#### COEN 146: Computer Networks

**Lab 1: Basic Linux and C programming skills**

**Objectives**

##### To review Linux/Unix commands

##### To write sample C programs that demonstrate the use of threads and files

##### 

##### **Guidelines**

##### For all COEN 146L you need to develop a good knowledge of Linux/Unix commands and to write C in a Linux development environment. You are highly encouraged to use command line tools in developing, compiling, and running your programs. For some labs, you will also demonstrate some of the networking concepts in Python. This Lab is designed to get you familiar with the basic programming skills that you will need for the upcoming labs.

##### Please pay attention to your coding style and good programming practices, if your program is not worth documenting, it probably isn't worth running.[[1]](#footnote-1) Please follow the GNU coding standards available on: <https://www.gnu.org/prep/standards/html_node/Writing-C.html>.

##### **Unix/ Linux**

##### Linux is a Unix-like operating system, following the design principles of Unix monolithic kernel in process control, cpu scheduling, memory management, file systems, networking, and access to the peripherals. Please read details on <https://en.wikipedia.org/wiki/Linux>. Unix/ Linux has a culture of distinctive art and powerful design philosophy since its inception in 1969. Software technologies come and go but Unix/ Linux remained dominant and continued to evolve on a wide variety of machines ranging from supercomputers and PCs to handheld devices and embedded networking hardware.  C language is ubiquitous and a central technology of Unix/ Linux. It is very hard to imagine developing applications at the core system level without C. The POSIX, Portable Operating System Standard, the Unix API (Application Programming Interface) is used to write truly portable software that can run across a heterogeneous mix of computers. TCP/IP (which you will learn in this class) and Unix represent the core technologies of the Internet. For more details, see [[2]](#footnote-2).

##### 

##### It is recommended that you install Linux on your, either as a bare operating system or as a virtual machine. For details, please check https://www.linux.com. If you are using Mac OS, you can use Linux commands in a terminal window, including vi and gcc compiler.

**Command-line**

##### The use of command-line programs is a tradition in Unix\Linux. Commands are executable programs that can run with variety of arguments (options). Command-line options are single letters preceded by a single hyphen, including:

##### -a: all, -b: buffer, -c: command, -d: debug, -e: execute, -f: file, -l: list, -o: output, -u: user

Some of the basic commands are:

* ls: lists all files and directories (try with options: -a, -al)
* cat: displays file content (try cat file1 file2 > file3)
* mv: moves a file to a new location (try mv file1 file2)
* rm: deletes a file
* cp: copy file
* man: gives help information on a command
* history: gives a list of past commands
* clear: clear the terminal
* mkdir: creates a new directory
* rmdir: deletes a directory
* echo: writes arguments to the standard output (try echo ‘Hello World’ > myfile)
* df: shows disk usage
* apt -get: install and update packages
* mail -s ‘subject’ -c ‘cc-address’ -b ‘bcc-address’ ‘to-address’ < filename: sends email with attachment
* chown/ chmod: change ownership/ permission of file or directory
* date: show the current date and time
* ps: displays active processes
* kill: kills process
* sh: bourne shell – command interpreter (good to learn about shell programming)
* grep: searches for pattern in files
* Ctrl+c: halts current command
* Ctrl+z: stops current command and resumes with foreground
* Ctrl+d (exit): logout of current session

##### In general, you will most probably use the commands - ls, more, mv, rm, mkdir, rmdir, cd, cp, chmod, who, ps, kill, ctrl+c, cmp, grep, cat, and man – more often. The man helps you to learn about a specific command and its use in Linux. For example, $man cat displays the meaning and usage of cat.

**System calls**

##### System calls are often called kernel calls. They are c libraries that execute at the kernel level to allow users to interact with the operating system for services that include:

* Process creation and management (e.g. fork(), exec(), wait(), exit())
* File management (e.g. open(), read(), write(), close())
* Communication (e.g. pipe(), shmget(), mmap())
* Networking (e.g. socket(), bind(), connect(), listen(), sendto(), recvfrom())

**Multi-threading**

##### Skills in multithread programming is required for developing client-server applications, as a mechanism to create parallelism for a better and yet expedited performance. For example, a server spawns a separate thread for every client connection to look after the specific client requests independently. A thread is a single sequence stream within in a process, and it is often referred to as a lightweight process. Threads operate faster than processes during their creation and termination, context switching, and communication. Threads are not independent like processes and they share with other threads their code and data, open file descriptors. Threads maintain their own program counters, registers, and stack.

##### The following library function are used for setting up threads.

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, pthread\_attr\_t \*attr,

##### void \*(\*start\_routine) (void \*arg), void \*arg);

##### **Copying files**

Problem: Write a C program to copy binary/ text files simultaneously.

Analysis:

* Input: pass file names as arguments to main(), so your main function needs to be defined as follows:

int main(int argc, char \* argv[])

Your files: src.dat and dest.dat files.

* File reading can be accomplished by using either:
  + Functions: fopen, fwrite, and fread for binary files or fprintf and fscanf for text files
    - FILE \*fopen(const char \*filename, const char \*mode)
    - fwrite( ptr, int size, int n, FILE \*fp ); or fprintf() (for text files)
    - fread( ptr, int size, int n, FILE \*fp ); or fscanf() (for text files)
    - fclose(ptr);

e.g.

FILE \*fp;

fp = fopen(“src.dat”,"r”);

fp = fopen(“dest.dat","w”);

fwrite(&buf,sizeof(buf),1,fp);

fread(&buf,sizeof(buf),1,fp);

fclose(fp);

OR

* + System calls: open, read, write
    - int open (const char\* Path, int flags [, int mode ]);
    - size\_t read (int fd, void\* buf, size\_t cnt);
    - size\_t write (int fd, void\* buf, size\_t cnt);

e.g.:

int fd = open("foo.txt", O\_RDWR);

           int nw = write(fd, buf, strlen(buf));

int nr = read(fd, buf, 40);

close (fd);

You need to include the following libraries:

* + - #include<sys/types.h>
    - #include<sys/stat.h>
    - #include <fcntl.h>

You may create files of random data with different sizes/ bytes. You may use “cat” and “head” commands ( i.e. $cat /dev/random | head -c <bytecount>). /dev/random are special files that serve as pseudorandom number generator. “cat” is used to display the content and “head” is used to display the specified number of lines. So the result of “cat” is sent to the upstream end of PIPE and “head” receives these results and redirects the content of the specified bytes to a file.

$cat /dev/random | head -c 100000 > src1.dat 🡪 creates a file with 100KB

$cat /dev/random | head -c 1000000 > src2.dat 🡪 creates a file with 1MB

Check the size of the files with the command “ls -la”

1. [15%] Write your C program to copy files (binary and text) using functions, compile, debug, run, and test
2. [15%] Write your C program to copy files (binary and text) using system calls, compile, debug, run, and test
3. [15%] Calculate the time taken to copy files for both step 1 and 2. You may use clock()function (in time.hlibrary). Call the clock function at the beginning and end of the code to measure the time, subtract the values, and then divide by CLOCKS\_PER\_SEC (the number of clock ticks per second) to get processor time. You may use the following code snippet:

#include <time.h>

clock\_t start, end;

**double** cpu\_time\_used;

start = clock();

... /\* Time consuming process. \*/

end = clock();

cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;

1. [15%] Write your C program to copy multiple files (say 10 files) by creating threads, each of which will be responsible of copying one file. Use good style practices, compile, debug, run, and test for large sizes of matrices.

##### **Circuit switching and packet switching**

Problem: Write a C program that implements quantitative comparisons between circuit switching and packet switching according to the following description

Analysis:

Variables

* The bandwidth of a network link is denoted by int linkBandwidth;
* The bandwidth required for a given user is denoted by int userBandwidth;
* The number of circuit switching users is denoted by int nCSusers;
* The number of packet switching users is denoted by int nPSusers;
* The percentage of time a packet switching user needs to transmit is denoted by double tPSuser;
* The probability that a given (*specific*) packet switching user is busy transmitting is denoted by double pPSusersBusy;
* The probability that one (*specific*) packet switching user is not busy transmitting is denoted by double pPSusersNotBusy;

Scenarios: Consider the following two scenarios:

1. A circuit-switching scenario in which nCSusers users, each requiring a bandwidth of userBandwidth Mbps, must share a link of capacity linkBandwidth Mbps.
2. A packet-switching scenario with nPSusers users sharing a linkBandwidth Mbps link, where each user again requires userBandwidth Mbps when transmitting, but only needs to transmit at a percentage of tPSuser.

Computations:

1. [15%] Circuit switching scenario
   1. The number of circuit-switched users that can be supported is computed as follows:

nCSusers = linkBandwidth/ userBandwidth;

1. [25%] Packet switching scenario
   1. The probability that a given (*specific*) user is busy transmitting is computed as:

pPSusers = tPSusers;

* 1. The probability that one specific other user is not busy is computed as:

pPSusersNotBusy = 1 - pPSusers;

* 1. The probability that *all* of the other specific other users are not busy is computed as:

(1 – pPSusers) *nPSusers -1*;

* 1. The probability that one specific user is transmitting and the remaining users are not transmitting is computed as:

*pPSusers1* \* pPSusersNotBusy *nPSusers -1*

* 1. The probability that exactly one (any one) of the nPSusers users is busy is  pPSusers times the probability that a given specific user is transmitting and the remaining users are not transmitting, i.e.:

nPSusers \*( *pPSusers1* \* pPSusersNotBusy *nPSusers -1*)

* 1. The probability that 10 specific users of nPSusers are transmitting and the others are idle is computed as:

*pPSusers10* \* pPSusersNotBusy *nPSusers -10*

* 1. The probability that any 10 users of nPSusers are transmitting and the others are idle is computed as:

(nPSusers, 10) \* *pPSusers10* \* pPSusersNotBusy *nPSusers -10*, where

(nPSusers, 10) = nPSusers!/(10!\*(nPSusers – 10)!) co-efficient of binomial distribution

* 1. The probability that more than 10 users of nPSusers are transmitting and the others are idle is computed as:

Σ *i=11..nPSusers* (nPSusers, i) \* *pPSusersi* \* pPSusersNotBusy *nPSusers -i*

Demonstrate steps 6 and 7 for your program to the TA with the following inputs:

linkBandwidth = 200 Mbps

userBandwidth = 20 Mbps

tPSuser = 0.10

nPSusers = 19

**Requirements to complete the lab**

1. Demo to the TA correct execution of your programs [recall: a successfull demo is 30% of the grade]
2. Submit the source code for all your programs as .c file(s) and upload to Camino.

Please start each program/ text with a descriptive block that includes minimally the following information:

# Name: <your name>

# Date: <date> (the day you have lab)

# Title: Lab1 – task

# Description: This program computes … <you should

# complete an appropriate description here.>

1. Jonathan Nagler, "Coding Style and Good Computing Practices”, *PS: Political Science and Politics*, Volume 28, Issue 3, 1995, pp. 488-492. [↑](#footnote-ref-1)
2. Eric S. Raymond, *The Art of Unix Programming*, Pearson, 2004 [↑](#footnote-ref-2)