1 Introduction

Professor: *Welcome!*

Student: *(screams) Ah! What is this place?*

Professor: *It's a book, silly! Don't be scared. You're just here to learn.*

Student: *(calming down) Uh... it's just that, well, I've never been in a book before.*

Professor: *I know, I know, it takes some getting used to. You'll get there.*

Student: *So what am I here to learn?*

Professor: *Good question! I think the answer is easy: basically, how a computer works, and how to use and program one effectively.*

Student: *(perking up) Hmm... that sounds pretty cool, actually. I've always liked computers, even before I was in a book about them.*

Professor: *It is cool, truly! But there's so much to learn.*

Student: *That's OK. Despite all appearances, I'm actually pretty sharp and, more importantly, hardworking. So where do we start?*

Professor: *Well, as Lewis Carrol once wrote [C65], "Begin at the beginning ... and go on till you come to the end: then stop." So, let's do that.*

Student: *Sounds good. Down the rabbit hole we go!*

1.1 Introduction

Despite everything you ever may have heard, computer systems are actually relatively simple and understandable. There exists a deep and fundamental reason for this: *they are made by people!* Unlike nature, with its quantum mysteries, gravitational enigmas, and biological conundrums, all of which we must slowly reverse engineer to make sense of the world – a never ending process called **science** – computer systems are human artifacts, and thus you should be able to make sense of them. True, many previous computer system designers were clever folks, but not *that* clever; even at some of the lowest levels of a computer system (such as code written in **assembly language**, which you'll learn about later), you will find that things are actually quite comprehensible, once you have the necessary background.

The purpose of this book is to give you that background, or at least, to start you down that path. To do so, you'll be learning about computer systems from three different vantage points. The first is from the view of a user typing commands (i.e., programs that the user wishes to run) into the **command-line interface** of a **UNIX** operating system; we'll call this the "UNIX perspective".

The second view is of a developer writing in the **C programming language**. We'll call this the "C perspective". C programming, while antiquated in some regards, is still a necessary part of a strong computer science education. C programming is also a needed skill in many parts of industry; for example, the Linux operating system, which runs on millions of systems around the world and is at the heart of the modern datacenter, is written primarily in C; the same is true for the core of macOS, which runs on desktops, laptops, and phones.

The third view provided by this book is at **machine level**. Diving down even deeper into how C code runs on a modern processor, we'll find that programs run by executing a sequence of logical, arithmetic, and related operations called **instructions**. Amazingly, modern processors can execute billions of these little instructions every second. Although each instruction isn't that interesting (e.g., take this number and add one to it), by processing billions and billions of them, amazing applications can be built.

These three vantage points, UNIX, C, and machine-level, form the three parts of this book, and hence the name "**Computer Systems: Three Easy Pieces**". In learning about each of these pieces, you'll have the advantage of examining how the computer system operates from multiple vantage points. It's like seeing the computer system in three dimensions, rather than one[1]; you just get a much richer and more complete picture of how things work if you have all of these pieces in mind rather than any single one of them.

1.2 The Big Idea: One Step At A Time

While these three perspectives differ in many regards, there is a single underlying idea that unites them: *take one step at a time*. As

[1]This reminds us of the famous book "Flatland: A Romance of Many Dimensions" [A84], a short book that every geometry-loving person should read. However, be wary of the part about how two-dimensional creatures eat and then, er..., void, which is a logical but unsettling surprise.

you'll see in the examples below, every aspect of the computer system can be unraveled if you slow down and make sure to think through what happens first, then next, then after that, and so forth. At machine level, that is how a processor processes instructions; at C level, that is how a C program runs, one C statement after the other; finally, at UNIX level, that is how a shell interprets a user's commands, one command before the next. At the core of understanding is your adoption of this one-step-at-a-time approach; read the shell script, C program, or assembly code, and make sense of what each is doing one step after the next.

In this regard, understanding how a computer works takes the same skill as following a recipe to prepare some food. To illustrate this, we step through one of our favorite recipes, the Cheeseboard Currant Scone [C03]. Here are the steps:

- Line a baking sheet with parchment paper.
- Whisk together 3 $\frac{1}{2}$ cups flour, $\frac{1}{2}$ teaspoon baking soda, 1 tablespoon baking powder, $\frac{1}{2}$ teaspoon kosher salt, and $\frac{3}{4}$ cup sugar together in a large bowl.
- Add in 1 cup butter (chilled, cut into small cubes), toss to coat, smush together until butter is mostly blended in.
- Stir in 1 cup dried currants; add $\frac{3}{4}$ cup cream and $\frac{3}{4}$ cup buttermilk, mixing until ingredients come together into a uniform whole. *// It's ok if some flour is left over.*
- Mix 1 tablespoon sugar and 1 tablespoon cinnamon for topping.
- Divide dough into 12 balls (about 2 inches in diameter); sprinkle sugar/cinnamon topping over each once on baking sheet.
- Bake at 375F for 25 to 30 minutes, until golden brown.

Although not as precise as the computer programs we will see, the process of following a recipe is much like what the computer does when it runs a program. Thus, if you're able to follow a recipe and understand it, you should also be able to read and understand how computer programs run, whether they are shell scripts, C programs, or even low-level instructions.

Note that the recipe is quite like a computer program in more ways than one. For example, the recipe not only includes the instructions themselves, but also a **comment** ("It's ok if some flour is left over"). Comments are found throughout computer programs; they are little bits of wisdom outside the instructions themselves that the programmer leaves for future readers of the program, and useful because sometimes the instructions themselves can be non-obvious to readers. One (selfish) reason to comment a program properly is that the future reader is often your forgetful self!

So, can you follow the steps in the recipe above, and make some delicious scones? (Seriously, these are the absolute best scones, because they are loaded with butter; try them!) If so, you're ready to begin! If not, well, keep reading the recipe above, going from top to bottom, one step at a time, until you understand how recipes work. If you can't make sense of it, ask some Chef that you know – even though they don't know it, Chefs already have a deep understanding of the fundamentals of computing.

1.3 Start At The Top With UNIX

Our investigation of computer systems begins with a fundamental component called the **operating system**. While there are a few different operating systems in the world, our work will focus on one of the most important and influential, the **UNIX** operating system.

We'll focus on the old-fashioned (but powerful, and awesome) interface UNIX provides to users: the **command-line interface** (**CLI**, and also known as the **command-line interpreter**). The CLI enables users to type **commands** into the system and thus accomplish their desired programming tasks. Knowing how to do so efficiently and effectively is thus one of the main goals of the first part of the book.

To demonstrate the power of the command line, let's work through an example. Imagine you have a **file** filled with numbers, with one number per line of the file. Here are the contents of the file:

```
10
5
3
5
-1
```

The file is a core abstraction provided by UNIX systems. Files store information such as PDF documents, Excel spreadsheets, QuickTime movies, C source code, and even executable programs; we'll learn a lot more about the file abstraction and its centrality to UNIX in subsequent chapters.

Now, let's say you want to sort the file, displaying the sorted output to the screen from lowest number to highest. A seasoned UNIX hacker knows there is already a program, or **command**, that can do this work for us, and so let's use it here. Assuming the numbers are in a file called `numbers.txt`, you can type the following at the command-line prompt:

```
prompt> sort -n numbers.txt
-1
3
5
5
10
prompt>
```

Voilà! That's pretty easy. If you know which commands exist, that is. Knowing useful and powerful commands is a key part of becoming good at using UNIX systems, and thus one of the focuses of the first part of the book is to introduce you to a hundred or so of the most useful ones.

Knowing each command is not quite enough, however. Each command usually has a number of **command-line parameters** (or **flags**) which let you change the behavior of the program in useful and interesting ways. In the example above, we used the flag `-n` to tell the sort program that the input is numeric; otherwise, the program would have produced an alphabetical sort of the numbers, which leads to a different output (try it yourself to see!). Thus, knowing some of the most popular command-line parameters for each program is also a necessary part of mastering UNIX.

On Linux, the `sort` program has these parameters[2]:

- `-b` ignore leading blanks
- `-d` consider only blanks and alphanumeric characters
- `-f` fold lower case to upper case characters
- `-g` compare according to general numerical value
- `-i` consider only printable characters
- `-M` compare months
- `-h` compare human-readable numbers (e.g., 2K 1G)
- `-n` compare according to numerical value
- `-R` shuffle, but group identical keys; See shuf(1)
- `-r` reverse the result of comparisons
- `-V` natural sort of (version) numbers within text
- `-c` check for sorted input; do not sort
- `-C` like -c, but do not report first bad line
- `-k` sort via a key; KEYDEF gives location and type
- `-m` merge already sorted files; do not sort
- `-o` write result to FILE instead of standard output
- `-s` stabilize sort by disabling last-resort comparison
- `-S` use SIZE for main memory buffer
- `-t` use SEP instead of non-blank to blank transition
- `-T` use DIR for temporaries, not $TMPDIR or /tmp
- `-u` output only the first of an equal run
- `-z` line delimiter is NUL, not newline
- `--help` help
- `--version` output version information and exit

Yikes! That's a lot to know. Even seasoned UNIX users aren't likely to know all of these flags, for each of the hundreds of commands available. Rather, they will know a useful subset, which slowly

[2]Actually, we even omitted a few options for brevity(!); type `man sort` for the full thing).

grows over time through need. For example, with `sort`, the flags `-n` (which does numeric sort), `-r` (which reverses the results), and `-k` (which sorts based on a particular key location in a line) are all commonly used and useful.

But knowing each command, and even some useful parameters, is still not enough (alas!). The real power of UNIX comes when harnessing not just one program to solve a task, but stringing together a number of programs to do so.

In this example, we use `sort` to first sort those numbers, but then also `uniq` to only print out each unique number once, with a count of how many times it appeared:

```
prompt> sort -n numbers.txt > tmp1.txt
prompt> uniq -c tmp1.txt
   1 -1
   1 3
   2 5
   1 10
```

To accomplish this task, we utilize a feature known as **output redirection**, via the > symbol above. Instead of printing the output of the `sort` program to screen, output redirection sends it to the file specified to the right of the redirection symbol, in this case, `tmp1.txt`. The `uniq` program is then run over this newly created file, and the desired output printed to screen.

Let's take this one step further, by not only sorting the numbers and producing a unique printing of each (with counts) as above, but also by filtering the results, i.e., showing only numbers that appear more than once in the original file. Here are the set of commands needed to accomplish this task:

```
prompt> sort -n numbers.txt > tmp1.txt
prompt> uniq -c tmp1.txt > tmp2.txt
prompt> awk '($1>1) {print $2, $1}' tmp2.txt
5 2
```

In this example, we utilize the powerful programming language `awk` to filter lines out that don't have their first value (`$1`) greater than 1, and then print out the number (`$2`) followed by the count (`$1`), thus flipping the output order of the `uniq` program (just because we wanted to, not because it is important to do so). Imagine writing a Java program to accomplish this same task – you'd likely write hundreds of lines of code to achieve a similar level of functionality.

You might also notice that our one-step-at-a-time mantra works well here. The first step is to sort the data. The next step is to find the unique numbers in that sorted list, and print out their counts. The last step is to use awk and only print numbers that appear more than once. Even UNIX shells work one step at a time, although the steps are powerful.

However, there is one little problem here: writing out those three lines are a bit clumsy, because of the temporary files we have to create (tmp1.txt and tmp2.txt) and the extra typing that entails. The originators of unix also found this clumsy, and came up with a new primitive, called the **pipe** (and represented by the vertical bar symbol, |). Here's how to use a pipe to accomplish the same task as above, but much more concisely:

```
prompt> sort -n numbers.txt | uniq -c |
        awk '($1>1) {print $2, $1}' file.2
5 2
```

Wow! That's cool. Pipes allow users to assemble programs into a more powerful whole, by taking the output of the program to the left of the pipe and seamlessly making it the input of the program on the right. We'll learn more about redirection, pipes, and related features later on.

While the shell is useful for interactive commands, i.e., where you sit and type in these commands, it is also useful for creating **shell scripts**, which enable you to save some series of commands and make it into a command you can run repeatedly.

For example, here we take the combined sort/uniq/awk line from above and turn it into a shell script (popular_numbers.sh), which can perform this same task for any file we give it. This one is written in **bash**, which is a popular choice for so doing:

```
#! /bin/bash

if (( $# != 1 )); then
    echo "usage: popular_numbers.sh <filename>"
    exit 1
fi

# sort the numbers, and print out unique numbers
# that appear in the sorted list more than once
sort -n $1 | uniq -c | awk '($1>1) {print $2, $1}'
```

When we then run the script as follows, we get the desired output:

```
prompt> popular_numbers.sh numbers.txt
5 2
```

By giving a different file name to this script, we can now perform this task readily on different data sets. Beyond learning how to use a shell interactively, you'll also learn how to create such scripts, another key element of UNIX mastery.

1.4 Becoming A Systems Hacker With C

Once you have gained familiarity with UNIX commands (such as those listed above, and many more), you might start to wonder: how are these commands implemented? Specifically, what programming language is used?

On the earliest systems, there was one and only one answer: the **C programming language** [KR88]. On modern systems, interestingly enough, the answer is usually the same. C is often used to implement the operating system itself (such as Linux or macOS), and many of the command-line tools you'll learn to love such as `cat`, `ls`, and many others. Thus, for any low-level systems hacker, C is a good language to learn.

Furthermore, when you understand what is going on when a C program is running (i.e., when it **executes**), you'll find that you have a much better idea of how computer systems actually work. Thus, through the lens of a C programmer, you can become more knowledgeable about the general and important topic of computer systems.

C programs also follow the mantra of "one step at a time"; any C program, no matter how complex, can be understood by thinking through what each step does. For example, here is a little program that implements a limited form of the `uniq` command line tool; the real `uniq` is (of course) much more complex and general, but the idea is similar. See if you can step through the logic of the program (even without knowing C).

```
// assume array is already sorted
int data[] = { -1, 3, 5, 5, 10 };
int data_length = 5;

int i, last_value;
for (i = 0; i < data_length; i++) {
    if ((i != 0) && (data[i] == last_value))
        continue;
    printf("%d\n", data[i]);
    last_value = data[i];
}
```

To understand this program, you just need to know a few things about C. The first is some basic data structures, such as an **integer** variable (**`int`**), of which there are a few: `data_length`, `i`, `last_value`, and `have_printed`. Computer programs use variables to track information in memory in a human-friendly manner, so we'll be learning a lot more about them during the second part of this course. C integers, more specifically, are positive, negative, or zero-valued numbers, but of limited size: the maximum value of a typical C integer is +2,147,483,647, and the minimum -2,147,483,648. Unlike mathematical integers, C integers are limited because they represent what the underlying machine can compute on efficiently.

You also have to understand a slightly more complex data type, the C **array**. C arrays are simple representations of a collection of similar data types. Here, the array `data` stores five integers; you can access each of those integers via a simple indexing operator, the square bracket (`[]`). Thus, to access the 0th element of the array, just access `data[0]`.

Data structures are not enough to comprehend this code snippet, however. You also need to know some **control** structures, which relate to how the program executes. The **`for`** loop is one such structure. As you might know from previous programming projects, the construct `for (XXX; YYY; ZZZ)  BLOCK` does the following. First, it executes the `XXX` once, before doing anything else (this is useful for initialization). Then, it checks if the condition `YYY` is true; if so, it will execute the code within the code block `BLOCK`, otherwise, it will skip over the loop and move on to the next part of the program. Each time the block gets executed, the statement `ZZZ` is executed afterwards, usually updating some variable value (in this case, the array index, `i`).

Finally, you need to be able to understand conditional execution, such as the **`if`** statement found in the code. In this case, if the condition being tested is true, the code block following the `if` is executed; here, the **`continue`** statement is executed in this case, which simply moves on to the next iteration of the `for` loop without executing the rest of the body of code in the `for` loop.

Given all of this information, can you step through the code, one statement at a time, and figure out what it does? Try it! If you like, you can check out the code[3] for this simple `uniq.c` and run the program yourself; you can also run a verbose form of it, `uniq-verbose.c`, to see the one-step-at-a-time trace of what is happening.

1.5 Diving Into The Guts Of Systems At Machine Level

You've now seen two of the three perspective on systems this book introduces. The first is the UNIX viewpoint, where you use a shell to type in commands, usually that some other developer has written. The second is the C vantage point, where you become that developer, writing C code to implement a new program of some kind. But how does that C code actually run on the machine? At

[3]Found at https://github.com/remzi-arpacidusseau/cstep-code.

the lowest levels of the computer, how do computers actually compute?

To answer these questions, let us dive down one more level, to the level of the **machine** itself. At this level, you'll see that C programs do not execute directly on the machine hardware, but rather are translated into a lower-level language known as **assembly language**. Assembly language, in turn, gets (almost directly) translated into **binary code**, which the machine itself knows how to execute[4]. By understanding the system at this level, finally you'll be able to know how a computer computes!

Assembly code ...

There are a number of tools that make it easy to look at assembly code (but not so easy to understand, at least, not yet!). One such tool is called **`objdump`**, which is a program that can take a compiled C program as input and produce the low-level assembly code as output. Let's use it here to examine the assembly code of the `uniq.c` code we wrote above:

```
400683: 83 f8 01          cmp    $0x1,%eax
400686: 75 05             jne    40068d <main+0x97>
400688: 44 39 2b          cmp    %r13d,(%rbx)
40068b: 74 1a             je     4006a7 <main+0xb1>
40068d: 44 8b 2b          mov    (%rbx),%r13d
400690: 44 89 ea          mov    %r13d,%edx
400693: be 64 07 40 00    mov    $0x400764,%esi
400698: bf 01 00 00 00    mov    $0x1,%edi
40069d: b8 00 00 00 00    mov    $0x0,%eax
4006a2: e8 39 fe ff ff    callq  4004e0 <__printf_chk@plt>
4006a7: 48 83 c3 04       add    $0x4,%rbx
4006ab: b8 01 00 00 00    mov    $0x1,%eax
4006b0: 49 39 dc          cmp    %rbx,%r12
4006b3: 75 ce             jne    400683 <main+0x8d>
```

On the right of this output are the **assembly instructions** that this program executes in the main body of the loop. These instructions execute at an incredible rate on modern processors, something on the order of billions of instructions per second. Indeed, even after years in the world of computers, it is hard to appreciate just how fast that is.

These instructions, if new to you, look complex, but actually are quite simple: each one does so little work to move your program one more step towards completion. For example, the first instruction listed above (`cmp $0x1,%eax`) just compares the number one to the

[4]Assembly code and binary code are quite similar; it is relatively easy to map back and forth between them. Thus, we will often talk about them as if they are the same thing, usually focusing on the more human-readable of the two, which is assembly code.

value in **register** `%eax`[5]. The instruction then sets some flags based on whether the values are the same or different. Another instruction `add $0x4,%rbx` just adds four to the value in register `%rbx`. Overall, each instruction is straightforward, it's just a little hard to read assembly and make sense of it (indeed, this is one strong reason people started writing code in higher-level languages like C).

On the left, there are two other parts to the listing. The first is a number (such as `400683:`) which just shows where the instruction is located in memory (i.e., the instruction's address). On modern systems, when a program is running, both its instructions and data reside in the system's memory; each memory location has an address, which is what is shown here.

The second is the **binary code** we mentioned earlier. While humans want to talk about the lowest levels of code in assembly, the machine wants to operate on numbers. Thus, to run code, the final translation is to take assembly code and turn it into binary, which (as also stated before) is relatively straightforward. For example, the assembly code `cmp $0x1,%eax` maps directly to binary code `83 f8 01` (which is shown in **hexadecimal notation**, or base 16[6]).

Finally, at this level, we see what a computer does. It executes one small instruction (like `cmp`, `mov`, `add`, or `call`), then the next, then the next, then the next. It follows the "recipe" of the program faithfully, one step at a time. By executing billions of these instructions per second, the computer can do amazing things, like simulate hyper-realistic virtual worlds, index and search through the entirety of human knowledge as found on the world wide web, and make new data-driven discoveries in science and other fields. By understanding the machine at this third and lowest vantage point, you will have gained a low-level view of the very nature of computing.

1.6 Summary

As Churchill said, "... this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning." Was he talking about the first chapter of a book? Not so much. But it's a great quote, so we include it here.

There are some things to remember. One is that computers are not magic! Rather, they are made by people, and thus you should be able to understand how they work deeply, with the right amount of effort and study. Two is that to understand computers, in most cases, you can understand them "one step at a time". This reduc-

[5]Registers are just places to store information on a processor; you'll learn more about them later.

[6]Humans are used to base 10, or decimal notation, but to understand the guts of computer systems, you'll have to learn some other bases, especially binary and hexadecimal.

tionist approach works quite well, turning large, complex programs into small chunks, each of which is comprehensible. Three is that there are three perspectives to learn from in this book: UNIX, C, the machine. By becoming knowledgeable at all three levels, you will have gained real insight as to how computer systems work.

References

[A84] "Flatland: A Romance of Many Dimensions" by Edwin Abbott Abbott. Seeley & Co. of London, 1884. *Originally published under the pseudonym "A. Square", Abbott's book discusses the ins and outs of what it would be like to live in a flat (two-dimensional world). The worst part is when he thinks through how creatures in a such a world must eat and ... well ... the opposite of eat. Gross! But a fun read nonetheless.*

[C65] "Alice in Wonderland" by Lewis Carroll (a.k.a. Charles Lutwidge Dodgson). Macmillan and Co., 1865. *Started as a tale spun to entertain some children, and then written down and published some time later. Introduced lots of fun things that we still say today, including the famous Cheshire Cat grin.*

[C03] "The Cheese Board: Collective Works: Bread, Pastry, Cheese, Pizza" by Cheese Board Collective Staff. Cheese Board Colletive, 2003. *The Cheese Board is likely the best place on earth. Why? Well, you can get bread there. You can get cheese there. And you can get awesome amazing pizza there. This recipe book will let you make reasonable facsimiles of what comes from the Cheese Board, but the best thing to do is head to Berkeley, pull out your wad of cash, and start loading on the carbs.*

[KR88] "The C Programming Language" by Brian Kernighan and Dennis Ritchie. Prentice-Hall, April 1988. *The C programming reference that is on every system programmer's bookshelf, written by the people who invented the language.*