Operating System Development Primer

GWU CSCI 3411 - Fall 2017 - Lab 1
Prepared by James Taylor

August 31, 2017

Topics

You will need to become proficient in each of these topics very quickly. By week three, your performance in lab, your development in this course, and your ultimate grade will be dependent upon basic familiarity and general proficiency with the following topics. The purpose of Lab 1 is to accelerate you as fast as possible with an introduction to the tools and fundamentals required for you to survive and thrive in this course. A number of references have been provided in each section, so it is in your best interest to review these references as soon as possible and to practice these topics on your own time before these requirements become critical to your success.

1. Basic Introduction to Linux
2. Software Configuration Management (SCM) using `git`
3. Working with Makefiles
4. Fundamental C Review

Conventions

Code or commands presented in this and future lab documents will be presented in `console` typeface. Text in console typeface can generally be used in the appropriate software without significant changes with some exceptions. Unique exceptions will be noted when appropriate; however, one general exception will be the usage of paired angle braces with text in between the braces, *i.e.* `<anexample>`. Text in console typeface between angled braces cannot be copied directly with one exception in C code, *e.g.* `#include <stdlib.h>` (should be copied directly but all other cases follow the general replacement convention). Angled braces will denote that the information between the braces should be replaced with an appropriate term that fits the description between the braces. For example, if `<url>` appears in a document, you are expected to replace `<url>` with an appropriate URL (*Note*: Drop the angled braces when replacing with the appropriate information).

Homework

Lab will generally not demand additional work outside of the lab with two exceptions. These exceptions are:

1. You are required to work outside of this lab, *i.e.* Lab 1, to prepare yourself and the tools that you will rely on throughout the semester.
2. If you do not satisfactorily complete lab work within the allotted lab time, you are heavily encouraged to complete the lab work before the subsequent lab session.

By next week (Lab 2), you must complete the following:

1. Prepare an Ubuntu 14.04 32-bit Linux system for use throughout the semester. **You must bring your Ubuntu system to lab each week**. You can install and use a different version of Ubuntu, but we are grading using 14.04, so we advise you to ensure that your programs work in that environment. You may use virtualization software such as VMWare or VirtualBox to host Linux on your own computer or you can create a dual boot partition. Each choice has its ups and downs, so we encourage you to discuss which option suits your needs on Piazza before you attempt to set up an Ubuntu system.
2. For this class, create a directory on your Ubuntu system for all of your future course and lab work.
3. On your Ubuntu system, install `gcc`, `make`, and `git`, refer to Section 1.4 for how to install software on Ubuntu.
4. Accept the github classroom invitation for Lab 1. The invitation to the Lab 1 repository is posted on Piazza along with this document.
5. In your dedicated course/lab work directory, clone the LAB1-OS-FA17 repository onto your Ubuntu system.
6. Change to the lab1 directory that is created by clone.
7. Compile your program using `make`. `make` will invoke `gcc`.
8. Execute `./hello` and verify that "Hello World!" is printed to the console.
9. Familiarize yourself with one of the recommended text editors. At the minimum, you should be able to open and close the program as well as open, edit, and save text documents.
10. Prepare to be able to answer all questions in this document to the best of your knowledge. You will not hand in answers; however, you will be randomly asked to answer questions from this document in Lab 2. You are encouraged to practice C to develop practical knowledge rather than a conceptual knowledge of C. In Lab 2, you will begin writing C on your own.

1 Basic Linux

We will be working in the Linux environment throughout this course. You will need to develop a working knowledge of Linux in order to succeed. This section is intended to inform you of the most basic Linux concepts and operations that you must become proficient with.

1.1 Linux Console

We will work from the Linux console (*Note:* The terms "console", "terminal", and "command line" may be used interchangably throughout this course). While many operations can be accomplished using a GUI, everything on the Linux system can be controlled through the console interface. For many systems, there is no GUI accessible, so a working knowledge of the console interface is a critical skill for low-level system interaction and development (It is also a critical skill for remote interaction with Linux based systems). The earlier you adopt and practice working with the console, the faster you will develop in all aspects of this course and the more successful you will be overall. You will not be able to successfully complete the course work if you are dependent on graphical interfaces.

1.2 Questions

1. What is a path?
2. What do the "`.`", "`..`", and "`/`" denote in a Linux path? What do "`/`" and "`~/`" when used as leading tokens denote in a Linux path?
3. What are wildcards?
4. What does the Linux console command `man` do?
5. What does the Linux console command `ls` do?

6. What does the Linux console command `cd` do?
7. What does the Linux console command `mkdir` do?
8. What does the Linux console command `rm` do? Why should you be extremely careful using the `rm` command with wildcards?
9. What does the Linux console command `cp` do?
10. What does the Linux console command `mv` do? Should you have similar concerns with `mv` as `rm`?
11. What does the Linux console command `gcc` do?
12. What does the Linux console command `grep` do?
13. What does the Linux console command `cat` do?
14. What does the Linux console command `sudo` do?
15. What do either of the Linux console commands `vi` or `emacs` do?

1.3 Text Editors

There are a number of text editors available for Linux. Built into the Linux system is a very basic text editor called `nano`. While `nano` is a viable tool, it is very primitive and there are at least two highly regarded and full featured text editors that are available, `vi` and `emacs`. You will need to become familiar with one or the other. These two programs have comparable features, but they use very different interfaces, so knowledge of one does not immediately transfer to the other. Gabe prefers `emacs` and I prefer `vi`. In lab, I will use `vi` or its sibling `vim`.

1.4 Installing software on Ubuntu

Ubuntu uses the Advanced Packaging Tool (`apt`) to install software. To install a package, use the `apt-get` command. The syntax for installing a package is as follows:

```
sudo apt-get install <packagename>
```

To get help information on `apt-get`, refer to the man pages on `apt` or `apt-get` or use the following command:

```
apt-get --help
```

1.4.1 To install `gcc` and `make` on Ubuntu

```
sudo apt-get install build-essential
```

1.4.2 To install `git` on Ubuntu

```
sudo apt-get install git
```

1.4.3 To install `emacs` and `vim` on Ubuntu

```
sudo apt-get install emacs vim
```

1.5 References

- Linux - http://www.linuxdevcenter.com/excerpt/LinuxPG_quickref/linux.pdf
- Linux - https://files.fosswire.com/2007/08/fwunixref.pdf
- emacs - https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf
- vi - http://web.mit.edu/merolish/Public/vi-ref.pdf

2 Using git

Source Configuration Management (SCM) encompasses the tools and practice of maintaining a comprehensive history of the changes and different configurations within a single codebase. There are a number of different SCM tools currently available with varying features such as git, Mercurial, and SVN. The principal benefit of SCM is to allow programmers to protect themselves from breaking a system too severely between updates; however, there are a number of additional benefits that have evolved through different SCM applications.

In this course, we will use git for SCM. There is particular git terminology that you will need to familiarize yourself with for us to communicate and there are a number of commands that you will need to become proficient with in order to use git. This section will list the most fundamental terminology and commands that you will need to understand git.

Note

This section is primarily focused on single user git. Later in the term, we will revisit git with a focus on multiple users.

2.1 git Terminology

- Repository
- Remote
- Origin
- Clone
- Branch
- Add
- Commit
- Checkout
- Merge
- Pull
- Push
- Stash

2.2 `git` Commands

- `git --help`
- `git <command> --help`
- `git init`
- `git clone <url>`
- `git remote -v`
- `git branch`
- `git branch <branchname>`
- `git add <filename>`
- `git add <pathspec>`
- `git commit`
- `git commit -m "<commitmessage>"`
- `git checkout <filename>`
- `git checkout <pathspec>`
- `git checkout -b <newbranch>`
- `git pull`
- `git pull <remote> <branch>`
- `git push`
- `git push <remote> <branch>`
- `git stash`

2.3 Basic `git` Workflows

2.3.1 Cloning an existing repository from a remote `origin`

```
git clone <url>
```

2.3.2 Update a file and commit to the current branch of the local repository

```
git add <filename>
git commit -m "<commitmessage>"
```

2.3.3 Update a file and commit to the current branch of both local and `origin` repositories

```
git add <filename>
git commit -m "<commitmessage>"
git push
```

2.3.4 Pull (and merge) remote changes with your local changes from the `origin`'s `master` branch

```
git pull origin master
```

2.3.5 Push the current state of the local repository's `master` branch to the `origin`'s remote repository `master` branch

```
git push origin master
```

2.4 Questions

1. Which `git` command creates a repository on your local machine?
2. Which `git` command copies a repository from a remote git server onto your local machine and creates a reference to that remote as the `origin`?
3. If you commit one or more files, what repositories are automatically updated?
4. What steps are necessary to ensure that a local repository is synchronized with a remote repository?
5. What steps are necessary to ensure that a remote repository is synchronized with a local repository?
6. When you checkout a file from a repository, what repository does the file come from?

2.5 References

- `git` cheat sheet - https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf
- `git` saving changes - https://www.atlassian.com/git/tutorials/saving-changes
- `git` undoing changes - https://www.atlassian.com/git/tutorials/undoing-changes
- `git` reference - https://git-scm.com/docs

3 Using `make` and Makefiles

Makefiles facilitate consistent and reliable compilation of complex projects that are dependent on a number of source files and libraries. While a compiler can be invoked directly from the command line, compilation may require the input of a large number of paths, references, or multiple commands. A `Makefile` allows a programmer to manage and maintain the complex set of dependencies for a program in a script and to invoke the potentially complex, multiphase build process through the one word command `make`.

You will need at least a basic understanding of `Makefile` syntax to succeed in this course. The syntax used in a `Makefile` can be quite sophisticated and variable, and there are a number of approaches that may be used to achieve the same goals. We will begin exercising with a very basic `Makefile`; however, we will quickly transition to a much more complex `Makefile` to build XV6 as even a basic operating system is sufficiently complex. It will be difficult to anticipate what you will need to understand in terms of `Makefile` syntax in advance, so be prepared to spend some time analyzing the XV6 `Makefile` when XV6 is introduced in lab.

3.1 `make` Terminology

- Building
- Cleaning

3.2 `make` Commands

`make` commands are subject to definitions within the `Makefile`. The `make` command alone will build the project, but parameters that may follow the `make` command are entirely dependent on what is defined in the `Makefile`. Any parameters used in the following list are generally standardized approaches, but their implementations are subject to the design of the `Makefile`.

- `make`
- `make clean`

3.3 References

- `make` manual - https://www.gnu.org/software/make/manual/make.pdf
- Debugging `make` - http://www.oreilly.com/openbook/make3/book/ch12.pdf

4 Fundamental C Review

Operating systems integrate applications, hardware, and users, so operating system performance is critical to effective use of computer systems. Operating systems must be fast and must interface with hardware at a very low-level, so operating systems require a language that compiles into machine code. While we might develop operating systems in a very low-level language like assembly, our abilities to develop, port, and maintain assembly code for multiple platforms and to understand the system would be challenging and costly. C code can be compiled into compatible machine code given an appropriate set of tools for a given architecture, C code is abstract and flexible enough that it can be generalized for use on a number of different platforms, and C code is expressive enough that humans can understand and write it with relative ease. For these reasons, C is an essential tool in operating system development.

The purpose of this section is two-fold: to remind you of concepts that you have learned previously and to encourage consideration of technical approaches in C that you may never have encountered. In a few weeks, you will begin working in an existing C codebase that uses a wide variety of approaches. Each of the questions in this section is designed to prepare you for working with that codebase.

4.1 Questions

1. What tools are necessary for C code to become a compatible program on a target platform? Who might develop these tools, what does that development require, and how does a programmer get the information and tools necessary to develop a C compiler or C code for a given system? (Consider what you have learned in Computer Architecture and C Programming.)
2. A C compiler optimizes C code according to its own rules. What effects might compiler optimizations have on code you write?
3. How are comments annotated in C code? Why are comments an important part of programming? Who is responsible for writing comments? Describe any standards or guidelines for commenting code.
4. What is a coding standard? Why are coding standards important? Who determines coding standards for a given project?
5. How many bits are represented in a standard byte? How many hexidecimal characters are required to represent one standard byte? How do you convert hexidecimal to binary? How do you convert binary and hexidecimal to base 10.
6. Every C program has an entry point. How is the entry point defined in a *user space* C program? Operating Systems are typically written in C, but the system is not running when the OS kernel starts, so what is the entry point for the OS in *kernel space*? (Consider what you have learned in Computer Architecture.)

7. What is a preprocessor directive? How are preprocessor directives indicated to the C compiler? Give two examples of preprocessor directives (*Note*: There are two preprocessor directives of principle concern to operating system programming and they are the most basic. You should not concern yourself with conditional preprocessor directives for this course). What do these preprocessor directives do?

8. What is a *C declaration*? What is a *C definition*? Give an example of each.

9. Global declarations typically appear in what type of C source file? Definitions or *implementations* typically appear in what type of C source file? Why might declarations and definitions be split between different C source files?

10. What is an *object* file? What is a *binary executable* file? How are object and executable files differentiated on a Linux based system?

11. Why might recompiling a C program after correcting the underlying code not correct an issue? How can you ensure that the compiled program incorporates corrected code?

12. What is the processing order of a C program? What is the compilation order of a C compiler?

13. How is a C variable declared? How is a C variable's *type* differentiated from the variable's *name* in the declaration?

14. What are primitive types in C? Give three examples.

15. How many bytes does the `char` type require and how many values can a `char` represent? How is the alphanumeric value of a `char` determined?

16. What determines the size of an `int`? Historically, how many bytes have been used to represent the `int` type? Where is the sign bit typically stored in an `int`. What algorithm is used to store negative values in an `int` and why is it important to be aware of this algorithm as a C programmer?

17. What concerns should a C programmer working at a low level have when storing, restoring, or examining values that consist of multiple bytes?

18. How can C variables be converted from one type to another? Give C examples of explicit type conversion and implicit type conversion.

19. What does *scope* mean? Why is scope important?

20. What is the lifespan and visibility of a global variable? What is the initial value of a global variable?

21. How are C variables initialized and what are the implications of default variable values? What should a C programmer do to ensure that variables contain values that are constrained to expected bounds and behavior?

22. How is a block defined in C? Defining a block affects lifespan and visibility... what are the implications of lifespan and visibility inside and outside a block?

23. Can two variables share the same name? Explain.

24. What keyword allows definition of a custom variable C type?

25. Give an example of a custom variable C type that stores a `char` type and two `int` types.

26. What is a reference and how does it relate to a C pointer?

27. How is a C pointer declared? What does the type indicate in a pointer declaration?

28. When using a pointer to a primitive type, what syntax allows the C programmer to access the data referenced by the pointer? What term is used to describe accessing the underlying data referenced by a pointer?

29. When using a pointer to a custom type, what syntax allows the C programmer to access the data contained in the custom type? (*Note*: There is a special operator for this case which cannot be used with primitive types; however, the same syntax used for primitive types can be used to access data stored in a custom type.)

30. Give an example of an `int` pointer declaration.
31. Give an example of a custom variable C type that stores a `char` type, an `int`, and an `int` pointer.
32. What protection is offered for invalid pointers? How does the system behave if a pointer is invalid?
33. What does null mean? Does null imply zero? When the term "null pointer" is used, what does this imply? Why can we use null comparisons for pointers?
34. What does the `void` keyword indicate? Why can C variables not have `void` type? Why can C pointers have `void` type? Why are `void` pointers used?
35. Give three different examples of how a C `int` array might be declared?
36. An array variable has a type, but the variable itself is not of that type. What is the array variable?
37. For a fixed sized array, how and when is the array allocated?
38. For a dynamically sized array, how and when is the array allocated?
39. How is the size of an array communicated in C?
40. How are strings stored in C?
41. What does a null terminator mean in terms of C strings?
42. What are literals? Can literals be referenced? Can literals be changed dynamically?
43. How is a C function declared? How is a C function defined? Where should C functions be declared and where should C functions be defined? Is it necessary for C functions to be declared separately from definitions? Why are declarations typically separated from definitions?
44. Give an example of a C function that accepts two parameters and returns a value.
45. How are a C function's *return type*, *name*, and *parameters* differentiated?
46. What does the `return` keyword instruct the C compiler to do?
47. How many parameters can a C function have? How are parameters differentiated from one another? Why is the order of parameters significant?
48. How many values can be explicitly returned from a C function? Does this imply that a C function can only modify a single variable? Explain.
49. When is it possible for two C functions to be declared with the same name and when is it not possible?
50. If a typed variable is passed to a C function and the value of that parameter is modified within the C function, is the original value in the calling scope modified as well?
51. If a pointer variable is passed to a C function and the value that it points to is modified within the C function, is the original value in the calling scope modified as well?
52. How might C functions be declared with the `void` keyword, and in what roles may it appear?
53. `exit(...)`, `printf(...)`, `malloc(...)`, are special functions that are provided for C users. Who provides these functions, what do these functions do, and how do these functions do what they do? What is the technical name for this special class of system interacting functions?
54. What are the C syntaxes for branching control structures? Give an example of each of these control structures.
55. What are the C syntaxes for loop control structures? (*Hint*: there are three different syntactical forms for loops) Which looping syntax is most appropriate when the program must iterate through a set with fixed size? Which looping syntax is most appropriate when an algorithm iterates until a logical condition is true? Which looping syntax is only appropriate when the contents of the loop must be processed at least once?

56. Are loop control structures generally interchangable? Explain.
57. How do the `break` and `continue` keywords affect loop processing.
58. What does the `extern` keyword mean?
59. How is an array passed to a C function? What additional information might be passed along with the array itself?
60. C is a weakly typed language. How can this affect C code?

4.2 References

- Essential C - https://www2.seas.gwu.edu/~gparmer/courses/essentialC.pdf
- Pointers and memory - https://www2.seas.gwu.edu/~gparmer/courses/mem-ptrs.pdf
- Linked lists - https://www2.seas.gwu.edu/~gparmer/courses/ll.pdf
- Linked list problems - https://www2.seas.gwu.edu/~gparmer/courses/ll-probs.pdf
- Linux C style - https://www.kernel.org/doc/html/v4.10/process/coding-style.html
- Gabe's C style - https://github.com/gparmer/composite/blob/ppos/doc/style_guide/composite_coding_style.pdf
- Google C++ style - https://google.github.io/styleguide/cppguide.html