Revitalizing the Linux Programming Course with Go[*]

Saverio Perugini
Department of Computer Science
University of Dayton
Dayton, Ohio 45469
saverio@udayton.edu

Abstract

We present the design of a contemporary *Linux Programming* course, which includes the use of the Go programming language. To foster course adoption and adaptation, we discuss the design of the course, which includes progressive and thematic content modules, a series of supportive programming assignments, and a culminating, final project experience. The goal of this article is to revitalize the *Linux Programming* course with the use of Go, generate discussion in the community around it, and inspire and facilitate a similar use of Go.

1 Introduction

Nearly thirty years have passed since J. Wolfe published on the topic of reviving systems programming [10]. Given recent trends in big data and cloud computing, web frameworks, and concurrent programming models, we believe the time is ripe to revisit and revitalize the *Linux Programming* course—an enabling and support course for those and similar areas of computing in science and engineering. While many of the concepts and topics in this course have remained stable over the past thirty years, we have attempted to revitalize the course through a focus on concept and tool integration, coverage of Git as version control software, and, particularly, the use of the *Go programming language*[1] [9] as an improved C, especially to reinforce that Linux system calls

[*]Copyright ©2019 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

[1]Developed by Robert Griesemer, Rob Pike, and Ken Thompson, the latter two of which were involved in the original work on UNIX and C; see http://golang.org/.

can be called from any programming language.

Linux Programming is a course that provides an accessible introduction to programming in the Linux environment, especially in C and Go, and prepares students for developing software in that environment using those languages. Topics include libraries and system calls, shells, operating system structures and internals, concurrency, interprocess communication (pipes and signals), the client-server model, configuration and compilation management, regular expressions, pattern matching and filters, shell programming, and automatic program generation. The course distills these topics and concepts through a survey of various software tools supporting Linux programming, including `gcc`, `gdb`, `make`, `git`, `sed` and `awk`, and `lex` and `yacc`, with a thematic focus on the programming environment that these tools collectively foster which is calibrated toward productivity. The course does not aim to be comprehensive and focuses more on breadth than depth. Assignments are designed to provide students with a pragmatic exposure to these tools as well as issues faced by modern practitioners.

Linux Programming is a programming-intensive course. The prerequisite for the course is an operating systems course. The course assumes no prior experience with Linux, C, Go, or any other language used in the course, but expects that students are familiar with programming in some block-structured language.

The Linux Philosophy

Among the multiple themes constituting the Linux philosophy we simply mention the following two; it is our hope that students acquire an appreciation for Linux through observation of these recurring themes, and others, of Linux in this course.

Concurrency and Communication: Often in Linux programming we compose a solution to a problem by combining several small, atomic programs in creative ways through interprocess communication mechanisms such as pipes. Atomic programs are the building blocks; communication mechanisms are the glue. Such programs are easier to develop, debug, and maintain than large, all-encompassing, monolithic systems.

> If you give me the right kind of Tinker Toys, I can imagine the building. I can sit there and see primitives and recognize their power to build structures a half mile high, if only I had just one more to make it functionally complete. — Ken Thompson, creator of UNIX and the 1983 ACM A.M. Turing Award Recipient, in *IEEE Computer*, 32(5), 1999.

Uniform style of I/O: For instance, the function `fprintf` and the system call `write` can be used to write to standard output, a file, or a pipe.

The Student Learning Outcomes are:

- Establish a comfort with and proficiency in Linux and C/Go as a programming language/environment.
- Survey various important system-oriented software tools (`gcc`), including debuggers (`gdb`), and compilation (`make`) and configuration (`git`) managers.
- Establish an understanding of the power of a programmable shell.
- Establish an understanding of the power of the Linux filter style of concurrent system construction as a composition of atomic processes using pipes as the glue in stark contrast to the construction of a monolithic sequential program.
- Establish an understanding of the design and development of systems software, such as command interpreters (`ksh`) and compilers (`gcc`), through the study of system libraries (`libc`), pattern matching and filters (`grep`, `sed`, and `awk`), interprocess communication (pipes and FIFOs), automatic program generation (`lex` and `yacc`), and signals (`SIGINT`).
- Establish a competency in Linux internals (e.g., inodes) and establish an understanding of Linux system calls (e.g., `open`, `close`, `read`, `write`, `fork`, `wait`, `exec`).

2 Course Design

An instructor-authored book on Linux and C, made available to students for free, is the only required textbook. The recommended textbooks are [2, 4, 5, 7, 8, 9] and students have access to each through the University library.

This course tells a story: the first half of the course (Modules I and II) progressively cover the fundamentals of Linux and C/Go, and Linux systems programming, while the second half of the course demonstrates how the atomic tools and utilities covered in Module I can be creatively combined and composed with each other using the interprocess communication mechanisms, covered in Module II and built into the shell, to solve practical programming problems within an environment which supports software development in a timely fashion. The main themes running throughout Modules I and II are the uniform style of I/O in Linux, C, and Go; the interface of Linux and C/Go; and

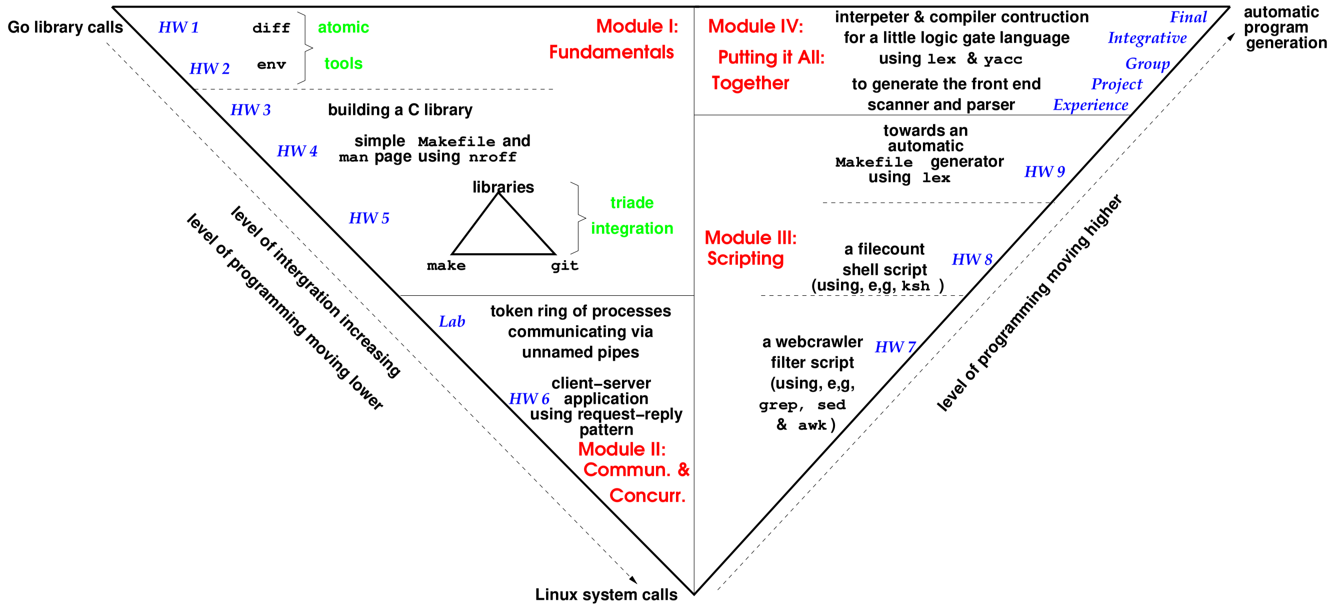

Figure 1: Graphical view of the sequence of course content modules, conceptual topics within each module, and the programming assignments therein.

the well-designed nature of the Linux OS and shell which fosters a powerful, stable, programming environment which has endured for half a century. The main theme running throughout Modules III and IV is the Linux filter style of programming (i.e., constructing software systems, such as specialized tools and utilities, by dynamically and creatively arranging multiple atomic existing tools as building blocks, using pipes as the interprocess communication mechanism). The central focus of the course is on establishing an understanding of these themes and ways of putting them into practice through supportive programming assignments. In other words, helping connect themes to applications. We can think of Modules II and IV as taking Modules I and III to a deeper level, respectively.

Homework assignments are programming exercises each of which require a fair amount of critical thought and design, and approximately 150 lines of code. The following is a list of sample assignment synopses, intended to relate the content and form of assignments that might be helpful to instructors inspired to teach a similar course. While adopters of this approach will undoubtedly tailor assignments, the guiding theme of the assignments as a whole is to serve as an evaluation mechanism within this course template. The series of assignments presented here represent one particular instantiation of that template, depicted graphically in Figure 1, and serve as a vehicle to convey the course motif. Thus, any series of programming and conceptual assignments that fits that general, or a similar, structure is appropriate.

(*MODULE I: FUNDAMENTALS*)

Homework #1 involves implementing a simplified version of the Linux file comparison utility **`diff` in Go**, which will accept input from standard input or file input or a combination of both. This first assignment helps (re-)orient students to programming with standard libraries, especially for I/O and text processing and manipulation.

Homework #2 has two parts—a conceptual exercise and a programming exercise—and both deal with the process environment. The conceptual exercise involves customizing the user environment through the setting and modification of shell variables (e.g., `ENV`, `PS1`, and `PATH`), exploring startup files (e.g., `.profile`, `.kshrc`, and `.vimrc`) and adding both Linux commands and shell-builtin commands to them (e.g., `alias` for creating command aliases). This is a fun exercise for students to tailor their programming environment to their tastes to help improve their productivity. We often spend class time as an activity-learning exercise for this part of the assignment. The programming exercise involves building the Linux utility `env` in Go (and is a modified version of [7, § 2.12 Exercise: An `env` Utility; pp. 54–55] in Go). This second part of the assignment introduces students to Linux system calls and the system call interface in Go (i.e., `package syscall`) and gives them experience with spawning and executing processes.

Homework #3 involves **building an application programming interface (API) in Go** (as a `package`) to a simple linked-list, which can be used for purposes of issue or bug tracking in the development of a software system. Students are given both the interface containing the signatures of the functions, which they must define, as well as a sample application which must be linked to their library. Neither must be modified. This assignment reinforces to students the idea of programming with an API and the idea of factoring a system into three components: i) a public interface (e.g., a `.h` header file), ii) a private library implementation (e.g., a `.a` `ar` file archive of a collection of pre-compiled `.o` object files), and iii) a client application containing the main program. This programming exercise is a modified version of [7, § 2.12 Exercise: Message Logging; pp. 55–56] in Go.

Homework #4 involves defining **a simple `Makefile`** for building both a utility `flip`, which converts DOS to UNIX newlines and verse versa, and its `man`page using `nroff`. Students are given multiple requirements on both the operation and style of the `Makefile` (i.e., naming conventions for targets such as `all` and `clean` and the use of variables such as `CC` to render it more readable and easily modifiable).

With this foundation in place, students are well equipped to start integrating some of the concepts and tools they now know.

Homework #5 integrates three important topics explored in this module: i) implementation of a library and its use in a client application, ii) compilation management (`make`), and iii) configuration management (`git`). Students progressively refine a `Makefile` for an application that utilizes two libraries for interacting with a linked-list data structure. Students [i)]

write and iteratively refine a `Makefile` for the project,

factor out some of the functions in the codebase and create a library from them, and

maintain versions of each iteration using Git on a local repository connected to a remote origin hosted in BitBucket.

Armed with this foundation of fundamentals, students are now ready to construct concurrent systems, which are built using the fundamental concepts from prior assignments (e.g., libraries), whose entities communicate with each other through interprocess communication mechanisms.

(*MODULE II: COMMUNICATION AND CONCURRENCY*)

Homework #6 involves **implementing and experimenting with a client-server application in Go**. The application uses a synchronization barrier, where the barrier is implemented as a server to which clients communicate through named pipes. Clients communicate with the server using a library (i.e., a `package` in Go) which students implement and install (leveraging their experience with implementing, packaging, and linking libraries in Go in Homework #3). This programming exercise is a modified version of [7, § 6.8 Exercise: Barriers; pp. 221–222] in Go.

Equipped with the knowledge of building integrated and concurrent Linux programming structures, we turn our attention from the low-level details of constructing those structures to harnessing off-the-shelf structures/mechanisms to creativity solve a variety of practical programming problems—a higher-level activity.

(*MODULE III: SCRIPTING*)

Homework #7 involves **implementing a Linux filter script** to scrape data from a webpage and apply a series of data transformations, using Linux filters, such as `cut`, `paste`, `join`, `sort`, `uniq`, `tr`, `grep`, `sed`, and `awk`, among a suite of others, to prepare the data for importation into a database system. Students are only given the final output; it is up to them to decide both which transformations to apply and concomitantly which filters to use. The goal of this assignment is to convey to students the plug-and-play flexibility (of the atomic processes) through the Linux filter style of programming. Another goal is to contrast the monolithic, construction of a (typically) sequential program versus the construction of a concurrent system as a composition of atomic processes using pipes as the glue. It is also important for students to understand that the processes in a pipeline execute concurrently, not sequentially, while all remain in synchronization due to the automatic blocking nature of Linux pipes.

Homework #8 involves **building a version of the Linux `find` utility** as a Korn shell script which traverses a series of directories given at the command line and reports the number of plain files, executables, directories, and symbolic links encountered therein. The goal of this assignment is to convey to students the power of a programmable shell, and to contrast (sequential) shell programming with (concurrent) filter style programming (in Homework #7).

Homework #9 involves **using `lex` to automatically generate a filter** capable of parsing a codebase of C and C++ source files and not only extracting, but also, differentiating between uncommented and commented header files therein. This assignment is the basis of an automatic `Makefile` generator, which is itself automatically generated using `lex`. The goal of this assignment is to introduce students to automatic program generation and lexical analysis as well as to expose them to yet another approach to filter construction.

(*MODULE IV: PUTTING IT ALL TOGETHER*)

The entire course is structured to prepare students for the **final, culminating project experience**, which ties many of a course concepts and themes into a robust and compelling, yet manageable, final project. The project involves building an interpreter and a compiler for a small logic language to C++. Students are advised to factor their system into the following three components: [i)]

a front end (i.e., a shift-reduce parser, automatically generated with `lex` and `yacc`, which produces a parse tree);

an interpreter (i.e., an expression evaluator); and

a compiler (i.e., a translator to C++).

3 Discussion

We have used and refined the approach espoused in this article in the *Linux Programming* course at the University of Dayton for nine consecutive offerings of it since Fall 2013 with documented student feedback. Feedback from anonymous student evaluations has revealed that this approach and, especially, the implementation-oriented nature of it, is effective at reinforcing core Linux concepts and themes.

We maintain a set of primary and supplemental material for this course online at `http://academic.udayton.edu/SaverioPerugini/LCP/`. The course webpage for the most recent offering of the *Linux Programming* course at the University of Dayton (Spring 2019), which contains links to the syllabus, course notes, readings, homework assignments, and projects, is available at `http://perugini.cps.udayton.edu/teaching/courses/Spring2019/cps444/`.

There is scope for customization within the general course framework presented here in both content and delivery. For instance, an instructor can make use of interactive learning and assessment tools such as *uAsssign* [1]. A focus on the Linux kernel through which to cover system calls (as opposed to the client-sever model as used here) is an alternate approach [3]. Similarly, more coverage of cybersecurity or mobile devices/OS can be infused into course [6].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Bailey and C. Zilles. uAssign: Scalable interactive activities for teaching the Unix terminal. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 70–76, New York, NY, 2019. ACM Press.

[2] S.P. Harbison and G.L. Steele Jr. *C: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, fourth edition, 1995.

[3] R. Hess and P. Paulson. Linux kernel projects for an undergraduate operating systems course. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 485–489, New York, NY, 2010. ACM Press.

[4] B.W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1984.

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[6] J.-F. Lalande, V. Viet Triem Tong, P. Graux, G. Hiet, W. Mazurczyk, H. Chaoui, and P. Berthomé. Teaching android mobile security. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 232–238, New York, NY, 2019. ACM Press.

[7] K.A. Robbins and S. Robbins. *UNIX Systems Programming: Communication, Concurrency, and Threads*. Prentice Hall, Upper Saddle River, NJ, second edition, 2003.

[8] W. Schotts. *The Linux Command Line: A Complete Introduction*. No Starch Press, 2019. `http://linuxcommand.org/tlcl.php` [Last accessed: 12 June 2019].

[9] M. Summerfield. *Programming in Go: Creating applications for the 21st century*. Addison Wesley, Boston, MA, 2012.

[10] J.L. Wolfe. Reviving systems programming. In *Proceedings of the 23rd ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 255–258, New York, NY, 1992. ACM Press.