual machine. If you do development on your own machine, please leave at least three days to make it work on Ubuntu. It might take much longer than you expect because of some incompatibility issues.

3. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command: **>>ps –aux | grep ee450**
   Identify the zombie processes and their process number and kill them by typing at the command-line: **>>kill -9 processnumber**

Academic Integrity:

All students are expected to write all their code on their own.

Copying code from friends is called **plagiarism** not **collaboration** and will result in an F for the entire course. Any libraries or pieces of code that you use and you did not write must be listed in your README file. All programs will be compared with automated tools to detect similarities; examples of code copying will get an F for the course. **IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR ISN'T ALLOWED ABOUT PLAGIARISM, TALK TO THE TA.** "I didn't know" is not an excuse.