EE450 Socket Programming Project, Fall 2016

Due Date : Thursday Nov 17th, 2016 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 this project you will implement a simple model of computational offloading where a single client offloads some 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 and communicates the same to the client in the required format. This is an example of how a cloud-computing service such Amazon Web Services might implement MapReduce to speed up a large computation task offloaded by the client.

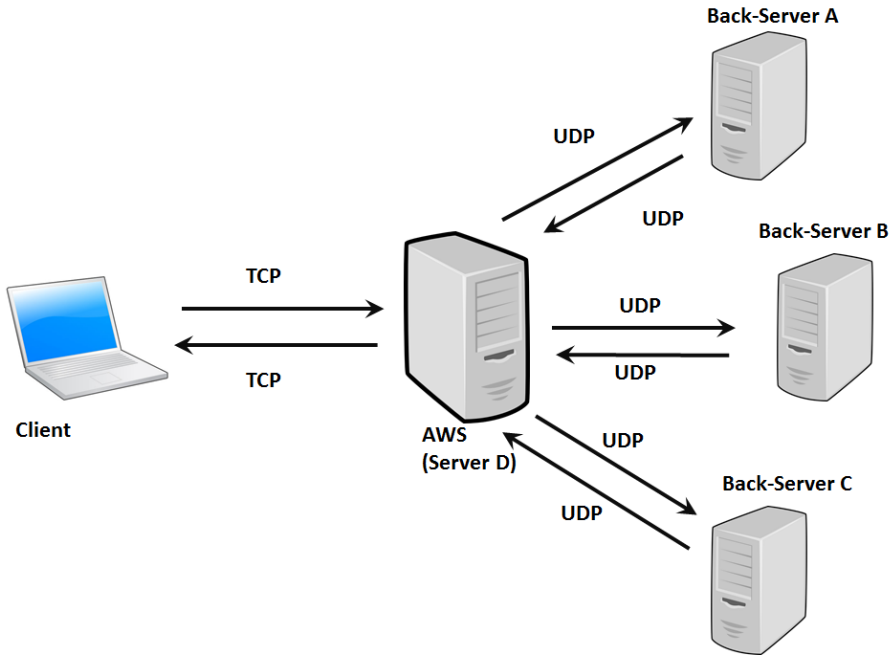

Figure 1. Illustration of the network

The server communicating with the client is called AWS (Amazon Web Server) and the three backend servers are named Back-Server A, Back-Server B and Back-Server C. 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. This setup is illustrated in Figure 1.

Input Files Used:

The files specified below will be used as **inputs** in your programs in order to **dynamically** configure the state of the system. The contents of the files should **NOT** be "**hardcoded**" in your source code, because during grading, the input files will be different, but the formats of the files will remain the same.

If you are working in an environment other than UNIX, **pay particular attention to line endings or newlines**. For this project, it is assumed that all files follow the **UNIX line ending convention**. This is particularly important while handling the input file(s). See the articles here and here for more information.

1. **nums.csv :** An ASCII file that contains a single column of integers. Each row consists of a single integer and ends with a newline. You may assume that each integer is within the range of a long signed integer type. The number of rows in the file will be a multiple of 3. This file will always reside in the same directory as the client.

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:

Phase1: (25 points)

All four server programs (AWS, 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 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. The format for running the client code is

```
./client <function_name>
```

where `<function_name>` can take a value from `{min, max, sum, sos}`. As an example, to find the sum of the all the numbers in the input file, the client should be run as follows:

```
./client sum
```

After booting up, the client establishes a TCP connection with AWS. After successfully establishing the connection, the client first sends the `<function_name>` to AWS. Once

the `<function_name>` is sent, the client should print a message in the format given int the table. The client then reads all integers from `nums.csv` and proceeds to send them to AWS over the same TCP connection. After successfully sending the integers, the client should print the number of integers sent to AWS. This ends Phase 1 and we now proceed to Phase 2.

Phase 2: (40 points)

In Phase 1, you read the numbers from the file and sent them to the AWS server over a TCP connection. Now in phase 2, this AWS server will divide the data into 3 non-overlapping components and send that to the 3 back-servers. If there are N numbers in the file, then the first N/3 numbers must be sent to back-server A, next N/3 to back-server B and the last N/3 numbers to back-server C. TAs will make sure that the number N is divisible by 3. Also the function to be performed needs to be communicated to the back-servers.

The communication between the AWS server and the back-servers happen over UDP. The AWS server will send the <function_name> along with the actual numbers. Note that the <function_name> can be MIN, MAX, SUM or SOS (sum of squares). The port numbers for 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).

Once a back-server receives the actual numbers (a total of N/3 numbers) and the function to be performed, it computes the function value. Let this value for server i as X_i. This step is also called as map in MapReduce. If the numbers received the back-server i are n_1, n_2 ……, then the Map operations it performs are as follows:

Table 1. Map operations at the back-server "i"

Function	X_i
MIN	$X_i = MIN(n_1, n_2, ...)$
MAX	$X_i = MAX(n_1, n_2, ...)$
SUM	$X_i = n_1 + n_2 + ...$
SOS (sum of squares)	$X_i = (n_1)^2 + (n_2)^2 + ...$

Each back-server calculates this value in phase 2 which needs to be sent to the AWS (server D) using UDP in phase 3.

Phase 3: (25 points)

At the end of Phase 2, all backend-servers have their answers ready. Let's call the value calculated by backend-server i as X_i. This is to be sent to the AWS server using UDP. The final answer needs to be calculated by the Frontend-server (AWS) in the reduce step and then handed over to the user.

The frontend-server (server D) looks at the type of reduction operation and calculates the final answer which we call X_{final} based on the answers it receives from the back-servers A, B and C. This step is also called as reduce in MapReduce. Now depending on the operation requested by the user we have:

Table 2. AWS Reduce operations upon getting final values from back-server	
Reduction Type	X_{final}
MIN	$X_{final} = MIN(X_A, X_B, X_C)$
MAX	$X_{final} = MAX(X_A, X_B, X_C)$
SUM	$X_{final} = X_A + X_B + X_C$
SOS	$X_{final} = X_A + X_B + X_C$

Now we need to send X_{final} to the client/user over the TCP connection.

When the client receives X_{final} it prints it on the screens and we are done!

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	-

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 on screen messages

Event	**On Screen Message (inside quotes)**
Booting Up:	“The Server A is up and running using UDP on port <port number>.”
Upon Receiving the numbers:	“The Server A has received <count> numbers”
After calculating the reduction: <reduction type> is one of MIN, MAX, SUM, or SOS	“The Server A has successfully finished the reduction <reduction type>: X_A ”
After sending the reduction value to the AWS server (D):	“The Server A has successfully finished sending the reduction value to AWS server.”

ON SCREEN MESSAGES:
Table 5. Backend-Server B on screen messages

Event	**On Screen Message (inside quotes)**
Booting Up:	“The Server B is up and running using UDP on port <port number>.”
Upon Receiving the numbers:	“The Server B has received <count> numbers”
After calculating the reduction: <reduction type> is one of MIN, MAX, SUM, or SOS	“The Server B has successfully finished the reduction <reduction type>: X_B ”
After sending the reduction value to the AWS server (D):	“The Server B has successfully finished sending the reduction value to AWS server.”

ON SCREEN MESSAGES:
Table 6. Backend-Server C on screen messages

Event	**On Screen Message (inside quotes)**

Booting Up:	"The Server C is up and running using UDP on port <port number>."
Upon Receiving the numbers:	"The Server C has received <count> numbers"
After calculating the reduction: <reduction type> is one of MIN, MAX, SUM, or SOS	"The Server C has successfully finished the reduction <reduction type>: X_C"
After sending the reduction value to the AWS server (D):	"The Server C has successfully finished sending the reduction value to AWS server."

ON SCREEN MESSAGES:
Table 7. AWS (D) on screen messages

Event	On Screen Message (inside quotes)
Booting Up:	"The AWS is up and running."
Upon Receiving the numbers from the client:	"The AWS has received <count> numbers 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 <count> numbers to Backend-Server <i>"
After receiving reduction calculated by Backend-Server (i): i is one of A, B, or C <reduction type> is one of MIN, MAX, SUM, or SOS	"The AWS received reduction result of <reduction type> from Backend-Server <i> using UDP over port <port number> and it is X_i"
After calculating the reduction: <reduction type> is one of MIN, MAX, SUM, or SOS	"The AWS has successfully finished the reduction <reduction type>: X_{final}"
After sending the reduction value to the client:	"The AWS has successfully finished sending the reduction 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."
Upong sending the reduction type:	"The client has sent the reduction type <reduction type> to AWS."
Upon sending the numbers:	"The client has sent <count> numbers to AWS"
After receiving the reduction: <reduction type> is one of MIN, MAX, SUM, or SOS	"The client has received reduction <reduction type>: X_{final}"

Example Output:

Backend-Server A Terminal:

The Server A is up and running using UDP on port 21319.
The Server A has received 30 numbers
The Server A has successfully finished the reduction SUM: 1000
The Server A has successfully finished sending the reduction value to AWS server.

Backend-Server B Terminal:

The Server B is up and running using UDP on port 22319.
The Server B has received 30 numbers
The Server B has successfully finished the reduction SUM: 1001
The Server B has successfully finished sending the reduction value to AWS server.

Backend-Server C Terminal:

The Server C is up and running using UDP on port 23319.
The Server C has received 30 numbers
The Server C has successfully finished the reduction SUM: 1002
The Server C has successfully finished sending the reduction value to AWS server.

AWS Terminal:

The AWS is up and running.
The AWS has received 90 numbers from the client using TCP over port 25319
The AWS has sent 30 numbers to Backend-Server A
The AWS has sent 30 numbers to Backend-Server B
The AWS has sent 30 numbers to Backend-Server C
The AWS received reduction result of SUM from Backend-Server A using UDP over port 24319 and it is 1000

The AWS received reduction result of SUM from Backend-Server B using UDP over port 24319 and it is 1001
The AWS received reduction result of SUM from Backend-Server C using UDP over port 24319 and it is 1002
The AWS has successfully finished the reduction SUM: 3003
The AWS has successfully finished sending the reduction value to client.

Client Terminal:
The client is up and running.
The client has sent the reduction type SUM to AWS.
The client has sent 90 numbers to AWS
The client has received reduction SUM: 3003

Assumptions:

1. It is recommended 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** (from past logins or previous runs of code that are still not terminated and hold the port busy). 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. If you have zombie processes you can kill them using unix commands: **kill -9 <pid>** or **killall <proc name>**

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. Use gethostbyname() to obtain the IP address of nunki.usc.edu or the local host however the host name must be hardcoded as nunki.usc.edu or localhost in all pieces of code.

3. You can either terminate all processes after completion of phase3 or assume that the user will terminate them at the end by pressing ctrl-C.

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 codes must run on ***nunki*** (nunki.usc.edu) and only ***nunki***. It is a SunOS machine at USC. You should all have access to ***nunki,*** if you are a USC student.

2. You are not allowed to run and test your code on any other USC Sun machines. This is a policy strictly enforced by ITS and we must abide by that.

3. No MS-Windows programs will be accepted.

4. You can easily connect to nunki if you are using an on-campus network (all the user room computers have xwin already installed and even some ssh connections already configured).

5. If you are using your own computer at home or at the office, you must download, install and run xwin on your machine to be able to connect to nunki.usc.edu and here's how:
    a. Open http://itservices.usc.edu/software/ in your web browser.
    b. Log in using your username and password (the one you use to check your USC email).
    c. Select your operating system (e.g. click on windows 8) and download the latest xwin.
    d. Install it on your computer.
    e. Then check the following webpage: http://itservices.usc.edu/unix/xservers/xwin32/ for more information as to how to connect to USC machines.

6. Please also check this website for all the info regarding "getting started" or "getting connected to USC machines in various ways" if you are new to USC: http://www.usc.edu/its/

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/

Once you run xwin and open an ssh connection to nunki.usc.edu, 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 nunki 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 -lsocket -lnsl -lresolv
g++ -o yourfileoutput yourfile.cpp -lsocket -lnsl -lresolv
```

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 (10 points):

1. Along with your code files, include a **README file & 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. What the TA should do to run your programs. (Any specific order of events should be mentioned.)
   f. 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.)

   About the Makefile: makefile should support following functions:

make all	Compiles all your files and creates executables
make serverA	Runs server A
make serverB	Runs server B
make serverC	Runs server C
make aws	Runs AWS
./client <function>	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. On the fifth terminal they will start the client as ./client sum or ./client max, ./client min or ./client sos. The terminals should display the messages shown in table 4, 5, 6, 7 and 8.

Submissions WITHOUT README AND Makefiles WILL NOT BE GRADED.

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 file name 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 nunki.usc.edu, go to the directory which has all your project files. Remove all executable and other unnecessary files. Only include the required source code files and the README file. Now run the following commands:

   b. **you@nunki>>** tar cvf **ee450_yourUSCusername_session#.tar** * - Now, you will find a file named "ee450_yourUSCusername_session#.tar" in the same directory.

   c. **you@nunki>>** gzip **ee450_yourUSCusername_session#.tar** – Now, you will find a file named "ee450_yourUSCusername_session#.tar.gz" in the same directory.

   d. Transfer this file from your directory on nunki.usc.edu to your local machine. You need to use an FTP program such as CoreFtp to do so. (The FTP programs are available at http://itservices.usc.edu/software/ and you can download and install them on your windows machine.)

3. Upload "ee450_yourUSCusername_session#.tar.gz" to the Digital Dropbox (available under Tools) on the DEN website. 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. Right after submitting the project, send a one-line email to your designated TA (NOT all TAs) informing him or her that you have submitted the project to the Digital Dropbox. Please do NOT forget to email the TA or your project submission will be considered late and will automatically receive a zero.

5. You will receive a confirmation email from the TA to inform you whether your project is received successfully, so please do check your emails well before the deadline to make sure your attempt at submission is successful.

6. You must allow at least 12 hours before the deadline to submit your project and receive the confirmation email from the TA.

7. By the announced deadline all Students must have already successfully submitted their projects and received a confirmation email from the TA.

8. 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.

9. Please do not wait till the last 5 minutes to upload and submit your project because you will not have enough time to email the TA and receive a confirmation email before the deadline.

10. Sometimes the first attempt at submission does not work and the TA will respond to your email and asks you to resubmit, so you must allow enough time (12 hours at least) before the deadline to resolve all such issues.

11. **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:

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. If your codes compile but when executed only perform phase1 correctly, you will receive 35 out of 100.

8. If your code compiles and performs all tasks up to the end of 2 phases correctly and error-free, and your README file conforms to the requirements mentioned before, you will receive 75 out of 100.

9. If your code compiles and performs all tasks of all 3 phases correctly and error-free, and your README file and Makefile conforms to the requirements mentioned before, you will receive 100 out of 100.

10. 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.)

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

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

13. 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.

14. 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.

15. 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.

16. **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.)

17. 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.

18. Your code will not be altered in any ways for grading purposes and however it will be tested with different input files. 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.

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 *nunki.usc.edu*. It is strongly recommended that students develop their code on nunki. 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 nunki.

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 even from your past logins to nunki, try this command: ps –aux | grep <your_username>

4. Identify the zombie processes and their process number and kill them by typing at the command-line:
   Kill -9 processnumber
5. There is a cap on the number of concurrent processes that you are allowed to run on nunki. If you forget to terminate the zombie processes, they accumulate and exceed the cap and you will receive a warning email from ITS. Please make sure you terminate all such processes before you exit nunki.

6. Please do remember to terminate all zombie or background processes, otherwise they hold the assigned port numbers and sockets busy and we will not be able to run your code in our account on nunki when we grade your project.

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.