Eng 67: Lab 1

In general, each lab in Eng 67 has two components that must be completed each week:

1) A set of programming **TASKS** that must be completed.
2) A set of **TOPICS** related to the NEXT lab that must be studied outside lab.

The course pre-requisites assume that you already know how to program in C; the TOPICS for the first lab have therefore already been completed and you are ready to proceed with the tasks (… well maybe).

Unfortunately, each student will have a different level of exposure to programming, will recall different topics with more or less clarity, and will have more or less exposure to Linux. With that in mind, the first week and a half will be spent bringing everyone up to the same level and, hopefully, giving each student the opportunity to delve a little more deeply into some of the concepts.

Lab 1 Learning Objectives:

1. Establish/refresh a basic C programming skills, including knowledge of:
- Pointers and linked structures
- Structures (struct)
- Type Definitions (typedef)
- Dynamic Memory Allocation (malloc)
- Casting
- Anonymity (void)
- Command Line Arguments
- Passing *functions* as arguments
- Interfaces and separate compilation (.c, .o, and .h files)

2. Ensure that you have a basic familiarity with a variety of industry standard systems programming tools available in Linux – **ssh, emacs, make, and gcc.**

3. Ensure you are able to write a *re-usable program module* that can be developed, compiled, and tested independently.

Background. This course will explore parallel programming using a high-performance scalable architecture called a ***Blade Server***. These servers are the basis for cloud computing systems and are built by coupling together a large number of rack-mounted ***blade chassis***, each of which holds some number of identical ***blades.*** In this course we will be using a machine – named "hub" – depicted below, that contains 16 blades. Each of the vertical lines shows a handle used to pull a single blade out of the chassis:

Each blade in the machine contains two, quad-core Intel processors with an associated hard drive running Linux. In our machine, there are 16 blades – named ***hub1, hub2***, …., ***hub16*** – a total of 128 processing cores. Each blade is connected to a gigabit Ethernet switch, within the chassis, that allows all processors to communicate with each other (and with other chassis in a larger machine). In this manner the entire 128-core machine can be used as a single computing device. This course is about how to program such machines so that a single program can use **ALL** of the cores at the same time to solve a single problem.

Basic Modular Programming Ideas: In order to develop large systems programs, it is important that the system can be *decomposed into modules* that can be developed, compiled, tested, and maintained separately by *different* programmers and then reused for subsequent projects.

For example, we might construct a ***queue*** module (e.g. in a source file ***queue.c***) that can be reused to represent a queue of people in a store, or a queue of processes in an operating system, or a queue of items on a shopping list. The programs that use the queue are all different, but the implementation of the queue, once developed, should remain the same.

Generally, we describe the *interface* to a module through a *header file* associated with the module (e.g. ***queue.h)***. The crucial point is that in order to use the compiled queue implementation (i.e. contained in the object file ***queue.o***), *you do not have to understand how the queue is implemented* – only the input and output behavior of the functions it provides. This input / output behavior is described by the *function prototypes* listed in the interface.

TASKS

1. **SSH:** From either the M210 lab or your personal computer, log into one of the *remote* hub blades using the Putty **ssh** tool. [HINT use any blade as a machine name e.g "hub1" or "hub13"].

2. **LINUX:** If you are unfamiliar with Linux, use the web and manual pages (i.e. the command "**man**") to gain a basic understanding of the hierarchical file structure. Ensure you understand what the following commands do on the remote machine (here I separate each command is separated by a comma):

   man, ls, more, cat, pwd, mkdir, rmdir, cd, touch, cp, mv, rm, grep, wc, chmod, make
   ls –a, ls –l, cd .., cd ~, make -n
   fg, bg, top, ps, ps –ale
   Control-p, Control-n, !!

   Also be sure you understand the following operators:

   ; (sequential operator), **|** (pipe or *concurrent* operator), **>** (redirection operator), **>>** (append operator)

3. **EMACS:** Inside the console window on the remote system, run the **emacs** command to start the emacs editor. If you are not familiar with emacs, take the tutorial and learn the basic operations – make sure that you understand the core operations for editing programs especially the following key sequences (Note: C means "*Hold down the Control key with a key press*", M means "*Hit Esc before a key press*"):

   Useful Starters:

   C-h t --- execute the emacs **T**utorial
   C-x, C-c – **C**ome out of emacs (exit)
   C-g -- **G**et out of what I am doing and back to insert mode

   Moving Around:

   Just type text to add it to the exit buffer.
   C-f , C-b --- move cursor **F**orward / **B**ackward a character
   C-p, C-n --- move cursor to **P**revious / **N**ext line
   C-a, C-e --- move cursor **B**eginning / **E**nd of line (not C-b already used)
   M-f, M-b --- move cursor **F**orward / **B**ackward a word
   M-<, M-> --- point to **T**op / **B**ottom of file
   C-v, M-v --- **D**own /**U**p page

   Modifying and Finding Text

   Use the delete key to delete characters
   C-k, C-y – **K**ill a line / **Y**ank it back

C-s --- **S**earch file for a string
M-x query-replace --- optionally replace a string multiple times

File operations

C-x C-f --- **F**ind and load a file into the current edit buffer window
C-x C-w --- **W**rite buffer window to file
C-x, C-s – **S**ave the current buffer to its file

Buffer operations

Allow you to kill text in one buffer and yank it back in another
C-x 2--- split into **2** buffers
C-x o --- go to the **O**ther buffer
C-x 1 --- go to **1** buffer

Note: Although there are many alternative short cuts to these commands, often they do not work on non-standard keyboards, non-conforming emacs implementations, or over networks to non-standard shells. I recommend learning the basic key-strokes as are easy to remember with the notations I have provided but usually work when everything else fails.

4. **GCC/MAKE/INTERFACE USE:** Create a directory to hold your first lab – **Lab1**. Copy (using ***cp***) the following files contained in:

/thayerfs/courses/software/engs067/lab1

into your directory:

Makefile –--- a make file for using the queue module
queue.h ---– the interface to the queue implementation
queue.o --- the compiled implementation of a queue
shop.c – an empty shopping list program

Take a look at the make file and ensure that you understand how it works by reading the manual pages for make and/or using the web. You might experiment with "make –n", "make", "touch shop.c" and see what happens.

Write a program **shop.c** that represents a ***shopping list*** using the queue module. Each entry on the shopping list is composed of an item to purchase (string) and the number of the items to purchase (integer). Your program should open a queue, repeatedly accept entries from the keyboard, add them to the queue. When all the items have been entered, print out the shopping list, remove all the entries from the queue, close the queue, and exit. If a command line argument ***n*** is present, then it specifies the number of lines to place on the shopping list before printing the list. Your program must compile with NO WARNINGS OR ERRORS.

5. **MODULE DESIGN:** Write a simple queue module (ie. queue.c) that provides only the following functions described in the interface: ***qopen, qput, qget, qapply, and qclose.***

6. **[ENG115: GRADUATE STUDENTS ONLY: Implement the full interface and exercise it]**

TOPICS

Research the following concepts:

- Processes – what is an operating system process ?
- Thread – what is a thread and how is it different from a process?
- POSIX threads -- how do you:
    i. Create a thread
    ii. Wait for all threads to terminate
    iii. Pass an argument to a thread
    iv. Pass data from one thread to another
- What is mutual exclusion?
- What is a mutex and what operations are provided on them?
- What is a semaphore and what operations are provided on them?
- What is deadlock and how can it be avoided?