TU, DO HOANG

OPERATING SYSTEMS: FROM 0 TO 1

Contents

III   Kernel Programming   275

Preface

Greetings!

You've probably asked yourself at least once how an operating system is written from the ground up. You might even have years of programming experience under your belt, yet your understanding of operating systems may still be a collection of abstract concepts not grounded in actual implementation. To those who've never built one, an operating system may seem like magic: a mysterious thing that can control hardware while handling a programmer's requests via the API of their favorite programming language. Learning how to build an operating system seems intimidating and difficult; no matter how much you learn, it never feels like you know enough. You're probably reading this book right now to gain a better understanding of operating systems to be a better software engineer.

If that is the case, this book is for you. By going through this book, you will be able to find the missing pieces that are essential and enable you to implement your own operating system from scratch! Yes, from scratch, without going through any existing operating system layer to prove to yourself that you are an operating system developer. You may ask,"Isn't it more practical to learn the internals of Linux?".

Yes...

and no.

Learning Linux can help your workflow at your day job. However, if you follow that route, you still won't achieve the ultimate goal of writing an actual operating system. By writing your own operating system, you will gain knowledge that you will not be able to glean just from learn-

ing Linux.

Here's a list of some benefits of writing your own OS:

▷ You will learn how a computer works at the hardware level, and you will learn to write software to manage that hardware directly.

▷ You will learn the fundamentals of operating systems, allowing you to adapt to any operating system, not just Linux

▷ To hack on Linux internals suitably, you'll need to write at least one operating system on your own. This is just like applications programming: to write a large application, you'll need to start with simple ones.

▷ You will open pathways to various low-level programming domains such as reverse engineering, exploits, building virtual machines, game console emulation and more. Assembly language will become one of your most indispensable tools for low-level analysis. (But that does not mean you have to write your operating system in Assembly!)

▷ Writing an operating system is fun!

Why another book on Operating Systems?

There are many books and courses on this topic made by famous professors and experts out there already. Who am I to write a book on such an advanced topic? While it's true that many quality resources exist, I find them lacking. Do any of them show you how to compile your C code and the C runtime library independent of an existing operating system? Most books on operating system design and implementation only discuss the software side; how the operating system communicates with the hardware is skipped. Important hardware details are skipped, and it's difficult for a self-learner to find relevant resources on the Internet. The aim of this book is to bridge that gap: not only will you learn how to program hardware directly, but also how to read official documents from hardware vendors to program it. You no longer have to seek out resources to help yourself interpret hardware manuals and documentation: you can do it yourself. Lastly, I wrote this book from an autodidact's perspective. I made this book as self-contained as possible so you can spend more

time learning and less time guessing or seeking out information on the Internet.

One of the core focuses of this book is to guide you through the process of reading official documentation from vendors to implement your software. Official documents from hardware vendors like Intel are critical for implementing an operating system or any other software that directly controls the hardware. At a minimum, an operating system developer needs to be able to comprehend these documents and implement software based on a set of hardware requirements. Thus, the first chapter is dedicated to discussing relevant documents and their importance.

Another distinct feature of this book is that it is "Hello World" centric. Most examples revolve around variants of a "Hello World" program, which will acquaint you with core concepts. These concepts must be learned before attempting to write an operating system. Anything beyond a simple "Hello World" example gets in the way of teaching the concepts, thus lengthening the time spent on getting started writing an operating system.

Let's dive in. With this book, I hope to provide enough foundational knowledge that will open doors for you to make sense of other resources. This book will be beneficial to students who've just finished their first C/C++ course greatly. Imagine how cool it would be to show prospective employers that you've already built an operating system.

Prerequisites

▷ Basic knowledge of circuits

  – Basic Concepts of Electricity: atoms, electrons, proton, neutron, current flow.

  – Ohm's law

  If you are unfamiliar with these concepts, you can quickly learn them here: http://www.allaboutcircuits.com/textbook/, by reading chapter 1 and chapter 2.

▷ C programming. In particular:

- Variable and function declarations/definitions

- While and for loops

- Pointers and function pointers

- Fundamental algorithms and data structures in C

▷ Linux basics:

- Know how to navigate directory with the command line

- Know how to invoke a command with options

- Know how to pipe output to another program

▷ Touch typing. Since we are going to use Linux, touch typing helps. I know typing speed does not relate to problem-solving, but at least your typing speed should be fast enough not to let it get in the way and degrade the learning experience.

In general, I assume that the reader has basic C programming knowledge, and can use an IDE to build and run a program.

What you will learn in this book

▷ How to write an operating system from scratch by reading hardware datasheets. In the real world, you will not be able to consult Google for a quick answer.

▷ Write code independently. It's pointless to copy and paste code. Real learning happens when you solve problems on your own. Some examples are provided to help kick start your work, but most problems are yours to conquer. However, the solutions are available online for you after giving a good try.

▷ A big picture of how each layer of a computer related to each other, from hardware to software.

▷ How to use Linux as a development environment and common tools for low-level programming.

▷ How a program is structured so that an operating system can run.

▷ How to debug a program running directly on hardware with `gdb` and QEMU.

▷ Linking and loading on bare metal x86_64, with pure C. No standard library. No runtime overhead.

What this book is not about

▷ ELECTRICAL ENGINEERING: The book discusses some concepts from electronics and electrical engineering only to the extent of how software operates on bare metal.

▷ HOW TO USE LINUX OR ANY OS TYPES OF BOOKS: Though Linux is used as a development environment and as a medium to demonstrate high-level operating system concepts, it is not the focus of this book.

▷ LINUX KERNEL DEVELOPMENT: There are already many high-quality books out there on this subject.

▷ OPERATING SYSTEM BOOKS FOCUSED ON ALGORITHMS: This book focuses more on actual hardware platform - Intel x86_64 - and how to write an OS that utilizes of OS support from the hardware platform.

The organization of the book

Part 1 provides a foundation for learning operating system.

▷ Chapter 1 briefly explains the importance of domain documents. Documents are crucial for the learning experience, so they deserve a chapter.

▷ Chapter 2 explains the layers of abstractions from hardware to software. The idea is to provide insight into how code runs physically.

▷ Chapter 3 provides the general architecture of a computer, then introduces a sample computer model that you will use to write an operating system.

▷ Chapter 4 introduces the x86 assembly language through the use of the Intel manuals, along with commonly used instructions. This chapter gives detailed examples of how high-level syntax corresponds to low-level assembly, enabling you to read generated assembly code comfortably. It is necessary to read assembly code when debugging an operating system.

▷ Chapter 5 dissects ELF in detail. Only by understanding how the structure of a program at the binary level, you can build one that runs on bare metal.

▷ Chapter 6 introduces `gdb` debugger with extensive examples for commonly used commands. After acquainting the reader with `gdb`, it then provides insight on how a debugger works. This knowledge is essential for building a debuggable program on the bare metal.

Part 2 presents how to write a bootloader to bootstrap a kernel. Hence the name *"Groundwork"*. After mastering this part, the reader can continue with the next part, which is a guide for writing an operating system. However, if the reader does not like the presentation, he or she can look elsewhere, such as OSDev Wiki: http://wiki.osdev.org/.

▷ Chapter 7 introduces what the bootloader is, how to write one in assembly, and how to load it on QEMU, a hardware emulator. This process involves typing repetitive and long commands, so GNU Make is applied to improve productivity by automating the repetitive parts and simplifying the interaction with the project. This chapter also demonstrates the use of GNU Make in context.

▷ Chapter 8 introduces linking by explaining the relocation process when combining object files. In addition to a bootloader and an operating system written in C, this is the last piece of the puzzle required for building debuggable programs on bare metal, including the bootloader written in Assembly and an operating system written in C.

Part 3 provides guidance on how to write an operating system, as you should implement an operating system on your own and be proud of your creation. The guidance consists of simpler and coherent explanations of necessary concepts, from hardware to software, to implement

the features of an operating system. Without such guidance, you will waste time gathering information spread through various documents and the Internet. It then provides a plan on how to map the concepts to code.

Acknowledgments

Thank you, my beloved family. Thank you, the contributors.

Part I

Preliminary

1

Domain documents

Problem domains

In the real world, software engineering is not only focused on software, but also the problem domain it is trying to solve.

> A *problem domain* is *the part of the world* where the computer is to produce effects, together with the means available to produce them, directly or indirectly. (Kovitz, 1999)

problem domain

A *problem domain* is anything outside of programming that a software engineer needs to understand to produce correct code that can achieve the desired effects. "Directly" means include anything that the software can control to produce the desired effects, e.g. keyboards, printers, monitors, other software, etc. "Indirectly" means anything not part of the software but relevant to the problem domain e.g. appropriate people to be informed by the software when some event happens, students that move to correct classrooms according to the schedule generated by the software. To write a finance application, a software engineer needs to learn sufficient finance concepts to understand the *requirements* of a customer and implement such requirements, correctly.

requirements

> Requirements are the effects that the machine is to exert in the problem domain by virtue of its programming.

Programming alone is not too complicated; programming to solve a problem domain, is [1]. Not only a software engineer needs to understand how to implement the software, but also the problem domain that it tries to solve, which might require in-depth expert knowledge. The software engineer must also select the right programming techniques that apply to the problem domain he is trying to solve because many techniques that are effective in one domain might not be in another. For example, many types of applications do not require performant written code, but a short time to market. In this case, interpreted languages are widely popular because it can satisfy such need. However, for writing huge 3D games or operating system, compiled languages are dominant because it can generate the most efficient code required for such applications.

Often, it is too much for a software engineer to learn non-trivial domains (that might require a bachelor degree or above to understand the domains). Also, it is easier for a *domain expert* to learn enough programming to break down the problem domain into parts small enough for the software engineers to implement. Sometimes, domain experts implement the software themselves.
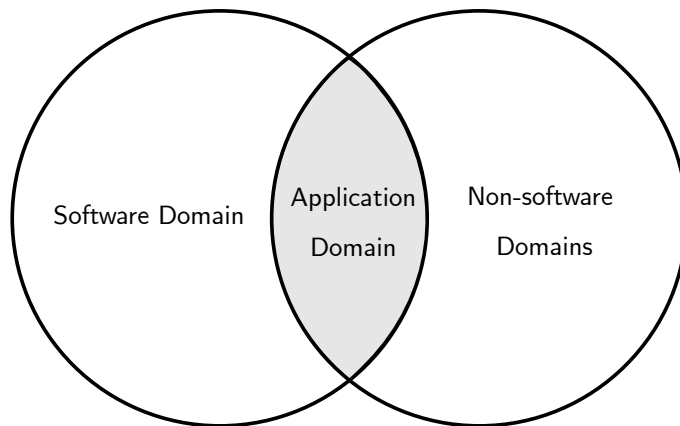
[1] We refer to the concept of "programming" here as someone able to write code in a language, but not necessary know any or all software engineering knowledge.

Figure 1.1.1: Problem domains: Software and Non-software.

One example of such scenario is the domain that is presented in this book: *operating system*. A certain amount of electrical engineering (EE) knowledge is required to implement an operating system. If a computer science (CS) curriculum does not include minimum EE courses, students in the curriculum have little chance to implement a working operating system. Even if they can implement one, either they need to invest a significant amount of time to study on their own, or they fill code in a pre-

defined framework just to understand high-level algorithms. For that reason, EE students have an easier time to implement an OS, as they only need to study a few core CS courses. In fact, only *"C programming"* and *"Algorithms and Data Structures"* classes are usually enough to get them started writing code for device drivers, and later generalize it into an *operating system.*
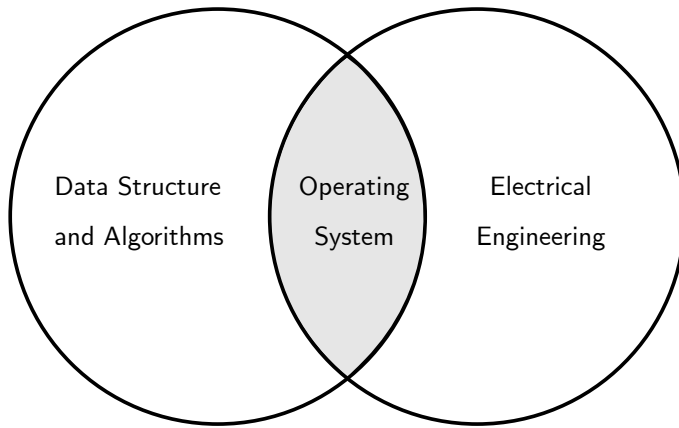
Figure 1.1.2: Operating System domain.

One thing to note is that software is its own problem domain. A problem domain does not necessarily divide between software and itself. Compilers, 3D graphics, games, cryptography, artificial intelligence, etc., are parts of software engineering domains (actually it is more of a computer science domain than a software engineering domain). In general, a software-exclusive domain creates software to be used by other software. Operating System is also a domain, but is overlapped with other domains such as electrical engineering. To effectively implement an operating system, it is required to learn enough of the external domain. How much learning is enough for a software engineer? At the minimum, a software engineer should be knowledgeable enough to understand the documents prepared by hardware engineers for using (i.e. programming) their devices.

Learning a programming language, even C or Assembly, does not mean a software engineer can automatically be good at hardware programming or any related low-level programming domains. One can spend 10 years, 20 years or his entire life writing C/C++ code, and he still cannot write an operating system, simply because of the ignorance of relevant domain knowledge. Just like learning English does not mean a person automatically becomes good at reading Math books written in English. Much

more than that is needed. Knowing one or two programming languages is not enough. If a programmer writes software for a living, he had better be specialized in one or two problem domains outside of software if he does not want his job taken by domain experts who learn programming in their spare time.

1.2   Documents for implementing a problem domain

Documents are essential for learning a problem domain (and actually, anything) since information can be passed down in a reliable way. It is evident that this written text has been used for thousands of years to pass knowledge from generation to generation. Documents are integral parts of non-trivial projects. Without the documents:

▷   New people will find it much harder to join a project.

▷   It is harder to maintain a project because people may forget important unresolved bugs or quirks in their system.

▷   It is challenging for customers to understand the product they are going to use. However, documents do not need to be written in book format. It can be anything from HTML format to database format to be displayed by a graphical user interface. Important information must be stored somewhere safe, readily accessible.

There are many types of documents. However, to facilitate the understanding of a problem domain, these two documents need to be written: *software requirement document* and *software specification.*

1.2.1 Software Requirement Document

Software requirement document includes both a list of requirements and   ***Software requirement***
a description of the problem domain (Kovitz, 1999).

   A software solves a business problem. But, which problems to solve, are requested by a customer. Many of these requests make a list of requirements that our software needs to fulfill. However, an enumerated list of features is seldom useful in delivering software. As stated in the

previous section, the tricky part is not programming alone but programming according to a problem domain. The bulk of software design and implementation depends upon the knowledge of the problem domain. The better understood the domain, the higher quality software can be. For example, building a house is practiced over thousands of years and is well understood, and it is easy to build a high-quality house; software is no different. Code that is difficult to understand is usually due to the author's ignorance of a problem domain. In the context of this book, we seek to understand the low-level working of various hardware devices.

Because software quality depends upon the understanding of the problem domain, the amount of software requirement document should consist of problem domain description.

Be aware that software requirements are not:

What vs How "what" and "how" are vague terms. What is the "what"? Is it nouns only? If so, what if a customer requires his software to perform specific steps of operations, such as purchasing procedure for a customer on a website. Does it include "verbs" now? However, isn't the "how" supposed to be step by step operations? Anything can be the "what" and anything can be the "how".

Sketches Software requirement document is all about the problem domain. It should not be a high-level description of an implementation. Some problems might seem straightforward to map directly from its domain description to the structure of an implementation. For example:

▷ Users are given a list of books in a ***drop-down menu*** to choose.

▷ Books are stored in a ***linked list***".

▷ etc

In the future, instead of a drop-down menu, all books are listed directly on a page in thumbnails. Books might be reimplemented as a graph, and each node is a book for finding related books, as a recommender is going to be added in the next version. The requirement document needs updating again to remove all the outdated implementation details, thus required additional efforts to maintain the requirement doc-

ument, and when the effort for syncing with the implementation is too much, the developers give up documentation, and everyone starts ranting how useless documentation is.

More often than not there is no straightforward one-to-one mapping. For example, a regular computer user expects an OS to be something that runs some program with GUI, or their favorite computer games. But for such requirements, an operating system is implemented as multiple layers, each hiding the details from the upper layers. To implement an operating system, a large body of knowledge from multiple fields is required, especially if the operating system runs on non-PC devices.

It's best to include information related to the problem domain in the requirement document. A good way to test the quality of a requirement document is to provide it to a domain expert for proofreading, to ensure he can understand the material thoroughly. A requirement document is also useful as a help document later, or for writing one much easier.

1.2.2 Software Specification

Software specification document states rules relating desired behavior of the output devices to all possible behavior of the input devices, as well as any rules that other parts of the problem domain must obey.Kovitz (1999)

Software specification

Simply put, software specification is interface design, with constraints for the problem domain to follow e.g. the software can accept certain types of input such as the software is designed to accept English but no other language. For a hardware device, a specification is always needed, as software depends on its hardwired behaviors. And in fact, it is mostly the case that hardware specifications are well-defined, with the tiniest details in it. It needs to be that way because once hardware is physically manufactured, there's no going back, and if defects exist, it's a devastating damage to the company on both finance and reputation.

Note that, similar to a requirement document, a specification only concerns interface design. If implementation details leak in, it is a burden

to sync between the actual implementation and the specification, and soon to be abandoned.

Another important remark is that, though a specification document is important, it does not have to be produced *before* the implementation. It can be prepared in any order: before or after a complete implementation; or at the same time with the implementation, when some part is done, and the interface is ready to be recorded in the specification. Regardless of methods, what matter is a complete specification at the end.

1.3   Documents for writing an x86 Operating System

When problem domain is different from software domain, requirement document and specification are usually separated. However, if the problem domain is inside software, specification most often includes both, and content of both can be mixed with each other. As demonstrated by previous sections the importance of documents, to implement an OS, we will need to collect relevant documents to gain sufficient domain knowledge. These documents are as follow:

▷ Intel® 64 and IA-32 Architectures Software Developer's Manual (Volume 1, 2, 3)

▷ Intel® 3 Series Express Chipset Family Datasheet

▷ System V Application Binary Interface

Aside from the Intel's official website, the website of this book also hosts the documents for convenience[2].

[2] Intel may change the links to the documents as they update their website, so this book doesn't contain any link to the documents to avoid confusion for readers.

Intel documents divide the requirement and specification sections clearly, but call the sections with different names. The corresponding to the requirement document is a section called *"Functional Description"*, which consists mostly of domain description; for specification, *"Register Description"* section describes all programming interfaces. Both documents carry no unnecessary implementation details[3]. Intel documents are also great examples of how to write well requirements/specifications, as explained in this chapter.

[3] As it should be; those details are trade secret.

Other than the Intel documents, other documents will be introduced in the relevant chapters.

2

From hardware to software:

Layers of abstraction

This chapter gives an intuition on how hardware and software connected together, and how software is represented physically.

2.1   The physical implementation of a bit

All electronic devices, from simple to complex, manipulate this flow to achieve desired effects in the real world. Computers are no exception. When we write software, we indirectly manipulate electrical current at the physical level, in such a way that the underlying machine produces desired effects. To understand the process, we consider a simple light bulb. A light bulb can change two states between on and off with a switch, periodically: an off means number 0, and an on means 1.

However, one problem is that such a switch requires manual intervention from a human. What is required is an automatic switch based on the voltage level, as described above. To enable automatic switching of electrical signals, a device called *transistor*, invented by William Shockley, John Bardeen and Walter Brattain. This invention started the whole computer industry.

Figure 2.1.1: A lightbulb
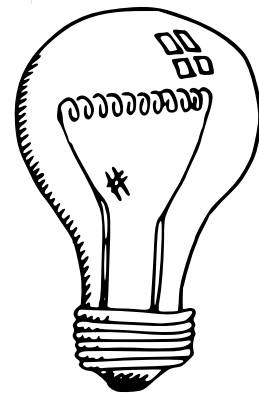
At the core, a *transistor* is just a resistor whose values can vary based on an input voltage value.

With this property, a transistor can be used as a current amplifier (more voltage, less resistance) or switch electrical signals off and on (block and unblock an electron flow) based on a voltage level. At 0 v, no current can pass through a transistor, thus it acts like a circuit with an open switch (light bulb off) because the resistor value is enough to block the electrical flow. Similarly, at +3.5 v, current can flow through a transistor because the resistor value is lessened, effectively enables electron flow, thus acts like a circuit with a closed switch.

A bit has two states: 0 and 1, which is the building block of all digital systems and software. Similar to a light bulb that can be turned on and off, bits are made out of this electrical stream from the power source: Bit 0 are represented with 0 v (no electron flow), and bit 1 is +3.5 v to +5 v (electron flow). Transistor implements a bit correctly, as it can regulate the electron flow based on voltage level.

transistor

Figure 2.1.2: Modern transistor

If you want a deeper explanation electrons move, you should look at the video "How semiconductors work" on Youtube, by Ben Eater.

2.1.1 MOSFET transistors

The classic transistors invented open a whole new world of micro digital devices. Prior to the invention, vacuum tubes - which are just fancier light bulbs - were used to present 0 and 1, and required human to turn it on and off. *MOSFET*, or **M***etal*–**O***xide*–**S***emiconductor* **F***ield*-**E***ffect* **T***ransistor*, invented in 1959 by Dawon Kahng and Martin M. (John) Atalla at Bell Labs, is an improved version of classic transistors that is more suitable for digital devices, as it requires shorter switching time between two states 0 and 1, more stable, consumes less power and easier to produce.

There are also two types of MOSFETs analogous to two types of transistors: n-MOSFET and p-MOSFET. n-MOSFET and p-MOSFET are also called NMOS and PMOS transistors for short.

MOSFET

2.2   Beyond transistors: digital logic gates

All digital devices are designed with logic gates. A *logic gate* is a device that implements a boolean function. Each logic gate includes a number

logic gate

of inputs and an output. All computer operations are built from the combinations of logic gates, which are just combinations of boolean functions.

2.2.1 The theory behind logic gates

Logic gates accept only binary inputs[1] and produce binary outputs. In other words, logic gates are functions that transform binary values. Fortunately, a branch of math that deals exclusively with binary values already existed, called *Boolean Algebra*, developed in the 19^{th}century by George Boole. With a sound mathematical theory as a foundation logic gates were created. As logic gates implement Boolean functions, a set of Boolean functions is *functionally complete*, if this set can construct all other Boolean functions can be constructed from. Later, Charles Sanders Peirce (during $1880 - 1881$) proved that either Boolean function of NOR or NAND alone is enough to create all other Boolean logic functions. Thus NOR and NAND gates are functionally complete Peirce (1933). Gates are simply the implementations of Boolean logic functions, therefore NAND or NOR gate is enough to implement **all** other logic gates. The simplest gates CMOS circuit can implement are inverters (NOT gates) and from the inverters, comes NAND gates. With NAND gates, we are confident to implement everything else. This is why the inventions of transistors, then CMOS circuit revolutionized computer industry.

We should realize and appreciate how powerful boolean functions are available in all programming languages.

2.2.2 Logic Gate implementation: CMOS circuit

Underlying every logic gate is a circuit called **CMOS - *C*** *omplementary* **MOS** *FET*. CMOS consists of two complementary transistors, *NMOS* and *PMOS.* The simplest CMOS circuit is an inverter or a *NOT* gate:

Figure 2.2.1: Example: NAND gate
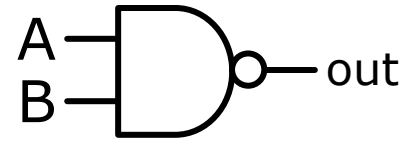
[1] Input that is either a 0 or 1.

functionally complete

If youwant to understand why and how fromNAND gate we cancreate all Boolean functions and a computer, I suggest the course *Build a ModernComputer fromFirst Principles: to From Tetris* available on Coursera: https://www.coursera.org/learn/build-a-computer. Go even further, after the course, youshould take the series *Computational Structures* on Edx.

CMOS

(a) When input is low   **(b)** When input is high
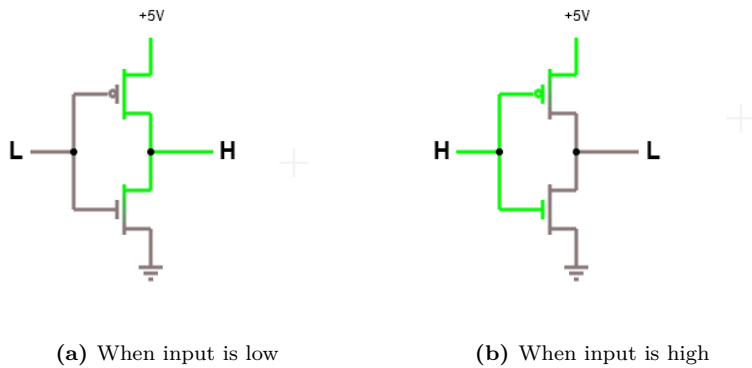
From NOT gate, a NAND gate can be created:

Figure 2.2.3: Electron flows of a NAND gate.

(a) Input = 00, Ouput = 1   **(b)** Input = 01, Ouput = 1

(c) Input = 10, Output = 1   **(d)** Input = 11, Output = 0
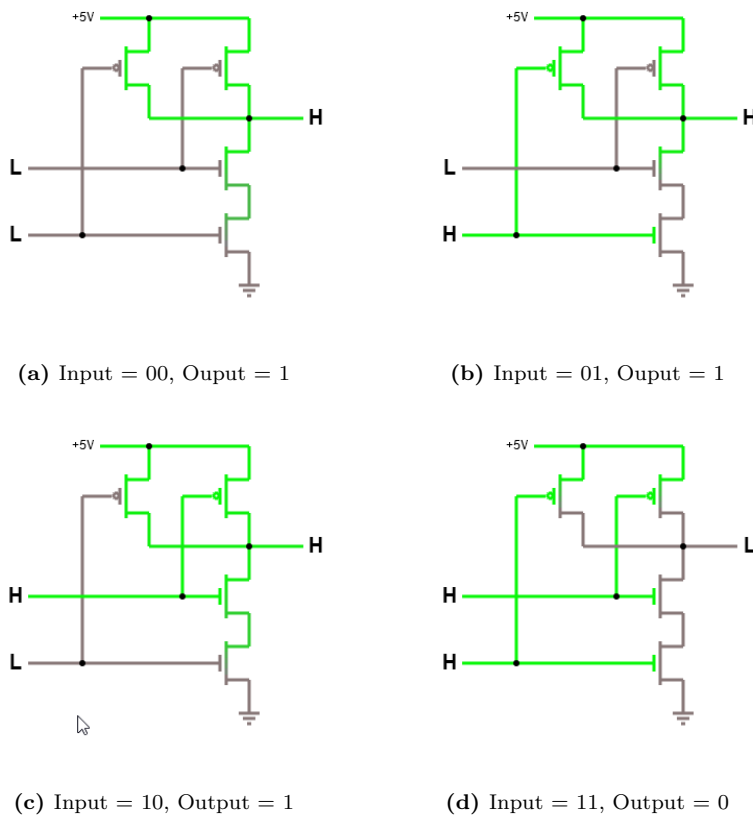
From NAND gate, we have all other gates. As demonstrated, such a simple circuitry performs the logical operators in day-to-day program languages e.g. NOT operator ~ is executed directly by an inverter circuit, and operator & is executed by an AND circuit and so on. Code does not run on a magic black box. In contrast, code execution is precise and transparent, often as simple as running some hardwired circuit. When

we write software, we simply manipulate electrical current at the physical level to run appropriate circuits to produce desired outcomes. However, this whole process somehow does not relate to any thought involving electrical current. That is the real magic and will be explained soon.

One interesting property of CMOS is that **a k-input gate uses k PMOS and k NMOS transistors** (Wakerly, 1999). All logic gates are built by pairs of NMOS and PMOS transistors, and gates are the building blocks of all digital devices from simple to complex, including any computer. Thanks to this pattern, it is possible to separate between the actual physical circuit implementation and logical implementation. Digital designs are done by designing with logic gates then later be "compiled" into physical circuits. In fact, later we will see that logic gates become a language that describes how circuits operate. Understanding how CMOS works is important to understand how a computer is designed, and as a consequence, how a computer works[2].

Finally, an implemented circuit with its wires and transistors is stored physically in a package called a *chip*. A *chip* is a substrate that an integrated circuit is etched onto. However, a chip also refers to a completely packaged integrated circuit in consumer market. Depends on the context, it is understood differently.

[2] Again, if you want to understand how logic gates make a computer, consider the suggested courses on Coursera and Edx earlier.

Figure 2.2.4: 74HC00 chip physical view

Example 2.2.1. 74HC00 is a chip with four 2-input NAND gates. The chip comes with 8 input pins and 4 output pins, 1 pin for connecting to a voltage source and 1 pin for connecting to the ground. This device is the physical implementation of NAND gates that we can physically touch and use. But instead of just a single gate, the chip comes with 4 gates that can be combined. Each combination enables a different logic function, effective creating other logic gates. This feature is what make the chip popular.

Each of the gates above is just a simple NAND circuit with the electron flows, as demonstrated earlier. Yet, many these NAND-gates chips combined can build a simple computer. Software, at the physical level, is just electron flows.
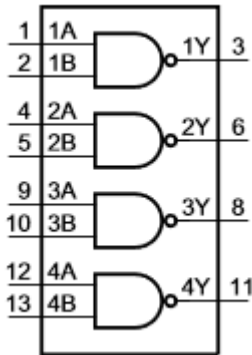
(a) Logic diagram of 74HC00
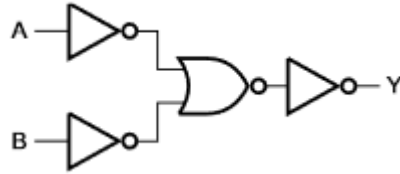
(b) Logic diagram of one NAND gate

Figure 2.2.5: 74HC00 logic diagrams (Source: 74HC00 datasheet, http://www.scrpdf.com/pdf/ Semiconductors_new/Logic/ 74HCT/74HC_HCT00.pdf)
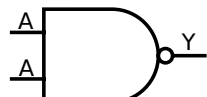
(a) NOT gate
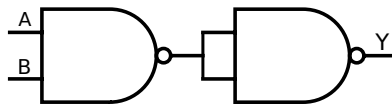
(b) AND gate

Figure 2.2.6: Gates built from NAND gates, each accepts 2 input signals and generate 1 output signal.
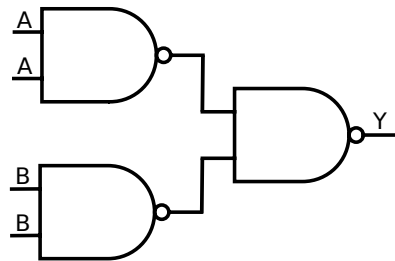
(c) OR gate
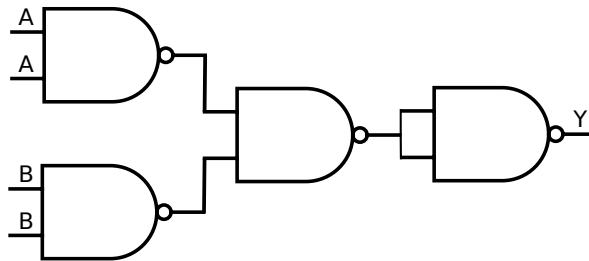
(d) NOR gate

How can the above gates be created with 74HC00? It is simple: as every gate has 2 input pins and 1 output pin, we can write the output of 1 NAND gate to an input of another NAND gate, thus chaining NAND gates together to produce the diagrams as above.

2.3   Beyond Logic Gates: Machine Language

2.3.1 Machine language

Being built upon gates, as gates only accept a series of 0 and 1, a hardware device only understands 0 and 1. However, a device only takes 0 and 1 in a systematic way. *Machine language* is a collection of unique bit patterns that a device can identify and perform a corresponding action. A *machine instruction* is a unique bit pattern that a device can identify. In a computer system, a device with its language is called **CPU** - **C**entral **P**rocessing **U**nit, which controls all activities going inside a computer. For example, in the x86 architecture, the pattern `10100000` means telling a CPU to add two numbers, or `000000101` to halt a computer. In the early days of computers, people had to write completely in binary.

Machine language

Why does such a bit pattern cause a device to do something? The reason is that underlying each instruction is a small circuit that implements the instruction. Similar to how a function/subroutine in a computer program is called by its name, a bit pattern is a name of a little function inside a CPU that got executed when the CPU finds one.

Note that CPU is not the only device with its language. CPU is just a name to indicate a hardware device that controls a computer system. A hardware device may not be a CPU but still has its language. A device with its own machine language is a *programmable device*, since a user can use the language to command the device to perform different actions. For example, a printer has its set of commands for instructing it how to print a page.

Example 2.3.1. A user can use 74HC00 chip without knowing its internal, but only the interface for using the device. First, we need to know its layout:
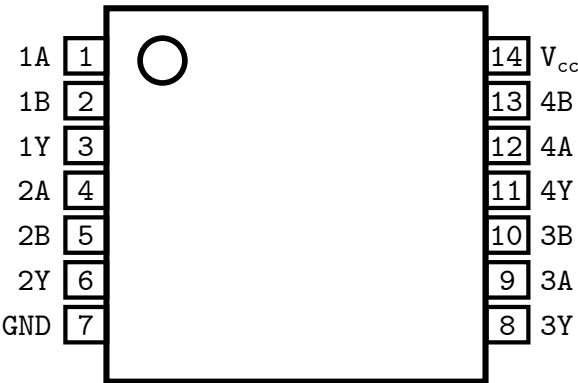
Then, the functionality of each pin:

Symbol	Pin	Description
1A to 4A	1, 4, 9, 12	data input
1B to 4B	2, 5, 10, 13	data input
1Y to 4Y	3, 6, 8, 11	data output
GND	7	ground (0 V)
V_{cc}	14	supply voltage

Finally, how to use the pins:

Input		Output
nA	**nB**	**nY**
L	L	H
L	X	H
X	L	H
H	H	L

Table 2.3.2: Functional Description

The functional description provides a truth table with all possible pin inputs and outputs, which also describes the usage of all pins in the device. A user needs not to know the implementation, but on such a table to use the device. We can say that the truth table above is the machine language of the device. Since the device is digital, its language is a collection of binary strings:

▷ n is a number, either 1, 2, 3, or 4

▷ H = HIGH voltage level; L = LOW voltage level; X = don't care.

▷ The device has 8 input pins, and this means it accepts binary strings of 8 bits.

▷ The device has 4 output pins, and this means it produces binary strings of 4 bits from the 8-bit inputs.

The number of input strings is what the device understand, and the number of output strings is what the device can speak. Together, they make the language of the device. Even though this device is simple, yet the language it can accept contains quite many binary strings: $2^8 + 2^4 = 272$. However, the number is a tiny fraction of a complex device like a CPU, with hundreds of pins.

When leaving as is, 74HC00 is simply a NAND device with two 4-bit inputs[3].

[3] Or simply 4-bit NAND gate, as it can only accept 4 bits of input at the maximum.

	Input								Output			
Pin	1A	1B	2A	2B	3A	3B	4A	4B	1Y	2Y	3Y	4Y
Value	1	1	0	0	1	1	0	0	0	1	0	1

The inputs and outputs as visually presented:
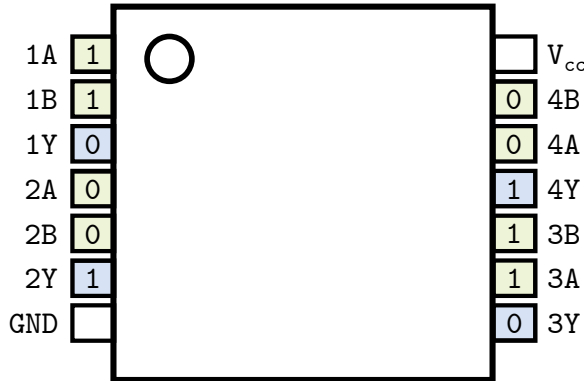
Figure 2.3.2: Pins when receiving digital signals that correspond to a binarystring. Green signals are inputs; blue signals are outputs.

On the other hand, if OR gate is implemented, we can only build a 2-input OR gate from 74HC00, as it requires 3 NAND gates: 2 input NAND gates and 1 output NAND gate. Each input NAND gate represents only a 1-bit input of the OR gate. In the following figure, the pins of each input NAND gates are always set to the same values (either both inputs are A or both inputs are B) to represent a single bit input for the final OR gate:

(a) 2-bit OR gate logic diagram, built from 3 NAND gates with 4 pins just for 2 bits of input.

(b) Pin 3A and 3B take the values from 1Y and 2Y.
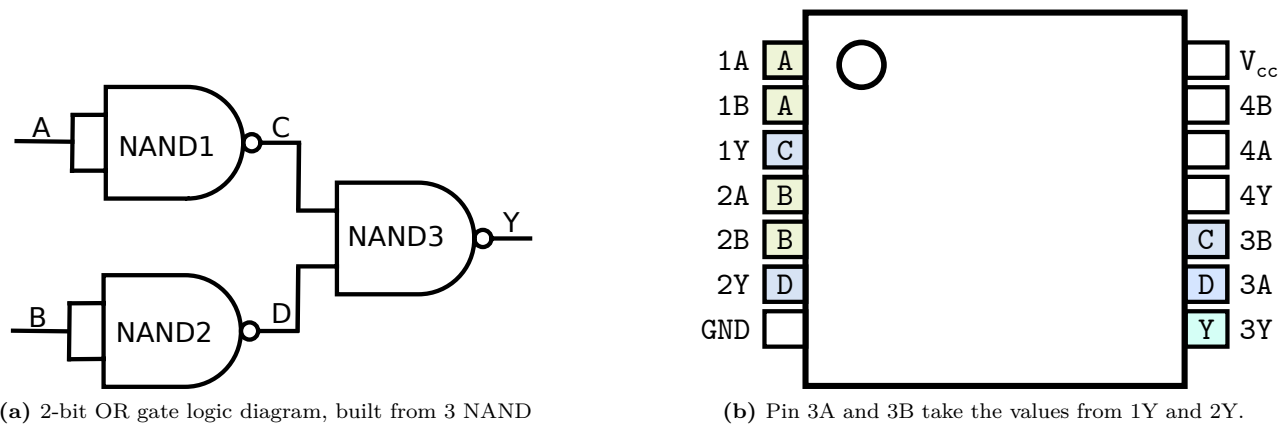
Figure 2.3.3: 2-bit OR gate implementation

Table 2.3.3: Truth table of OR logic diagram.

A	B	C	D	Y
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

To implement a 4-bit OR gate, we need a total of four of 74HC00 chips configured as OR gates, packaged as a single chip as in figure 2.3.4.
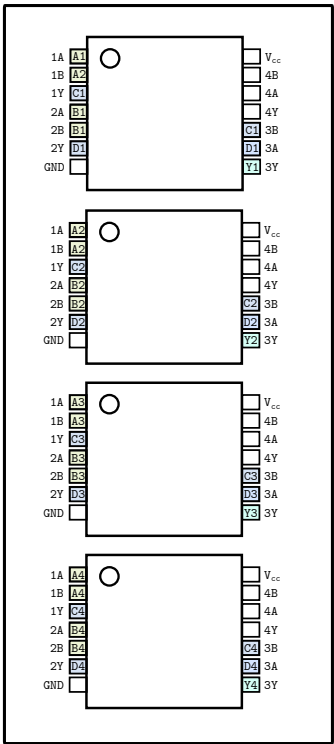
Figure 2.3.4: 4-bit OR chip made from four 74HC00 devices

2.3.2 Assembly Language

Assembly language is the symbolic representation of binary machine code, by giving bit patterns mnemonic names. It was a vast improvement when programmers had to write 0 and 1. For example, instead of writing `000000101`, a programmer simply write `hlt` to stop a computer. Such an abstraction makes instructions executed by a CPU easier to remember, and thus more instructions could be memorized, less time spent looking up CPU manual to find instructions in bit forms and as a result, code was written faster.

Understand assembly language is crucial for low-level programming domains, even to this day. The more instructions a programmer want to understand, the deeper understanding of machine architecture is required.

Example 2.3.2. We can build a device with 2 assembly instructions:

```
or   <op1>, <op2>
nand <op1>, <op2>
```

▷ `or` accepts two 4-bit operands. This corresponds to a 4-input OR gate device built from 4 74HC00 chips.

▷ `nand` accepts two 4-bit operands. This corresponds to a single 74HC00 chips, leave as is.

Essentially, the gates in the example 2.3.1 implements the instructions. Up to this point, we only specify input and output and manually feed it to a device. That is, to perform an operation:

▷ Pick a device by hands.

▷ Manually put electrical signals into pins.

First, we want to automate the process of device selection. That is, we want to simply write assembly instruction and the device that implements the instruction is selected correctly. Solving this problem is easy:

▷ Give each instruction an index in binary code, called *operation code* or *opcode* for short, and embed it as part of input. The value for each instruction is specified as in table 2.3.4.

Table 2.3.4: Instruction-Opcode mapping.

Instruction	Binary Code
nand	00
or	01

Each input now contains additional data at the beginning: an opcode. For example, the instruction:

```
nand 1100, 1100
```

corresponds to the binary string: 0011001100. The first two bits 00 encodes a `nand` instruction, as listed in the table above.

▷ Add another device to select a device, based on a binary code peculiar to an instruction.

Such a device is called a *decoder*, an important component in a CPU that decides which circuit to use. In the above example, when feeding 0011001100 to the decoder, because the opcode is 00, data are sent to NAND device for computing.

Finally, writing assembly code is just an easier way to write binary strings that a device can understand. When we write assembly code and save in a text file, a program called an *assembler* translates the text file   **assembler** into binary strings that a device can understand. So, how can an assembler exist in the first place? Assume this is the first assembler in the world, then it is written in binary code. In the next version, life is easier: the programmers write the assembler in the assembly code, then use the first version to compile itself. These binary strings are then stored in another device that later can be retrieved and sent to a decoder. A *storage de-*   **storage device** *vice* is the device that stores machine instructions, which is an array of circuits for saving 0 and 1 states.

A decoder is built out of logic gates similar to other digital devices. However, a storage device can be anything that can store 0 and 1 and is retrievable. A storage device can be a magnetized device that uses magnetism to store information, or it can be made out of electrical circuits that can change and rermember states when a voltage is applied. Regardless of the technology used, as long as the device can store data and is accessible to retrieve data, it suffices. Indeed, the modern devices are so complex that it is impossible and unnecessary to understand every implementation detail. Instead, we only need to learn the interfaces, e.g. the pins, that the devices expose.

current instruction

Storage

0011001100
0111111111
0111101100
0010101110
....
....
....

retrieve data

Decoder

send data

4-bit OR

1A A1   ○   V_cc
1B A2   4B
1Y C1   4A
2A B1   4Y
2B B1   C1 3B
2Y D1   D1 3A
GND   Y1 3Y

1A A2   ○   V_cc
1B A2   4B
1Y C2   4A
2A B2   4Y
2B B2   C2 3B
2Y D2   D2 3A
GND   Y2 3Y

1A A3   ○   V_cc
1B A3   4B
1Y C3   4A
2A B3   4Y
2B B3   C3 3B
2Y D3   D3 3A
GND   Y3 3Y

1A A4   ○   V_cc
1B A4   4B
1Y C4   4A
2A B4   4Y
2B B4   C4 3B
2Y D4   D4 3A
GND   Y4 3Y

1A 1   ○   V_cc
1B 1   0 4B
1Y 0   0 4A
2A 0   4-bit NAND 1 4Y
2B 0   1 3B
2Y 1   1 3A
GND   0 3Y

Figure 2.3.5: A decoder retrieves the current instruction pointed by the arrow and selects the NAND device to execute the `nand` instruction.

A computer essentially implements this process:

▷ *Fetch* an instruction from a storage device.

▷ *Decode* the instruction.

▷ *Execute* the instruction.

Or in short, a fetch – decode – execute cycle. The above device is extremely rudimentary, but it already represents a computer with a *fetch – decode – execute* cycle. More instructions can be implemented by adding more devices and allocating more opcodes for the instructions, then update the decoder accordingly. The Apollo Guidance Computer, a digital computer produced for the Apollo space program from 1961 – 1972, was built entirely with NOR gates - the other choice to NAND gate for creating

other logic gates. Similarly, if we keep improving our hypothetical device, it eventually becomes a full-fledge computer.

2.3.3 Programming Languages

Assembly language is a step up from writing 0 and 1. As time goes by, people realized that many pieces of assembly code had repeating patterns of usages. It would be nice if instead of writing all the repeating blocks of code all over again in all places, we simply refer to such blocks of code with easier to use text forms. For example, a block of assembly code checks whether one variable is greater than another and if so, execute a block of code, else execute another block of code; in C, such block of assembly code is represented by an `if` statement that is close to human language.
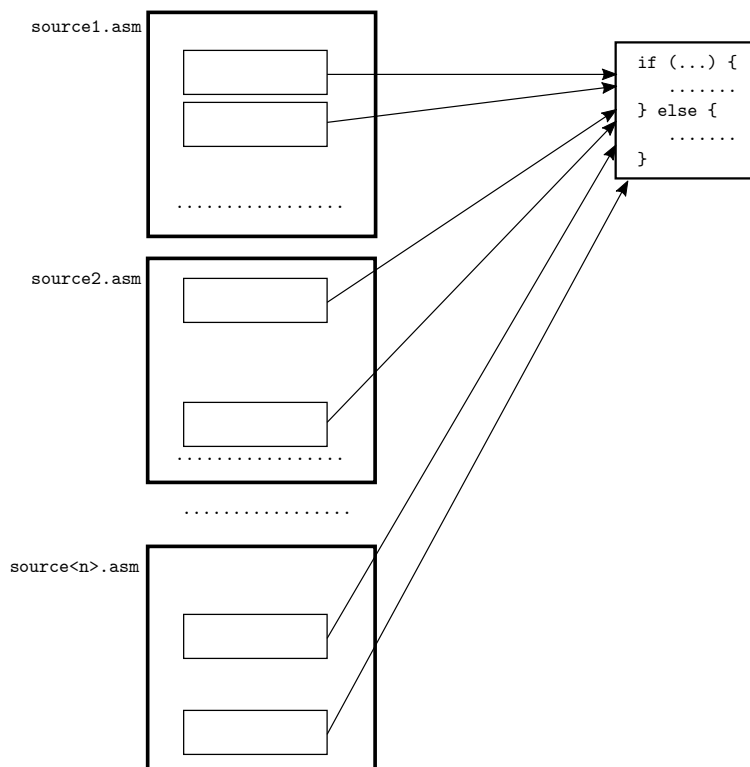
Figure 2.3.6: Repeated assembly patterns are generalized into a new language.

People created text forms to represent common blocks of assembly code, such as the `if` syntax above, then write a program to translate the text forms into assembly code. The program that translates such text forms to machine code is called a *compiler*:   *compiler*

Any software logic a programming language can implement, hardware

```
if (argc) {
    i = 1;
} else {
    i = 0;
}
```

Compiler

```
cmp     DWORD PTR [ebp+0x8],0x0
je      80483f7 <main+0x1c>
mov     DWORD PTR [ebp-0x4],0x1
jmp     80483fe <main+0x23>
mov     DWORD PTR [ebp-0x4],0x0
```
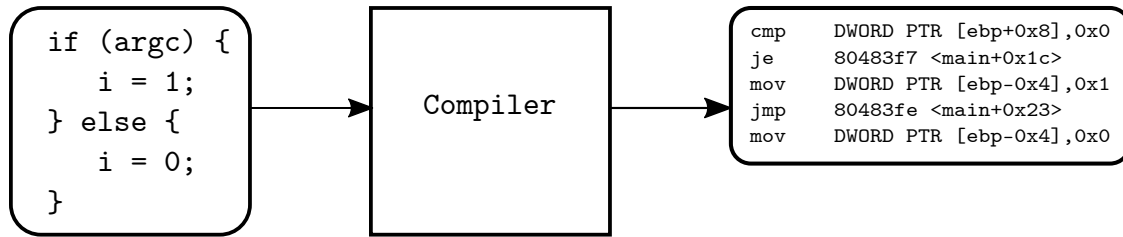
Figure 2.3.7: From high-level language back to low-level language.

can also implement. The reverse is also true: any hardware logic that is implemented in a circuit can be reimplemented in a programming language. The simple reason is that programming languages, or assembly languages, or machine languages, or logic gates are just languages to express computations. It is impossible for software to implement something hardware is incapable of because programming language is just a simpler way to use the underlying hardware. At the end of the day, programming languages are translated to machine instructions that are valid to a CPU. Otherwise, code is not runnable, thus a useless software. In reverse, software can do everything hardware (that run the software) can, as programming languages are just an easier way to use the hardware.

In reality, even though all languages are equivalent in power, not all of them are capable of express programs of each other. Programming languages vary between two ends of a spectrum: high level and low level.

The higher level a programming language is, the more distant it becomes from the hardware. In some high-level programming languages, such as Python, a programmer cannot manipulate underlying hardware, despite being able to deliver the same computations as low-level programming languages. The reason is that high-level languages want to hide hardware details to free programmers from dealing with irrelevant details not related to current problem domains. Such convenience, however, is not free: it requires software to carry an extra code for managing hardware details (e.g. memory) thus making the code run slower, and it makes hardware programming difficult or impossible. The more abstractions a programming language imposes, the more difficult it is for writing low-level software, such as hardware drivers or an operating system. This is the reason why C is usually a language of choice for writing an operating system, since C is just a thin wrapper of the underlying hardware, making

it easy to understand how exactly a hardware device runs when executing a certain piece of C code.

Each programming language represents a way of thinking about programs. Higher-level programming languages help to focus on problem domains that are not related to hardware at all, and where programmer performance is more important than computer performance. Lower-level programming languages help to focus on the inner-working of a machine, thus are best suited for problem domains that are related to control hardware. That is why so many languages exist. Use the right tools for the right job to achieve the best results.

2.4   Abstraction

Abstraction is a technique for hiding complexity that is irrelevant to the problem in context. For example, writing programs without any other layer except the lowest layer: with circuits. Not only a person needs an in-depth understanding of how circuits work, making it much more obscure to design a circuit because the designer must look at the raw circuits but think in higher-level such as logic gates. It is a distracting process, as a designer must constantly translate the idea into circuits. It is possible for a designer simply thinks his high-level ideas straight, and later translate the ideas into circuits. Not only it is more efficient, but it is also more accurate as a designer can focus all his efforts into verifying the design with high-level thinking. When a new designer arrives, he can easily understand the high-level designs, thus can continue to develop or maintain existing systems.

2.4.1 Why abstraction works

In all the layers, abstractions manifest itself:

▷ Logic gates abstract away the details of CMOS.

▷ Machine language abstracts away the details of logic gates.

▷ Assembly language abstracts away the details of machine languages.

▷ Programming language abstracts away the details of assembly languages.

We see repeating patterns of how lower-layers build upper-layers:

▷ A lower layer has a recurring pattern. Then, this recurring pattern is taken out and built a language on top of it.

▷ A higher layer strips away layer-specific (non-recurring) details to focus on the recurring details.

▷ The recurring details are given a new and simpler language than the languages of the lower layers.

What to realize is that every layer is just *a more convenient language to describe the lower layer*. Only after a description is fully created with the language of the higher layer, it is then be *implemented* with the language of the lower layer.

▷ CMOS layer has a recurring pattern that makes sure logic gates are reliably translated to CMOS circuits: **a k-input gate uses k PMOS and k NMOS transistors** (Wakerly, 1999). Since digital devices use CMOS exclusively, a language arose to describe higher level ideas while hiding CMOS circuits: Logic Gates.

▷ Logic Gates hides the language of circuits and focuses on how to implement primitive Boolean functions and combine them to create new functions. All logic gates receive input and generate output as binary numbers. Thanks to this recurring patterns, logic gates are hidden away for the new language: Assembly, which is a set of predefined binary patterns that cause the underlying gates to perform an action.

▷ Soon, people realized that many recurring patterns arisen from within Assembly language. Repeated blocks of Assembly code appear in Assembly source files that express the same or similar idea. There were many such ideas that can be reliably translated into Assembly code. Thus, the ideas were extracted for building into the high level programming languages that everyone programmer learns today.

Recurring patterns are the key to abstraction. Recurring patterns are why abstraction works. Without them, no language can be built, and thus

no abstraction. Fortunately, human already developed a systematic discipline for studying patterns: Mathematics. As quoted from the British mathematician G. H. Hardy (2005):

> A mathematician, like a painter or a poet, is a maker of patterns. If his patterns are more permanent than theirs, it is because they are made with ideas.

Isn't that a mathematical formula a representation of a pattern? A variable represents values with the same properties given by constraints? Mathematics provides a formal system to identify and describe existing patterns in nature. For that reason, this system can certainly be applied in the digital world, which is just a subset of the real world. Mathematics can be used as a common language to help translation between layers easier, and help with the understanding of layers.
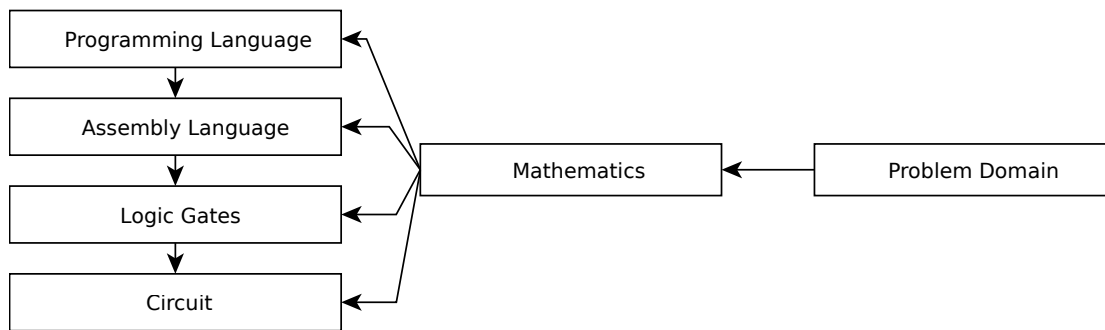
Figure 2.4.1: Mathematics as a universal language for all layers. Since all layers can express mathematics with their technologies, each layer can be translated into another layer.

2.4.2 Why abstraction reduces complexity

Abstraction by building language certainly leverages productivity by stripping irrelevant details to a problem. Imagine writing programs without any other layout except the lowest layer: with circuits. This is how complexity emerges: when high-level ideas are expressed with lower-level language, as the example above demonstrated. Unfortunately, this is the case with software as programming languages at the moment are more emphasized on software rather than the problem domains. That is, without prior knowledge, code written in a language is unable to express itself the knowledge of its target domain. In other words, *a language is expressive if its syntax is designed to express the problem domain it is trying to solve.* Consider this example: That is, the *what* it will do rather

the *how* it will do.

Example 2.4.1. Graphviz (http://www.graphviz.org/) is a visualization software that provides a language, called `dot`, for describing graph:

```
digraph {
    a -> b;
    b -> c;
    a -> c;
    d -> c;
}
```
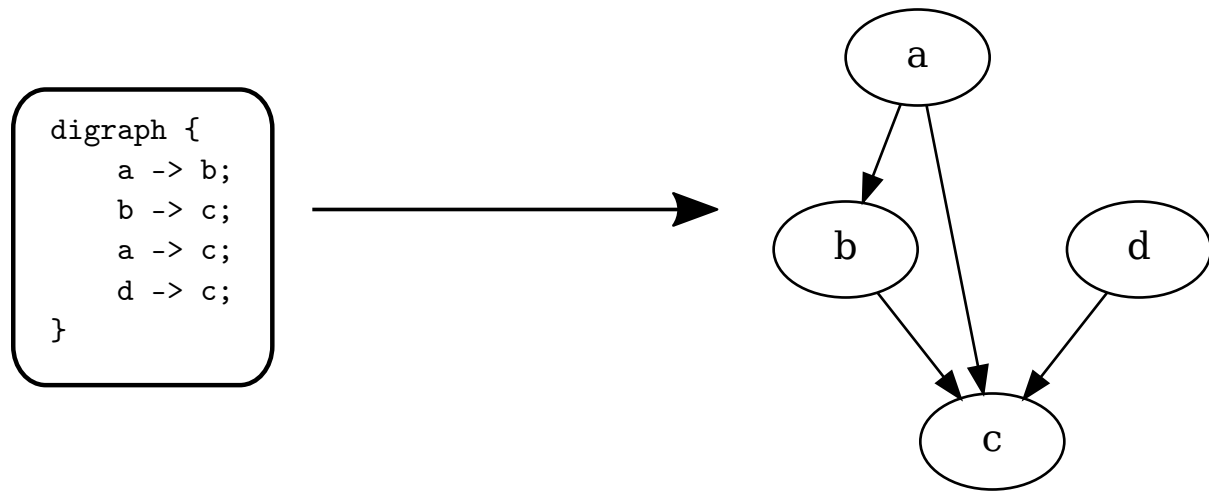
Figure 2.4.2: From graph description to graph.

As can be seen, the code perfectly expresses itself how the graph is connected. Even a non-programmer can understand and use such language easily. An implementation in C would be more troublesome, and that's assuming that the functions for drawing graphs are already available. To draw a line, in C we might write something like:

```
draw_line(a, b);
```

However, it is still verbose compared with:

```
a -> b;
```

Also, `a` and `b` must be defined in C, compared to the implicit nodes in the `dot` language. However, if we do not factor in the verbosity, then C still has a limitation: it cannot change its syntax to suit the problem domain. A domain-specific language might even be more verbose, but it makes a domain more understandable. If a problem domain must be expressed in C, then it is constraint by the syntax of C. Since C is not a

specialized language for a problem domain that, but is a *general-purpose* programming language, the domain knowledge is buried within the implementation details. As a result, a C programmer is needed to decipher and extract the domain knowledge out. If the domain knowledge cannot be extracted, then the software cannot be further developed.

Example 2.4.2. Linux is full of applications controlled by many domain-specific languages and are placed in `/etc` directory, such as a web server. Instead of reprogramming the software, a domain-agnostic language is made for it.

In general, code that can express a problem domain must be understandable by a domain expert. Even within the software domain, building a language out of repeated programming patterns is useful. It helps people aware the existence of such patterns in code and thus making software easier to maintain, as software structure is visible as a language. Only a programming language that is capable of morphing itself to suit a problem domain can achieve that goal. Such language is called a *programmable programming language.* Unfortunately, this approach of turning software structure visible is not favored among programmers, as a new language must be made out of it along with new toolchain to support it. Thus, software structure and domain knowledge are buried within code written in the syntax of a general-purpose language, and if a programmer is not familiar or even aware of the existence of a code pattern, then it is hopeless to understand the code. A prime example is reading C code that controls hardware, e.g. an operating system: if a programmer knows absolutely nothing about hardware, then it is impossible to read and write operating system code in C, even if he could have 20 years of writing application C code.

With abstraction, a software engineer can also understand the inner-working of a device without specialized knowledge of physical circuit design, enables the software engineer to write code that controls a device. The separation between logical and physical implementation also entails that gate designs can be reused even when the underlying technologies

changed. For example, in some distant future biological computer could be a reality, and gates might not be implemented as CMOS but some kind of biological cells e.g. as living cells; in either technology: electrical or biological, as long as logic gates are physically realized, the same computer design could be implemented.

<div align="center">

3

Computer Architecture

</div>

To write lower level code, a programmer must understand the architecture of a computer. It is similar to when one writes programs in a software framework, he must know what kinds of problems the framework solves, and how to use the framework by its provided software interfaces. But before getting to the definition of what computer architecture is, we must understand what exactly is a computer, as many people still think that a computer is a regular computer we put on a desk, or at best, a server. Computers come in various shapes and sizes and are devices that people never imagine they are computers, and that code can run on such devices.

3.1   What is a computer?

A *computer* is a hardware device that consists of at least a processor (CPU),   ***computer***
a memory device and input/output interfaces. All the computers can be
grouped into two types:

Single-purpose computer is a computer built at the *hardware level* for
   specific tasks. For example, dedicated application encoders/decoders ,
   timer, image/video/sound processors.

General-purpose computer is a computer that can be programmed (without modifying its hardware) to emulate various features of single-purpose

computers.

3.1.1 Server

A *server* is a general-purpose high-performance computer with huge re-
sources to provide large-scale services for a broad audience. The audi-
ence are people with their personal computer connected to a server.

server

Figure 3.1.1: Blade servers. Each
blade server is a computer with a
modular design optimize for the use
of physical space and energy.The
enclosure of blade servers is called a
chassis.(Source: Wikimedia, author:
Victorgrigas)

3.1.2 Desktop Computer

A *desktop computer* is a general-purpose computer with an input and out-
put system designed for a human user, with moderate resources enough
for regular use. The input system usually includes a mouse and a key-
board, while the output system usually consists of a monitor that can
display a large mount of pixels. The computer is enclosed in a chassis
large enough for putting various computer components such as a proces-
sor, a motherboard, a power supply, a hard drive, etc.

desktop computer

Figure 3.1.2: A typical desktop
computer.

3.1.3 Mobile Computer

A *mobile computer* is similar to a desktop computer with fewer resources but can be carried around.

mobile computer
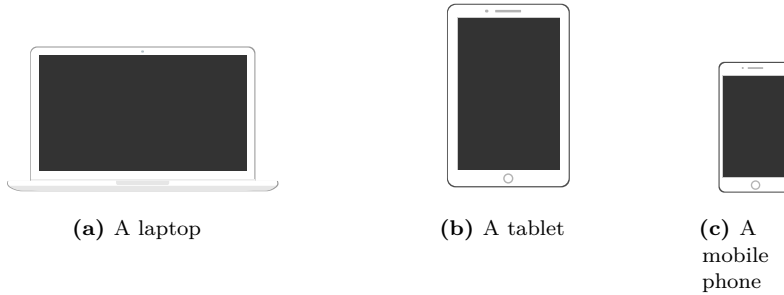
Figure 3.1.3: Mobile computers

(a) A laptop   **(b)** A tablet   **(c)** A mobile phone

3.1.4 Game Consoles

Game consoles are similar to desktop computers but are optimized for gaming. Instead of a keyboard and a mouse, the input system of a game console are game controllers, which is a device with a few buttons for controlling on-screen objects; the output system is a television. The chassis is similar to a desktop computer but is smaller. Game consoles use custom processors and graphic processors but are similar to ones in desktop computers. For example, the first Xbox uses a custom Intel Pentium III processor.

(a) A Play Station 4   **(b)** A Xbox One   **(c)** A Wii U

Figure 3.1.4: Current-gen Game Consoles

Handheld game consoles are similar to game consoles, but incorporate both the input and output systems along with the computer in a single package.

(a) A Nintendo DS

(b) A PS Vita

Figure 3.1.5: Some Handheld Consoles

3.1.5 Embedded Computer

An *embedded computer* is a single-board or single-chip computer with limited resources designed for integrating into larger hardware devices.

A *microcontroller* is an embedded computer designed for controlling other hardware devices. A microcontroller is mounted on a chip. Microcontrollers are general-purpose computers, but with limited resources so that it is only able to perform one or a few specialized tasks. These computers are used for a single purpose, but they are still general-purpose since it is possible to program them to perform different tasks, depends on the requirements, without changing the underlying hardware.

Another type of embedded computer is *system-on-chip*. A *system-on-chip* is a full computer on a single chip. Though a microcontroller is housed on a chip, its purpose is different: to control some hardware. A microcontroller is usually simpler and more limited in hardware resources as it specializes only in one purpose when running, whereas a system-on-chip is a general-purpose computer that can serve multiple purposes. A system-on-chip can run like a regular desktop computer that is capable of loading an operating system and run various applications. A system-on-chip typically presents in a smartphone, such as Apple A5 SoC used in Ipad2 and iPhone 4S, or Qualcomm Snapdragon used in many Android phones.

Be it a microcontroller or a system-on-chip, there must be an environment where these devices can connect to other devices. This environment is a circuit board called a *PCB* – *P*rinted *C*ircuit *B*oard. A *printed circuit board* is a physical board that contains lines and pads to enable electron flows between electrical and electronics components. Without a PCB, devices cannot be combined to create a larger device. As long as these

embedded computer

Figure 3.1.6: An Intel 82815 Graphics and Memory Controller Hub embedded on a PC motherboard. (Source: Wikimedia, author: Qurren)

Figure 3.1.7: A PIC microcontroller. (Soure: Microchip)

microcontroller

Figure 3.1.8: Apple A5 SoC

devices are hidden inside a larger device and contribute to a larger device that operates at a higher level layer for a higher level purpose, they are embedded devices. Writing a program for an embedded device is therefore called *embedded programming*. Embedded computers are used in automatically controlled devices including power tools, toys, implantable medical devices, office machines, engine control systems, appliances, remote controls and other types of embedded systems.
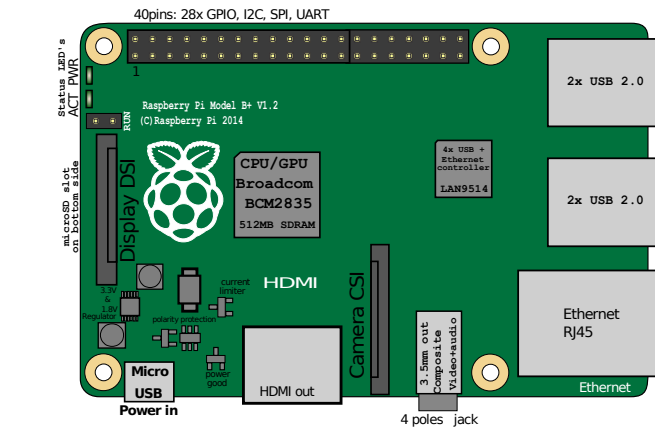
(a) Functional View.
The SoC is a Broadcom BCM2835.
The microcontroller is the Ethernet Controller LAN9514.
(Source: Wikimedia, author: Efa2)

(b) Physical View

Figure 3.1.9: Raspberry Pi B+ Rev 1.2, a single-board computer that includes both a system-on-chip and a microcontroller.

The line between a microcontroller and a system-on-chip is blurry. If hardware keeps evolving more powerful, then a microcontroller can get enough resources to run a minimal operating system on it for multiple specialized purposes. In contrast, a system-on-chip is powerful enough to handle the job of a microcontroller. However, using a system-on-chip as a microcontroller would not be a wise choice as price will rise significantly, but we also waste hardware resources since the software written for a microcontroller requires little computing resources.

3.1.6 Field Gate Programmable Array

Field Programmable Gate Array (*FPGA*) is a hardware an array of reconfigurable gates that makes circuit structure programmable after it is shipped away from the factory[1]. Recall that in the previous chapter, each 74HC00 chip can be configured as a gate, and a more sophisticated device can be built by combining multiple 74HC00 chips. In a similar

Field Programmable Gate Array

[1] This is why it is called **Field** Gate Programmable Array. It is changeable "in the field" where it is applied.

manner, each FPGA device contains thousands of chips called *logic blocks*, which is a more complicated chip than a 74HC00 chip that can be configured to implement a Boolean logic function. These logic blocks can be chained together to create a high-level hardware feature. This high-level feature is usually a dedicated algorithm that needs high-speed processing.
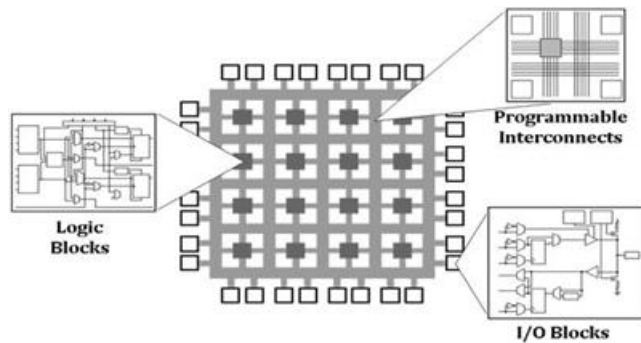
Figure 3.1.10: FPGA Architecture (Source: National Instruments)

Digital devices can be designed by combining logic gates, without regarding actual circuit components, since the physical circuits are just multiples of CMOS circuits. Digital hardware, including various components in a computer, is designed by writing code, like a regular programmer, by using a language to describe how gates are wired together. This language is called a *Hardware Description Language*. Later the hardware description is compiled to a description of connected electronic components called a *netlist*, which is a more detailed description of how gates are connected.

The difference between FPGA and other embedded computers is that programs in FPGA are implemented at the digital logic level, while programs in embedded computers like microcontrollers or system-on-chip devices are implemented at assembly code level. An algorithm written for a FPGA device is a description of the algorithm in logic gates, which the FPGA device then follows the description to configure itself to run the algorithm. An algorithm written for a microcontroller is in assembly instructions that a processor can understand and act accordingly.

FPGA is applied in the cases where the specialized operations are unsuitable and costly to run on a regular computer such as real-time medical image processing, cruise control system, circuit prototyping, video en-

coding/decoding, etc. These applications require high-speed processing that is not achievable with a regular processor because a processor wastes a significant amount of time in executing many non-specialized instructions - which might add up to thousands of instructions or more - to implement a specialized operation, thus more circuits at physical level to carry the same operation. A FPGA device carries no such overhead; instead, it runs a single specialized operation implemented in hardware directly.

3.1.7 Application-Specific Integrated Circuit

An **A**pplication-**S**pecific **I**ntegrated **C**ircuit (or *ASIC*) is a chip designed for a particular purpose rather than for general-purpose use. ASIC does not contain a generic array of logic blocks that can be reconfigured to adapt to any operation like an FPGA; instead, every logic block in an ASIC is made and optimized for the circuit itself. FPGA can be considered as the prototyping stage of an ASIC, and ASIC as the final stage of circuit production. ASIC is even more specialized than FPGA, so it can achieve even higher performance. However, ASICs are very costly to manufacture and once the circuits are made, if design errors happen, everything is thrown away, unlike the FPGA devices which can simply be reprogrammed because of the generic gate array.

3.2   Computer Architecture

The previous section examined various classes of computers. Regardless of shapes and sizes, every computer is designed for an architect from high level to low level.

$$Computer\,Architecture = Instruction\,Set\,Architecture + Computer\,Organization + Hardware$$

At the highest-level is the Instruction Set Architecture.

At the middle-level is the Computer Organization.

At the lowest-level is the Hardware.

3.2.1 Instruction Set Architecture

An *instruction set* is the basic set of commands and instructions that a microprocessor understands and can carry out.

An **I***nstruction* **S***et* **A***rchitecture*, or **ISA**, is the design of an environment that implements an instruction set. Essentially, a runtime environment similar to those interpreters of high-level languages. The design includes all the instructions, registers, interrupts, memory models (how memory are arranged to be used by programs), addressing modes, I/O, etc., of a CPU. The more features (e.g. more instructions) a CPU has, the more circuits are required to implement it.

3.2.2 Computer organization

Computer organization is the functional view of the design of a computer. In this view, hardware components of a computer are presented as boxes with input and output that connects to each other and form the design of a computer. Two computers may have the same ISA, but different organizations. For example, both AMD and Intel processors implement x86 ISA, but the hardware components of each processor that make up the environments for the ISA are not the same.

Computer organization

Computer organizations may vary depend on a manufacturer's design, but they are all originated from the Von Neumann architecture[2]:
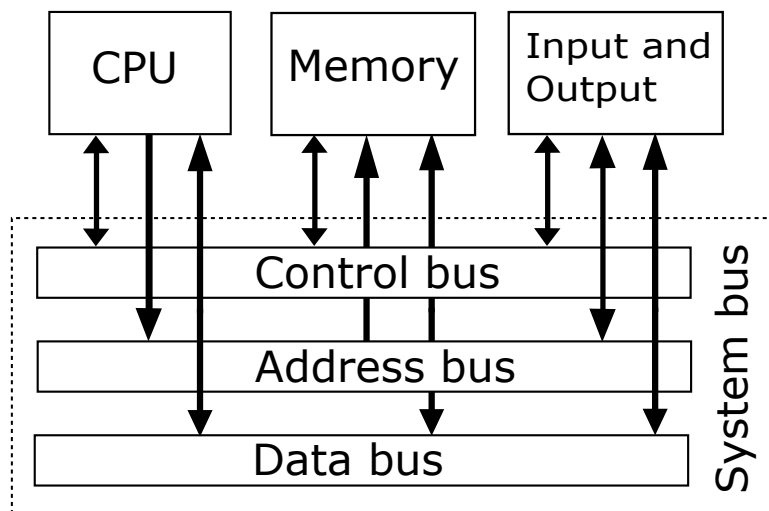
[2] *John von Neumann* was a mathematician and physicist who invented a computer architecture.

Figure 3.2.1: Von-Neumann Architecture

CPU fetches instructions continuously from main memory and execute.

Memory stores program code and data.

Bus are electrical wires for sending raw bits between the above components.

I/O Devices are devices that give input to a computer i.e. keyboard, mouse, sensor, etc, and takes the output from a computer i.e. monitor takes information sent from CPU to display it, LED turns on/off according to a pattern computed by CPU, etc.

The Von-Neumann computer operates by storing its instructions in main memory, and CPU repeatedly fetches those instructions into its internal storage for executing, one after another. Data are transferred through a data bus between CPU, memory and I/O devices, and where to store in the devices is transferred through the address bus by the CPU. This architecture completely implements the *fetch – decode – execute* cycle.

The earlier computers were just the exact implementations of the Von Neumann architecture, with CPU and memory and I/O devices communicate through the same bus. Today, a computer has more buses, each is specialized in a type of traffic. However, at the core, they are still Von Neumann architecture. To write an OS for a Von Neumann computer, a programmer needs to be able to understand and write code that controls the cores components: CPU, memory, I/O devices, and bus.

CPU, or **C**entral **P**rocessing **U**nit, is the heart and brain of any computer system. Understand a CPU is essential to writing an OS from scratch:

▷ To use these devices, a programmer needs to controls the CPU to use the programming interfaces of other devices. CPU is the only way, as CPU is the only direct device a programmer can use and the only device that understand code written by a programmer.

▷ In a CPU, many OS concepts are already implemented directly in hardware, e.g. task switching, paging. A kernel programmer needs to know how to use the hardware features, to avoid duplicating such concept in software, thus wasting computer resources.

▷ CPU built-in OS features boost both OS performance and developer productivity because those features are actual hardware, the lowest possible level, and developers are free to implement such features.

▷ To effectively use the CPU, a programmer needs to understand the documentation provided from CPU manufacturer. For example, Intel® 64 and IA-32 Architectures Software Developer Manuals.

▷ After understanding one CPU architecture well, it is easier to learn other CPU architectures.

A CPU is an implementation of an ISA, effectively the implementation of an assembly language (and depending on the CPU architecture, the language may vary). Assembly language is one of the interfaces that are provided for software engineers to control a CPU, thus control a computer. But how can every computer device be controlled with only the access to the CPU? The simple answer is that a CPU can communicate with other devices through these two interfaces, thus commanding them:

Registers are a hardware component for high-speed data access and com-   ***Registers***
  munication with other hardware devices. Registers allow software to control hardware directly by writing to registers of a device, or receive information from hardware device when reading from registers of a device.

  Not all registers are used for communication with other devices. In a CPU, most registers are used as high-speed storage for temporary data. Other devices that a CPU can communicate always have a set of registers for interfacing with the CPU.

Port is a specialized register in a hardware device used for communica-   ***Port***
  tion with other devices. When data are written to a port, it causes a hardware device to perform some operation according to values written to the port. The different between a port and a register is that port does not store data, but delegate data to some other circuit.

These two interfaces are extremely important, as they are the only interfaces for controlling hardware with software. Writing device drivers is essentially learning the functionality of each register and how to use them properly to control the device.

  Memory is a storage device that stores information. Memory consists   ***Memory***
of many cells. Each cell is a byte with its address number, so a CPU can

use such address number to access an exact location in memory. Memory
is where software instructions (in the form of machine language) is stored
and retrieved to be executed by CPU; memory also stores data needed
by some software. Memory in a Von Neumann machine does not distin-
guish between which bytes are data and which bytes are software instruc-
tions. It's up to the software to decide, and if somehow data bytes are
fetched and executed as instructions, CPU still does it if such bytes rep-
resents valid instructions, but will produce undesirable results. To a CPU,
there's no code and data; both are merely different types of data for it
to act on: one tells it how to do something in a specific manner, and one
is necessary materials for it to carry such action.

The RAM is controlled by a device called a *memory controller*. Currently,
most processors have this device embedded, so the CPU has a dedicated
memory bus connecting the processor to the RAM. On older CPU[3], how-
ever, this device was located in a chip also known as **MCH** or ***M**emory*
***C**ontroller **H**ub*. In this case, the CPU does not communicate directly
to the RAM, but to the MCH chip, and this chip then accesses the mem-
ory to read or write data. The first option provides better performance
since there is no middleman in the communications between the CPU
and the memory.

[3] Prior to the CPU's produced in 2009

(a) Old CPU

(b) Modern CPU
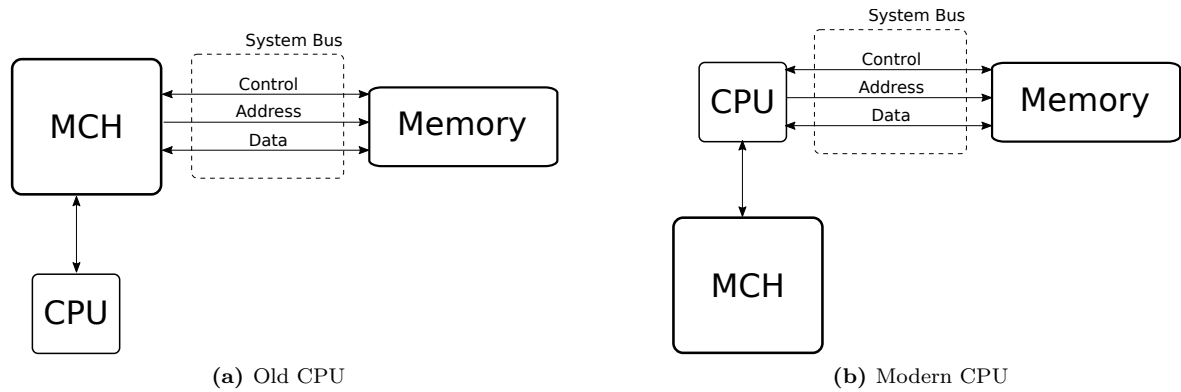
Figure 3.2.2: CPU - Memory
Communication

capacitor

At the physical level, RAM is implemented as a grid of cells that each
contain a transistor and an electrical device called a *capacitor*, which stores
charge for short periods of time. The transistor controls access to the ca-
pacitor; when switched on, it allows a small charge to be read from or
written to the capacitor. The charge on the capacitor slowly dissipates,

requiring the inclusion of a refresh circuit to periodically read values from the cells and write them back after amplification from an external power source.

Bus is a subsystem that transfers data between computer components or between computers. Physically, buses are just electrical wires that connect all components together and each wire transfer a single big chunk of data. The total number of wires is called *bus width*, and is dependent on how many wires a CPU can support. If a CPU can only accept 16 bits at a time, then the bus has 16 wires connecting from a component to the CPU, which means the CPU can only retrieve 16 bits of data a time.

Bus

bus width

3.2.3 Hardware

Hardware is a specific implementation of a computer. A line of processors implement the same instruction set architecture and use nearly identical organizations but differ in hardware implementation. For example, the Core i7 family provides a model for desktop computers that is more powerful but consumes more energy, while another model for laptops is less performant but more energy efficient. To write software for a hardware device, seldom we need to understand a hardware implementation if documents are available. Computer organization and especially the instruction set architecture are more relevant to an operating system programmer. For that reason, the next chapter is devoted to study the x86 instruction set architecture in depth.

3.3   x86 architecture

A *chipset* is a chip with multiple functions. Historically, a chipset is actually a set of individual chips, and each is responsible for a function, e.g. memory controller, graphic controllers, network controller, power controller, etc. As hardware progressed, the set of chips were incorporated into a single chip, thus more space, energy, and cost efficient. In a desktop computer, various hardware devices are connected to each other through a PCB called a *motherboard*. Each CPU needs a compatible motherboard that can host it. Each motherboard is defined by its chipset model that

determine the environment that a CPU can control. This environment typically consists of

▷ a slot or more for CPU

▷ a chipset of two chips which are the Northbridge and Southbridge chips

 − Northbridge chip is responsible for the high-performance communication between CPU, main memory and the graphic card.

 − Southbridge chip is responsible for the communication with I/O devices and other devices that are not performance sensitive.

▷ slots for memory sticks

▷ a slot or more for graphic cards.

▷ generic slots for other devices, e.g. network card, sound card.

▷ ports for I/O devices, e.g. keyboard, mouse, USB.

   To write a complete operating system, a programmer needs to understand how to program these devices. After all, an operating system manages hardware automatically to free application programs doing so. However, of all the components, learning to program the CPU is the most important, as it is the component present in any computer, regardless of what type a computer is. For this reason, the primary focus of this book will be on how to program an x86 CPU. Even solely focused on this device, a reasonably good minimal operating system can be written. The reason is that not all computers include all the devices as in a normal desktop computer. For example, an embedded computer might only have a CPU and limited internal memory, with pins for getting input and producing an output; yet, operating systems were written for such devices.

   However, learning how to program an x86 CPU is a daunting task, with 3 primary manuals written for it: almost 500 pages for volume 1, over 2000 pages for volume 2 and over 1000 pages for volume 3. It is an impressive feat for a programmer to master every aspect of x86 CPU programming.
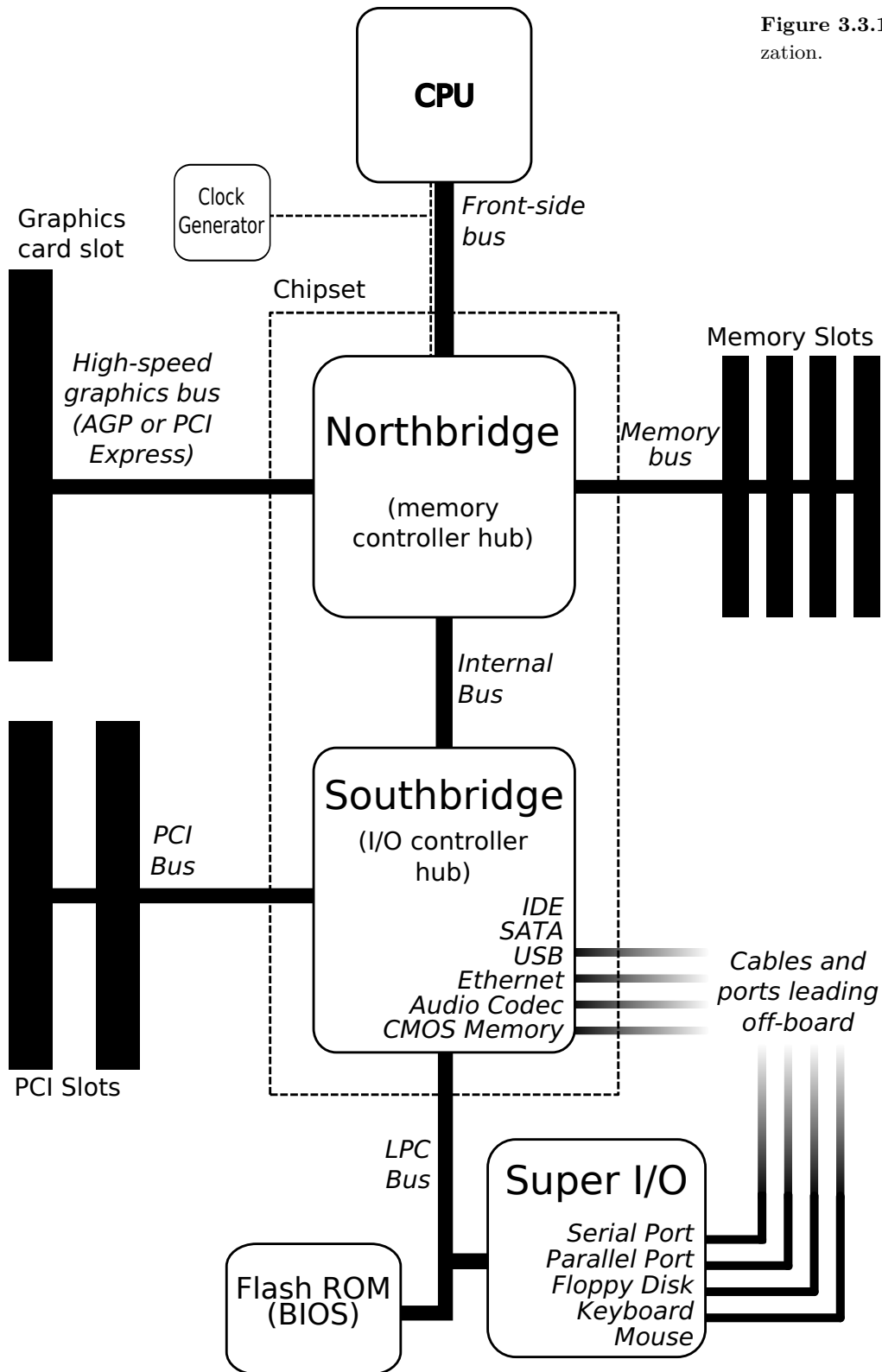
Figure 3.3.1: Motherboard organization.

CPU

Clock
Generator

Graphics
card slot

*Front-side
bus*

Chipset

Memory Slots

*High-speed
graphics bus
(AGP or PCI
Express)*

Northbridge

(memory
controller hub)

*Memory
bus*

*Internal
Bus*

Southbridge

(I/O controller
hub)

*PCI
Bus*

*IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory*

*Cables and
ports leading
off-board*

PCI Slots

*LPC
Bus*

Super I/O

*Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse*

Flash ROM
(BIOS)

3.4   Intel Q35 Chipset

Q35 is an Intel chipset released September 2007. Q35 is used as an example of a high-level computer organization because later we will use QEMU to emulate a Q35 system, which is latest Intel system that QEMU can emulate. Though released in 2007, Q35 is relatively modern to the current hardware, and the knowledge can still be reused for current chipset model. With a Q35 chipset, the emulated CPU is also relatively up-to-date with features presented in current day CPUs so we can use the latest software manuals from Intel.

Figure 3.3.1 on the facing page is a typical current-day motherboard organization, in which Q35 shares similar organization.

3.5   x86 Execution Environment

An *execution environment* is an environment that provides the facility to make code executable. The execution environment needs to address the following question:

▷ SUPPORTED OPERATIONS? data transfer, arithmetic, control, floating-point, etc.

▷ WHERE ARE OPERANDS STORED? registers, memory, stack, accumulator

▷ HOW MANY EXPLICIT OPERANDS ARE THERE FOR EACH INSTRUCTION? 0, 1, 2, or 3

▷ HOW IS THE OPERAND LOCATION SPECIFIED? register, immediate, indirect, etc.

▷ WHAT TYPE AND SIZE OF OPERANDS ARE SUPPORTED? byte, int, float, double, string, vector, etc.

▷ ETC.

For the remain of this chapter, please carry on the reading to chapter 3 in Intel Manual Volume 1, *"Basic Execution Environment"* .

4

x86 Assembly and C

In this chapter, we will explore assembly language, and how it connects to C. But why should we do so? Isn't it better to trust the compiler, plus no one writes assembly anymore?

Not quite. Surely, the compiler at its current state of the art is trustworthy, and we do not need to write code in assembly, *most of the time.* A compiler can generate code, but as mentioned previously, a high-level language is a collection of patterns of a lower-level language. It does not cover everything that a hardware platform provides. As a consequence, not every assembly instruction can be generated by a compiler, so we still need to write assembly code for these circumstances to access hardware-specific features. Since hardware-specific features require writing assembly code, debugging requires reading it. We might spend even more time reading than writing. Working with low-level code that interacts directly with hardware, assembly code is unavoidable. Also, understand how a compiler generates assembly code could improve a programmer's productivity. For example, if a job or school assignment requires us to write assembly code, we can simply write it in C, then let `gcc` does the hard working of writing the assembly code for us. We merely collect the generated assembly code, modify as needed and be done with the assignment.

We will learn `objdump` extensively, along with how to use Intel documents to aid in understanding x86 assembly code.

4.1   objdump

`objdump` is a program that displays information about object files. It will be handy later to debug incorrect layout from manual linking. Now, we use `objdump` to examine how high level source code maps to assembly code. For now, we ignore the output and learn how to use the command first. Supposed that we have a executable binary named `hello` compiled from a `hello.c` thath prints "Hello World', it is simple to use `objdump` :

```
$ objdump -d hello
```

`-d` option only displays assembled contents of executable sections. A *section* is a block of memory that contains either program code or data. A code section is executable by the CPU, while a data section is not executable. Non-executable sections, such as `.data` and `.bss` (for storing program data), debug sections, etc, are not displayed. We will learn more about section when studying ELF binary file format in chapter 5 on page 107 . On the other hand:

```
$ objdump -D hello
```

where `-D` option displays assembly contents of all sections. If `-D`, `-d` is implicitly assumed. `objdump` is mostly used for inspecting assembly code, so `-d` is the most useful and thus is set by default.

The output overruns the terminal screen. To make it easy for reading, send all the output to `less`:

```
$ objdump -d hello | less
```

To intermix source code and assembly, the binary must be compiled with `-g` option to include source code in it, then add `-S` option:

```
$ objdump -S hello | less
```

The default syntax used by `objdump` is AT&T syntax. To change it to the familiar Intel syntax:

```
$ objdump -M intel -D hello | less
```

When using `-M` option, option `-D` or `-d` must be explicitly supplied. Next, we will use `objdump` to examine how compiled C data and code are represented in machine code.

Finally, we will write a 32-bit kernel, therefore we will need to compile a 32-bit binary and examine it in 32-bit mode:

```
$ objdump -M i386,intel -D hello | less
```

`-M i386` tells objdump to display assembly content using 32-bit layout. Knowing the difference between 32-bit and 64-bit is crucial for writing kernel code. We will examine this matter later on when writing our kernel.

4.2   Reading the output

At the start of the output displays the file format of the object file:

```
hello: file format elf64-x86-64
```

After the line is a series of disassembled sections:

```
Disassembly of section .interp:
...
Disassembly of section .note.ABI-tag:
...
Disassembly of section .note.gnu.build-id:
...
...
etc
```

Finally, each disassembled section displays its actual content - which is a sequence of assembly instructions - with the following format:

```
4004d6:        55                      push   rbp
```

▷ The first column is the address of an assembly instruction. In the above example, the address is `0x4004d6`.

▷ The second column is assembly instruction in raw hex values. In the above example, the value is `0x55`.

▷ The third column is the assembly instruction. Depends on the section, the assembly instruction might be meaningful or meaningless. For example, if the assembly instructions are in a `.text` section, then the assembly instructions are actual program code. On the other hand, if the assembly instructions are displayed in a `.data` section, then we can safely ignore the displayed instructions. The reason is that `objdump` doesn't know which hex values are code and which are data, so it blindly translates every hex values into assembly instructions. In the above example, the assembly instruction is `push %rbp`.

▷ The optional fourth column is a comment - appears when there is a reference to an address - to inform where the address originates. For example, the comment in blue:

```
lea r12,[rip+0x2008ee] # 600e10 <__frame_dummy_init_array_entry>
```

is to inform that the referenced address from `[rip+0x2008ee]` is `0x600e10`, where the variable `__frame_dummy_init_array_entry` resides.

In a disassembled section, it may also contain *labels*. A label is a name given to an assembly instruction. The label denotes the purpose of an assembly block to a human reader, to make it easier to understand. For example, `.text` section carries many of such labels to denote where code in a program start; `.text` section below carries two functions: `_start` and `deregister_tm_clones`. The `_start` function starts at address `4003e0`, is annotated to the left of the function name. Right below `_start` label is also the instruction at address `4003e0`. This whole thing means that a label is simply a name of a memory address. The function `deregister_tm_clones` also shares the same format as every function in the section.

```
00000000004003e0 <_start>:
  4003e0:       31 ed                     xor     ebp,ebp
  4003e2:       49 89 d1                  mov     r9,rdx
  4003e5:       5e                        pop     rsi
...more assembly code....
0000000000400410 <deregister_tm_clones>:
  400410:       b8 3f 10 60 00            mov     eax,0x60103f
  400415:       55                        push    rbp
  400416:       48 2d 38 10 60 00         sub     rax,0x601038
...more assembly code....
```

4.3 Intel manuals

The best way to understand and use assembly language properly is to understand precisely the underlying computer architecture and what each machine instruction does. To do so, the most reliable source is to refer to documents provided by vendors. After all, hardware vendors are the one who made their machines. To understand Intel's instruction set, we need the document "*Intel 64 and IA-32 architectures software developer's manual combined volumes 2A, 2B, 2C, and 2D: Instruction set reference, A-Z*". The document can be retrieved here: https://software.intel.com/en-us/articles/intel-sdm.

▷ Chapter 1 provides brief information about the manual, and the comment notations used in the book.

▷ Chapter 2 provides an in-depth explanation of the anatomy of an assembly instruction, which we will investigate in the next section.

▷ Chapter 3 - 5 provide the details of every instruction of the x86_64 architecture.

▷ Chapter 6 provides information about safer mode extensions. We won't need to use this chapter.

The first volume "*Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*" describes the basic architecture and programming environment of Intel processors. In the book, Chapter

5 gives the summary of all Intel instructions, by listing instructions into different categories. We only need to learn general-purpose instructions listed *chapter 5.1* for our OS. *Chapter 7* describes the purpose of each category. Gradually, we will learn all of these instructions.

Exercise 4.3.1. Read section 1.3 in volume 2, exclude sections 1.3.5 and 1.3.7.

4.4   Experiment with assembly code

The subsequent sections examine the anatomy of an assembly instruction. To fully understand, it is necessary to write code and see the code in its actual form displayed as hex numbers. For this purpose, we use `nasm` assembler to write a few line of assembly code and see the generated code.

Example 4.4.1. Suppose we want to see the machine code generated for this instruction:

```
jmp eax
```

Then, we use an editor e.g. Emacs, then create a new file, write the code and save it in a file, e.g. `test.asm`. Then, in the terminal, run the command:

```
$ nasm -f bin test.asm -o test
```

`-f` option specifies the file format, e.g. ELF, of the final output file. But in this case, the format is `bin`, which means this file is just a flat binary output without any extra information. That is, the written assembly code is translated to machine code as is, without the overhead of the metadata from file format like ELF. Indeed, after compiling, we can examine the output using this command:

```
$ hd test
```

hd (short for hexdump) is a program that displays the content of a file in hex format. And get the following output:

```
00000000  66 ff e0                                           |f..|
00000003
```

Though its name is short for hexdump, hd can display in different base, e.g. binary, other than hex.

The file only consists of 3 bytes: 66 ff e0, which is equivalent to the instruction jmp eax.

Example 4.4.2. If we were to use elf as file format:

```
$ nasm -f elf test.asm -o test
```

It would be more challenging to learn and understand assembly instructions with all the added noise[1]:

[1] The output from hd.

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  40 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |@.......4.....(.|
00000030  05 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000060  00 00 00 00 00 00 00 00  01 00 00 00 01 00 00 00  |................|
00000070  06 00 00 00 00 00 00 00  10 01 00 00 02 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00  10 00 00 00 00 00 00 00  |................|
00000090  07 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  20 01 00 00 21 00 00 00  00 00 00 00 00 00 00 00  | ...!...........|
000000b0  01 00 00 00 00 00 00 00  11 00 00 00 02 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  50 01 00 00 30 00 00 00  |........P...0...|
000000d0  04 00 00 00 03 00 00 00  04 00 00 00 10 00 00 00  |................|
000000e0  19 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000000f0  80 01 00 00 0d 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

```
00000110  ff e0 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  00 2e 74 65 78 74 00 2e  73 68 73 74 72 74 61 62  |..text..shstrtab|
00000130  00 2e 73 79 6d 74 61 62  00 2e 73 74 72 74 61 62  |..symtab..strtab|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000160  01 00 00 00 00 00 00 00  00 00 00 00 04 00 f1 ff  |................|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 03 00 01 00  |................|
00000180  00 74 65 73 74 2e 61 73  6d 00 00 00 00 00 00 00  |.disp8-5.asm....|
00000190
```

Thus, it is better just to use flat binary format in this case, to experiment instruction by instruction.

With such a simple workflow, we are ready to investigate the structure of every assembly instruction.

Note: Using the bin format puts `nasm` by default into 16-bit mode. To enable 32-bit code to be generated, we must add this line at the beginning of an `nasm` source file:

```
bits 32
```

4.5   Anatomy of an Assembly Instruction

Chapter 2 of the instruction reference manual provides an in-depth of view of instruction format. But, the information is too much that it can overwhelm beginners. This section provides an easier instruction before reading the actual chapter in the manual.

Recall that an assembly instruction is simply a fixed-size series of bits. The length of an instruction varies and depends on how complicated an instruction is. What every instruction shares is a common format described in the figure above that divides the bits of an instruction into smaller parts that encode different types of information. These parts are:

Instruction Prefixes appears at the beginning of an instruction. Prefixes
   are optional. A programmer can choose to use a prefix or not because
   in practice, a so-called prefix is just another assembly instruction to

Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Prefixes of 1 byte each (optional)[1,2]	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2 or 4 bytes or none[3]	Immediate data of 1, 2 or 4 bytes or none[3]

7	6 5		3 2		0
Mod	Reg/ Opcode		R/M		

7	6 5		3 2		0
Scale	Index		Base		
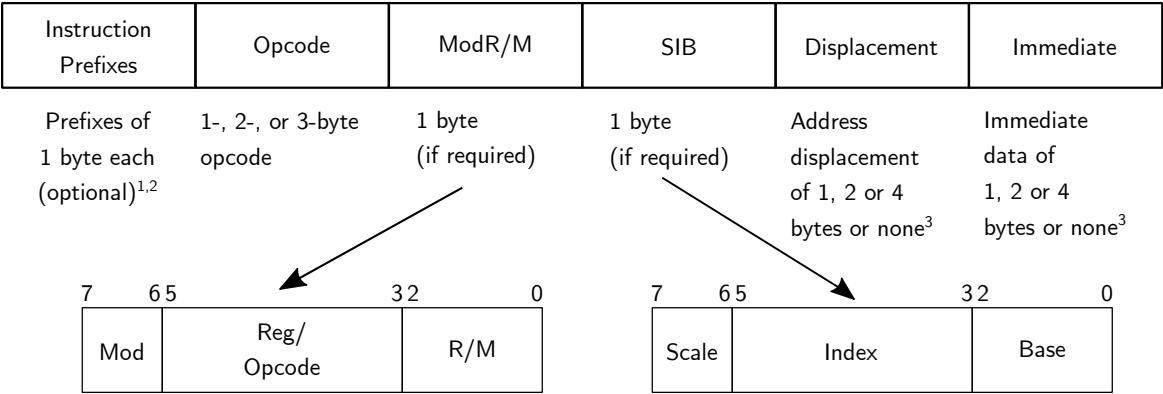
1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, "REX Prefixes" in the manual for additional information.
2. For VEX encoding information, see Section 2.3, "Intel® Advanced Vector Extensions (Intel® AVX)" in the manual.
3. Some rare instructions can take an 8B immediate or 8B displacement.

Figure 4.5.1: Intel 64 and IA-32 Architectures Instruction Format

be inserted before another assembly instruction that such prefix is applicable. Instructions with 2 or 3-bytes opcodes include the prefixes by default.

Opcode is a unique number that identifies an instruction. Each opcode is given an mnemonic name that is human readable, e.g. one of the opcodes for instruction `add` is 04. When a CPU sees the number 04 in its instruction cache, it sees instruction `add` and execute accordingly. Opcode can be 1,2 or 3 bytes long and includes an additional 3-bit field in the `ModR/M` byte when needed.

Example 4.5.1. This instruction:

```
jmp [0x1234]
```

generates the machine code:

```
ff 26 34 12
```

The very first byte, `0xff` is the opcode, which is unique to `jmp` instruction.

ModR/M specifies operands of an instruction. Operand can either be a

register, a memory location or an immediate value. This component
of an instruction consists of 3 smaller parts:

▷ *mod* field, or *modifier* field, is combined with `r/m` field for a total of
5 bits of information to encode 32 possible values: 8 registers and
24 addressing modes.

▷ *reg/opcode* field encodes either a register operand, or extends the
`Opcode` field with 3 more bits.

▷ *r/m* field encodes either a register operand or can be combined with
`mod` field to encode an addressing mode.

The tables 4.5.1 and 4.5.2 list all possible 256 values of `ModR/M` byte
and how each value maps to an addressing mode and a register, in 16-
bit and 32-bit modes.

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP[1]	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Values of ModR/M Byte (In Hexadecimal)**							
[BX + SI]	00	000	00	08	10	18	20	28	30	38
[BX + DI]		001	01	09	11	19	21	29	31	39
[BP + SI]		010	02	0A	12	1A	22	2A	32	3A
[BP + DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX + SI] + disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX + DI] + disp8		001	41	49	51	59	61	69	71	79
[BP + SI] + disp8		010	42	4A	52	5A	62	6A	72	7A
[BP + DI] + disp8		011	43	4B	53	5B	63	6B	73	7B
[SI] + disp8		100	44	4C	54	5C	64	6C	74	7C
[DI] + disp8		101	45	4D	55	5D	65	6D	75	7D
[BP] + disp8		110	46	4E	56	5E	66	6E	76	7E
[BX] + disp8		111	47	4F	57	5F	67	6F	77	7F
[BX + SI] + disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX + DI] + disp16		001	81	89	91	99	A1	A9	B1	B9
[BP + SI] + disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP + DI] + disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI] + disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI] + disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP] + disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX] + disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.

2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.

3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 4.5.1: 16-Bit   Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Values of ModR/M Byte (In Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--]^1		100	04	0C	14	1C	24	2C	34	3C
disp32^2		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX] + disp8^3	01	000	40	48	50	58	60	68	70	78
[ECX] + disp8		001	41	49	51	59	61	69	71	79
[EDX] + disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX] + disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--] + disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP] + disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI] + disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI] + disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX] + disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX] + disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX] + disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX] + disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--] + disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP] + disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI] + disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI] + disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.

2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.

3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 4.5.2: 32-Bit   Addressing Forms with the ModR/M Byte

In an instruction, next to the opcode is a `ModR/M` byte. Then, look up the byte value in this table to get the corresponding operands in the row and column.

Example 4.5.2. An instruction uses this addressing mode:

```
jmp [0x1234]
```

Then, the machine code is:

```
ff 26 34 12
```

`0xff` is the opcode. Next to it, `0x26` is the `ModR/M` byte. Look up in the 16-bit table , the first operand is in the row, equivalent to a `disp16`, which means a 16-bit offset. Since the instruction does not have a second operand, the column can be ignored.

Remember, using `bin` format generates 16-bit code by default

Example 4.5.3. An instruction uses this addressing mode:

```
add eax, ecx
```

Then the machine code is:

```
66 01 c8
```

The interesting feature of this instruction is that `0x66` is the not the opcode. `0x01` is the opcode. So then, what is `0x66`? Recall that for every assembly instruction, there will be an optional instruction prefix, and that is what `0x66` is. According to the Intel manual, vol 1:

> The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

If the CPU is switched to 32-bit mode, when it runs an instruction with `0x66` prefix, the instruction operands are limited to only 16-bit width.

On the other hand, if the CPU is in 16-bit environment, as a result, 32-bit is considered non-standard and as such, instruction operands are temporary upgraded to 32-bit width while the instructions without the prefix use 16-bit operands.

Next to it, `c8` is the `ModR/M` byte. Look up in the 16-bit table at `c8` value, the row tells the first operand is `ax` , the column tells the second operand is `cx`; the column can't be ignored as the second operand is in the instruction.

Remember, using bin format generates 16-bit code by default

Why is the first operand in the row and the second in a column? Let's break down the `ModR/M` byte, with an example value c8, into bits:

mod		reg/opcode			r/m		
1	1	0	0	1	0	0	0

The `mod` field divides addressing modes into 4 different categories. Further combines with the `r/m` field, exactly one addressing mode can be selected from one of the 24 rows. If an instruction only requires one operand, then the column can be ignored. Then the `reg/opcode` field finally provides an extra register or different variants, if an instruction requires one.

SIB is ***S****cale*-***I****ndex*-***B****ase* byte. This byte encodes ways to calculate the memory position into an element of an array. SIB is the name that is based on this formula for calculating an effective address:

$$\texttt{Effective address} = \texttt{scale} * \texttt{index} + \texttt{base}$$

▷ `Index` is an offset into an array.

▷ `Scale` is a factor of `Index`. `Scale` is one of the values 1, 2, 4 or 8; any other value is invalid. To scale with values other than 2, 4 or 8, the scale factor must be set to 1, and the offset must be calculated manually. For example, if we want to get the address of the nth element in an array and each element is `12`-bytes long. Because each element is 12-bytes long instead of 1, 2, 4 or 8, `Scale` is set to 1 and a compiler needs to calculate the offset:

$$\mathtt{Effective\ address} = 1 * (12 * \mathtt{n}) + \mathtt{base}$$

Why do we bother with SIB when we can manually calculate the offset? The answer is that in the above scenario, an additional `mul` instruction must be executed to get the offset, and the `mul` instruction consumes more than 1 byte, while the SIB only consumes 1 byte. More importantly, if the element is repeatedly accessed many times in a loop, e.g. millions of times, then an extra `mul` instruction can detriment the performance as the CPU must spend time executing millions of these additional `mul` instructions.

The values 2, 4 and 8 are not random chosen. They map to 16-bit (or 2 bytes), 32-bit (or 4 bytes) and 64-bit (or 8 bytes) numbers that are often used for intensive numeric calculations.

▷ `Base` is the starting address.

Below is the table listing all 256 values of `SIB` byte, with the lookup rule similar to `ModR/M` tables:

Example 4.5.4. This instruction:

```
jmp [eax*2 + ebx]
```

generates the following code:

```
00000000  67 ff 24 43
```

First of all, the first byte, `0x67` is *not* an opcode but a *prefix*. The number is a predefined prefix for address-size override prefix. After the prefix, comes the opcode `0xff` and the `ModR/M` byte `0x24`. The value from `ModR/M` suggests that there exists a `SIB` byte that follows. The `SIB` byte is `0x43`.

Look up in the `SIB` table, the row tells that `eax` is scaled by 2, and the column tells that the base to be added is in `ebx`.

Displacement is the offset from the start of the base index.

Example 4.5.5. This instruction:

```
jmp [0x1234]
```

r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**SS**	**R/M**	**Values of SIB Byte (In Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

Table 4.5.3: 32-Bit   Addressing Forms with the SIB Byte

generates machine code is:

```
ff 26 34 12
```

0x1234, which is generated as 34 12 in raw machine code, is the displacement and stands right next to 0x26, which is the ModR/M byte.

Example 4.5.6. This instruction:

```
jmp [eax * 4 + 0x1234]
```

generates the machine code:

```
67 ff 24 85 34 12 00 00
```

▷ 0x67 is an address-size override prefix. Its meaning is that if an instruction runs a default address size e.g. 16-bit, the use of prefix enables the instruction to use non-default address size, e.g. 32-bit or 64-bit. Since the binary is supposed to be 16-bit, 0x67 changes the instruction to 32-bit mode.

▷ 0xff is the opcode.

▷ 0x24 is the ModR/M byte. According to table 4.5.2, the value suggests that a SIB byte follows, .

▷ 0x85 is the SIB byte. According to table 4.5.3, the byte 0x85 can be destructured into bits as follow:

SS		R/M			REG		
1	0	0	0	0	1	0	1

The above values are obtained through the columns **SS**, **R/M** and finally the 8 column of **REG** respectively. The total bits combined into the value 10000101, which is 0x85 in hex value. By default, if a register after the displacement is not specified, it is set to EBP register, and thus the 6th column (bit pattern 101) is always chosen. If the example uses another register:

Example 4.5.7. For example:

```
jmp [eax * 4 + eax + esi]
```

the SIB byte becomes 0x86 instead of , which is in the 7th column. Try to verify with the table 4.5.3 again.

▷ `34 12 00 00` is the displacement. As can be seen, the displacement is 4 bytes in size, which is equivalent to 32-bit, due to address-size override prefix.

Immediate When an instruction accepts a fixed value, e.g. `0x1234`, as an operand, this optional field holds the value. Note that this field is different from displacement: the value is not necessary used an offset, but an arbitrary value of anything.

Example 4.5.8. This instruction:

```
mov eax, 0x1234
```

generates the code:

```
66 b8 34 12 00 00
```

▷ `0x66` is operand-sized override prefix. Similar to address-size override prefix, this prefix enables operand-size to be non-default.

▷ `0xb8` is one of the opcodes for `mov` instruction.

▷ `0x1234` is the value to be stored in register `eax`. It is just a value for storing directly into a register, and nothing more. On the other hand, displacement value is an offset for some address calculation.

Exercise 4.5.1. Read section 2.1 in Volume 2 for even more details.

Exercise 4.5.2. Skim through section 5.1 in volume 1. Read chapter 7 in volume 1. If there are terminologies that you don't understand e.g. segmentation, don't worry as the terms will be explained in later chapters or ignored.

4.6   Understand an instruction in detail

In the instruction reference manual (Volume 2), from chapter 3 onward, every x86 instruction is documented in detail. Whenever the precise behavior of an instruction is needed, we always consult this document first. However, before using the document, we must know the writing conventions first. Every instruction has the following common structure for organizing information:

`Opcode table` lists all possible opcodes of an assembly instruction.

Each table contains the following fields, and can have one or more rows:

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID	Description
				Feature flag	

Opcode shows a unique hexadecimal number assigned to an instruction. There can be more than one opcode for an instruction, each encodes a variant of the instruction. For example, one variant requires one operand, but another requires two. In this column, there can be other notations aside from hexadecimal numbers. For example, `/r` indicates that the `ModR/M` byte of the instruction contains a `reg` operand and an `r/m` operand. The detail listing is in section *3.1.1.1* and *3.1.1.2* in the Intel's manual, volume 2.

Instruction gives the syntax of the assembly instruction that a programmer can use for writing code. Aside from the mnemonic representation of the opcode, e.g. `jmp`, other symbols represent operands with specific properties in the instruction. For example, `rel8` represents a relative address from 128 bytes before the end of the instruction to 127 bytes after the end of instruction; similarly `rel16/rel32` also represents relative addresses, but with the operand size of 16/32-bit instead of 8-bit like `rel8`. For a detailed listing, please refer to section `3.1.1.3` of volume 2.

Op/En is short for ***Op**erand/**En**coding*. An operand encoding specifies how a `ModR/M` byte encodes the operands that an instruction requires. If a variant of an instruction requires operands, then an additional table named *"Instruction Operand Encoding"* is added for explaining the operand encoding, with the following structure:

Op/En	Operand 1	Operand 2	Operand 3	Operand 4

Most instructions require one to two operands. We make use of these instructions for our OS and skip the instructions that require three or four operands. The operands can be readable or writable or both. The symbol `(r)` denotes a readable operand, and `(w)` denotes a writable operand. For example, when `Operand 1` field contains `ModRM:r/m`

(r), it means the first operand is encoded in `r/m` field of `ModR/M` byte, and is only `readable`.

64/32-bit mode indicates whether the opcode sequence is supported in a 64-bit mode and possibly 32-bit mode.

CPUID Feature Flag indicates a particular CPU feature must be available to enable the instruction. An instruction is invalid if a CPU does not support the required feature.

Compat/Leg Mode Many instructions do not have this field, but instead is replaced with `Compat/Leg Mode`, which stands for *Compatibility or Legacy Mode*. This mode enables 64-bit variants of instructions to run normally in 16 or 32-bit mode.

Description briefly explains the variant of an instruction in the current row.

Description specifies the purpose of the instructions and how an instruction works in detail.

Operation is pseudo-code that implements an instruction. If a description is vague, this section is the next best source to understand an assembly instruction. The syntax is described in section *3.1.1.9* in volume 2.

Flags affected lists the possible changes to system flags in `EFLAGS` register.

Exceptions list the possible errors that can occur when an instruction cannot run correctly. This section is valuable for OS debugging. Exceptions fall into one of the following categories:

▷ Protected Mode Exceptions

▷ Real-Address Mode Exception

▷ Virtual-8086 Mode Exception

▷ Floating-Point Exception

▷ SIMD Floating-Point Exception

▷ Compatibility Mode Exception

In Linux, the command:

`cat /proc/cpuinfo`

lists the information of available CPUs and   its features in `flags` field.

Table 4.6.1: Notations in `Compat/Leg Mode`

Notation	Description
Valid	Supported
I	Not supported
N.E.	The 64-bit opcode cannot be encoded as it overlaps with existing 32-bit opcode.

▷ 64-bit Mode Exception

For our OS, we only use *Protected Mode Exceptions* and *Real-Address Mode Exceptions*. The details are in section *3.1.1.13* and *3.1.1.14*, volume 2.

4.7   Example: `jmp` instruction

Let's look at our good old `jmp` instruction. First, the opcode table:

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
EB cb	JMP rel8	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 cw	JMP rel16	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 cd	JMP rel32	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP r/m16	M	N.S.	Valid	Jump near, absolute indirect, address = zero- extended r/m16. Not supported in 64-bit mode
FF /4	JMP r/m32	M	N.S.	Valid	Jump near, absolute indirect, address given in r/m32. Not supported in 64-bit mode
FF /4	JMP r/m64	M	Valid	N.E	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA cd	JMP ptr16:16	D	Inv.	Valid	Jump far, absolute, address given in operand
EA cp	JMP ptr16:32	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP m16:16	D	Valid	Valid	Jump far, absolute indirect, address given in m16:16
FF /5	JMP m16:32	D	Valid	Valid	Jump far, absolute indirect, address given in m16:32
REX.W + FF /5	JMP m16:64	D	Valid	N.E.	Jump far, absolute indirect, address given in m16:64

Table 4.7.1: `jmp` opcode table

   Each row lists a variant of `jmp` instruction. The first column has the opcode `EB cb`, with an equivalent symbolic form `jmp rel8`. Here, `rel8` means 128 bytes offset, counting from the end of the instruction. The end of an instruction is the next byte after the last byte of an instruction. To make it more concrete, consider this assembly code:

```
main:
  jmp main
  jmp main2
  jmp main
```

```
main2:
    jmp 0x1234
```

generates the machine code:

main				main2					
↓				↓					

Address	00	01	02	03	04	05	06	07	08	09
Opcode	eb	fe	eb	02	eb	fa	e9	2b	12	00

The first `jmp main` instruction is generated into `eb fe` and occupies the addresses `00` and `01`; the end of the first `jmp main` is at address `02`, past the last byte of the first `jmp main` which is located at the address `01`. The value `fe` is equivalent to `-2`, since `eb` opcode uses only a byte (8 bits) for relative addressing. The offset is `-2`, and the end address of the first `jmp main` is `02`, adding them together we get `00` which is the destination address for jumping to.

Similarly, the `jmp main2` instruction is generated into `eb 02`, which means the offset is `+2`; the end address of `jmp main2` is at `04`, and adding together with the offset we get the destination address is `06`, which is the start instruction marked by the label `main2`.

The same rule can be applied to `rel16` and `rel32` encoding. In the example code, `jmp 0x1234` uses `rel16` (which means 2-byte offset) and is generated into `e9 2b 12`. As the table 4.7.1 shows, `e9` opcode takes a `cw` operand, which is a 2-byte offset (section *3.1.1.1*, volume 2). Notice one strange issue here: the offset value is `2b 12`, while it is supposed to be `34 12`. There is nothing wrong. Remember, `rel8/rel16/rel32` is an *offset*, not an *address*. A offset is a distance from a point. Since no label is given but a number, the offset is calculated from the start of a program. In this case, the start of the program is the address `00`, the end of `jmp 0x1234` is the address `09`[2], so the offset is calculated as `0x1234 - 0x9` `= 0x122b`. That solved the mystery!

The `jmp` instructions with opcode `FF /4` enable jumping to a *near, absolute* address stored in a general-purpose register or a memory location; or in short, as written in the description, *absolute indirect*. The symbol `/4` is the column with digit 4 in table 4.5.1[3]. For example:

[2] which means 9 bytes was consumed, starting from address 0.

[3] The column with the following fields:
```
AH
SP
ESP
M45
XMM4
4
100
```

```
jmp [0x1234]
```

is generated into:

```
ff 26 34 12
```

Since this is 16-bit code, we use table 4.5.1. Looking up the table, `ModR/M` value `26` means `disp16`, which means a 16-bit offset from the start of current index[4], which is the base address stored in `DS` register. In this case, `jmp [0x1234]` is implicitly understood as `jmp [ds:0x1234]`, which means the destination address is `0x1234` bytes away from the start of a data segment.

[4] Look at the note under the table.

The `jmp` instruction with opcode `FF /5` enables jumping to a *far, absolute* address stored in a *memory location* (as opposed to `/4`, which means stored in a register); in short, *a far pointer*. To generate such instruction, the keyword `far` is needed to tell `nasm` we are using a far pointer:

```
jmp far [eax]
```

is generated into:

```
67 ff 28
```

Since `28` is the value in the 5th column of the table 4.5.2[5] that refers to `[eax]`, we successfully generate an instruction for a far jump. After CPU runs the instruction, the program counter `eip` and code segment register `cs` is set to the memory address, stored in the memory location that `eax` points to, and CPU starts fetching code from the new address in `cs` and `eip`. To make it more concrete, here is an example:

[5] Remember the prefix `67` indicates the instruction is used as 32-bit. The prefix only added if the default environment is assumed as 16-bit when generating code by an assembler.
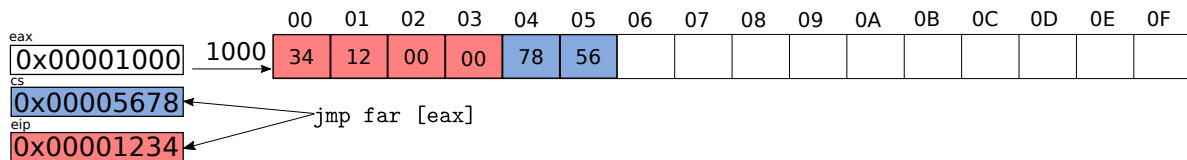
The far address consumes total of 6 bytes in size for a 16-bit segment and 32-bit address, which is encoded as `m16:32` from the table 4.7.1. As

Figure 4.7.1: far `jmp` example, with the destination memory stored at address `0x1000`, which is stored in `eax` to be dereferenced. After CPU executes the instruction, code segment register `cs` and instruction pointer `eip`

can be seen from the figure above, the blue part is a segment address, loaded into `cs` register with the value `0x5678`; the red part is the memory address within that segment, loaded into `eip` register with the value `0x1234` and start executing from there.

Finally, the `jmp` instructions with `EA` opcode jump to a direct absolute address. For example, the instruction:

```
jmp 0x5678:0x1234
```

is generated into:

```
ea 34 12 78 56
```

The address `0x5678:0x1234` is right next to the opcode, unlike `FF /5` instruction that needs an indirect address in `eax` register.

We skip the jump instruction with `REX` prefix, as it is a 64-bit instruction.

4.8   Examine compiled data

In this section, we will examine how data definition in C maps to its assembly form. The generated code is extracted from `.bss` section. That means, the assembly code displayed has no[6], aside from showing that such a value has an equivalent assembly opcode that represents an instruction.

[6] Actually, code is just a type of data, and is often used for hijacking into a running program to execute such code. However, we have no use for it in this book.

The code-assembly listing is not random, but is based on *Chapter 4* of Volume 1, *"Data Type"*. The chapter lists fundamental data types that x86 hardware operates on, and through learning the generated assembly code, it can be understood how close C maps its syntax to hardware, and then a programmer can see why C is appropriate for OS programming. The specific `objdump` command used in this section will be:

```
$ objdump -z -M intel -S -D -j .data -j .bss <object
file> | less
```

Note: zero bytes are hidden with three dot symbols: ... To show all the zero bytes, we add `-z` option.

4.8.1 Fundamental data types

The most basic types that x86 architecture works with are based on sizes, each is twice as large as the previous one: 1 byte (8 bits), 2 bytes (16 bits), 4 bytes (32 bits), 8 bytes (64 bits) and 16 bytes (128 bits).

Figure 4.8.1: Fundamental Data Types

These types are simplest: they are just chunks of memory at different sizes that enables CPU to access memory efficiently. From the manual, *section 4.1.1*, volume 1:

> Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

> Some instructions that operate on double quadwords require memory operands to be aligned on a natural boundary. These instructions generate a general-protection exception (#GP) if an unaligned operand is specified. A natural boundary for a double quadword is any address evenly divisible by 16. Other instructions that operate on double quadwords

permit unaligned access (without generating a general-protection exception). However, additional memory bus cycles are required to access unaligned data from memory.

In C, the following primitive types (must include `stdint.h`) maps to the fundamental types:

Source

```c
#include <stdint.h>


uint8_t byte = 0x12;

uint16_t word = 0x1234;

uint32_t dword = 0x12345678;

uint64_t qword = 0x123456789abcdef;

unsigned __int128 dqword1 = (__int128) 0x123456789abcdef;

unsigned __int128 dqword2 = (__int128) 0x123456789abcdef << 64;


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

        return 0;

}
```

Assembly

```
0804a018 <byte>:
  804a018:       12 00                   adc    al,BYTE PTR [eax]
0804a01a <word>:
  804a01a:       34 12                   xor    al,0x12
0804a01c <dword>:
  804a01c:       78 56                   js     804a074 <_end+0x48>
  804a01e:       34 12                   xor    al,0x12
0804a020 <qword>:
  804a020:       ef                      out    dx,eax
  804a021:       cd ab                   int    0xab
  804a023:       89 67 45                mov    DWORD PTR [edi+0x45],esp
  804a026:       23 01                   and    eax,DWORD PTR [ecx]
0000000000601040 <dqword1>:
  601040:       ef                      out    dx,eax
  601041:       cd ab                   int    0xab
  601043:       89 67 45                mov    DWORD PTR [rdi+0x45],esp
  601046:       23 01                   and    eax,DWORD PTR [rcx]
```

```
   601048:        00 00                         add     BYTE PTR [rax],al
   60104a:        00 00                         add     BYTE PTR [rax],al
   60104c:        00 00                         add     BYTE PTR [rax],al
   60104e:        00 00                         add     BYTE PTR [rax],al
0000000000601050 <dqword2>:
   601050:        00 00                         add     BYTE PTR [rax],al
   601052:        00 00                         add     BYTE PTR [rax],al
   601054:        00 00                         add     BYTE PTR [rax],al
   601056:        00 00                         add     BYTE PTR [rax],al
   601058:        ef                            out     dx,eax
   601059:        cd ab                         int     0xab
   60105b:        89 67 45                      mov     DWORD PTR [rdi+0x45],esp
   60105e:        23 01                         and     eax,DWORD PTR [rcx]
```

gcc generates the variables byte, word, dword, qword, dqword1, dword2, written earlier, with their respective values highlighted in the same colors; variables of the same type are also highlighted in the same color. Since this is data section, the assembly listing carries no meaning. When byte is declared with uint8_t, gcc guarantees that the size of byte is always 1 byte. But, an alert reader might notice the 00 value next to the 12 value in the byte variable. This is normal, as gcc avoid memory misalignment by adding extra *padding bytes*. To make it easier to see, we look at readelf output of .data section:

```
$ readelf -x .data hello
```

the output is (the colors mark which values belong to which variables):

```
Hex dump of section '.data':
  0x00601020 00000000 00000000 00000000 00000000 ................
  0x00601030 12003412 78563412 efcdab89 67452301 ..4.xV4.....gE#.
  0x00601040 efcdab89 67452301 00000000 00000000 ....gE#.........
  0x00601050 00000000 00000000 efcdab89 67452301 ............gE#.
```

As can be seen in the readelf output, variables are allocated storage space according to their types and in the declared order by the program-

mer (the colors correspond the the variables). Intel is a little-endian machine, which means smaller addresses hold bytes with smaller values, larger addresses hold byte with larger values. For example, `0x1234` is displayed as `34 12`; that is, `34` appears first at address `0x601032`, then `12` at `0x601033`. The decimal values within a byte is unchanged, so we see `34 12` instead of `43 21`. This is quite confusing at first, but you will get used to it soon.

Also, isn't it redundant when `char` type is always 1 byte already and why do we bother adding `int8_t`? The truth is, `char` type is not guaranteed to be 1 byte in size, but only the minimum of 1 byte in size. In C, a byte is defined to be the size of a char, and a char is defined to be smallest addressable unit of the underlying hardware platform. There are hardware devices that the smallest addressable unit is 16 bit or even bigger, which means char is 2 bytes in size and a "byte" in such platforms is actually 2 units of 8-bit bytes.

Not all architectures support the double quadword type. Still, `gcc` does provide support for 128-bit number and generate code when a CPU supports it (that is, a CPU must be 64-bit). By specifying a variable of type `__int128` or `unsigned __int128`, we get a 128-bit variable. If a CPU does not support 64-bit mode, `gcc` throws an error.

The data types in C, which represents the fundamental data types, are also called *unsigned numbers*. Other than numerical calculations, unsigned numbers are used as a tool for structuring data in memory; we will see this application later on the book, when various data structures are organized into bit groups.

In all the examples above, when the value of a variable with smaller size is assigned to a variable with larger size, the value easily fits in the larger variable. On the contrary, the value of a variable with larger size is assigned to a variable with smaller size, two scenarios occur:

▷ The value is greater than the maximum value of the variable with smaller layout, so it needs truncating to the size of the variable and causing incorrect value.

▷ The value is smaller than the maximum value of the variable with a smaller layout, so it fits the variable.

However, the value might be unknown until runtime and can be value, it is best not to let such implicit conversion handled by the compiler, but explicitly controlled by a programmer. Otherwise it will cause subtle bugs that are hard to catch as the erroneous values might rarely be used to reproduce the bugs.

4.8.2 Pointer Data Types

Pointers are variables that hold memory addresses. x86 works with 2 types of pointers:

Near pointer is a 16-bit/32-bit offset within a segment, also called *effective address.*

Far pointer is also an offset like a near pointer, but with an explicit segment selector.

Near Pointer
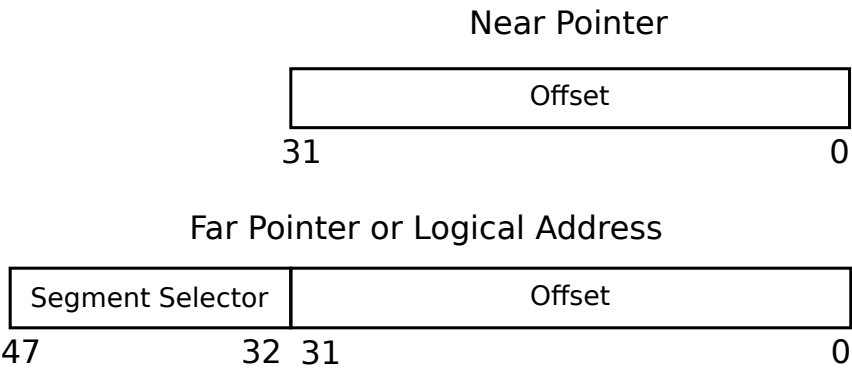
Offset
31   0

Figure 4.8.2: Numeric Data Types

Far Pointer or Logical Address

Segment Selector	Offset
47 32	31 0

    C only provides support for near pointers, since far pointers are platform dependent, such as x86. In application code, you can assume that the address of current segment starts at 0, so the offset is actually any memory address from 0 to the maximum address.

Source

```c
#include <stdint.h>


int8_t i = 0;
int8_t *p1 = (int8_t *) 0x1234;
int8_t *p2 = &i;
```

```
int main(int argc, char *argv[]) {
      return 0;
}
```

Assembly   0000000000601030 <p1>:

```
     601030:        34 12                       xor     al,0x12
     601032:        00 00                       add     BYTE PTR [rax],al
     601034:        00 00                       add     BYTE PTR [rax],al
     601036:        00 00                       add     BYTE PTR [rax],al
   0000000000601038 <p2>:
     601038:        41 10 60 00                 adc     BYTE PTR [r8+0x0],spl
     60103c:        00 00                       add     BYTE PTR [rax],al
     60103e:        00 00                       add     BYTE PTR [rax],al
   Disassembly of section .bss:
   0000000000601040 <__bss_start>:
     601040:        00 00                       add     BYTE PTR [rax],al
   0000000000601041 <i>:
     601041:        00 00                       add     BYTE PTR [rax],al
     601043:        00 00                       add     BYTE PTR [rax],al
     601045:        00 00                       add     BYTE PTR [rax],al
     601047:        00                          .byte 0x0
```

The pointer p1 holds a direct address with the value 0x1234. The pointer
p2 holds the address of the variable i. Note that both the pointers are
8 bytes in size (or 4-byte, if 32-bit).

4.8.3 Bit Field Data Type

A *bit field* is a contiguous sequence of bits. Bit fields allow data structur-
ing at bit level. For example, a 32-bit data can hold multiple bit fields
that represent multiples different pieces of information, such as bits 0-4
specifies the size of a data structure, bit 5-6 specifies permissions and so
on. Data structures at the bit level are common for low-level program-
ming.

Source

```
struct bit_field {
    int data1:8;
```

.

Bit Field

Field Length

Least
Significant
Bit
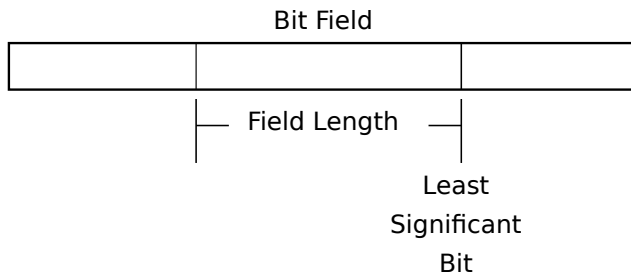
```c
    int data2:8;
    int data3:8;
    int data4:8;
};


struct bit_field2 {
    int data1:8;
    int data2:8;
    int data3:8;
    int data4:8;
    char data5:4;
};


struct normal_struct {
    int data1;
    int data2;
    int data3;
    int data4;
};


struct normal_struct ns = {
    .data1 = 0x12345678,
    .data2 = 0x9abcdef0,
    .data3 = 0x12345678,
    .data4 = 0x9abcdef0,
};
```

```
int i = 0x12345678;

struct bit_field bf = {
    .data1 = 0x12,
    .data2 = 0x34,
    .data3 = 0x56,
    .data4 = 0x78
};

struct bit_field2 bf2 = {
    .data1 = 0x12,
    .data2 = 0x34,
    .data3 = 0x56,
    .data4 = 0x78,
    .data5 = 0xf
};

int main(int argc, char *argv[]) {
    return 0;
}
```

Assembly Each variable and its value are given a unique color in the assembly listing below:

```
0804a018 <ns>:
 804a018: 78 56                      js     804a070 <_end+0x34>
 804a01a: 34 12                      xor    al,0x12
 804a01c: f0 de bc 9a 78 56 34    lock fidivr WORD PTR [edx+ebx*4+0x12345678]
 804a023: 12
 804a024: f0 de bc 9a 78 56 34    lock fidivr WORD PTR [edx+ebx*4+0x12345678]
 804a02b: 12
0804a028 <i>:
 804a028: 78 56                      js     804a080 <_end+0x44>
 804a02a: 34 12                      xor    al,0x12
0804a02c <bf>:
 804a02c: 12 34 56                   adc    dh,BYTE PTR [esi+edx*2]
```

```
  804a02f: 78 12                        js      804a043 <_end+0x7>
0804a030 <bf2>:
  804a030: 12 34 56                     adc     dh,BYTE PTR [esi+edx*2]
  804a033: 78 0f                        js      804a044 <_end+0x8>
  804a035: 00 00                        add     BYTE PTR [eax],al
  804a037: 00                           .byte 0x0
```

The sample code creates 4 variables: ns, i, bf, bf2. The definition of normal_struct
and bit_field structs both specify 4 integers. bit_field specifies ad-
ditional information next to its member name, separated by a colon, e.g.
.data1 : 8. This extra information is the bit width of each bit group.
It means, even though defined as an int, .data1 only consumes 8 bit of
information. If additional data members are specified after .data1, two
scenarios happen:

▷ If the new data members fit within the remaining bits after .data, which
   are 24 bits[7], then the total size of bit_field struct is still 4 bytes, or
   32 bits.

▷ If the new data members don't fit, then the remaining 24 bits (3 bytes)
   are still allocated. However, the new data members are allocated brand
   new storages, without using the previous 24 bits.

In the example, the 4 data members: .data1, .data2, .data3 and .data4,
each can access 8 bits of information, and together can access all of 4 bytes
of the integer first declared by .data1. As can be seen by the generated
assembly code, the values of bf are follow natural order as written in the
C code: 12 34 56 78, since each value is a separate members. In con-
trast, the value of i is a number as a whole, so it is subject to the rule
of little endianess and thus contains the value 78 56 34 12. Note that
at 804a02f, is the address of the final byte in bf, but next to it is a num-
ber 12, despite 78 is the last number in it. This extra number 12 does
not belong to the value of bf. objdump is just being confused that 78 is
an opcode; 78 corresponds to js instruction, and it requires an operand.
For that reason, objdump grabs whatever the next byte after 78 and put
it there. objdump is a tool to display assembly code after all. A better
tool to use is gdb that we will learn in the next chapter. But for this chap-
ter, objdump suffices.

[7] Since .data1 is declared as an int, 32 bits are still allocated, but .data1 can only access 8 bits of information.

Unlike `bf`, each data member in `ns` is allocated fully as an integer, 4 bytes each, 16 bytes in total. As we can see, bit field and normal struct are different: bit field structure data at the bit level, while normal struct works at byte level.

Finally, the struct of `bf2`[8] is the same of `bf`[9], except it contains one more data member: `.data5`, and is defined as a `char`. For this reason, another 4 bytes are allocated just for `.data5`, even though it can only access 4 bits of information, and the final value of `bf2` is: 12 34 56 78 0f 00 00 00. The remaining 3 bytes must be accessed by the mean of a pointer, or casting to another data type that can fully access all 4 bytes..

[8] bit_field2
[9] bit_field

Exercise 4.8.1. What happens when the definition of `bit_field` struct and `bf` variable are changed to:

```
struct bit_field {
    int data1:8;
};
struct bit_field bf = {
    .data1 = 0x1234,
};
```

What will be the value of `.data1`?

Exercise 4.8.2. What happens when the definition of `bit_field2` struct is changed to:

```
struct bit_field2 {
    int data1:8;
    int data5:32;
};
```

What is layout of a variable of type `bit_field2`?

4.8.4 String Data Types

Although share the same name, string as defined by x86 is different than a string in C. x86 defines string as *"continuous sequences of bits, bytes, words, or doublewords"*. On the other hand, C defines a string as an array of 1-byte characters with a zero as the last element of the array to

make a *null-terminated string*. This implies that strings in x86 are arrays, not C strings. A programmer can define an array of bytes, words or doublewords with `char` or `uint8_t`, `short` or `uint16_t` and `int` or `uint32_t`, except an array of bits. However, such a feature can be easily implemented, as an array of bits is essentially any array of bytes, or words or doublewords, but operates at the bit level.

The following code demonstrates how to define array (string) data types:

Source

```c
#include <stdint.h>


uint8_t a8[2] = {0x12, 0x34};
uint16_t a16[2] = {0x1234, 0x5678};
uint32_t a32[2] = {0x12345678, 0x9abcdef0};
uint64_t a64[2] = {0x123456789abcdef0, 0x123456789abcdef0
    };


int main(int argc, char *argv[])
{
    return 0;
}
```

Assembly

```
0804a018 <a8>:
 804a018:  12 34 00               adc    dh,BYTE PTR [eax+eax*1]
 804a01b:  00 34 12               add    BYTE PTR [edx+edx*1],dh
0804a01c <a16>:
 804a01c:  34 12                  xor    al,0x12
 804a01e:  78 56                  js     804a076 <_end+0x3a>
0804a020 <a32>:
 804a020:  78 56                  js     804a078 <_end+0x3c>
 804a022:  34 12                  xor    al,0x12
 804a024:  f0 de bc 9a f0 de bc   lock fidivr WORD PTR [edx+ebx*4-0x65432110]
 804a02b:  9a
0804a028 <a64>:
 804a028:  f0 de bc 9a 78 56 34   lock fidivr WORD PTR [edx+ebx*4+0x12345678]
 804a02f:  12
 804a030:  f0 de bc 9a 78 56 34   lock fidivr WORD PTR [edx+ebx*4+0x12345678]
```

```
804a037:	12
```

Despite a8 is an array with 2 elements, each is 1-byte long, but it is still allocated with 4 bytes. Again, to ensure natural alignment for best performance, gcc pads extra zero bytes. As shown in the assembly listing, the actual value of a8 is 12 34 00 00, with a8[0] equals to 12 and a8[1] equals to 34.

Then it comes a16 with 2 elements, each is 2-byte long. Since 2 elements are 4 bytes in total, which is in the natural alignment, gcc pads no byte. The value of a16 is 34 12 78 56, with a16[0] equals to 34 12 and a16[1] equals to 78 56. Note that, objdump is confused again, as de is the opcode for the instruction fidivr (short of reverse divide) that requires another operand, so objdump grabs whatever the next bytes that makes sense to it for creating "an operand". Only the highlighted values belong to a32.

Next is a32, with 2 elements, 4 bytes each. Similar to above arrays, the value of a32[0] is 78 56 34 12, the value of a32[1] is f0 de bc 9a, exactly what is assigned in the C code.

Finally is a64, also with 2 elements, but 8 bytes each. The total size of a64 is 16 bytes, which is in the natural alignment, therefore no padding bytes added. The values of both a64[0] and a64[1] are the same: f0 de bc 9a 78 56 34 12, that got misinterpreted to fidivr instruction.

```
a8:    | 12 | 34 |
a16:   | 34 12    | 78 56 |
a32:   | 78 56 34 12        | f0 de bc 9a |
a64:   | f0 de bc 9a 78 56 34 12            | f0 de bc 9a 78 56 34 12 |
```
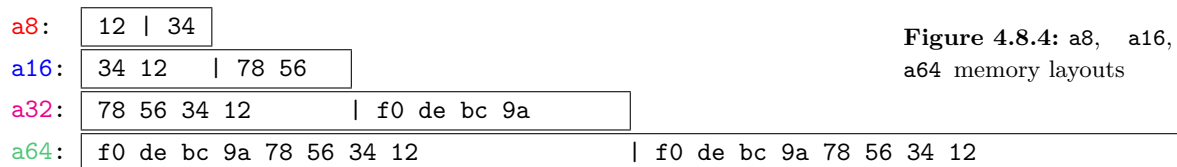
Figure 4.8.4: a8, a16, a32 and a64 memory layouts

However, beyond one-dimensional arrays that map directly to hardware string type, C provides its own syntax for multi-dimensional arrays:

Source
```
#include <stdint.h>

uint8_t a2[2][2] = {
    {0x12, 0x34},
    {0x56, 0x78}
};
```

```
uint8_t a3[2][2][2] = {
    {{0x12, 0x34},
     {0x56, 0x78}},
    {{0x9a, 0xbc},
     {0xde, 0xff}},
};


int main(int argc, char *argv[]) {
    return 0;
}
```

Assembly   0804a018 <a2>:

```
    804a018: 12 34 56                  adc    dh,BYTE PTR [esi+edx*2]
    804a01b: 78 12                     js     804a02f <_end+0x7>
  0804a01c <a3>:
    804a01c: 12 34 56                  adc    dh,BYTE PTR [esi+edx*2]
    804a01f: 78 9a                     js     8049fbb <_DYNAMIC+0xa7>
    804a021: bc                        .byte 0xbc
    804a022: de ff                     fdivrp st(7),st
```

Technically, multi-dimensional arrays are like normal arrays: in the end, the total size is translated into flat allocated bytes. A 2 x 2 array is allocated with 4 bytes; a $2 \times 2 \times 2$ array is allocated with 8 bytes, as can be seen in the assembly listing of a2[10] and a3. In low-level assembly code, the representation is the same between a[4] and a[2][2]. However, in high-level C code, the difference is tremendous. The syntax of multi-dimensional array enables a programmer to think with higher level concepts, instead of translating manually from high-level concepts to low-level code and work with high-level concepts in his head at the same time.

[10] Again, objdump is confused and put the number 12 next to 78 in a3 listing.

Example 4.8.1. The following two-dimensional array can hold a list of 2 names with the length of 10:

```
char names[2][10] = {
    "John␣Doe",
    "Jane␣Doe"
```

```
};
```

To access a name, we simply adjust the column index[11] e.g. `names[0]`, `names[1]`. To access individual character within a name, we use the row index[12] e.g. `names[0][0]` gives the character "J", `names[0][1]` gives the character "o" and so on.

Without such syntax, we need to create a 20-byte array e.g. `names[20]`, and whenever we want to access a character e.g. to check if the names contains with a number in it, we need to calculate the index manually. It would be distracting, since we constantly need to switch thinkings between the actual problem and the translate problem.

Since this is a repeating pattern, C abstracts away this problem with the syntax for define and manipulating multi-dimensional array. Through this example, we can clearly see the power of abstraction through language can give us. It would be ideal if a programmer is equipped with such power to define whatever syntax suitable for a problem at hands. Not many languages provide such capacity. Fortunately, through C macro, we can partially achieve that goal .

In all cases, an array is guaranteed to generate contiguous bytes of memory, regardless of the dimensions it has.

Exercise 4.8.3. What is the difference between a multi-dimensional array and an array of pointers, or even pointers of pointers?

[11] The left index is called column index since it changes the index based on a column.

[12] Same with column index, the right index is called row index since it changes the index based on a row.

4.9   Examine compiled code

This section will explore how compiler transform high level code into assembly code that CPU can execute, and see how common assembly patterns help to create higher level syntax. `-S` option is added to `objdump` to better demonstrate the connection between high and low level code.

In this section, the option `--no-show-raw-insn` is added to objdump command to omit the opcodes for clarity:

```
$ objdump --no-show-raw-insn -M intel -S -D <object
file> | less
```

4.9.1 Data Transfer

Previous section explores how various types of data are created, and how they are laid out in memory. Once memory storages are allocated for variables, they must be accessible and writable. Data transfer instructions move data (bytes, words, doublewords or quadwords) between memory and registers, and between registers, effectively read from a storage source and write to another storage source.

Source

```c
#include <stdint.h>


int32_t i = 0x12345678;


int main(int argc, char *argv[]) {
        int j = i;
        int k = 0xabcdef;


        return 0;
}
```

Assembly   080483db <main>:

```
    #include <stdint.h>
    int32_t i = 0x12345678;
    int main(int argc, char *argv[]) {
     80483db:        push    ebp
     80483dc:        mov     ebp,esp
     80483de:        sub     esp,0x10
            int j = i;
     80483e1:        mov     eax,ds:0x804a018
     80483e6:        mov     DWORD PTR [ebp-0x8],eax
            int k = 0xabcdef;
     80483e9:        mov     DWORD PTR [ebp-0x4],0xabcdef
            return 0;
     80483f0:        mov     eax,0x0
    }
     80483f5:        leave
```

```
80483f6:        ret
80483f7:        xchg    ax,ax
80483f9:        xchg    ax,ax
80483fb:        xchg    ax,ax
80483fd:        xchg    ax,ax
80483ff:        nop
```

The general data movement is performed with the `mov` instruction. Note that despite the instruction being called `mov`, it actually copies data from one destination to another.

The red instruction copies data from the register `esp` to the register `ebp`. This `mov` instruction moves data between registers and is assigned the opcode `89`.

The blue instructions copies data from one memory location (the `i` variable) to another (the `j` variable). There exists no data movement from memory to memory; it requires two `mov` instructions, one for copying the data from a memory location to a register, and one for copying the data from the register to the destination memory location.

The pink instruction copies an immediate value into memory. Finally, the green instruction copies immediate data into a register.

4.9.2 Expressions

Source

```
int expr(int i, int j)
{
    int add = i + j;
    int sub = i - j;
    int mul = i * j;
    int div = i / j;
    int mod = i % j;
    int neg = -i;
    int and = i & j;
    int or = i | j;
    int xor = i ^ j;
    int not = ~i;
    int shl = i << 8;
```

```
    int shr = i >> 8;

    char equal1 = (i == j);

    int equal2 = (i == j);

    char greater = (i > j);

    char less = (i < j);

    char greater_equal = (i >= j);

    char less_equal = (i <= j);

    int logical_and = i && j;

    int logical_or = i || j;

    ++i;

    --i;

    int i1 = i++;

    int i2 = ++i;

    int i3 = i--;

    int i4 = --i;


    return 0;
}


int main(int argc, char *argv[]) {
    return 0;
}
```

Assembly The full assembly listing is really long. For that reason, we ex-
amine expression by expression.

Expression: int add = i + j;

```
    80483e1:       mov     edx,DWORD PTR [ebp+0x8]
    80483e4:       mov     eax,DWORD PTR [ebp+0xc]
    80483e7:       add     eax,edx
    80483e9:       mov     DWORD PTR [ebp-0x34],eax
```

The assembly code is straight forward: variable i and j are stored
in eax and edx respectively, then added together with the add in-
struction, and the final result is stored into eax. Then, the result
is saved into the local variable add, which is at the location [ebp-0x34].

Expression: `int sub = i - j;`

```
80483ec:        mov     eax,DWORD PTR [ebp+0x8]
80483ef:        sub     eax,DWORD PTR [ebp+0xc]
80483f2:        mov     DWORD PTR [ebp-0x30],eax
```

Similar to `add` instruction, x86 provides a `sub` instruction for subtraction. Hence, `gcc` translates a subtraction into `sub` instruction, with `eax` is reloaded with `i`, as `eax` still carries the result from previous expression. Then, `j` is subtracted from `i`. After the subtraction, the value is saved into the variable `sub`, at location `[ebp-0x30]`.

Expression: `int mul = i * j;`

```
80483f5:        mov     eax,DWORD PTR [ebp+0x8]
80483f8:        imul    eax,DWORD PTR [ebp+0xc]
80483fc:        mov     DWORD PTR [ebp-0x34],eax
```

Similar to `sub` instruction, only `eax` is reloaded, since it carries the result of previous calculation. `imul` performs signed multiply[13]. `eax` is first loaded with `i`, then is multiplied with `j` and stored the result back into `eax`, then stored into the variable `mul` at location `[ebp-0x34]`.

[13] Unsigned multiply is perform by `mul` instruction.

Expression: `int div = i / j;`

```
80483ff:        mov     eax,DWORD PTR [ebp+0x8]
8048402:        cdq
8048403:        idiv    DWORD PTR [ebp+0xc]
8048406:        mov     DWORD PTR [ebp-0x30],eax
```

Similar to `imul`, `idiv` performs sign divide. But, different from `imul` above `idiv` only takes one operand:

1. First, `i` is reloaded into `eax`.

2. Then, `cdq` converts the double word value in `eax` into a quad-word value stored in the pair of registers `edx:eax`, by copying the signed (bit 31th) of the value in `eax` into every bit position in `edx`. The pair `edx:eax` is the dividend, which is the variable i, and the operand to `idiv` is the divisor, which is the variable j.

3. After the calculation, the result is stored into the pair `edx:eax` registers, with the quotient in `eax` and remainder in `edx`. The quotient is stored in the variable `div`, at location `[ebp-0x30]`.

Expression: `int mod = i % j;`

```
8048409:        mov     eax,DWORD PTR [ebp+0x8]
804840c:        cdq
804840d:        idiv    DWORD PTR [ebp+0xc]
8048410:        mov     DWORD PTR [ebp-0x2c],edx
```

The same `idiv` instruction also performs the modulo operation, since it also calculates a remainder and stores in the variable `mod`, at location `[ebp-0x2c]`.

Expression: `int neg = -i;`

```
8048413:        mov     eax,DWORD PTR [ebp+0x8]
8048416:        neg     eax
8048418:        mov     DWORD PTR [ebp-0x28],eax
```

`neg` replaces the value of operand (the destination operand) with its two's complement (this operation is equivalent to subtracting the operand from 0). In this example, the value `i` in `eax` is replaced replaced with `-i` using `neg` instruction. Then, the new value is stored in the variable neg at `[ebp-0x28]`.

Expression: `int and = i & j;`

```
804841b:        mov     eax,DWORD PTR [ebp+0x8]
804841e:        and     eax,DWORD PTR [ebp+0xc]
8048421:        mov     DWORD PTR [ebp-0x24],eax
```

`and` performs a bitwise `AND` operation on two operands, and stores the result in the destination operand, which is the variable `and` at `[ebp-0x24]`.

Expression: `int or = i | j;`

```
8048424:        mov     eax,DWORD PTR [ebp+0x8]
8048427:        or      eax,DWORD PTR [ebp+0xc]
804842a:        mov     DWORD PTR [ebp-0x20],eax
```

Similar to `and` instruction, `or` performs a bitwise `OR` operation on two operands, and stores the result in the destination operand, which is the variable `or` at `[ebp-0x20]` in this case.

Expression: `int xor = i ^ j;`

```
804842d:        mov     eax,DWORD PTR [ebp+0x8]
8048430:        xor     eax,DWORD PTR [ebp+0xc]
8048433:        mov     DWORD PTR [ebp-0x1c],eax
```

Similar to and/or instruction, xor performs a bitwise XOR opera-
tion on two operands, and stores the result in the destination operand,
which is the variable xor at [ebp-0x1c].

Expression: int not = ~i;

```
8048436:        mov     eax,DWORD PTR [ebp+0x8]
8048439:        not     eax
804843b:        mov     DWORD PTR [ebp-0x18],eax
```

not performs a bitwise NOT operation (each 1 is set to 0, and each
0 is set to 1) on the destination operand and stores the result in
the destination operand location, which is the variable not at [ebp-0x18].

Expression: int shl = i << 8;

```
804843e:        mov     eax,DWORD PTR [ebp+0x8]
8048441:    shl     eax,0x8
8048444:    mov     DWORD PTR [ebp-0x14],eax
```

shl (shift logical left) shifts the bits in the destination operand to
the left by the number of bits specified in the source operand. In
this case, eax stores i and shl shifts eax by 8 bits to the left. A dif-
ferent name for shl is sal *(*shift arithmetic left). Both can be used
synonymous. Finally, the result is stored in the variable shl at [ebp-0x14].
Here is a visual demonstration of shl/sal and shr instructions:
After shifting to the left, the right most bit is set for Carry Flag in
EFLAGS register.

Expression: int shr = i >> 8;

```
8048447:        mov     eax,DWORD PTR [ebp+0x8]
804844a:    sar     eax,0x8
804844d:    mov     DWORD PTR [ebp-0x10],eax
```

sar is similar to shl/sal, but shift bits to the right and extends
the sign bit. For right shift, shr and sar are two different instruc-
tions. shr differs to sar is that it does not extend the sign bit. Finally,
the result is stored in the variable shr at [ebp-0x10].

Initial State
CF

X

1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 **1**

After 1-bit SHL/SAL instruction

1   0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 **0**   ← 0

After 1-bit SHL/SAL instruction

0   0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 **1** 0 0 0 0 0 0 0 0 0 0 0 0   ← 0

(a) SHL/SAL (Source: Figure 7-6, Volume 1)

Initial State
CF

1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 **1**   X

After 1-bit SHR instruction

0 →   0 **1** 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 **1**   → **1**

After 10-bit SHR instruction

0 →   0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0   → 0

(b) SHR (Source: Figure 7-7, Volume 1)

In the figure 4.9.1(b), notice that initially, the sign bit is 1, but after 1-bit and 10-bit shiftings, the shifted-out bits are filled with zeros.

Figure 4.9.1: Shift Instructions (red is the start bit, blue is the end bit.)

Initial State (Positive Operand)
Operand   CF

0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 **1**   X

After 1-bit SAR instruction

0 **0** 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1   → **1**

Initial State (Negative Operand)
Operand

1 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 **1**   → X

After 10-bit SAR instruction

1 **1** 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1   → **1**
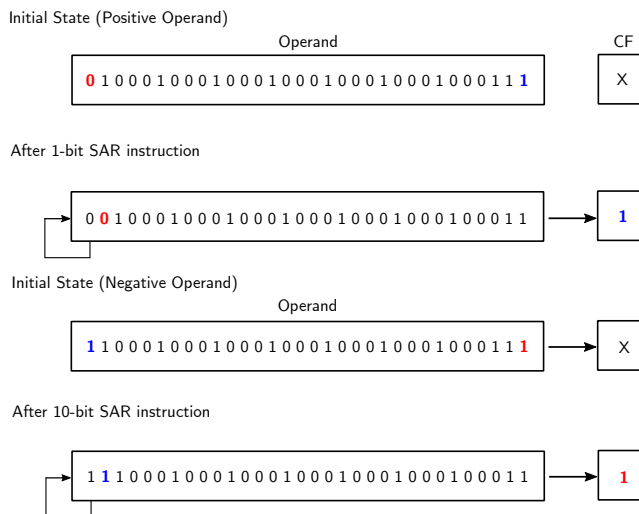
Figure 4.9.2: SAR Instruction Operation (Source: Figure 7-8, Volume 1)

With `sar`, the sign bit (the most significant bit) is preserved. That is, if the sign bit is 0, the new bits always get the value 0; if the sign bit is 1, the new bits always get the value 1.

Expression: `char equal1 = (i == j);`

```
8048450:        mov     eax,DWORD PTR [ebp+0x8]
8048453:        cmp     eax,DWORD PTR [ebp+0xc]
8048456:        sete    al
8048459:        mov     BYTE PTR [ebp-0x41],al
```

cmp and variants of the variants of set instructions make up all the logical comparisons. In this expression, cmp compares variable i and j; then sete stores the value 1 to al register if the comparison from cmp earlier is equal, or stores 0 otherwise. The general name for variants of set instruction is called SETcc. The suffix cc denotes the condition being tested for in EFLAGS register. Appendix B in volume 1, *"EFLAGS Condition Codes"*, lists the conditions it is possible to test for with this instruction. Finally, the result is stored in the variable equal1 at [ebp-0x41].

Expression: int equal2 = (i == j);

```
804845c:        mov     eax,DWORD PTR [ebp+0x8]
804845f:        cmp     eax,DWORD PTR [ebp+0xc]
8048462:        sete    al
8048465:        movzx   eax,al
8048468:        mov     DWORD PTR [ebp-0xc],eax
```

Similar to equality comparison, this expression also compares for equality, with an exception that the result is stored in an int type. For that reason, one more instruction is a added: movzx instruction, a variant of mov that copies the result into a destination operand and fills the remaining bytes with 0. In this case, since eax is 4-byte wide, after copying the first byte in al, the remaining bytes of eax are filled with 0 to ensure the eax carries the same value as al.

| 12 | 34 | 56 | 78 |

(a) eax before movzx

| 00 | 00 | 00 | 78 |

(b) after movzx eax, al

Figure 4.9.3: movzx instruction

Expression: char greater = (i > j);

```
804846b:        mov     eax,DWORD PTR [ebp+0x8]
804846e:        cmp     eax,DWORD PTR [ebp+0xc]
8048471:        setg    al
8048474:        mov     BYTE PTR [ebp-0x40],al
```

Similar to equality comparison, but used setg for **g**reater comparison instead.

Expression: char less = (i < j);

```
8048477:        mov     eax,DWORD PTR [ebp+0x8]
```

```
804847a:        cmp     eax,DWORD PTR [ebp+0xc]
804847d:        setl    al
8048480:        mov     BYTE PTR [ebp-0x3f],al
```

Applied `setl` for **l**ess comparison.

Expression: `char greater_equal = (i >= j);`

```
8048483:        mov     eax,DWORD PTR [ebp+0x8]
8048486:        cmp     eax,DWORD PTR [ebp+0xc]
8048489:        setge   al
804848c:        mov     BYTE PTR [ebp-0x3e],al
```

Applied `setge` for **g**reater or **e**qual comparison.

Expression: `char less_equal = (i <= j);`

```
804848f:        mov     eax,DWORD PTR [ebp+0x8]
8048492:        cmp     eax,DWORD PTR [ebp+0xc]
8048495:        setle   al
8048498:        mov     BYTE PTR [ebp-0x3d],al
```

Applied `setle` for **l**ess than or **e**qual comparison.

Expression: `int logical_and = (i && j);`

```
804849b:        cmp     DWORD PTR [ebp+0x8],0x0
804849f:        je      80484ae <expr+0xd3>
80484a1:        cmp     DWORD PTR [ebp+0xc],0x0
80484a5:        je      80484ae <expr+0xd3>
80484a7:        mov     eax,0x1
80484ac:        jmp     80484b3 <expr+0xd8>
80484ae:        mov     eax,0x0
80484b3:        mov     DWORD PTR [ebp-0x8],eax
```

Logical `AND` operator `&&` is one of the syntaxes that is made entirely in software[14] with simpler instructions. The algorithm from the assembly code is simple:

1. First, check if `i` is `0` with the instruction at `0x804849b`.

   (a) If true, jump to `0x80484ae` and set eax to 0.

   (b) Set the variable `logical_and` to 0, as it is the next instruction after `0x80484ae`.

[14] That is, there is no equivalent assembly instruction implemented in hardware.

2. If `i` is not `0`, check if `j` is `0` with the instruction at `0x80484a1`.

    (a) If true, jump to `0x80484ae` and set eax to 0.

    (b) Set the variable `logical_and` to 0, as it is the next instruction after `0x80484ae`.

3. If both `i` and `j` are not `0`, the result is certainly `1`, or `true`.

    (a) Set it accordingly with the instruction at `0x80484a7`.

    (b) Then jump to the instruction at `0x80484b3` to set the variable `logical_and` at `[ebp-0x8]` to 1.

Expression: `int logical_or = (i || j);`

```
80484b6:        cmp     DWORD PTR [ebp+0x8],0x0
80484ba:        jne     80484c2 <expr+0xe7>
80484bc:        cmp     DWORD PTR [ebp+0xc],0x0
80484c0:        je      80484c9 <expr+0xee>
80484c2:        mov     eax,0x1
80484c7:        jmp     80484ce <expr+0xf3>
80484c9:        mov     eax,0x0
80484ce:        mov     DWORD PTR [ebp-0x4],eax
```

Logical `OR` operator `||` is similar to logical and above. Understand the algorithm is left as an exercise for readers.

Expression: `++i;` and `--i;` (or `i++` and `i--`)

```
80484d1:        add     DWORD PTR [ebp+0x8],0x1
80484d5:        sub     DWORD PTR [ebp+0x8],0x1
```

The syntax of increment and decrement is similar to logical `AND` and logical `OR` in that it is made from existing instruction, that is `add`. The difference is that the CPU actually does has a built-in instruction, but `gcc` decided not to use the instruction because `inc` and `dec` cause a *partial flag register stall*, occurs when an instruction modifies a part of the flag register and the following instruction is dependent on the outcome of the flags (*section 3.5.2.6*, Intel Optimization Manual, 2016b). The manual even suggests that `inc` and `dec` should be replaced with `add` and `sub` instructions (section *3.5.1.1*, Intel Optimization Manual, 2016b).

Expression: `int i1 = i++;`

```
80484d9:        mov     eax,DWORD PTR [ebp+0x8]
80484dc:        lea     edx,[eax+0x1]
80484df:        mov     DWORD PTR [ebp+0x8],edx
80484e2:        mov     DWORD PTR [ebp-0x10],eax
```

First, `i` is copied into `eax` at 80484d9. Then, the value of `eax + 0x1`
is copied into `edx` as an *effective address* at 80484dc. The `lea` (*load
effective address*) instruction copies a memory address into a reg-
ister. According to Volume 2, the source operand is a memory ad-
dress specified with one of the processors addressing modes. This
means, the source operand must be specified by the addressing modes
defined in 16-bit/32-bit ModR/M Byte tables, 4.5.1 and 4.5.2.

After loading the incremented value into `edx`, the value of `i` is in-
creased by 1 at 80484df. Finally, the *previous* `i` value is stored back
to `i1` at `[ebp-0x8]` by the instruction at 80484e2.

Expression: `int i2 = ++i;`

```
80484e5:        add     DWORD PTR [ebp+0x8],0x1
80484e9:        mov     eax,DWORD PTR [ebp+0x8]
80484ec:        mov     DWORD PTR [ebp-0xc],eax
```

The primary differences between this increment syntax and the pre-
vious one are:

▷ `add` is used instead of `lea` to increase `i` directly.

▷ the newly incremented `i` is stored into `i2` instead of the old value.

▷ the expression only costs 3 instructions instead of 4.

This prefix-increment syntax is faster than the post-fix one used
previously. It might not matter much which version to use if the
increment is only used once or a few hundred times in a small loop,
but it matters when a loop runs millions or more times. Also, de-
pends on different circumstances, it is more convenient to use one
over the other e.g. if `i` is an index for accessing an array, we want
to use the old value for accessing previous array element and newly
incremented `i` for current element.

Expression: `int i3 = i--;`

```
80484ef:        mov     eax,DWORD PTR [ebp+0x8]
```

```
80484f2:        lea     edx,[eax-0x1]

80484f5:        mov     DWORD PTR [ebp+0x8],edx

80484f8:        mov     DWORD PTR [ebp-0x8],eax
```

Similar to `i++` syntax, and is left as an exercise to readers.

Expression: `int i4 = --i;`

```
80484fb:        sub     DWORD PTR [ebp+0x8],0x1

80484ff:        mov     eax,DWORD PTR [ebp+0x8]

8048502:        mov     DWORD PTR [ebp-0x4],eax
```

Similar to `++i` syntax, and is left as an exercise to readers.

Exercise 4.9.1. Read section *3.5.2.4, "Partial Register Stalls"* to understand register stalls in general.

Exercise 4.9.2. Read the sections from *7.3.1* to *7.3.7* in volume 1.

4.9.3 Stack

A stack is a contiguous array of memory locations that holds a collection of discrete data. When a new element is added, a stack *grows down* in memory toward lesser addresses, and *shrinks up* toward greater addresses when an element is removed. x86 uses the `esp` register to point to the top of the stack, at the newest element. A stack can be originated anywhere in main memory, as `esp` can be set to any memory address. x86 provides two operations for manipulating stacks:

▷ `push` instruction and its variants add a new element on top of the stack

▷ `pop` instructions and its variants remove the top-most element from the stack.

(a) Initial state at address `0x10004`

(b) After   executing `push word`
`0x5678`

(c) After executing `pop word`
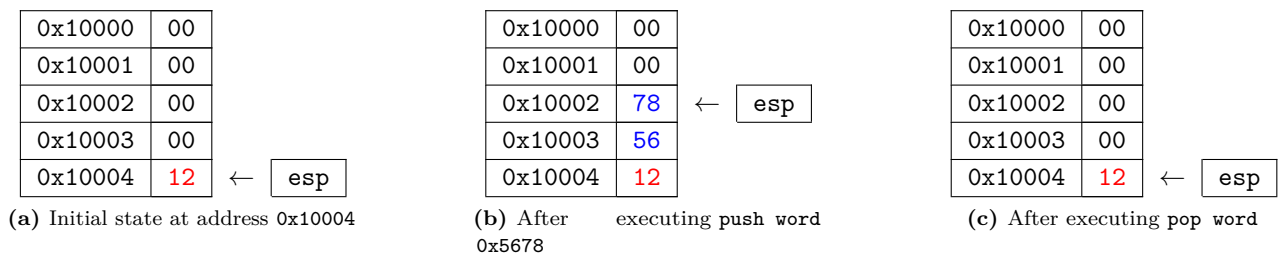
Figure 4.9.4: Stack operations

4.9.4 Automatic variables

Local variables are variables that exist within a scope. A scope is delimited by a pair of braces: `{..}`. The most common scope to define local variables is at function scope. However, scope can be unnamed, and variables created inside an unnamed scope do not exist outside of its scope and its inner scope.

Example 4.9.1. Function scope:

```
void foo() {
    int a;
    int b;
}
```

a and b are variables local to the function `foo`.

Example 4.9.2. Unnamed scope:

```
int foo() {
    int i;

    {
        int a = 1;
        int b = 2;
        {
            return i = a + b;
        }
    }
}
```

a and b are local to where it is defined and local into its inner child scope that return `i = a + b`. However, they do not exist at the function scope that creates `i`.

When a local variable is created, it is pushed on the stack; when a local variable goes out of scope, it is pop out of the stack, thus destroyed. When an argument is passed from a caller to a callee, it is pushed on the stack; when a callee returns to the caller, the arguments are popped out

the stack. The local variables and arguments are automatically allocated upon enter a function and destroyed after exiting a function, that's why it's called *automatic variables.*

A base frame pointer points to the start of the current function frame, and is kept in `ebp` register. Whenever a function is called, it is allocated with its own dedicated storage on stack, called *stack frame.* A stack frame is where all local variables and arguments of a function are placed on a stack[15].

[15] Data and only data are exclusively allocated on stack for every stack frame. No code resides here.

When a function needs a local variable or an argument, it uses `ebp` to access a variable:

▷ All local variables are allocated after the `ebp` pointer. Thus, to access a local variable, a number is subtracted from `ebp` to reach the location of the variable.

▷ All arguments are allocated before `ebp` pointer. To access an argument, a number is added to `ebp` to reach the location of the argument.

▷ The `ebp` itself pointer points to the return address of its caller.

Previous Frame					Current Frame						
Function Arguments						ebp	Local variables				
A1	A2	A3	....	An	Return Address	Old ebp	L1	L2	L3	....	Ln

Figure 4.9.5: Function arguments and local variables

`A` = Argument

`L` = Local Variable

Here is an example to make it more concrete:

Source

```
int add(int a, int b) {

    int i = a + b;


    return i;
}
```

Assembly   080483db <add>:

```
    #include <stdint.h>
    int add(int a, int b) {
     80483db:        push    ebp
```

```
80483dc:        mov     ebp,esp
80483de:        sub     esp,0x10
    int i = a + b;
80483e1:        mov     edx,DWORD PTR [ebp+0x8]
80483e4:        mov     eax,DWORD PTR [ebp+0xc]
80483e7:        add     eax,edx
80483e9:        mov     DWORD PTR [ebp-0x4],eax
    return i;
80483ec:        mov     eax,DWORD PTR [ebp-0x4]
}
80483ef:        leave
80483f0:        ret
```

In the assembly listing, `[ebp-0x4]` is the local variable `i`, since it is allocated *after* `ebp`, with the length of 4 bytes (an `int`). On the other hand, `a` and `b` are arguments and can be accessed with `ebp`:

▷ `[ebp+0x8]` accesses `a`.

▷ `[ebp+0xc]` access `b`.

For accessing arguments, the rule is that the closer a variable on stack to `ebp`, the closer it is to a function name.

	ebp+0xc				ebp+0x8				ebp+0x4				ebp			
	↓				↓				↓				↓			
	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
0x10000	b				a				Return Address				Old ebp			

									ebp+0x8				ebp+0x4			
									↓				↓			
	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
0xffe0												N		i		
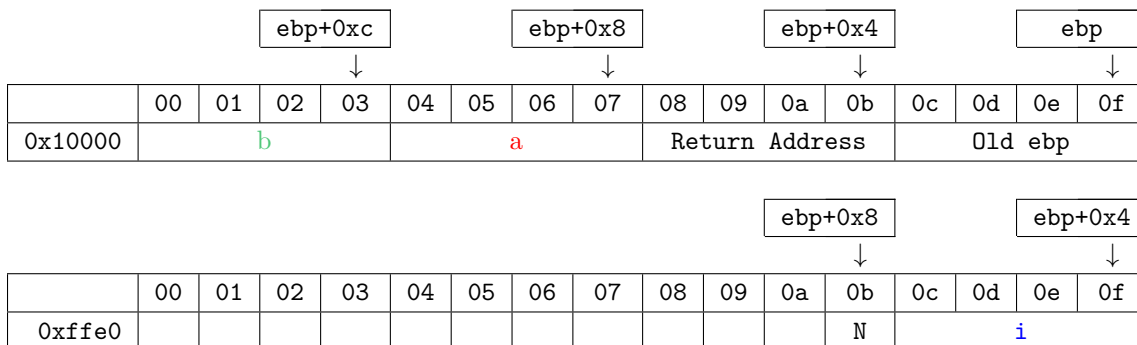
Figure 4.9.6: Function arguments and local variables in memory

`N` = Next local variable starts here

From the figure, we can see that `a` and `b` are laid out in memory with the exact order as written in C, relative to the return address.

4.9.5 Function Call and Return

Source

```
#include <stdio.h>


int add(int a, int b) {
    int local = 0x12345;


    return a + b;
}


int main(int argc, char *argv[]) {
    add(1,1);


    return 0;
}
```

Assembly For every function call, `gcc` pushes arguments on the stack in
reversed order with the push instructions. That is, the arguments pushed
on stack are in reserved order as it is written in high level C code, to
ensure the relative order between arguments, as seen in previous sec-
tion how function arguments and local variables are laid out. Then,
`gcc` generates a `call` instruction, which then implicitly pushes a re-
turn address before transferring the control to `add` function:

```
080483f2 <main>:
int main(int argc, char *argv[]) {
 80483f2:        push    ebp
 80483f3:        mov     ebp,esp
    add(1,2);
 80483f5:        push    0x2
 80483f7:        push    0x1
 80483f9:        call    80483db <add>
 80483fe:        add     esp,0x8
    return 0;
 8048401:        mov     eax,0x0
 }
 8048406:        leave
```

```
8048407:        ret
```

Upon finishing the call to `add` function, the stack is restored by adding `0x8` to stack pointer `esp` (which is equivalent to 2 `pop` instructions). Finally, a `leave` instruction is executed and main returns with a `ret` instruction. A `ret` instruction transfers the program execution back to the caller to the instruction right after the `call` instruction, the `add` instruction. The reason `ret` can return to such location is that the return address implicitly pushed by the `call` instruction, which is the address right after the `call` instruction; whenever the CPU executes `ret` instruction, it retrieves the return address that sits right after all the arguments on the stack:

At the end of a function, `gcc` places a `leave` instruction to clean up all spaces allocated for local variables and restore the frame pointer to frame pointer of the caller.

```
080483db <add>:
#include <stdio.h>
int add(int a, int b) {
 80483db:        push    ebp
 80483dc:        mov     ebp,esp
 80483de:        sub     esp,0x10
    int local = 0x12345;
 80483e1:        DWORD PTR [ebp-0x4],0x12345
    return a + b;
 80483e8:        mov     edx,DWORD PTR [ebp+0x8]
 80483eb:        mov     eax,DWORD PTR [ebp+0xc]
 80483ee:        add     eax,edx
}
 80483f0:        leave
 80483f1:        ret
```

Exercise 4.9.3. The above code that `gcc` generated for function calling is actually the standard method x86 defined. Read chapter 6, *"Produce Calls, Interrupts, and Exceptions",* Intel manual volume 1.

4.9.6 Loop

Loop is simply resetting the instruction pointer to an already executed
instruction and starting from there all over again. A loop is just one ap-
plication of `jmp` instruction. However, because looping is a pervasive pat-
tern, it earned its own syntax in C.

Source

```
#include <stdio.h>


int main(int argc, char *argv[]) {
    for (int i = 0; i < 10; i++) {
    }


    return 0;
}
```

Assembly   080483db <main>:

```
    #include <stdio.h>
    int main(int argc, char *argv[]) {
     80483db:         push   ebp
     80483dc:         mov    ebp,esp
     80483de:         sub    esp,0x10
        for (int i = 0; i < 10; i++) {
     80483e1:         mov    DWORD PTR [ebp-0x4],0x0
     80483e8:         jmp    80483ee <main+0x13>
     80483ea:         add    DWORD PTR [ebp-0x4],0x1
     80483ee:         cmp    DWORD PTR [ebp-0x4],0x9
     80483f2:         jle    80483ea <main+0xf>
        }
        return 0;
     80483f4: b8 00 00 00 00        mov    eax,0x0
    }
     80483f9: c9                    leave
     80483fa: c3                    ret
     80483fb: 66 90                 xchg   ax,ax
     80483fd: 66 90                 xchg   ax,ax
```

```
80483ff: 90                        nop
```

The colors mark corresponding high level code to assembly code:

1. The red instruction initialize `i` to `0`.

2. The green instructions compare i to 10 by using `jle` and compare it to `9`. If true, jump to `80483ea` for another iteration.

3. The blue instruction increase i by 1, making the loop able to terminate once the terminate condition is satisfied.

Exercise 4.9.4. Why does the increment instruction (the blue instruction) appears before the compare instructions (the green instructions)?

Exercise 4.9.5. What assembly code can be generated for `while` and `do...while`?

4.9.7 Conditional

Again, conditional in C with `if...else...` construct is just another application of `jmp` instruction under the hood. It is also a pervasive pattern that earned its own syntax in C.

Source
```c
#include <stdio.h>


int main(int argc, char *argv[]) {
    int i = 0;


    if (argc) {
        i = 1;
    } else {
        i = 0;
    }


    return 0;
}
```

Assembly   int main(int argc, char *argv[]) {
```
    80483db:          push    ebp
```

```
80483dc:        mov     ebp,esp
80483de:        sub     esp,0x10
    int i = 0;
80483e1:        mov     DWORD PTR [ebp-0x4],0x0
    if (argc) {
80483e8:        cmp     DWORD PTR [ebp+0x8],0x0
80483ec:        je      80483f7 <main+0x1c>
        i = 1;
80483ee:        mov     DWORD PTR [ebp-0x4],0x1
80483f5:        jmp     80483fe <main+0x23>
    } else {
        i = 0;
80483f7:        mov     DWORD PTR [ebp-0x4],0x0
    }
    return 0;
80483fe:        mov     eax,0x0
}
8048403:        leave
8048404:        ret
```

The generated assembly code follows the same order as the corresponding high level syntax:

▷ red instructions represents `if` branch.

▷ blue instructions represents `else` branch.

▷ green instruction is the exit point for both `if` and `else` branch.

`if` branch first compares whether `argc` is *false* (equal to 0) with `cmp` instruction. If true, it proceeds to `else` branch at 80483f7. Otherwise, `if` branch continues with the code of its branch, which is the next instruction at 80483ee for copying 1 to `i`. Finally, it skips over `else` branch and proceeds to 80483fe, which is the next instruction pasts the `if..else...` construct.

`else` branch is entered when `cmp` instruction from `if` branch is true. `else` branch starts at 80483f7, which is the first instruction of `else` branch. The instruction copies 0 to `i`, and proceeds naturally to the next instruction pasts the `if...else...` construct without any jump.

5

The Anatomy of a Program

Every program consists of code and data, and only those two components made up a program. However, if a program consists purely code and data of its own, from the perspective of an operating system (as well as human), it does not know in a program, which block of binary is a program and which is just raw data, where in the program to start execution, which region of memory should be protected and which is free to modify. For that reason, each program carries extra metadata to communicate with the operating system how to handle the program.

When a source file is compiled, the generated machine code is stored into an *object file*, which is just a block of binary. One or more object files can be combined to produce an *executable binary*, which is a complete program runnable in an operating system.

object file

executable binary

`readelf` is a program that recognizes and displays the ELF metadata of a binary file, be it an object file or an executable binary. ***ELF***, or ***E**xecutable and **L**inkable **F**ormat*, is the content at the very beginning of an executable to provide an operating system necessary information to load into main memory and run the executable. ELF can be thought of similar to the table of contents of a book. In a book, a table of contents list the page numbers of the main sections, subsections, sometimes even figures and tables for easy lookup. Similarly, ELF lists various sections used for code and data, and the memory addresses of each symbol along with other in-

formation.

An ELF binary is composed of:

▷ An *ELF header*: the very first section of an executable that describes the file's organization.

ELF header

▷ A *program header table*: is an array of fixed-size structures that describes segments of an executable.

program header table

▷ A *section header table*: is an array of fixed-size structures that describes sections of an executable.

section header table

▷ *Segments and section*s are the main content of an ELF binary, which are the code and data, divided into chunks of different purposes.

*Segments and section*s

A *segment* is a composition of zero or more sections and is directly loaded by an operating system at runtime.

A *section* is a block of binary that is either:

– actual program code and data that is available in memory when a program runs.

– metadata about other sections used only in the linking process, and disappear from the final executable.

Linker uses sections to build segments.
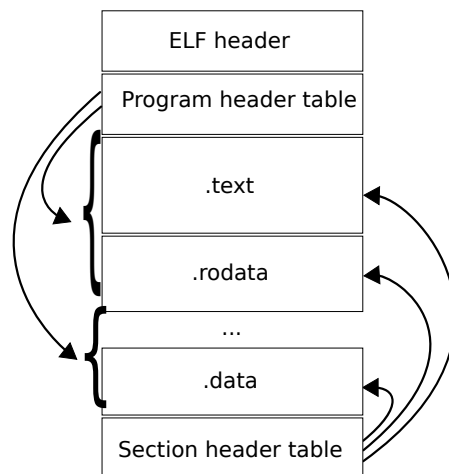
Figure 5.0.1: ELF - Linking View vs Executable View (Source: Wikipedia)

Later we will compile our kernel as an ELF executable with GCC, and explicitly specify how segments are created and where they are loaded

in memory through the use a *linker script*, a text file to instruct how a linker should generate a binary. For now, we will examine the anatomy of an ELF executable in detail.

5.1   Reference documents:

The ELF specification is bundled as a `man` page in Linux: ***ELF specification***

```
$ man elf
```

It is a useful resource to understand and implement ELF. However, it will be much easier to use after you finish this chapter, as the specification mixes implementation details in it.

The default specification is a generic one, in which every ELF implementation follows. However, each platform provides extra features unique to it. The ELF specification for x86 is currently maintained on Github by H.J. Lu: https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI.

Platform-dependent details are referred to as "processor specific" in the generic ELF specification. We will not explore these details, but study the generic details, which are enough for crafting an ELF binary image for our operating system.

5.2   ELF header

To see the information of an ELF header:

```
$ readelf -h hello
```

The output:

```
Output

    ELF Header:
      Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
      Class:                             ELF64
      Data:                              2's complement, little endian
```

```
        Version:                        1 (current)

        OS/ABI:                         UNIX - System V

        ABI Version:                    0

        Type:                           EXEC (Executable file)

        Machine:                        Advanced Micro Devices X86-64

        Version:                        0x1

        Entry point address:            0x400430

        Start of program headers:       64 (bytes into file)

        Start of section headers:       6648 (bytes into file)

        Flags:                          0x0

        Size of this header:            64 (bytes)

        Size of program headers:        56 (bytes)

        Number of program headers:      9

        Size of section headers:        64 (bytes)

        Number of section headers:      31

    Section header string table index: 28
```

Let's go through each field:

Magic Displays the raw bytes that uniquely addresses a file is an ELF
executable binary. Each byte gives a brief information.

In the example, we have the following magic bytes:

Output

```
    Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

Examine byte by byte:

Byte	Description
7f 45 4c 46	Predefined values. The first byte is always 7F, the remaining 3 bytes represent the string "ELF".
02	See Class field below.
01	See Data field below.
01	See Version field below.
00	See OS/ABI field below.
00 00 00 00 00 00 00 00	Padding bytes. These bytes are unused and are always set to 0. Padding bytes are added for proper alignment, and is reserved for future use when more information is needed.

Class A byte in Magic field. It specifies the class or capacity of a file.

Possible values:

Value	Description
0	Invalid class
1	32-bit objects
2	64-bit objects

Data A byte in Magic field. It specifies the data encoding of the processor-specific data in the object file.

Possible values:

Value	Description
0	Invalid data encoding
1	Little endian, 2's complement
2	Big endian, 2's complement

Version A byte in Magic. It specifies the ELF header version number.

Possible values:

Value	Description
0	Invalid version
1	Current version

OS/ABI A byte in `Magic` field. It specifies the target operating system ABI. Originally, it was a padding byte.

Possible values: Refer to the latest ABI document, as it is a long list of different operating systems.

Type Identifies the object file type.

Value	Description
0	No file type
1	Relocatable file
2	Executable file
3	Shared object file
4	Core file
0xff00	Processor specific, lower bound
0xffff	Processor specific, upper bound

The values from `0xff00` to `0xffff` are reserved for a processor to define additional file types meaningful to it.

Machine Specifies the required architecture value for an ELF file e.g. x86_64, MIPS, SPARC, etc. In the example, the machine is of `x86_64` architecture.

Possible values: Please refer to the latest ABI document, as it is a long list of different architectures.

Version Specifies the version number of the current *object file* (not the version of the ELF header, as the above `Version` field specified).

Entry point address Specifies the memory address where the very first code to be executed. The address of `main` function is the default in a normal application program, but it can be any function by explicitly specifying the function name to `gcc`. For the operating system we are going to write, this is the single most important field that we need to retrieve to bootstrap our kernel, and everything else can be ignored.

Start of program headers The offset of the program header table, in
bytes. In the example, this number is `64` bytes, which means the `65th`
byte, or `<start address> + 64`, is the start address of the program
header table. That is, if a program is loaded at address `0x10000` in mem-
ory, then the start address is `0x10000` (the very first byte of `Magic` field,
where the value `0x7f` resides) and the start address of program header
table is `0x10000 + 0x40 = 0x10040`.

Start of section headers The offset of the section header table in bytes,
similar to the start of program headers. In the example, it is `6648` bytes
into file.

Flags Hold processor-specific flags associated with the file. When the
program is loaded, in a x86 machine, `EFLAGS` register is set according
to this value. In the example, the value is `0x0`, which means `EFLAGS`
register is in a clear state.

Size of this header Specifies the total size of ELF header's size in bytes.
In the example, it is `64` bytes, which is equivalent to Start of program
headers. Note that these two numbers are not necessarily equivalent,
as program header table might be placed far away from the ELF header.
The only fixed component in the ELF executable binary is the ELF
header, which appears at the very beginning of the file.

Size of program headers Specifies the size of *each* program header
in bytes. In the example, it is `64` bytes.

Number of program headers Specifies the total number of program
headers. In the example, the file has a total of `9` program headers.

Size of section headers Specifies the size of *each* section header in
bytes. In the example, it is 64 bytes.

Number of section headers Specifies the total number of section head-
ers. In the example, the file has a total of `31` section headers. In a sec-
tion header table, the first entry in the table is always an empty sec-
tion.

Section header string table index Specifies the index of the header
in the section header table that points to the section that holds all

null-terminated strings. In the example, the index is 28, which means it's the 28th entry of the table.

5.3   Section header table

As we know already, code and data compose a program. However, not all types of code and data have the same purpose. For that reason, instead of a big chunk of code and data, they are divided into smaller chunks, and each chunk must satisfy these conditions (according to gABI):

▷ Every section in an object file has exactly one section header describing it. But, section headers may exist that do not have a section.

▷ Each section occupies one contiguous (possibly empty) sequence of bytes within a file. That means, there's no two regions of bytes that are the same section.

▷ Sections in a file may not overlap. No byte in a file resides in more than one section.

▷ An object file may have inactive space. The various headers and the sections might not "cover" every byte in an object file. The contents of the inactive data are unspecified.

To get all the headers from an executable binary e.g. `hello`, use the following command:

```
$ readelf -S hello
```

Here is a sample output (do not worry if you don't understand the output. Just skim to get your eyes familiar with it. We will dissect it soon enough):

```
There are 31 section headers, starting at offset 0x19c8:
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
```

```
[ 0]                     NULL            0000000000000000  00000000
     0000000000000000  0000000000000000          0        0        0
[ 1] .interp            PROGBITS        0000000000400238  00000238
     000000000000001c  0000000000000000   A        0        0        1
[ 2] .note.ABI-tag      NOTE            0000000000400254  00000254
     0000000000000020  0000000000000000   A        0        0        4
[ 3] .note.gnu.build-i  NOTE            0000000000400274  00000274
     0000000000000024  0000000000000000   A        0        0        4
[ 4] .gnu.hash          GNU_HASH        0000000000400298  00000298
     000000000000001c  0000000000000000   A        5        0        8
[ 5] .dynsym            DYNSYM          00000000004002b8  000002b8
     0000000000000048  0000000000000018   A        6        1        8
[ 6] .dynstr            STRTAB          0000000000400300  00000300
     0000000000000038  0000000000000000   A        0        0        1
[ 7] .gnu.version       VERSYM          0000000000400338  00000338
     0000000000000006  0000000000000002   A        5        0        2
[ 8] .gnu.version_r     VERNEED         0000000000400340  00000340
     0000000000000020  0000000000000000   A        6        1        8
[ 9] .rela.dyn          RELA            0000000000400360  00000360
     0000000000000018  0000000000000018   A        5        0        8
[10] .rela.plt          RELA            0000000000400378  00000378
     0000000000000018  0000000000000018  AI        5       24        8
[11] .init              PROGBITS        0000000000400390  00000390
     000000000000001a  0000000000000000  AX        0        0        4
[12] .plt               PROGBITS        00000000004003b0  000003b0
     0000000000000020  0000000000000010  AX        0        0       16
[13] .plt.got           PROGBITS        00000000004003d0  000003d0
     0000000000000008  0000000000000000  AX        0        0        8
[14] .text              PROGBITS        00000000004003e0  000003e0
     0000000000000192  0000000000000000  AX        0        0       16
[15] .fini              PROGBITS        0000000000400574  00000574
     0000000000000009  0000000000000000  AX        0        0        4
[16] .rodata            PROGBITS        0000000000400580  00000580
     0000000000000004  0000000000000004  AM        0        0        4
```

```
[17] .eh_frame_hdr     PROGBITS          0000000000400584  00000584
     000000000000003c  0000000000000000   A       0      0     4
[18] .eh_frame         PROGBITS          00000000004005c0  000005c0
     0000000000000114  0000000000000000   A       0      0     8
[19] .init_array       INIT_ARRAY        0000000000600e10  00000e10
     0000000000000008  0000000000000000   WA      0      0     8
[20] .fini_array       FINI_ARRAY        0000000000600e18  00000e18
     0000000000000008  0000000000000000   WA      0      0     8
[21] .jcr              PROGBITS          0000000000600e20  00000e20
     0000000000000008  0000000000000000   WA      0      0     8
[22] .dynamic          DYNAMIC           0000000000600e28  00000e28
     00000000000001d0  0000000000000010   WA      6      0     8
[23] .got              PROGBITS          0000000000600ff8  00000ff8
     0000000000000008  0000000000000008   WA      0      0     8
[24] .got.plt          PROGBITS          0000000000601000  00001000
     0000000000000020  0000000000000008   WA      0      0     8
[25] .data             PROGBITS          0000000000601020  00001020
     0000000000000010  0000000000000000   WA      0      0     8
[26] .bss              NOBITS            0000000000601030  00001030
     0000000000000008  0000000000000000   WA      0      0     1
[27] .comment          PROGBITS          0000000000000000  00001030
     0000000000000034  0000000000000001   MS      0      0     1
[28] .shstrtab         STRTAB            0000000000000000  000018b6
     000000000000010c  0000000000000000           0      0     1
[29] .symtab           SYMTAB            0000000000000000  00001068
     0000000000000648  0000000000000018          30     47     8
[30] .strtab           STRTAB            0000000000000000  000016b0
     0000000000000206  0000000000000000           0      0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

The first line:

```
    There are 31 section headers, starting at offset 0x19c8
```

summarizes the total number of sections in the file, and where the
address where it starts. Then, comes the listing section by section with
the following header, is also the format of each section output:

Output

```
    [Nr] Name              Type           Address          Offset
         Size              EntSize        Flags  Link  Info  Align
```

Each section has two lines with different fields:

Nr The index of each section.

Name The name of each section.

Type This field (in a section header) identifies the type of each section.
Types are used to classify sections.

Address The starting *virtual* address of each section. Note that the ad-
dresses are virtual only when a program runs in an OS with support
for virtual memory enabled. In our OS, we run on the bare metal, the
addresses will all be physical.

Offset is a distance in bytes, from the first byte of a file to the start of
an object, such as a section or a segment in the context of an ELF bi-
nary file.

Size The size in bytes of each section.

EntSize Some sections hold a table of fixed-size entries, such as a sym-
bol table. For such a section, this member gives the size in bytes of
each entry. The member contains 0 if the section does not hold a ta-
ble of fixed-size entries.

Flags describes attributes of a section. Flags together with a type de-
fines the purpose of a section. Two sections can be of the same type,
but serve different purposes. For example, even though .data and .text
share the same type, .data holds the initialized data of a program while

`.text` holds executable instructions of a program. For that reason, `.data` is given read and write permission, but not executable. Any attempt to execute code in `.data` is denied by the running OS: in Linux, such invalid section usage gives a *segmentation fault*.

ELF gives information to enable an OS with such protection mechanism. However, running on bare metal, nothing can prevent from doing anything. Our OS can execute code in data section, and vice versa, writing to code section.

Table 5.3.1: Section Flags

Flag	Descriptions
W	Bytes in this section are writable during execution.
A	Memory is allocated for this section during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.
X	The section contains executable instructions.
M	The data *in the section* may be merged to eliminate duplication. Each element in the section is compared against other elements in sections with the same name, type and flags. Elements that would have identical values at program run-time may be merged.
S	The data elements in the section consist of null-terminated character strings. The size of each character is specified in the section header's *EntSize* field.
l	Specific large section for x86_64 architecture. This flag is not specified in the Generic ABI but in x86_64 ABI.
I	The *Info* field of this section header holds an index of a section header. Otherwise, the number is the index of something else.
L	Preserve section ordering when linking. If this section is combined with other sections in the output file, it must appear in the same relative order with respect to those sections, as the linked-to section appears with respect to sections the linked-to section is combined with. Apply when the *Link* field of this section's header references another section (the linked-to section)
G	This section is a member (perhaps the only one) of a section group.
T	This section holds **T**hread-**L**ocal **S**torage, meaning that each thread has its own distinct instance of this data. A thread is a distinct execution flow of code. A program can have multiple threads that pack different pieces of code and execute separately, at the same time. We will learn more about threads when writing our kernel.

E	Link editor is to exclude this section from executable and shared library that it builds when those objects are not to be further relocated.
x	Unknown flag to `readelf`. It happens because the linking process can be done manually with a linker like `GNU ld` (we will later later). That is, section flags can be specified manually, and some flags are for a customized ELF that the open-source `readelf` doesn't know of.
O	This section requires special OS-specific processing (beyond the standard linking rules) to avoid incorrect behavior. A link editor encounters sections whose headers contain OS-specific values it does not recognize by Type or Flags values defined by ELF standard, the link editor should combine those sections.
o	All bits included in this flag are reserved for operating system-specific semantics.
p	All bits included in this flag are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Link and Info are numbers that references the indexes of sections, symbol table entries, hash table entries. `Link` field only holds the index of a section, while `Info` field holds an index of a section, a symbol table entry or a hash table entry, depends on the type of a section.

Later when writing our OS, we will handcraft the kernel image by explicitly linking the object files (produced by `gcc`) through a linker script. We will specify the memory layout of sections by specifying at what addresses they will appear in the final image. But we will not assign any section flag and let the linker take care of it. Nevertheless, knowing which flag does what is useful.

Align is a value that enforces the offset of a section should be divisible by the value. Only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraint.

Example 5.3.1. Output of `.interp` section:

```
Output
     [Nr] Name              Type            Address         Offset
          Size              EntSize         Flags Link Info Align
     [ 1] .interp           PROGBITS        0000000000400238  00000238
          000000000000001c  0000000000000000  A     0    0    1
```

Nr is 1.

Type is PROGBITS, which means this section is part of the program.

Address is 0x0000000000400238, which means the program is loaded at this virtual memory address at runtime.

Offset is 0x00000238 *bytes* into file.

Size is 0x000000000000001c in bytes.

EntSize is 0, which means this section does not have any fixed-size entry.

Flags are A (Allocatable), which means this section consumes memory at runtime.

Info and Link are 0 and 0, which means this section links to no section or entry in any table.

Align is 1, which means no alignment.

Example 5.3.2. Output of the .text section:

```
Output    [14] .text              PROGBITS          00000000004003e0  000003e0
               0000000000000192  0000000000000000  AX        0     0      16
```

Nr is 14.

Type is PROGBITS, which means this section is part of the program.

Address is 0x00000000004003e0, which means the program is loaded at this virtual memory address at runtime.

Offset is 0x000003e0 *bytes* into file.

Size is 0x0000000000000192 in bytes.

EntSize is 0, which means this section does not have any fixed-size entry.

Flags are A (Allocatable) and X (Executable), which means this section consumes memory and can be executed as code at runtime.

Info and Link are 0 and 0, which means this section links to no section or entry in any table.

Align is 16, which means the starting address of the section should be divisible by 16, or 0x10. Indeed, it is: 0x3e0/0x10 = 0x3e.

5.4   Understand Section in-depth

In this section, we will learn different details of section types and the purposes of special sections e.g. `.bss, .text, .data`, etc, by looking at each section one by one. We will also examine the content of each section as a hexdump with the commands:

```
$ readelf -x <section name|section number> <file>
```

For example, if you want to examine the content of section with index 25 (the `.bss` section in the sample output) in the file `hello`:

```
$ readelf -x 25 hello
```

Equivalently, using name instead of index works:

```
$ readelf -x .data hello
```

If a section contains strings e.g. string symbol table, the flag `-x` can be replaced with `-p`.

NULL marks a section header as inactive and does not have an associated section. NULL section is always the first entry of section header table. It means, any useful section starts from 1.

Example 5.4.1. The sample output of `NULL` section:

```
Output
     [Nr] Name                 Type               Address            Offset
          Size                 EntSize            Flags  Link  Info  Align
     [ 0]                      NULL               0000000000000000 00000000
          0000000000000000 0000000000000000            0      0     0
```

Examining the content, the section is empty:

```
Output
        Section " has no data to dump.
```

NOTE marks a section with special information that other programs will
check for conformance, compatibility, etc, by a vendor or a system builder.

Example 5.4.2. In the sample output, we have 2 NOTE sections:

```
Output

     [Nr] Name              Type             Address          Offset
          Size              EntSize          Flags  Link  Info  Align
     [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
          0000000000000020  0000000000000000   A        0     0     4
     [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
          0000000000000024  0000000000000000   A        0     0     4
```

Examine 2nd section with the command:

```
$ readelf -x 2 hello
```

we have:

```
Output

     Hex dump of section '.note.ABI-tag':
       0x00400254 04000000 10000000 01000000 474e5500  ............GNU.
       0x00400264 00000000 02000000 06000000 20000000  ............ ...
```

PROGBITS indicates a section holding the main content of a program, either code or data.

Example 5.4.3. There are many PROGBITS sections:

```
Output

     [Nr] Name              Type             Address          Offset
          Size              EntSize          Flags  Link  Info  Align
     [ 1] .interp           PROGBITS         0000000000400238  00000238
          000000000000001c  0000000000000000   A        0     0     1
     ...
     [11] .init             PROGBITS         0000000000400390  00000390
          000000000000001a  0000000000000000   AX       0     0     4
     [12] .plt              PROGBITS         00000000004003b0  000003b0
          0000000000000020  0000000000000010   AX       0     0    16
     [13] .plt.got          PROGBITS         00000000004003d0  000003d0
```

```
              0000000000000008  0000000000000000  AX       0     0     8
      [14] .text            PROGBITS         00000000004003e0  000003e0
              0000000000000192  0000000000000000  AX       0     0     16
      [15] .fini            PROGBITS         0000000000400574  00000574
              0000000000000009  0000000000000000  AX       0     0     4
      [16] .rodata          PROGBITS         0000000000400580  00000580
              0000000000000004  0000000000000004  AM       0     0     4
      [17] .eh_frame_hdr    PROGBITS         0000000000400584  00000584
              000000000000003c  0000000000000000  A        0     0     4
      [18] .eh_frame        PROGBITS         00000000004005c0  000005c0
              0000000000000114  0000000000000000  A        0     0     8
  ...
      [23] .got             PROGBITS         0000000000600ff8  00000ff8
              0000000000000008  0000000000000008  WA       0     0     8
      [24] .got.plt         PROGBITS         0000000000601000  00001000
              0000000000000020  0000000000000008  WA       0     0     8
      [25] .data            PROGBITS         0000000000601020  00001020
              0000000000000010  0000000000000000  WA       0     0     8
      [27] .comment         PROGBITS         0000000000000000  00001030
              0000000000000034  0000000000000001  MS       0     0     1
```

For our operating system, we only need the following section:

.text This section holds all the compiled code of a program.

.data This section holds the initialized data of a program. Since the
data are initialized with actual values, `gcc` allocates the section with
actual byte in the executable binary.

.rodata This section holds read-only data, such as fixed-size strings
in a program, e.g. "Hello World", and others.

.bss This section, shorts for **B**lock **S**tarted by **S**ymbol, holds unini-
tialized data of a program. Unlike other sections, no space is allo-
cated for this section in the image of the executable binary on disk.
The section is allocated only when the program is loaded into main
memory.

Other sections are mainly needed for dynamic linking, that is code linking at runtime for sharing between many programs. To enable such feature, an OS as a runtime environment must be presented. Since we run our OS on bare metal, we are effectively creating such environment. For simplicity, we won't add dynamic linking to our OS.

SYMTAB and DYNSYM These sections hold symbol table. A *symbol table* is an array of entries that describe symbols in a program. A *symbol* is a name assigned to an entity in a program. The types of these entities are also the types of symbols, and these are the possible types of an entity:

Example 5.4.4. In the sample output, section 5 and 29 are symbol tables:

```
Output
    [Nr] Name                Type            Address           Offset
         Size                EntSize         Flags  Link  Info  Align
    [ 5] .dynsym             DYNSYM          00000000004002b8  000002b8
         0000000000000048    0000000000000018  A      6     1     8

    ...
    [29] .symtab             SYMTAB          0000000000000000  00001068
         0000000000000648    0000000000000018         30    47     8
```

To show the symbol table:

```
$ readelf -s hello
```

Output consists of 2 symbol tables, corresponding to the two sections above, .dynsym and .symtab:

```
Output
    Symbol table '.dynsym' contains 4 entries:
       Num:    Value          Size Type    Bind   Vis      Ndx Name
         0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
         1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
         2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
         3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

```
    Symbol table '.symtab' contains 67 entries:
       Num:    Value          Size Type    Bind   Vis      Ndx Name
       .........................................
        59: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   26 _end
        60: 0000000000400430    42 FUNC    GLOBAL DEFAULT   14 _start
        61: 0000000000601038     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
        62: 0000000000400526    32 FUNC    GLOBAL DEFAULT   14 main
        63: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
        64: 0000000000601038     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
        65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
        66: 00000000004003c8     0 FUNC    GLOBAL DEFAULT   11 _init
```

TLS The symbol is associated with a Thread-Local Storage entity.

Num is the index of an entry in a table.

Value is the virtual memory address where the symbol is located.

Size is the size of the entity associated with a symbol.

Type is a symbol type according to table.

   NOTYPE The type of a symbol is not specified.

   OBJECT The symbol is associated with a data object. In C, any variable definition is of `OBJECT` type.

   FUNC The symbol is associated with a function or other executable code.

   SECTION The symbol is associated with a section, and exists primarily for relocation.

   FILE The symbol is the name of a source file associated with an executable binary.

   COMMON The symbol labels an uninitialized variable. That is, when a variable in C is defined as global variable without an initial value, or as an external variable using the extern keyword. In other words, these variables stay in `.bss` section.

Bind is the scope of a symbol.

   LOCAL are symbols that are only visible in the object files that defined them. In C, the `static` modifier marks a symbol (e.g. a variable/function) as local to only the file that defines it.

Example 5.4.5. If we define variables and functions with `static` modifer:

hello.c

```
static int global_static_var = 0;


static void local_func() {
}


int main(int argc, char *argv[])
{
    static int local_static_var = 0;


    return 0;
}
```

Then we get the `static` variables listed as local symbols after compiling:

```
$ gcc -m32 hello.c -o hello
$ readelf -s hello
```

Output

```
Symbol table '.dynsym' contains 5 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
     2: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     3: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (2)
     4: 080484bc     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
Symbol table '.symtab' contains 72 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND

        ......... output omitted .........

    38: 0804a020     4 OBJECT  LOCAL  DEFAULT   26 global_static_var
    39: 0804840b     6 FUNC    LOCAL  DEFAULT   14 local_func
```

```
      40: 0804a024      4 OBJECT  LOCAL   DEFAULT    26 local_static_var.1938
......... output omitted .........
```

GLOBAL *are* symbols that are accessible by other object files when linking together. These symbols are primarily non-`static` functions and non-`static` global data. The `extern` modifier marks a symbol as externally defined elsewhere but is accessible in the final executable binary, so an `extern` variable is also considered `GLOBAL`.

Example 5.4.6. Similar to the `LOCAL` example above, the output lists many `GLOBAL` symbols such as `main`:

```
  Num:    Value  Size Type    Bind    Vis      Ndx Name
......... output omitted .........
  66: 080483e1    10 FUNC    GLOBAL DEFAULT   14 main
......... output omitted .........
```

WEAK are symbols whose definitions can be redefined. Normally, a symbol with multiple definitions are reported as an error by a compiler. However, this constraint is lax when a definition is explicitly marked as weak, which means the default implementation can be replaced by a different definition at link time.

Example 5.4.7. Suppose we have a default implementation of the function `add`:

<div align="center">hello.c</div>

```c
#include <stdio.h>


__attribute__((weak)) int add(int a, int b) {
    printf("warning: function is not implemented.\n")
        ;
    return 0;
}


int main(int argc, char *argv[])
{
```

```
    printf("add(1,2) is %d\n", add(1,2));
    return 0;
}
```

`__attribute__((weak))` is a function attribute. A *function attribute* is extra information for a compiler to handle a function differently from a normal function. In this example, `weak` attribute makes the function `add` a weak function,which means the default implementation can be replaced by a different definition at link time. Function attribute is a feature of a compiler, not standard C.

function attribute

If we do not supply a different function definition in a different file (must be in a different file, otherwise `gcc` reports as an error), then the default implementation is applied. When the function `add` is called, it only prints the message: `"warning: function not implemented"`and returns 0:

```
$ ./hello
warning: function is not implemented.
add(1,2) is 0
```

However, if we supply a different definition in another file e.g. `math.c`:

math.c

```
int add(int a, int b) {
    return a + b;
}
```

and compile the two files together:

```
$ gcc math.c hello.c -o hello
```

Then, when running `hello`, no warning message is printed and the correct value is returned.

Weak symbol is a mechanism to provide a default implementation, but replaceable when a better implementation is available (e.g. more specialized and optimized) at link-time.

Vis is the visibility of a symbol. The following values are available:

Table 5.4.1: Symbol Visibility

Value	Description
DEFAULT	The visibility is specified by the binding type of asymbol. ▷ Global and weak symbols are visible outside of their defining component (executable file or shared object). ▷ Local symbols are hidden. See HIDDEN below.
HIDDEN	A symbol is hidden when the name is not visible to any other program outside of its running program.
PROTECTED	A symbol is protected when it is shared outside of its running program or shared libary and cannot be overridden. That is, there can only be one definition for this symbol across running programs that use it. No program can define its own definition of the same symbol.
INTERNAL	Visibility is processor-specific and is defined by processor-specific ABI.

Ndx is the index of a section that the symbol is in. Aside from fixed index numbers that represent section indexes, index has these special values:

Table 5.4.2: Symbol Index

Value	Description
ABS	The index will not be changed by any symbol relocation.
COM	The index refers to an unallocated common block.
UND	The symbol is undefined in the current object file, which means the symbol depends on the actual definition in another file. Undefined symbols appears when the object file refers to symbols that are available at runtime, from shared library.
LORESERVE HIRESERVE	LORESERVE is the lower boundary of the reserve indexes. Its value is 0xff00. HIREVERSE is the upper boundary of the reserve indexes. Its value is 0xffff. The operating system reserves exclusive indexes between LORESERVE and HIRESERVE, which do not map to any actual section header.
XINDEX	The index is larger than LORESERVE. The actual value will be contained in the section SYMTAB_SHNDX, where each entry is a mapping between a symbol, whose Ndx field is a XINDEX value, and the actual index value.

Others	Sometimes, values such as `ANSI_COM`, `LARGE_COM`, `SCOM`, `SUND` appear. This means that the index is processor-specific.

`Name` is the symbol name.

Example 5.4.8. A C application program always starts from symbol `main`. The entry for `main` in the symbol table in `.symtab` section is:

```
    Num:                 Value  Size Type    Bind    Vis       Ndx Name
     62:     0000000000400526    32 FUNC    GLOBAL DEFAULT    14 main
```

The entry shows that:

▷ `main` is the 62th entry in the table.

▷ `main` starts at address `0x0000000000400526`.

▷ `main` consumes 32 bytes.

▷ `main` is a function.

▷ `main` is in global scope.

▷ `main` is visible to other object files that use it.

▷ `main` is inside the 14th section, which is `.text`. This is logical, since `.text` holds all program code.

STRTAB hold a table of null-terminated strings, called *string table*. The first and last byte of this section is always a NULL character. A string table section exists because a string can be reused by more than one section to represent symbol and section names, so a program like `readelf` or `objdump` can display various objects in a program, e.g. variable, functions, section names, in a human-readable text instead of its raw hex address.

Example 5.4.9. In the sample output, section 28 and 30 are of `STRTAB` type:

```
[Nr] Name                 Type            Address           Offset
     Size                 EntSize         Flags  Link  Info  Align
[28] .shstrtab            STRTAB          0000000000000000  000018b6
     000000000000010c     0000000000000000            0     0     1
[30] .strtab              STRTAB          0000000000000000  000016b0
     0000000000000206     0000000000000000            0     0     1
```

.shstrtab holds all the section names.

.strtab holds the symbols e.g. variable names, function names, struct
names, etc., in a C program, but not fixed-size null-terminated C strings;
the C strings are kept in `.rodata` section.

Example 5.4.10. Strings in those section can be inspected with the
command:

```
$ readelf -p 29 hello
```

The output shows all the section names, with the offset (also the string
index) into `.shstrtab` the table to the left:

```
String dump of section '.shstrtab':
  [     1]   .symtab
  [     9]   .strtab
  [    11]   .shstrtab
  [    1b]   .interp
  [    23]   .note.ABI-tag
  [    31]   .note.gnu.build-id
  [    44]   .gnu.hash
  [    4e]   .dynsym
  [    56]   .dynstr
  [    5e]   .gnu.version
  [    6b]   .gnu.version_r
  [    7a]   .rela.dyn
  [    84]   .rela.plt
```

```
[     8e]   .init
[     94]   .plt.got
[     9d]   .text
[     a3]   .fini
[     a9]   .rodata
[     b1]   .eh_frame_hdr
[     bf]   .eh_frame
[     c9]   .init_array
[     d5]   .fini_array
[     e1]   .jcr
[     e6]   .dynamic
[     ef]   .got.plt
[     f8]   .data
[     fe]   .bss
[    103]   .comment
```

The actual implementation of a string table is a contiguous array of null-terminated strings. The index of a string is the position of its first character in the array. For example, in the above string table, .symtab is at index 1 in the array (NULL character is at index 0). The length of .symtab is 7, plus the NULL character, which occurs 8 bytes in total. So, .strtab starts at index 9, and so on.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	\0	.	s	y	m	t	a	b	\0	.	s	t	r	t	a	b

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000010	\0	.	s	h	s	t	r	t	a	b	\0	.	i	n	t	e

.... and so on ....

Figure 5.4.1: String table in memory of .shstrtab. A red number is the starting index of a string.

Similarly, the output of .strtab:

Output

```
String dump of section '.strtab':
[      1]   crtstuff.c
[      c]   __JCR_LIST__
[     19]   deregister_tm_clones
```

```
[     2e]   __do_global_dtors_aux

[     44]   completed.7585

[     53]   __do_global_dtors_aux_fini_array_entry

[     7a]   frame_dummy

[     86]   __frame_dummy_init_array_entry

[     a5]   hello.c

[     ad]   __FRAME_END__

[     bb]   __JCR_END__

[     c7]   __init_array_end

[     d8]   _DYNAMIC

[     e1]   __init_array_start

[     f4]   __GNU_EH_FRAME_HDR

[    107]   _GLOBAL_OFFSET_TABLE_

[    11d]   __libc_csu_fini

[    12d]   _ITM_deregisterTMCloneTable

[    149]   j

[    14b]   _edata

[    152]   __libc_start_main@@GLIBC_2.2.5

[    171]   __data_start

[    17e]   __gmon_start__

[    18d]   __dso_handle

[    19a]   _IO_stdin_used

[    1a9]   __libc_csu_init

[    1b9]   __bss_start

[    1c5]   main

[    1ca]   _Jv_RegisterClasses

[    1de]   __TMC_END__

[    1ea]   _ITM_registerTMCloneTable
```

HASH holds a symbol hash table, which supports symbol table access.

DYNAMIC holds information for dynamic linking.

NOBITS is similar to `PROGBITS` but occupies no space.

Example 5.4.11. `.bss` section holds uninitialized data, which means

the bytes in the section can have any value. Until a operating system actually loads the section into main memory, there is no need to allocate space for the binary image on disk to reduce the size of a binary file. Here is the details of `.bss` from the example output:

```
[Nr] Name              Type            Address          Offset
     Size              EntSize         Flags  Link  Info  Align
[26] .bss              NOBITS          0000000000601038  00001038
     0000000000000008  0000000000000000  WA      0     0    1
[27] .comment          PROGBITS        0000000000000000  00001038
     0000000000000034  0000000000000001  MS      0     0    1
```

In the above output, the size of the section is only 8 bytes, while the offsets of both sections are the same, which means `.bss` consumes no byte of the executable binary on disk.

Notice that the `.comment` section has no starting address. This means that this section is discarded when the executable binary is loaded into memory.

REL holds relocation entries without explicit addends. This type will be explained in details in 8.1

RELA holds relocation entries with explicit addends. This type will be explained in details in 8.1

INIT_ARRAY is an array of function pointers for program initialization. When an application program runs, before getting to `main()`, initialization code in `.init` and this section are executed first. The first element in this array is an ignored function pointer.

It might not make sense when we can include initialization code in the `main()` function. However, for shared object files where there are no `main()`, this section ensures that the initialization code from an object file executes before any other code to ensure a proper environment for main code to run properly. It also makes an object file more modularity, as the main application code needs not to be responsible for initializing a proper environment for using a particular object file, but the object file itself. Such a clear division makes code cleaner.

However, we will not use any `.init` and `INIT_ARRAY` sections in our operating system, for simplicity, as initializing an environment is part of the operating-system domain.

Example 5.4.12. To use the `INIT_ARRAY`, we simply mark a function with the attribute `constructor`:

<div align="center">hello.c</div>

```c
#include <stdio.h>


__attribute__((constructor)) static void init1(){
    printf("%s\n", __FUNCTION__);
}


__attribute__((constructor)) static void init2(){
    printf("%s\n", __FUNCTION__);
}



int main(int argc, char *argv[])
{
    printf("hello world\n");


    return 0;
}
```

The program automatically calls the constructor without explicitly invoking it:

```
$ gcc -m32 hello.c -o hello
$ ./hello
init1
init2
hello world
```

Example 5.4.13. Optionally, a constructor can be assigned with a priority from 101 onward. The priorities from 0 to 100 are reserved

for `gcc`. If we want `init2` to run before `init1`, we give it a higher priority:

hello.c

```
#include <stdio.h>


__attribute__((constructor(102))) static void init1(){
    printf("%s\n", __FUNCTION__);
}


__attribute__((constructor(101))) static void init2(){
    printf("%s\n", __FUNCTION__);
}



int main(int argc, char *argv[])
{
    printf("hello␣world\n");


    return 0;
}
```

The call order should be exactly as specified:

```
$ gcc -m32 hello.c -o hello
$ ./hello
init2
init1
hello world
```

Example 5.4.14. We can add initialization functions using another method:

hello.c

```
#include <stdio.h>
```

```
void init1() {
    printf("%s\n", __FUNCTION__);
}


void init2() {
    printf("%s\n", __FUNCTION__);
}


/* Without typedef, init is a definition of a function
    pointer.
   With typedef, init is a declaration of a type.*/
typedef void (*init)();


__attribute__((section(".init_array"))) init init_arr[2]
    = {init1, init2};


int main(int argc, char *argv[])
{
    printf("hello world!\n");


    return 0;
}
```

The attribute `section("...")` put a function into a particular section rather than the default `.text`. In this example, it is `.init_array`. The section name is not necessary the same as the standard header in an ELF file (such as `.text` or `.init_array`, but can be anything. Non-standard section names are often used for controlling the final binary layout of a compiled program. We will explore this techinque in more details when learning the GNU `ld` linker and the linking process. Again, the program automatically calls the constructors without explicitly invoking it:

```
$ gcc -m32 hello.c -o hello
$ ./hello
init1
init2
hello world!
```

FINI_ARRAY is an array of function pointers for program termination, called after exiting `main()`. If the application terminate abnormally, such as through `abort()` call or a crash, the `.finit_array` is ignored.

Example 5.4.15. A destructor is automatically called after exiting main(), if one or more available:

hello.c

```c
#include <stdio.h>

__attribute__((destructor)) static void destructor(){
    printf("%s\n", __FUNCTION__);
}

int main(int argc, char *argv[])
{
    printf("hello world\n");

    return 0;
}
```

```
$ gcc -m32 hello.c -o hello
$ ./hello
hello world
destructor
```

PREINIT_ARRAY is an array of function pointers that are invoked before all other initialization functions in `INIT_ARRAY`.

Example 5.4.16. To use the `.preinit_array`, the only way to put functions into this section is to use the attribute `section()`:

<div align="center">hello.c</div>

```c
#include <stdio.h>


void preinit1() {
    printf("%s\n", __FUNCTION__);
}


void preinit2() {
    printf("%s\n", __FUNCTION__);
}


void init1() {
    printf("%s\n", __FUNCTION__);
}


void init2() {
    printf("%s\n", __FUNCTION__);
}



typedef void (*preinit)();
typedef void (*init)();

__attribute__((section(".preinit_array"))) preinit
    preinit_arr[2] = {preinit1, preinit2};
__attribute__((section(".preinit_array"))) init init_arr
    [2] = {init1, init2};

int main(int argc, char *argv[])
{
    printf("hello world!\n");
```

```
    return 0;
}
```

```
$ gcc -m32 hello2.c -o hello2
$ ./hello2
preinit1
preinit2
init1
init2
hello world!
```

GROUP defines a section group, which is the same section that appears in different object files but when merged into the final executable binary file, only one copy is kept and the rest in other object files are discarded. This section is only relevant in C++ object files, so we will not examine further.

SYMTAB_SHNDX is a section containing extended section indexes, that are associated with a symbol table. This section only appears when the `Ndx` value of an entry in the symbol table exceeds the `LORESERVE` value. This section then maps between a symbol and an actual index value of a section header.

Upon understanding section types, we can understand the number in `Link` and `Info` fields:

Exercise 5.4.1. Verify that the value of the `Link` field of a SYMTAB section is the index of a `STRTAB` section.

Exercise 5.4.2. Verify that the value of the `Info` field of a SYMTAB section is the index of last local symbol + 1. It means, in the symbol table, from the index listed by `Info` field onward, no local symbol appears.

Exercise 5.4.3. Verify that the value of the `Info` field of a `REL` section is the index of the SYMTAB section.

Exercise 5.4.4. Verify that the value of the `Link` field of a `REL` section is the index of the section where relocation is applied. For example. if the section is `.rel.text`, then the relocating section should be `.text`.

Type	Link	Info
DYNAMIC	Entries in this section uses the section index of the dynamic string table.	0
HASH GNU_HASH	The section index of the symbol table to which the hash table applies.	0
REL RELA	The section index of the associated symbol table.	The section index to which the relocation applies.
SYMTAB DYNSYM	The section index of the associated string table.	One greater than the symbol table index of the last local symbol.
GROUP	The section index of the associated symbol table.	The symbol index of an entry in the associated symbol table. The name of the specified symbol table entry provides a signature for the section group.
SYMTAB_SHNDX	The section header index of the associated symbol table.	

Table 5.4.3: The meannings of `Link` and `Info` depend on `section types.` interpretation

5.5   Program header table

A *program header table* is an array of program headers that defines the memory layout of a program at runtime.

A *program header* is a description of a program segment.

A *program segment* is a collection of related sections. A segment contains zero or more sections. An operating system when loading a program, *only use segments*, not sections. To see the information of a program header table, we use the `-l` option with `readelf:`

```
$ readelf -l <binary file>
```

Similar to a section, a program header also has types:

PHDR specifies the location and size of the program header table itself, both in the file and in the memory image of the program

INTERP specifies the location and size of a null-terminated path name to invoke as an interpreter for linking runtime libraries.

LOAD specifies a loadable segment. That is, this segment is loaded into main memory.

DYNAMIC specifies dynamic linking information.

NOTE specifies the location and size of auxiliary information.

TLS specifies the *Thread-Local Storage template*, which is formed from the combination of all sections with the flag TLS.

GNU_STACK indicates whether the program's stack should be made executable or not. Linux kernel uses this type.

A segment also has permission, which is a combination of these 3 values:

▷ Read (R)

▷ Write (W)

▷ Execute (E)

Table 5.5.1: Segment Permission

Permission	Description
R	Readable
W	Writable
E	Executable

Example 5.5.1. The command to get the program header table:

```
$ readelf -l hello
```

Output:

```
Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000070c 0x000000000000070c  R E    200000
```

```
     LOAD            0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                     0x0000000000000228 0x0000000000000230  RW     200000
     DYNAMIC         0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
                     0x00000000000001d0 0x00000000000001d0  RW      8
     NOTE            0x0000000000000254 0x0000000000400254 0x0000000000400254
                     0x0000000000000044 0x0000000000000044  R       4
     GNU_EH_FRAME    0x00000000000005e4 0x00000000004005e4 0x00000000004005e4
                     0x0000000000000034 0x0000000000000034  R       4
     GNU_STACK       0x0000000000000000 0x0000000000000000 0x0000000000000000
                     0x0000000000000000 0x0000000000000000  RW      10
     GNU_RELRO       0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                     0x00000000000001f0 0x00000000000001f0  R       1
  Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
 .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini
 .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

In the sample output, `LOAD` segment appears twice:

Output

```
     LOAD            0x0000000000000000 0x0000000000400000 0x0000000000400000
                     0x000000000000070c 0x000000000000070c  R E     200000
     LOAD            0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                     0x0000000000000228 0x0000000000000230  RW      200000
```

Why? Notice the permission:

▷ the upper `LOAD` has Read and Execute permission. This is a *text* seg-

ment. A text segment contains read-only instructions and read-only data.

▷ the lower `LOAD` has Read and Write permission. This is a *data* segment. It means that this segment can be read and written to, but is not allowed to be used as executable code, for security reason.

Then, `LOAD` contains the following sections:

Output

```
    02      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini
.rodata .eh_frame_hdr .eh_frame
    03      .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
```

The first number is the index of a program header in program header table, and the remaining text is the list of all sections within a segment. Unfortunately, `readelf` does not print the index, so a user needs to keep track manually which segment is of which index. First segment starts at index 0, second at index 1 and so on. `LOAD` are segments at index 2 and 3. As can be seen from the two lists of sections, most sections are loadable and is available at runtime.

5.6   Segments vs sections

As mentioned earlier, an operating system loads program segments, not sections. However, a question arises: Why doesn't the operating system use sections instead? After all, a section also contains similar information to a program segment, such as the type, the virtual memory address to be loaded, the size, the attributes, the flags and align. As explained before, a segment is the perspective of an operating system, while a section is the perspective of a linker. To understand why, looking into the structure of a segment, we can easily see:

▷ A segment is a collection of sections. It means that sections are logically grouped together by their attributes. For example, all sections

in a `LOAD` segment are always loaded by the operating system; all sections have the same permission, either a `RE` (Read + Execute) for executable sections, or `RW` (Read + Write) for data sections.

▷ By grouping sections into a segment, it is easier for an operating system to batch load sections just once by loading the start and end of a segment, instead of loading section by section.

▷ Since a segment is for loading a program and a section is for linking a program, all the sections in a segment is *within its start and end virtual memory addresses of a segment.*

To see the last point clearer, consider an example of linking two object files. Suppose we have two source files:

hello.c

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
    printf("Hello World\n");
    return 0;
}
```

and:

math.c

```c
int add(int a, int b) {
    return a + b;
}
```

Now, compile the two source files as *object files*:

```
$ gcc -m32 -c math.c
$ gcc -m32 -c hello.c
```

Then, we check the sections of `math.o`:

```
$ readelf -S math.o
```

```
There are 11 section headers, starting at offset 0x1a8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00000d 00  AX  0   0  1
  [ 2] .data             PROGBITS        00000000 000041 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          00000000 000041 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        00000000 000041 000035 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        00000000 000076 000000 00      0   0  1
  [ 6] .eh_frame         PROGBITS        00000000 000078 000038 00   A  0   0  4
  [ 7] .rel.eh_frame     REL             00000000 00014c 000008 08   I  9   6  4
  [ 8] .shstrtab         STRTAB          00000000 000154 000053 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 0000b0 000090 10     10   8  4
  [10] .strtab           STRTAB          00000000 000140 00000c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

As shown in the output, all the section virtual memory addresses of
every section are set to 0. At this stage, each object file is simply a *block
of binary* that contains code and data. Its existence is to serve as a ma-
terial container for the final product, which is the executable binary. As
such, the virtual addresses in `hello.o` are all zeroes.

No segment exists at this stage:

```
$ readelf -l math.o
There are no program headers in this file.
```

The same happens to other object file:

Output

```
There are 13 section headers, starting at offset 0x224:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002e 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 0001ac 000010 08   I 11   1  4
  [ 3] .data             PROGBITS        00000000 000062 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000062 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        00000000 000062 00000c 00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 00006e 000035 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 0000a3 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        00000000 0000a4 000044 00   A  0   0  4
  [ 9] .rel.eh_frame     REL             00000000 0001bc 000008 08   I 11   8  4
  [10] .shstrtab         STRTAB          00000000 0001c4 00005f 00      0   0  1
  [11] .symtab           SYMTAB          00000000 0000e8 0000b0 10     12   9  4
  [12] .strtab           STRTAB          00000000 000198 000013 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

```
$ readelf -l hello.o
There are no program headers in this file.
```

Only when object files are combined into a final executable binary, sections are fully realized:

```
$ gcc -m32 math.o hello.o -o hello
$ readelf -S hello.
```

Output

```
There are 31 section headers, starting at offset 0x1804:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
```

```
[ 0]                       NULL            00000000 000000 000000 00       0   0  0
[ 1] .interp              PROGBITS        08048154 000154 000013 00    A  0   0  1
[ 2] .note.ABI-tag        NOTE            08048168 000168 000020 00    A  0   0  4
[ 3] .note.gnu.build-i    NOTE            08048188 000188 000024 00    A  0   0  4
[ 4] .gnu.hash            GNU_HASH        080481ac 0001ac 000020 04    A  5   0  4
[ 5] .dynsym              DYNSYM          080481cc 0001cc 000050 10    A  6   1  4
[ 6] .dynstr              STRTAB          0804821c 00021c 00004a 00    A  0   0  1
[ 7] .gnu.version         VERSYM          08048266 000266 00000a 02    A  5   0  2
[ 8] .gnu.version_r       VERNEED         08048270 000270 000020 00    A  6   1  4
[ 9] .rel.dyn             REL             08048290 000290 000008 08    A  5   0  4
[10] .rel.plt             REL             08048298 000298 000010 08   AI  5  24  4
[11] .init                PROGBITS        080482a8 0002a8 000023 00   AX  0   0  4
[12] .plt                 PROGBITS        080482d0 0002d0 000030 04   AX  0   0 16
[13] .plt.got             PROGBITS        08048300 000300 000008 00   AX  0   0  8
[14] .text                PROGBITS        08048310 000310 0001a2 00   AX  0   0 16
[15] .fini                PROGBITS        080484b4 0004b4 000014 00   AX  0   0  4
[16] .rodata              PROGBITS        080484c8 0004c8 000014 00    A  0   0  4
[17] .eh_frame_hdr        PROGBITS        080484dc 0004dc 000034 00    A  0   0  4
[18] .eh_frame            PROGBITS        08048510 000510 0000ec 00    A  0   0  4
[19] .init_array          INIT_ARRAY      08049f08 000f08 000004 00   WA  0   0  4
[20] .fini_array          FINI_ARRAY      08049f0c 000f0c 000004 00   WA  0   0  4
[21] .jcr                 PROGBITS        08049f10 000f10 000004 00   WA  0   0  4
[22] .dynamic             DYNAMIC         08049f14 000f14 0000e8 08   WA  6   0  4
[23] .got                 PROGBITS        08049ffc 000ffc 000004 04   WA  0   0  4
[24] .got.plt             PROGBITS        0804a000 001000 000014 04   WA  0   0  4
[25] .data                PROGBITS        0804a014 001014 000008 00   WA  0   0  4
[26] .bss                 NOBITS          0804a01c 00101c 000004 00   WA  0   0  1
[27] .comment             PROGBITS        00000000 00101c 000034 01   MS  0   0  1
[28] .shstrtab            STRTAB          00000000 0016f8 00010a 00       0   0  1
[29] .symtab              SYMTAB          00000000 001050 000470 10      30  48  4
[30] .strtab              STRTAB          00000000 0014c0 000238 00       0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
```

> O (extra OS processing required) o (OS specific), p (processor specific)

Every loadable section is assigned an address, highlighted in green. The reason each section got its own address is that in reality, gcc *does not combine an object by itself, but invokes the linker* ld. The linker ld uses the default script that it can find in the system to build the executable binary. In the default script, a segment is assigned a starting address 0x8048000 and sections belong to it. Then:

▷ 1st section address = starting segment address + section offset = 0x8048000 + 0x154 = 0x08048154

▷ 2nd section address = starting segment address + section offset = 0x8048000 + 0x168 = 0x08048168

▷ and so on until the last loadable section.

Indeed, the end address of a segment is also the end address of the final section. We can see this by listing all the segments:

```
$ readelf -l hello
```

And check, for example, LOAD segment which starts at 0x08048000 and end at 0x08048000 + 0x005fc = 0x080485fc:

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x8048310
There are 9 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x005fc 0x005fc R E 0x1000
  LOAD           0x000f08 0x08049f08 0x08049f08 0x00114 0x00118 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0004dc 0x080484dc 0x080484dc 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

```
   GNU_RELRO       0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R    0x1
 Section to Segment mapping:
  Segment Sections...
   00
   01      .interp
   02      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini
.rodata .eh_frame_hdr .eh_frame
   03      .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04      .dynamic
   05      .note.ABI-tag .note.gnu.build-id
   06      .eh_frame_hdr
   07
   08      .init_array .fini_array .jcr .dynamic .got
```

The last section in the first `LOAD` segment is `.eh_frame`. The `.eh_frame`
section starts at `0x0804851` because the start address is `0x08048000`, the
offset into the file is `0x510`. The end address of `.eh_frame` should be: $0x08048000 + 0x510 + 0xec = 0x080485fc$
because the segment size is `0xec`. This is exactly the same as the end ad-
dress of the first `LOAD` segment above: $0x08048000 + 0x5ec = 0x080485fc$.

Chapter 8 will explore this whole process in detail.

6

Runtime inspection and debug

A *debugger* is a program that allows inspection of a running program.   *debugger*
A debugger can start and run a program then stop at a specific line for
examining the state of the program at that point. The point where the
debugger stop (but not halt) is called a *breakpoint*.

We will be using the **GDB - G**NU **D**eb*ugger* for debugging our ker-
nel. `gdb` is the program name. `gdb` can do four main kinds of things:

▷ Start your program, specifying anything that might affect its behav-
ior.

▷ Make your program stop on specified conditions.

▷ Examine what has happened, when your program has stopped

▷ Change things in your program, so you can experiment with correct-
ing the effects of one bug and go on to learn about another

6.1   A sample program

There must be an existing program for debugging. The good old "Hello
World" program suffices for the educational purpose in this chapter:

hello.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

We compile it with debugging information with the option `-g`:

```
$ gcc -m32 -g hello.c -o hello
```

Finally, we start gdb with the program as argument:

```
$ gdb hello
```

6.2   Static inspection of a program

Before inspecting a program at runtime, `gdb` loads it first. Upon loading
into memory (but without running), a lot of useful information can be
retrieve for inspection. The commands in this section can be used be-
fore the program runs. However, they are also usable when the program
runs and can display even more information.

6.2.1 Command: `info target/info file/info files`

This command prints the information of the target being debugged. A
target is the debugging program.

Example 6.2.1. The output of the command from `hello` program, a
local target in detail:

```
(gdb) info target
```

Output

```
Symbols from "/tmp/hello".
Local exec file:
'/tmp/hello', file type elf32-i386.
Entry point: 0x8048310
0x08048154 - 0x08048167 is .interp
0x08048168 - 0x08048188 is .note.ABI-tag
0x08048188 - 0x080481ac is .note.gnu.build-id
0x080481ac - 0x080481cc is .gnu.hash
0x080481cc - 0x0804821c is .dynsym
0x0804821c - 0x08048266 is .dynstr
0x08048266 - 0x08048270 is .gnu.version
0x08048270 - 0x08048290 is .gnu.version_r
0x08048290 - 0x08048298 is .rel.dyn
0x08048298 - 0x080482a8 is .rel.plt
0x080482a8 - 0x080482cb is .init
0x080482d0 - 0x08048300 is .plt
0x08048300 - 0x08048308 is .plt.got
0x08048310 - 0x080484a2 is .text
0x080484a4 - 0x080484b8 is .fini
0x080484b8 - 0x080484cd is .rodata
0x080484d0 - 0x080484fc is .eh_frame_hdr
0x080484fc - 0x080485c8 is .eh_frame
0x08049f08 - 0x08049f0c is .init_array
0x08049f0c - 0x08049f10 is .fini_array
0x08049f10 - 0x08049f14 is .jcr
0x08049f14 - 0x08049ffc is .dynamic
0x08049ffc - 0x0804a000 is .got
0x0804a000 - 0x0804a014 is .got.plt
0x0804a014 - 0x0804a01c is .data
0x0804a01c - 0x0804a020 is .bss
```

The output displayed reports:

▷ Path of a symbol file. A *symbol file* is the file that contains the debug-
   ging information. Usually, this is the same file as the binary, but it is

common to separate between an executable binary and its debugging information into 2 files, especially for remote debugging. In the example, it is this line:

```
Symbols from "/tmp/hello".
```

▷ The path of the debugging program and its file type. In the example, it is this line:

```
Local exec file:
'/tmp/hello', file type elf32-i386.
```

▷ The entry point to the debugging program. That is, the very first code the program runs. In the example, it is this line:

```
Entry point: 0x8048310
```

▷ A list of sections with its starting and ending addresses. In the example, it is the remaining output.

Example 6.2.2. If the debugging program runs in a different machine, it is a remote target and **gdb** only prints a brief information:

```
(gdb) info target
```

Output

```
Remote serial target in gdb-specific protocol:
Debugging a target over a serial line.
```

6.2.2 Command: *maint info sections*

This command is similar to `info target` but give extra information about program sections, specifically the file offset and the flags of each section.

Example 6.2.3. Here is the output when running against **hello** program:

```
(gdb) maint info sections
```

Output

```
Exec file:
    '/tmp/hello', file type elf64-x86-64.
[0]      0x00400238->0x00400254 at 0x00000238: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
[1]      0x00400254->0x00400274 at 0x00000254: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
[2]      0x00400274->0x00400298 at 0x00000274: .note.gnu.build-id ALLOC LOAD READONLY DATA HAS_CONTENTS
[3]      0x00400298->0x004002b4 at 0x00000298: .gnu.hash ALLOC LOAD READONLY DATA HAS_CONTENTS
[4]      0x004002b8->0x00400318 at 0x000002b8: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
[5]      0x00400318->0x00400355 at 0x00000318: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
[6]      0x00400356->0x0040035e at 0x00000356: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
[7]      0x00400360->0x00400380 at 0x00000360: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
....remaining output omitted....
```

The output is similar to `info target`, but with more details. Next to the section names are the section flags, which are attributes of a section. Here, we can see that the sections with `LOAD` flag are from `LOAD` segment. The command can be combined with the section flags for filtered outputs:

ALLOBJ displays sections for all loaded object files, including shared libraries. Shared libraries are only displayed when the program is already running.

section names displays only named sections.

Example 6.2.4. The command:

```
(gdb) maint info sections .text .data .bss
```

only displays `.text`, `.data` and `.bss` sections:

Output

```
Exec file:
    '/tmp/hello', file type elf64-x86-64.
[13]     0x00400430->0x004005c2 at 0x00000430: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
[24]     0x00601028->0x00601038 at 0x00001028: .data ALLOC LOAD DATA HAS_CONTENTS
[25]     0x00601038->0x00601040 at 0x00001038: .bss ALLOC
```

section-flags displays only sections with specified section flags. Note that these section flags are specific to `gdb`, though it is based on the section attributes defined previously. Currently, `gdb` understands the following flags:

ALLOC Section will have space allocated in the process when loaded. Set for all sections except those containing debug information.

LOAD Section will be loaded from the file into the child process memory. Set for pre-initialized code and data, clear for .bss sections.

RELOC Section needs to be relocated before loading.

READONLY Section cannot be modified by the child process.

CODE Section contains executable code only.

DATA Section contains data only (no executable code).

ROM Section will reside in ROM.

CONSTRUCTOR Section contains data for constructor/destructor lists.

HAS_CONTENTS Section is not empty.

NEVER_LOAD An instruction to the linker to not output the section.

COFF_SHARED_LIBRARY A notification to the linker that the section contains COFF shared library information. COFF is an object file format, similar to ELF. While ELF is the file format for an executable binary, COFF is the file format for an object file.

IS_COMMON Section contains common symbols.

Example 6.2.5. We can restrict the output to only display sections that contain code with the command:

```
(gdb) maint info sections CODE
```

The output:

```
Exec file:
    '/tmp/hello', file type elf64-x86-64.
 [10]     0x004003c8->0x004003e2 at 0x000003c8: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
 [11]     0x004003f0->0x00400420 at 0x000003f0: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
 [12]     0x00400420->0x00400428 at 0x00000420: .plt.got ALLOC LOAD READONLY CODE HAS_CONTENTS
 [13]     0x00400430->0x004005c2 at 0x00000430: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
 [14]     0x004005c4->0x004005cd at 0x000005c4: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
```

6.2.3 Command: `info functions`

This commands list all function names and their loaded addresses. The names can be filtered with a regular expression.

Example 6.2.6. Run the command, we get the following output:

```
(gdb) info functions
```

Output

```
All defined functions:
File hello.c:
int main(int, char **);
Non-debugging symbols:
0x00000000004003c8  _init
0x0000000000400400  puts@plt
0x0000000000400410  __libc_start_main@plt
0x0000000000400430  _start
0x0000000000400460  deregister_tm_clones
0x00000000004004a0  register_tm_clones
0x00000000004004e0  __do_global_dtors_aux
0x0000000000400500  frame_dummy
0x0000000000400550  __libc_csu_init
0x00000000004005c0  __libc_csu_fini
0x00000000004005c4  _fini
```

6.2.4 Command: `info variables`

This command lists all global and static variable names, or filtered with a regular expression.

Example 6.2.7. If we add a global variable `int i` into the sample source program and recompile then run the command, we get the following output:

```
(gdb) info variables
```

Output

```
All defined variables:
File hello.c:
int i;
Non-debugging symbols:
0x00000000004005d0  _IO_stdin_used
0x00000000004005e4  __GNU_EH_FRAME_HDR
0x0000000000400708  __FRAME_END__
0x0000000000600e10  __frame_dummy_init_array_entry
0x0000000000600e10  __init_array_start
0x0000000000600e18  __do_global_dtors_aux_fini_array_entry
0x0000000000600e18  __init_array_end
0x0000000000600e20  __JCR_END__
0x0000000000600e20  __JCR_LIST__
0x0000000000600e28  _DYNAMIC
0x0000000000601000  _GLOBAL_OFFSET_TABLE_
0x0000000000601028  __data_start
0x0000000000601028  data_start
0x0000000000601030  __dso_handle
0x000000000060103c  __bss_start
0x000000000060103c  _edata
0x000000000060103c  completed
0x0000000000601040  __TMC_END__
0x0000000000601040  _end
```

6.2.5 Command: `disassemble/disas`

This command displays the assembly code of the executable file.

Example 6.2.8. gdb can display the assembly code of a function:

```
(gdb) disassemble main
```

Output

```
Dump of assembler code for function main:
   0x0804840b <+0>:   lea     ecx,[esp+0x4]
   0x0804840f <+4>:   and     esp,0xfffffff0
   0x08048412 <+7>:   push    DWORD PTR [ecx-0x4]
   0x08048415 <+10>:  push    ebp
   0x08048416 <+11>:  mov     ebp,esp
   0x08048418 <+13>:  push    ecx
   0x08048419 <+14>:  sub     esp,0x4
   0x0804841c <+17>:  sub     esp,0xc
   0x0804841f <+20>:  push    0x80484c0
   0x08048424 <+25>:  call    0x80482e0 <puts@plt>
   0x08048429 <+30>:  add     esp,0x10
   0x0804842c <+33>:  mov     eax,0x0
   0x08048431 <+38>:  mov     ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>:  leave
   0x08048435 <+42>:  lea     esp,[ecx-0x4]
   0x08048438 <+45>:  ret
End of assembler dump.
```

Example 6.2.9. It would be more useful if source is included:

```
(gdb) disassemble /s main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>: lea     ecx,[esp+0x4]
   0x0804840f <+4>: and     esp,0xfffffff0
   0x08048412 <+7>: push    DWORD PTR [ecx-0x4]
   0x08048415 <+10>: push   ebp
   0x08048416 <+11>: mov    ebp,esp
   0x08048418 <+13>: push   ecx
   0x08048419 <+14>: sub    esp,0x4
```

```
5      printf("Hello World!\n");
   0x0804841c <+17>: sub     esp,0xc
   0x0804841f <+20>: push    0x80484c0
   0x08048424 <+25>: call    0x80482e0 <puts@plt>
   0x08048429 <+30>: add     esp,0x10
6      return 0;
   0x0804842c <+33>: mov     eax,0x0
7 }
   0x08048431 <+38>: mov     ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: leave
   0x08048435 <+42>: lea     esp,[ecx-0x4]
   0x08048438 <+45>: ret
End of assembler dump.
```

Now the high level source (in green text) is included as part of the assembly dump. Each line is backed by the corresponding assembly code below it.

Example 6.2.10. If the option **/r** is added, raw instructions in hex are included, just like how **objdump** displays assembly code by default:

```
(gdb) disassemble /rs main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>:   8d 4c 24 04    lea     ecx,[esp+0x4]
   0x0804840f <+4>:   83 e4 f0       and     esp,0xfffffff0
   0x08048412 <+7>:   ff 71 fc       push    DWORD PTR [ecx-0x4]
   0x08048415 <+10>:  55    push     ebp
   0x08048416 <+11>:  89 e5    mov     ebp,esp
   0x08048418 <+13>:  51    push     ecx
   0x08048419 <+14>:  83 ec 04       sub     esp,0x4
5      printf("Hello World!\n");
   0x0804841c <+17>:  83 ec 0c       sub     esp,0xc
```

```
    0x0804841f <+20>:  68 c0 84 04 08 push    0x80484c0
    0x08048424 <+25>:  e8 b7 fe ff ff call    0x80482e0 <puts@plt>
    0x08048429 <+30>:  83 c4 10       add     esp,0x10
6       return 0;
    0x0804842c <+33>:  b8 00 00 00 00 mov     eax,0x0
7   }
    0x08048431 <+38>:  8b 4d fc       mov     ecx,DWORD PTR [ebp-0x4]
    0x08048434 <+41>:  c9 leave
    0x08048435 <+42>:  8d 61 fc       lea     esp,[ecx-0x4]
    0x08048438 <+45>:  c3 ret
End of assembler dump.
```

Example 6.2.11. A function in a specific file can also be specified:

```
(gdb) disassemble /sr 'hello.c'::main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
    0x0804840b <+0>:   8d 4c 24 04  lea     ecx,[esp+0x4]
    0x0804840f <+4>:   83 e4 f0     and     esp,0xfffffff0
    0x08048412 <+7>:   ff 71 fc     push    DWORD PTR [ecx-0x4]
    0x08048415 <+10>: 55     push    ebp
    0x08048416 <+11>: 89 e5 mov     ebp,esp
    0x08048418 <+13>: 51     push    ecx
    0x08048419 <+14>: 83 ec 04     sub     esp,0x4
5       printf("Hello World!\n");
    0x0804841c <+17>: 83 ec 0c     sub     esp,0xc
    0x0804841f <+20>: 68 c0 84 04 08 push    0x80484c0
    0x08048424 <+25>: e8 b7 fe ff ff call    0x80482e0 <puts@plt>
    0x08048429 <+30>: 83 c4 10     add     esp,0x10
6       return 0;
    0x0804842c <+33>: b8 00 00 00 00 mov     eax,0x0
7   }
```

```
    0x08048431 <+38>: 8b 4d fc        mov     ecx,DWORD PTR [ebp-0x4]
    0x08048434 <+41>: c9 leave
    0x08048435 <+42>: 8d 61 fc        lea     esp,[ecx-0x4]
    0x08048438 <+45>: c3 ret
End of assembler dump.
```

The filename must be included in a single quote, and the function must be prefixed by double colons e.g. 'hello.c'::main to specify disassembling of the function main in the file hello.c.

6.2.6   Command: x

This command examines the content of a given memory range.

Example 6.2.12. We can examine the raw content in main:

```
(gdb) x main
```

```
    0x804840b <main>: 0x04244c8d
```

By default, without any argument, the command only prints the content of a single memory address. In this case, that is the starting memory address in main.

Example 6.2.13. With format arguments, the command can print a range of memory in a specific format.

```
(gdb) x/20b main
```

```
    0x804840b <main>:    0x8d 0x4c 0x24 0x04 0x83 0xe40xf0 0xff
    0x8048413 <main+8>:  0x71 0xfc 0x55 0x89 0xe5 0x510x83 0xec
    0x804841b <main+16>: 0x04 0x83 0xec 0x0c
```

/20b main argument means that the command prints 20 bytes, where main starts in memory.

The general form for format argument is: /<repeated count><format letter>

If the repeated count is not supplied, by default `gdb` supplies the count as 1. The format letter is one the following values:

Letter	Description
o	Print the memory content in *octal* format.
x	Print the memory content in hex format.
d	Print the memory content in decimal format.
u	Print the memory content in *unsigned decimal* format.
t	Print the memory content in *binary* format.
f	Print the memory content in *float* format.
a	Print the memory content as *memory addresses.*
i	Print the memory content as a series of assembly instructions, similar to `disassemble` command.
c	Print the memory content as an array of ASCII characters.
s	Print the memory content as a string

Depends on the circumstance, certain format is advantageous than the others. For example, if a memory region contains floating-point numbers, then it is better to use the format `f` than viewing the number as separated 1-byte hex numbers.

6.2.7 Command: `print/p`

Examining raw memory is useful but usually it is better to have a more human-readable output. This command does precisely the task: it pretty-prints an expression. An expression can be a global variable, a local variable in current stack frame, a function, a register, a number, etc.

6.3   Runtime inspection of a program

The main use of a debugger is to examine the state of a program, when it is running. `gdb` provides a set of useful commands for retrieving useful runtime information.

6.3.1 Command: `run`

This command starts running the program.

Example 6.3.1. Run the hello program:

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Hello World!
[Inferior 1 (process 1002) exited normally]
```

The program runs successfully and printed the message "Hello World". However, it would not be useful if all **gdb** can do is run a program.

6.3.2 Command: *break/b*

This command sets a breakpoint at a location in the high-level source code. When **gdb** runs to a specific location marked by a breakpoint, it stops executing for a programmer to inspect the current state of a program.

Example 6.3.2. A breakpoint can be set on a line as displayed by an editor. Suppose we want to set a breakpoint at line 3 of the program, which is the start of **main** function:

hello.c

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      printf("Hello World!\n");
6      return 0;
7  }
```

When running a program, instead of running from start to finish, **gdb** stopped at line 3:

```
(gdb) b 3
```

Output

```
Breakpoint 1 at 0x400535: file hello.c, line 3.
```

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at hello.c:5
5       printf("Hello World!\n");
```

The breakpoint is at line 3, but `gdb` stopped line 5. The reason is that line 3 does not contain code, but a function signature; `gdb` only stops where it can execute code. The code in the function starts at line 5, the call to `printf`, so `gdb` stops there.

Example 6.3.3. Line of code is not always the reliable way to specify a breakpoint, as the source code can be changed. What if `gdb` should always stop at `main` function? In this case, a better method is to use the function name directly:

```
b main
```

Then, regardless of how the source code changes, `gdb` always stops at the `main` function.

Example 6.3.4. Sometimes, the debugging program does not contain debug info, or `gdb` is debugging assembly code. In that case, a memory address can be specified as a stop point. To get the function address, `print` command can be used:

```
(gdb) print main
```

Output

```
$3 = {int (int, char **)} 0x400526 <main>
```

Knowing the address of main, we can easily set a breakpoint with a memory address:

```
b *0x400526
```

Example 6.3.5. `gdb` can also set breakpoint in any source file. Suppose that `hello` program is composed not just one file but many files e.g. `hello1.c`, `hello2.c`, `hello3.c`... In that case, simply add the filename before either a line number:

```
b hello.c:3
```

Example 6.3.6. A function name in a specific file can also be set:

```
b hello.c:main
```

6.3.3 Command: *next/n*

This command executes the current line and stops at the next line. When the current line is a function call, steps over it.

Example 6.3.7. After setting a breakpoint at `main`, run a program and stop at the first `printf`:

```
(gdb) r
```

Output
```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at hello.c:5
5        printf("Hello World!\n");
```

Then, to proceed to the next statement, we use the **next** command:

```
(gdb) n
```

Output
```
Hello World!
6        return 0;
```

In the output, the first line shows the output produced after executing line 5; then, the next line shows where `gdb` stops currently, which is line 6.

6.3.4 Command: `step/s`

This command executes the current line and stops at the next line. When the current line is a function call, steps into it to the first next line in the called function.

Example 6.3.8. Suppose we have a new function `add`[1]:

<div align="center">hello.c</div>

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main(int argc, char *argv[])
{
    add(1, 2);
    printf("Hello World!\n");
    return 0;
}
```

[1] Why should we add a new function and function call instead of using the existing `printf` call? Stepping into shared library functions is tricky because to make debugging works, the debug info must be installed and loaded. It is not worth the trouble for demonstrating this simple command.

If `step` command is used instead of `next` on the function call `printf`, gdb steps inside the function:

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0xffffd154) at hello.c:11
11      add(1, 2);
```

```
(gdb) s
```

```
add (a=1, b=2) at hello.c:6
6       return a + b;
```

After executing the command **s**, **gdb** stepped into the **add** function where
the first statement is a **return**.

6.3.5 Command: *ni*

At the core, **gdb** operates on assembly instruction. Source line by line
debugging is simply an enhancement to make it friendlier for program-
mers. Each statement in C translates to one or more assembly instruc-
tion, as shown with **objdump** and **disassemble** command. With the de-
bug info available, **gdb** knows how many instructions belong to one line
of high-level code; line by line debugging is just a execution of assembly
instructions of a line when moving from the current line to the next.

This command executes the *one* assembly instruction belongs to the
current line. Until all assembly instructions of the current line are exe-
cuted, **gdb** will not move to the next line. If the current instruction is a
call, step over it to the next instruction.

Example 6.3.9. When breakpoint is on the **printf** call and **ni** is used,
it steps through each assembly instruction:

```
(gdb) disassemble /s main
```

```
Dump of assembler code for function main:
hello.c:
4 {
    0x0804840b <+0>:   lea     ecx,[esp+0x4]
    0x0804840f <+4>:   and     esp,0xfffffff0
    0x08048412 <+7>:   push    DWORD PTR [ecx-0x4]
    0x08048415 <+10>:  push    ebp
```

```
       0x08048416 <+11>: mov     ebp,esp
       0x08048418 <+13>: push    ecx
       0x08048419 <+14>: sub     esp,0x4
5      printf("Hello World!\n");
       0x0804841c <+17>: sub     esp,0xc
       0x0804841f <+20>: push    0x80484c0
       0x08048424 <+25>: call    0x80482e0 <puts@plt>
       0x08048429 <+30>: add     esp,0x10
6      return 0;
=> 0x0804842c <+33>: mov     eax,0x0
7 }
       0x08048431 <+38>: mov     ecx,DWORD PTR [ebp-0x4]
       0x08048434 <+41>: leave
       0x08048435 <+42>: lea     esp,[ecx-0x4]
       0x08048438 <+45>: ret
   End of assembler dump.
```

```
(gdb) r
```

Output

```
   Starting program: /tmp/hello
   Breakpoint 1, main (argc=1, argv=0xffffd154) at hello.c:5
5      printf("Hello World!\n");
```

```
(gdb) ni
```

Output

```
   0x0804841f 5     printf("Hello World!\n");
```

```
(gdb) ni
```

Output

```
   0x08048424 5     printf("Hello World!\n");
```

```
(gdb) ni
```

Output

```
Hello World!
0x08048429 5      printf("Hello World!\n");
```

```
(gdb)
```

Output

```
6      return 0;
```

Upon entering `ni`, `gdb` executes current instruction and display the
next instruction. That's why from the output, `gdb` only displays 3 ad-
dresses: 0x0804841f, 0x08048424 and 0x08048429. The instruction at
0x0804841c, which is the first instruction of `printf`, is not displayed be-
cause it is the first instruction that `gdb` stopped at. Assume that `gdb` stopped
at the first instruction of `printf` at 0x0804841c, the current instruction
can be displayed using `x` command:

```
(gdb) x/i $eip
```

Output

```
=> 0x804841c <main+17>: sub     esp,0xc
```

6.3.6 Command: *si*

Similar to `ni`, this command executes the current assembly instruction
belongs to the current line. But if the current instruction is a `call`, step
into it to the first next instruction in the called function.

Example 6.3.10. Recall that the assembly code generated from `printf`
contains a `call` instruction:

```
(gdb) disassemble /s main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
    0x0804840b <+0>: lea     ecx,[esp+0x4]
    0x0804840f <+4>: and     esp,0xfffffff0
    0x08048412 <+7>: push    DWORD PTR [ecx-0x4]
    0x08048415 <+10>: push   ebp
    0x08048416 <+11>: mov    ebp,esp
    0x08048418 <+13>: push   ecx
    0x08048419 <+14>: sub    esp,0x4
5    printf("Hello World!\n");
    0x0804841c <+17>: sub    esp,0xc
    0x0804841f <+20>: push   0x80484c0
    0x08048424 <+25>: call   0x80482e0 <puts@plt>
    0x08048429 <+30>: add    esp,0x10
6    return 0;
=> 0x0804842c <+33>: mov    eax,0x0
7 }
    0x08048431 <+38>: mov    ecx,DWORD PTR [ebp-0x4]
    0x08048434 <+41>: leave
    0x08048435 <+42>: lea    esp,[ecx-0x4]
    0x08048438 <+45>: ret
End of assembler dump.
```

We try instruction by instruction stepping again, but this time by running `si` at 0x08048424, where `call` resides:

```
(gdb) si
```

Output

```
0x0804841f 5          printf("Hello World!\n");
```

```
(gdb) si
```

Output

```
    0x08048424 5           printf("Hello World!\n");
```

```
  (gdb) x/i $eip
```

Output

```
    => 0x8048424 <main+25>: call    0x80482e0 <puts@plt>
```

```
  (gdb) si
```

Output

```
    0x080482e0 in puts@plt ()
```

The next instruction right after 0x8048424 is the first instruction at
0x080482e0 in puts function. In other words, gdb stepped into puts in-
stead of stepping over it.

6.3.7 Command: until

This command executes until the next line is greater than the current
line.

Example 6.3.11. Suppose we have a function that execute a long loop:

hello.c

```c
#include <stdio.h>

int add1000() {
    int total = 0;

    for (int i = 0; i < 1000; ++i){
        total += i;
    }

    printf("Done adding!\n");
```

```
    return total;
}


int main(int argc, char *argv[])
{
    add1000(1, 2);
    printf("Hello World!\n");
    return 0;
}
```

Using **next** command, we need to press 1000 times for finishing the loop. Instead, a faster way is to use **until**:

```
(gdb) b add1000
```

Output
```
    Breakpoint 1 at 0x8048411: file hello.c, line 4.
```

```
(gdb) r
```

Output
```
    Starting program: /tmp/hello
    Breakpoint 1, add1000 () at hello.c:4
    4      int total = 0;
```

```
(gdb) until
```

Output
```
    5      for (int i = 0; i < 1000; ++i){
```

```
(gdb) until
```

Output
```
    6          total += i;
```

```
(gdb) until
```

Output

```
5       for (int i = 0; i < 1000; ++i){
```

```
(gdb) until
```

Output

```
8       printf("Done adding!\n");
```

Executing the first **until**, **gdb** stopped at line 5 since line 5 is greater than line 4.

Executing the second **until**, **gdb** stopped at line 6 since line 6 is greater than line 5.

Executing the third **until**, **gdb** stopped at line 5 since the loop still continues. Because line 5 is less than line 6, with the fourth **until**, **gdb** kept executing until it does not go back to line 5 anymore and stopped at line 8. This is a great way to skip over loop in the middle, instead of setting unneeded breakpoint.

Example 6.3.12. until can be supplied with an argument to explicitly execute to a specific line:

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, add1000 () at hello.c:4
4     int total = 0;
```

```
(gdb) until 8
```

Output

```
add1000 () at hello.c:8
8       printf("Done adding!\n");
```

6.3.8 Command: `finish`

This command executes until the end of a function and displays the return value. `finish` is actually just a more convenient version of `until`.

Example 6.3.13. Using the `add1000` function from the previous example and use `finish` instead of `until`:

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, add1000 () at hello.c:4
4       int total = 0;
```

```
(gdb) finish
```

Output

```
Run till exit from #0  add1000 () at hello.c:4
Done adding!
0x08048466 in main (argc=1, argv=0xffffd154) at hello.c:15
15        add1000(1, 2);
Value returned is $1 = 499500
```

6.3.9 Command: `bt`

This command prints the *backtrace* of all stack frames. A *backtrace* is a list of currently active functions:

backtrace

Example 6.3.14. Suppose we have a chain of function calls:

hello.c

```
void d(int d) { };
void c(int c) { d(0); }
void b(int b) { c(1); }
void a(int a) { b(2); }


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

```
{
    a(3);
    return 0;
}
```

bt can visualize such a chain in action:

```
(gdb) b a
```

Output

```
Breakpoint 1 at 0x8048404: file hello.c, line 9.
```

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, a (a=3) at hello.c:9
9 void a(int a) { b(2); }
```

```
(gdb) s
```

Output

```
b (b=2) at hello.c:7
7 void b(int b) { c(1); }
```

```
(gdb) s
```

Output

```
c (c=1) at hello.c:5
5 void c(int c) { d(0); }
```

```
(gdb) s
```

Output
```
d (d=0) at hello.c:3
3 void d(int d) { };
```

```
(gdb) bt
```

Output
```
#0  d (d=0) at hello.c:3
#1  0x080483eb in c (c=1) at hello.c:5
#2  0x080483fb in b (b=2) at hello.c:7
#3  0x0804840b in a (a=3) at hello.c:9
#4  0x0804841b in main (argc=1, argv=0xffffd154) at hello.c:13
```

Most-recent calls are placed on top and least-recent calls are near the bottom. In this case, d is the most current active function, so it has the index 0. Next is c, the 2nd active function, has the index 1 and so on with function b, function a, and finally function main at the bottom, the least-recent function. That is how we read a backtrace.

6.3.10 Command: *up*

This command goes up one frame earlier the current frame.

Example 6.3.15. Instead of staying in d function, we can go up to c function and look at its state:

```
(gdb) bt
```

Output
```
#0  d (d=0) at hello.c:3
#1  0x080483eb in c (c=1) at hello.c:5
#2  0x080483fb in b (b=2) at hello.c:7
#3  0x0804840b in a (a=3) at hello.c:9
#4  0x0804841b in main (argc=1, argv=0xffffd154) at hello.c:13
```

```
(gdb) up
```

```
#1  0x080483eb in c (c=1) at hello.c:3
3 void b(int b) { c(1); }
```

The output displays the current frame is moved to c and where the
call to c is made, which is in function b at line 3.

6.3.11 Command: *down*

Similar to up, this command goes down one frame later then the current
frame.

Example 6.3.16. After inspecting c function, we can go back to d:

```
(gdb) bt
```

```
#0  d (d=0) at hello.c:3
#1  0x080483eb in c (c=1) at hello.c:5
#2  0x080483fb in b (b=2) at hello.c:7
#3  0x0804840b in a (a=3) at hello.c:9
#4  0x0804841b in main (argc=1, argv=0xffffd154) at hello.c:13
```

```
(gdb) up
```

```
#1  0x080483eb in c (c=1) at hello.c:3
3 void b(int b) { c(1); }
```

```
(gdb) down
```

```
#0  d (d=0) at hello.c:1
1 void d(int d) { };
```

6.3.12 Command: `info registers`

This command lists the current values in commonly used registers. This command is useful when debugging assembly and operating system code, as we can inspect the current state of the machine.

Example 6.3.17. Executing the command, we can see the commonly used registers:

```
(gdb) info registers
```

Output

```
eax             0xf7faddbc -134554180
ecx             0xffffd0c0 -12096
edx             0xffffd0e4 -12060
ebx             0x0 0
esp             0xffffd0a0 0xffffd0a0
ebp             0xffffd0a8 0xffffd0a8
esi             0xf7fac000 -134561792
edi             0xf7fac000 -134561792
eip             0x804841c 0x804841c <main+17>
eflags          0x286 [ PF SF IF ]
cs              0x23 35
ss              0x2b 43
ds              0x2b 43
es              0x2b 43
fs              0x0 0
gs              0x63 99
```

The above registers suffice for writing our operating system in later part.

6.4   How debuggers work: A brief introduction

6.4.1 How breakpoints work

When a programmer places a breakpoint somewhere in his code, what actually happens is that the *first* opcode of the *first* instruction of a state-

ment is replaced with another instruction, `int 3` with opcode `CCh`:

83	ec	0c
sub esp,0x4		

→
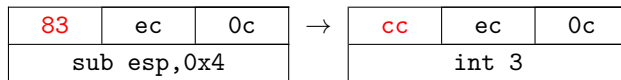
cc	ec	0c
int 3		

`int 3` only costs a single byte, making it efficient for debugging. When `int 3` instruction is executed, the operating system calls its breakpoint interrupt handler. The handler then checks what process reaches a breakpoint, pauses it and notifies the debugger it has paused a debugged process. The debugged process is only paused and that means a debugger is free to inspect its internal state, like a surgeon operates on an anesthetic patient. Then, the debugger replaces the `int 3` opcode with the original opcode and executes the original instruction normally.

cc	ec	0c
int 3		

→

83	ec	0c
sub esp,0x4		
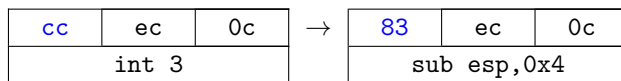
Figure 6.4.2: Restore the original opcode, after `int 3` was executed

Example 6.4.1. It is simple to see `int 3` in action. First, we add an `int 3` instruction where we need `gdb` to stop:

hello.c

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
    asm("int 3");
    printf("Hello World\n");
    return 0;
}
```

`int 3` precedes `printf`, so `gdb` is expected to stop at `printf`. Next, we compile with debug enable and with Intel syntax:

```
$ gcc -masm=intel -m32 -g hello.c -o hello
```

Finally, start `gdb`:

```
$ gdb hello
```

Running without setting any breakpoint, `gdb` stops at `printf` call, as expected:

```
(gdb) r
```

Output
```
Starting program: /tmp/hello
Program received signal SIGTRAP, Trace/breakpoint trap.
main (argc=1, argv=0xffffd154) at hello.c:6
6       printf("Hello World\n");
```

The blue text indicates that `gdb` encountered a breakpoint, and indeed it stopped at the right place: the `printf` call, where `int 3` preceded it.

6.4.2 Single stepping

When breakpoint is implemented, it is easy to implement single stepping: a debugger simply places another `int 3` opcode in the next instruction. So, when a programmer sets a breakpoint at an instruction, the next instruction is automatically set by the debugger, thus enable instruction by instruction debugging. Similarly, source line by line debugging is just the placements of the very first opcodes in the two statements with two `int 3` opcodes.

6.4.3 How a debugger understands high level source code

DWARF is a debugging file format used by many compilers and debuggers to support source level debugging. DWARF contains information that maps between entities in the executable binary with the source files. A program entity can either be data or code. A **DIE**, or *Debugging Information* **Debugging Information** *Entry*, is a description of a program entity. A DIE consists of a tag, which spe-**Entry** ifies the entity that the DIE describes, and a list of attributes that describes the entity. Of all the attributes, these two attributes enables source-level debugging:

▷ WHERE THE ENTITY APPEARS IN THE SOURCE FILES: which file and which line the entity appears.

▷ WHERE THE ENTITY APPEARS IN THE EXECUTABLE BINARY:
in which memory address the entity is loaded at runtime. With the
precise address, `gdb` can retrieve correct value for a data entity, or place
a correct breakpoint and stop accordingly for a code entity. Without
the information of these addresses, `gdb` would not know where the en-
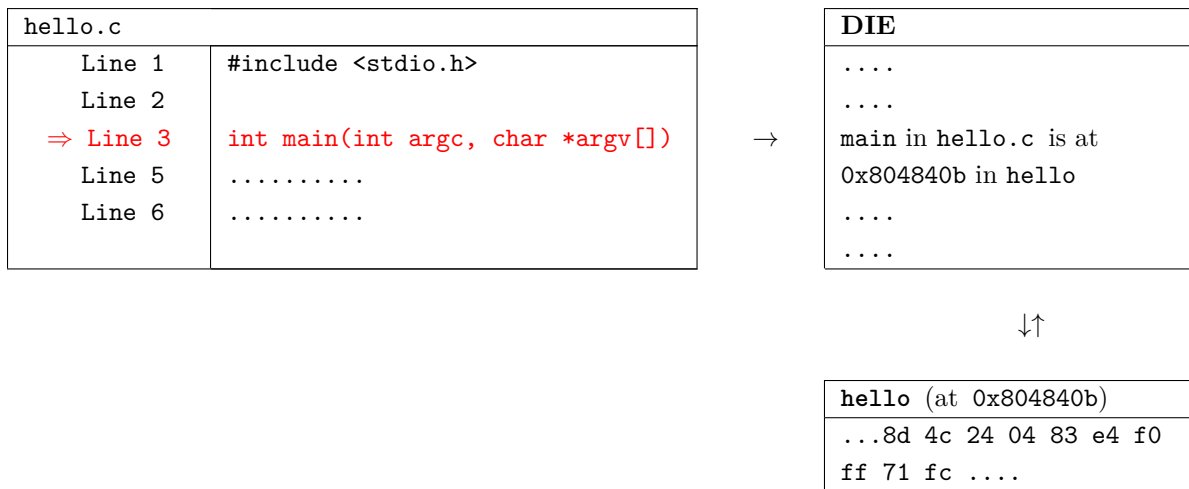tities are to inspect them.

hello.c	
Line 1	`#include <stdio.h>`
Line 2	
⇒ Line 3	`int main(int argc, char *argv[])`
Line 5	`..........`
Line 6	`..........`

→

DIE
....
....
`main` in `hello.c` is at `0x804840b` in `hello`
....
....

↓↑

`hello` (at 0x804840b)
...8d 4c 24 04 83 e4 f0
ff 71 fc ....

Figure 6.4.3: Source-binary
mapping with DIE

In addition to DIEs, another binary-to-source mapping is the *line num-
ber table*. The line number table maps between a line in the source code
and at which memory address is the start of the line in the executable
binary.

In sum, to successfully enable source-level debugging, a debugger needs
to know the precise location of the source files and the load addresses
at runtime. Address matching, between the image layout of the ELF bi-
nary and the address where it is loaded, is extremely important since de-
bug information relies on correct loading address at runtime. That is, it
assumes the addresses as recorded in the binary image at compile-time
the same as at runtime e.g. if the load address for `.text` section is recorded
in the executable binary at `0x800000`, then when the binary actually runs,
`.text` should really be loaded at `0x800000` for gdb to be able to correctly
match running instructions with high-level code statement. Address mis-
matching makes debug information useless, as actual code at one address
is displayed as code at another address. Without this knowledge, we will

not be able to build an operating system that can be debugged with `gdb`.

Example 6.4.2. When an executable binary contains debug info, `readelf` can display such information in a readable format. Using the good old hello world program:

hello.c

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
    printf("Hello World\n");


    return 0;
}
```

and compile with debug info:

```
$ gcc -m32 -g hello.c -o hello
```

With the binary ready, we can look at the line number table with the command:

```
$ readlelf -wL hello
```

`-w` option prints all the debug information. In combination with its sub-option, only specific information is displayed. For example, with `-L,` only the line number table is displayed:

Output

```
Decoded dump of debug contents of section .debug_line:
CU: hello.c:
File name                           Line number     Starting address
hello.c                                       6            0x804840b
hello.c                                       7            0x804841c
hello.c                                       9            0x804842c
```

```
    hello.c                              10              0x8048431
```

From the above output:

CU shorts for ***C****ompilation* ***U****nit*, a separately compiled source file. In
the example, we only have one file, `hello.c`.

File name displays the filename of the current compilation unit.

Line number is the line number in the source file of which the line is not
an empty line. In the example, line 8 is an empty line, so it does not
appear.

Starting address is the memory address where the line actually starts
in the executable binary.

With such crystal clear information, this is how `gdb` is able to set a break-
point on a line easily. For placing breakpoints on variables and functions,
it is time to look at the DIEs. To get the DIEs information from an ex-
ecutable binary, run the command:

```
$ readlelf -wi hello
```

`-wi` option lists all the DIE entries. This is one typical DIE entry:

```
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
   <c>   DW_AT_producer    : (indirect string, offset: 0xe): GNU C11 5.4.0 20160609 -masm=intel -m32
   <10>   DW_AT_language    : 12 (ANSI C99)
   <11>   DW_AT_name        : (indirect string, offset: 0xbe): hello.c
   <15>   DW_AT_comp_dir    : (indirect string, offset: 0x97): /tmp
   <19>   DW_AT_low_pc      : 0x804840b
   <1d>   DW_AT_high_pc     : 0x2e
   <21>   DW_AT_stmt_list   : 0x0
```

Red This left-most number indicates the current nesting level of a DIE
entry. `0` is the outer-most level DIE with its entity is the compilation
unit. This means subsequent DIE entries with higher nesting level are
all the children of this tag, the compilation unit. It makes sense, as
all the entities must originate from a source file.

Blue These numbers in hex format indicate the offsets into `.debug_info` section. Each meaningful information is displayed along with its offset. When an attribute references to another attribute, the offset is used to precisely identify the referenced attribute.

Green These names with `DW_AT_` prefix are the attributes attached to a DIE that describe an entity. Notable attributes:

`DW_AT_name`

`DW_AT_comp_dir` The filename of the compilation unit and the directory where compilation occurred. Without the filename and the path, **gdb** would not be able to display the high-level source, despite the availability of the debug info. Debug info only contains the mapping between source and binary, not the source code itself.

`DW_AT_low_pc`

`DW_AT_high_pc` The start and end of the current entity, which is the compilation unit, in the executable binary. The value in `DW_AT_low_pc` is the starting address. `DW_AT_high_pc` is the size of the compilation unit, when adding up to `DW_AT_low_pc` results in the end address of the entity. In this example, code compiled from **hello.c** starts at $0x804840b$ and end at $0x804840b + 0x2e = 0x8048439$. To really make sure, we verify with **objdump**:

Output

```
int main(int argc, char *argv[])
{
  804840b:        8d 4c 24 04              lea    ecx,[esp+0x4]
  804840f:        83 e4 f0                 and    esp,0xfffffff0
  8048412:        ff 71 fc                 push   DWORD PTR [ecx-0x4]
  8048415:        55                       push   ebp
  8048416:        89 e5                    mov    ebp,esp
  8048418:        51                       push   ecx
  8048419:        83 ec 04                 sub    esp,0x4
      printf("Hello World\n");
  804841c:        83 ec 0c                 sub    esp,0xc
  804841f:        68 c0 84 04 08           push   0x80484c0
```

```
8048424:        e8 b7 fe ff ff          call   80482e0 <puts@plt>

8048429:        83 c4 10                add    esp,0x10

    return 0;

804842c:        b8 00 00 00 00          mov    eax,0x0

}

8048431:        8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]

8048434:        c9                      leave

8048435:        8d 61 fc                lea    esp,[ecx-0x4]

8048438:        c3                      ret

8048439:        66 90                   xchg   ax,ax

804843b:        66 90                   xchg   ax,ax

804843d:        66 90                   xchg   ax,ax

804843f:        90                      nop
```

It is true: `main` starts at 804840b and end at 8048439, right after
the `ret` instruction at 8048438. The instructions after 8048439 are
just padding bytes inserted by `gcc` for alignment, which do not be-
long to `main`. Note that the output from `objdump` shows much more
code past `main`. It is not counted, as the code is outside of `hello.c`,
added by `gcc` for the operating system. `hello.c` contains only one
function: `main` and this is why `hello.c` also starts and ends the
same as `main`.

Pink This number displays the abbreviation form of a tag. An abbre-
viation is the form of a DIE. When debug info is displayed with `-wi`,
the DIEs are displayed with their values. `-wa` option shows abbrevi-
ations in the `.debug_abbrev` section:

Output

```
Contents of the .debug_abbrev section:

  Number TAG (0x0)

   1      DW_TAG_compile_unit    [has children]

    DW_AT_producer      DW_FORM_strp

    DW_AT_language      DW_FORM_data1

    DW_AT_name          DW_FORM_strp

    DW_AT_comp_dir      DW_FORM_strp
```

```
      DW_AT_low_pc       DW_FORM_addr

      DW_AT_high_pc      DW_FORM_data4

      DW_AT_stmt_list    DW_FORM_sec_offset

      DW_AT value: 0     DW_FORM value: 0
.... more abbreviations ....
```

The output is similar to a DIE output, with only attribute names and without any value. We can also say an abbreviation is a *type* of a DIE, as an abbreviation represents the structure of a particular DIE. Many DIEs share the same abbreviation, or structure, thus they are of the same type. An abbreviation number specifies which type a DIE is in the abbreviation table above. Abbreviations improve encoding efficiency (reduce binary size) because each DIE needs not to carry their structure information as pairs of attribute-value[2], but simply refers to an abbreviation for correct decoding.

[2] For example data format such as YAML or JSON encodes its attribute names along with its values. This simplifies encoding, but with overhead.

Here are all the DIEs of hello represented as a tree:

In the figure 6.4.4, `DW_TAG_subprogram` represents a function such as `main`. Its children are the DIEs of `argc` and `argv`. With such precise information, matching source to binary is an easy job for `gdb`.

If more than one compilation units exist in an executable binary, the DIE entries are sorted according to the compilation order from `gcc`. For example, suppose we have another `test.c` source file[3] and compile it together with hello:

[3] It can contain anything. Just a sample file.

```
$ gcc -masm=intel -m32 -g test.c hello.c -o hello
```

Then, the all DIE entries in `test.c` are displayed before the DIE entries in `hello.c`:

```
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)


  <c>   DW_AT_producer   : (indirect string, offset: 0x0): GNU C11 5.4.0 20160609
 -masm=intel -m32 -mtune=generic -march=i686 -g -fstack-protector-strong
     <10>   DW_AT_language   : 12      (ANSI C99)
```
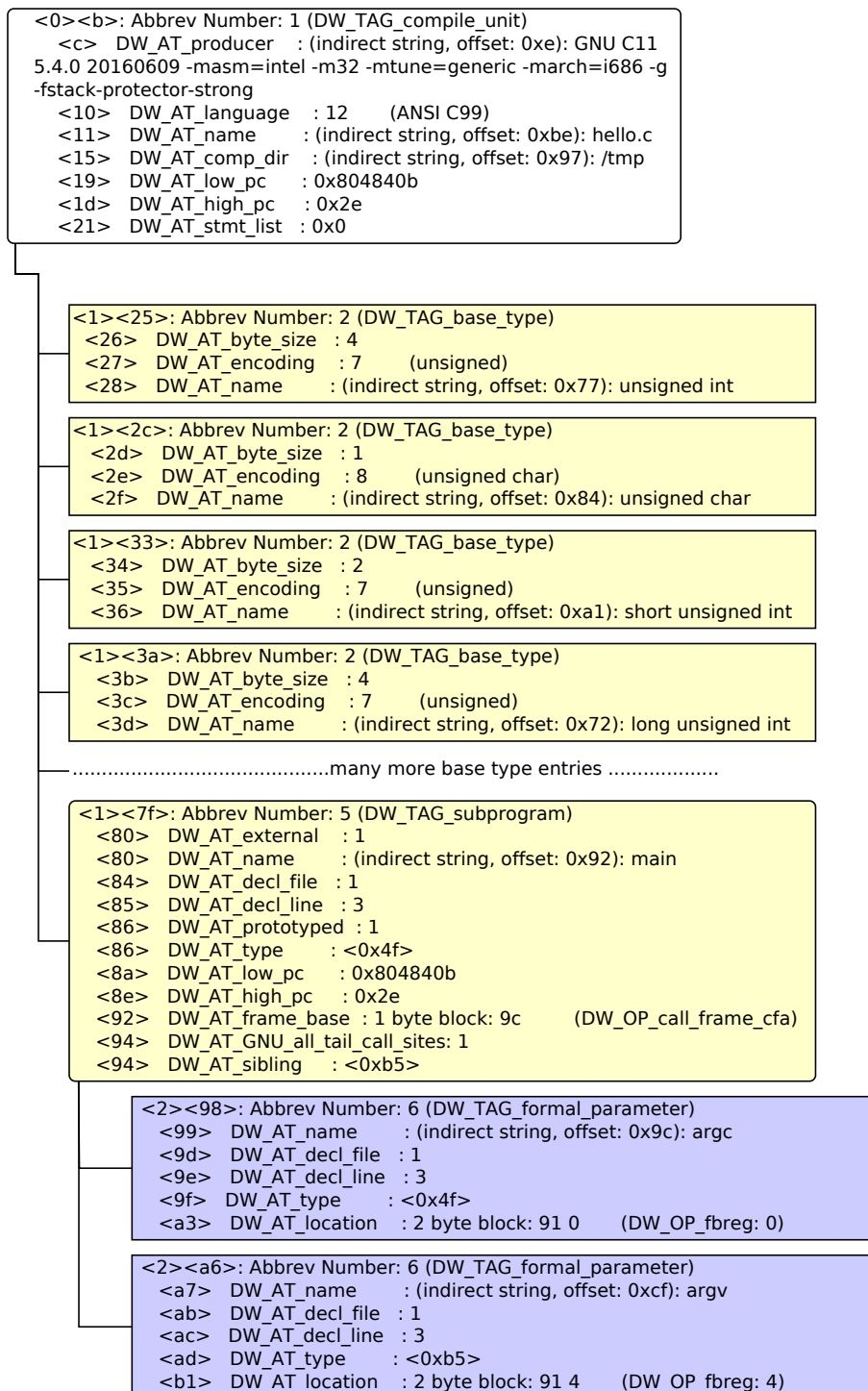
```
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
  <c>  DW_AT_producer   : (indirect string, offset: 0xe): GNU C11
5.4.0 20160609 -masm=intel -m32 -mtune=generic -march=i686 -g
-fstack-protector-strong
  <10>  DW_AT_language  : 12      (ANSI C99)
  <11>  DW_AT_name      : (indirect string, offset: 0xbe): hello.c
  <15>  DW_AT_comp_dir  : (indirect string, offset: 0x97): /tmp
  <19>  DW_AT_low_pc    : 0x804840b
  <1d>  DW_AT_high_pc   : 0x2e
  <21>  DW_AT_stmt_list : 0x0
```

```
<1><25>: Abbrev Number: 2 (DW_TAG_base_type)
  <26>  DW_AT_byte_size  : 4
  <27>  DW_AT_encoding   : 7       (unsigned)
  <28>  DW_AT_name       : (indirect string, offset: 0x77): unsigned int
```

```
<1><2c>: Abbrev Number: 2 (DW_TAG_base_type)
  <2d>  DW_AT_byte_size  : 1
  <2e>  DW_AT_encoding   : 8       (unsigned char)
  <2f>  DW_AT_name       : (indirect string, offset: 0x84): unsigned char
```

```
<1><33>: Abbrev Number: 2 (DW_TAG_base_type)
  <34>  DW_AT_byte_size  : 2
  <35>  DW_AT_encoding   : 7       (unsigned)
  <36>  DW_AT_name       : (indirect string, offset: 0xa1): short unsigned int
```

```
<1><3a>: Abbrev Number: 2 (DW_TAG_base_type)
  <3b>  DW_AT_byte_size  : 4
  <3c>  DW_AT_encoding   : 7       (unsigned)
  <3d>  DW_AT_name       : (indirect string, offset: 0x72): long unsigned int
```

....many more base type entries ....

```
<1><7f>: Abbrev Number: 5 (DW_TAG_subprogram)
  <80>  DW_AT_external   : 1
  <80>  DW_AT_name       : (indirect string, offset: 0x92): main
  <84>  DW_AT_decl_file  : 1
  <85>  DW_AT_decl_line  : 3
  <86>  DW_AT_prototyped : 1
  <86>  DW_AT_type       : <0x4f>
  <8a>  DW_AT_low_pc     : 0x804840b
  <8e>  DW_AT_high_pc    : 0x2e
  <92>  DW_AT_frame_base : 1 byte block: 9c      (DW_OP_call_frame_cfa)
  <94>  DW_AT_GNU_all_tail_call_sites: 1
  <94>  DW_AT_sibling    : <0xb5>
```

```
<2><98>: Abbrev Number: 6 (DW_TAG_formal_parameter)
  <99>  DW_AT_name       : (indirect string, offset: 0x9c): argc
  <9d>  DW_AT_decl_file  : 1
  <9e>  DW_AT_decl_line  : 3
  <9f>  DW_AT_type       : <0x4f>
  <a3>  DW_AT_location   : 2 byte block: 91 0     (DW_OP_fbreg: 0)
```

```
<2><a6>: Abbrev Number: 6 (DW_TAG_formal_parameter)
  <a7>  DW_AT_name       : (indirect string, offset: 0xcf): argv
  <ab>  DW_AT_decl_file  : 1
  <ac>  DW_AT_decl_line  : 3
  <ad>  DW_AT_type       : <0xb5>
  <b1>  DW_AT_location   : 2 byte block: 91 4     (DW_OP_fbreg: 4)
```

Figure 6.4.4: DIE entries visualized as a tree

```
    <11>    DW_AT_name          : (indirect string, offset: 0x64): test.c
    <15>    DW_AT_comp_dir      : (indirect string, offset: 0x5f): /tmp
    <19>    DW_AT_low_pc        : 0x804840b
    <1d>    DW_AT_high_pc       : 0x6
    <21>    DW_AT_stmt_list     : 0x0
 <1><25>: Abbrev Number: 2 (DW_TAG_subprogram)
    <26>    DW_AT_external      : 1
    <26>    DW_AT_name          : bar
    <2a>    DW_AT_decl_file     : 1
    <2b>    DW_AT_decl_line     : 1
    <2c>    DW_AT_low_pc        : 0x804840b
    <30>    DW_AT_high_pc       : 0x6
    <34>    DW_AT_frame_base  : 1 byte block: 9c         (DW_OP_call_frame_cfa)
    <36>    DW_AT_GNU_all_call_sites: 1
....after all DIEs in test.c listed....
<0><42>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <43>    DW_AT_producer      : (indirect string, offset: 0x0): GNU C11 5.4.0 20160609
-masm=intel -m32 -mtune=generic -march=i686 -g -fstack-protector-strong
    <47>    DW_AT_language      : 12        (ANSI C99)
    <48>    DW_AT_name          : (indirect string, offset: 0xc5): hello.c
    <4c>    DW_AT_comp_dir      : (indirect string, offset: 0x5f): /tmp
    <50>    DW_AT_low_pc        : 0x8048411
    <54>    DW_AT_high_pc       : 0x2e
    <58>    DW_AT_stmt_list     : 0x35
....then all DIEs in hello.c are listed....
```

Part II

Groundwork

7

Bootloader

A *bootloader* loads an OS, or an application [1] that runs and communicate directly with hardware. To run an OS, the first thing to write is a bootloader. In this chapter, we are going to write a rudimentary bootloader, as our main focus is writing an operating system, not a bootloader. More interestingly, this chapter will present related tools and techniques that are applicable for writing a bootloader as well as an operating system.

[1] Many embedded devices don't use an OS. In embedded systems, the bootloader is simply included in boot firmware and no bootloader is needed.

7.1 x86 Boot Process

After the POST process finished, the CPU's program counter is set to the address `FFFF:0000h` for executing BIOS code. *BIOS - **B**asic **I**nput/**O**utput **S**ystem* is a firmware that performs hardware initialization and provides a set of generic subroutines to control input/output devices. The BIOS checks all available storage devices (floppy disks and hard disks) if any device is bootable, by examining the last two bytes of the first sector whether it has the boot record signature of `0x55`, `0xAA`. If so, the BIOS loads the first sector to the address `7C00h`, set the program counter to that address and let the CPU executing code from there.

The first sector is called ***M**aster **B**oot **R**ecord,* or *MBR*. The program in the first sector is called *MBR Bootloader*.

7.2   Using BIOS services

BIOS provides many basic services for controlling the hardware at the
boot stage. A service is a group of routines that controls a particular hard-
ware device, or returns information of current system. Each service is given
an interrupt number. To call a BIOS routine, an `int` instruction must
be used with an interrupt number. Each BIOS service defines its own
numbers for its routines; to call a routine, a specific number must be writ-
ten to a register required by each service. The list of all BIOS interrupts
is available with Ralf Brown's Interrupt List at: http://www.cs.cmu.edu/
~ralf/files.html.
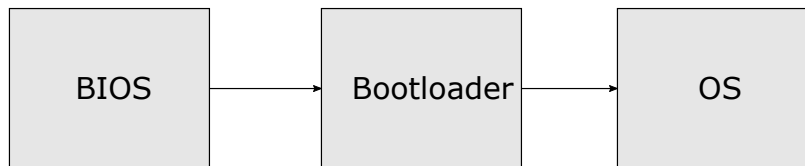
Figure 7.2.1: The boot process.

Example: Interrupt call `13h` (diskette service) requires number of sectors
to read, track number, sector number, head number and drive number
to read from a storage device. The content of the sector is stored in
memory at the address defined by the pair of registers `ES:BX`. The pa-
rameters are stored in registers like this:

```
1   ; Store sector content in the buffer 10FF:0000
2   mov dx, 10FFh
3   mov es, dx
4   xor bx, bx
5   mov al, 2 ; read 2 sector
6   mov ch, 0 ; read track 0
7   mov cl, 2 ; 2nd sector is read
8   mov dh, 0 ; head number
9   mov dl, 0 ; drive number. Drive 0 is floppy drive.
10  mov ah, 0x02 ; read floppy sector function
11  int 0x13 ; call BIOS - Read the sector
```

The BIOS is only available in real mode. However, when switching to
protected mode, then BIOS will not be usable anymore and the operat-

ing system code is responsible for controlling hardware devices. This is when the operating system stands on its own: it must provide its own kernel drivers for talking to hardware.

7.3 Boot process

1. BIOS transfers control to MBR bootloader by jumping to `0000:7c00h`, where bootloader is assumed to exist already.

2. Setup machine environment for booting by properly initialize segment registers to enable flat memory model.

3. Load the kernel:

   (a) Read kernel from disk.

   (b) Save it somewhere in the main memory.

   (c) Jump to the starting code address of the kernel and execute.

4. If error occurs, print a message to notify users something went wrong and halt.

7.4 Example Bootloader

Here is a simple bootloader that does nothing, except not crashing the machine but halt it gracefully. If the virtual machine does not halt but text repeatedly flashing, it means the bootloader does not load properly and the machine crashed. The machine crashed because it keeps executing until the near end of physical memory (1 MB in real mode), which is `FFFF:0000h`, which starts the whole BIOS boot process all over again. This is effectively a reset, but not fully, since machine environment from previous run is still reserved. For that reason, it is called a *warm reboot*. The opposite of warm reboot is *cold reboot*, in which the machine environment is reset to initial settings when the computer starts from a powerless state.

bootloader.asm

```
1  ;*****************************************
```

```asm
2   ; bootloader.asm
3   ; A Simple Bootloader
4   ;*****************************************
5   org 0x7c00
6   bits 16
7   start: jmp boot
8
9   ;; constant and variable definitions
10  msg db "Welcome to My Operating System!", 0ah, 0dh, 0h
11
12  boot:
13    cli ; no interrupts
14    cld ; all that we need to init
15    hlt ; halt the system
16
17  ; We have to be 512 bytes. Clear the rest of the bytes with
          0
18  times 510 - ($-$$) db 0
19  dw 0xAA55     ; Boot Signiture
```

7.5   Compile and load

We compile the code with **nasm** and write it to a disk image:

```
$ nasm -f bin bootloader.asm -o bootloader
```

Then, we create a 1.4 MB floppy disk and:

```
$ dd if=/dev/zero of=disk.img bs=512 count=2880
```

Output

```
2880+0 records in
2880+0 records out
1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.00625622 s, 236 MB/s
```

Then, we write the bootloader to the 1stsector:

```
$ dd conv=notrunc if=bootloader of=disk.img bs=512
count=1 seek=0
```

Output

```
1+0 records in
1+0 records out
512 bytes copied, 0.000102708 s, 5.0 MB/s
```

The option `conv=notrunc` preserves the original size of the floppy disk. Without this option, the 1.4 MB disk image will be completely replaced by the new `disk.img` with only 512 bytes, and we do not want that happens.

In the past, developing an operating system is complicated because a programmer needs to understand specific hardware he is using. Even though x86 was ubiquitous, the minute differences between models made some code written for a machine not run on another. Further, if you use the same physical computer you write your operating system take very long between runs, and also difficult to debug. Fortunately, today we can uniformly produce a virtual machine with a particular specification and avoid the incompatibility issue altogether, thus making an OS easier to write and test since everyone can reproduce the same machine environment.

We will be using *QEMU*, a generic and open source machine emulator and virtualizer. QEMU can emulate various types of machine, not limited to x86_64 only. Debug is easy since you can connect GDB to a virtual machine to debug code that runs on it, through QEMU's built-in GDB server. QEMU can use `disk.img` as a boot device e.g. a floppy disk:

```
$ qemu-system-i386 -machine q35 -fda disk.img -gdb
tcp::26000 -S
```

▷ With option `-machine q35`, QEMU emulates a `q35` machine model from Intel.[2].

[2] The following command lists all supported emulated machines from QEMU:

```
qemu-system-i386 -machine help
```

▷ With option `-fda disk.img`, QEMU uses `disk.img` as a floppy disk image.

▷ With option `-gdb tcp::26000`, QEMU allows `gdb` to connect to the virtual machine for remote debugging through a tcp socket with port 26000.

▷ With option `-S`, QEMU waits for `gdb` to connect before it starts running.

After the command is executed, a new console window that displays the screen output of the virtual machine. Open another terminal, run `gdb` and set the current architecture to `i8086`, since we are running in 16-bit mode:

```
(gdb) set architecture i8086
```

Output
```
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.
The target architecture is assumed to be i8086
```

Then, connect `gdb` to the waiting virtual machine with this command:

```
(gdb) target remote localhost:26000
```

Output
```
Remote debugging using localhost:26000
0x0000fff0 in ?? ()
```

Then, place a breakpoint at `0x7c00`:

```
(gdb) b *0x7c00
```

Output
```
Breakpoint 1 at 0x7c00
```

Note the asterick before the memory address. Without the asterisk, `gdb` treats the address as a symbol in a program rather than an address.

Then, for convenience, we use a split layout for viewing the assembly code
and registers together:

```
(gdb) layout asm
(gdb) layout reg
```

Finally, run the program:

```
(gdb) c
```

If the virtual machine successfully runs the bootloader, this is what
the QEMU screen should look like:

Figure 7.5.1: Boot succeeded.

7.5.1 Debugging

If, for some reason, the sample bootloader cannot get to such screen and
gdb does not stop at 0x7c00, then the following scenarios are likely:

▷ THE BOOTLOADER IS INVALID: the message "Boot failed: not a bootable
   disk" appears for floppy disk booting. Make sure the boot signature
   is at the last 2 bytes of the 512-byte first sector.

▷ THE MACHINE CANNOT FIND A BOOT DISK: the message "Boot failed: not a bootable disk" appears for floppy disk booting. Make sure the bootloader is correctly written to the first sector. It can be verify by check the disk with `hd`:

```
$ hd disk.img | less
```

If the first 512 bytes are all zeroes, then it is likely that the bootloader is incorrectly written to another sector.

▷ THE MACHINE CRASHES: When such scenario happens, it reset back to the beginning at `FFFF:0000h`. If the QEMU machine starts without waiting for `gdb`, then the console output window keeps flashing as the machine is repeatedly reset. It is likely some instruction in the bootloader code causing the fault.

Exercise 7.5.1. Print a welcome message

We loaded the bootloader successfully. But, it needs to do something useful other than halting our machine. The easiest thing to do is printing something on screen, like how an introduction to all programming language starts with "Hello World". Our bootloader prints "Welcome to my operating system"[3]. In this part, we will build a simple I/O library that allows us to set a cursor anywhere on the screen and print text there.

[3] Or whatever message you want.

First, create a file **io.asm** for I/O related routines. Then, write the following routines:

1. `MovCursor`

   Purpose: Move a cursor to a specific location on screen and remember this location.

   Parameters:

   ▷ bh = Y coordinate

   ▷ bl = X coordinate.

   Return: None

2. `PutChar`

   Purpose: Print a character on screen, at the cursor position previously set by `MovCursor` .

   Parameters:

   ▷ `al` = Character to print

   ▷ `bl` = text color

   ▷ `cx` = number of times the character is repeated

   Return: None

3. `Print`

   Purpose: Print a string.

   Parameters:

   ▷ `ds:si` = Zero terminated string

   Return: None

Test the routines by putting each in the bootloader source, compile and run. To debug, run GDB and set a breakpoint at a specific routine. The end result is that `Print` should display a welcome message on screen.

7.6   Loading a program from bootloader

Now that we get the feel of how to use the BIOS services, it is time for something more complicated. We will place our kernel on 2nd sector onward, and our bootloader reads 30 sectors starting from 2nd sector. Why 30 sectors? Our kernel will grow gradually, so we will preserve 30 sectors and save us time for modifying the bootloader each time the kernel size expands another sector.

The primary responsibility of a bootloader is to read an operating system from some storage device e.g. hard disk, then loads it into main memory and transfer the control to the loaded operating system, similar to

how the BIOS reads and loads a bootloader. At the moment, our bootloader does nothing more than just an assembly program loaded by the BIOS. To make our bootloader a real one, it must perform well the above two tasks: *read* and *load* an operating system.

7.6.1 Floppy Disk Anatomy

To read from a storage device, we must understand how the device works, and the provided interface for controlling it. First of all, a floppy disk is a storage device, similar to RAM, but can store information even when a computer is turned off, thus is called *persistent storage device.* A floppy disk also a persistent storage device, thus it provides a storage space up to 1.4 MB, or 1,474,560 bytes. When reading from a floppy disk, the smallest unit that can be read is a *sector*, a group of 512 contiguous bytes. A group of 18 sectors is a *track.* Each side of a floppy disk consists of 80 tracks. A floppy drive is required to read a floppy disk. Inside a floppy drive contains an arm with 2 *heads*, each head reads a side of a floppy drive; head 0 writes the upper side and head 1 writes the lower side of a floppy disk.

persistent storage device

Figure 7.6.1: Sector and Track.

Figure 7.6.2: Floppy disk platter with 2 sides.

When a floppy drive writes data to a brand new floppy disk, track 0 on the upper side is written first, by head 0. When the upper track 0 is full, the lower track 0 is used by head 1. When both the upper and lower side of a track 0 are full, it goes back to head 0 for writing data again, but this time the upper side of track 1 and so on, until no space left on the device. The same procedure is also applied for reading data from floppy disk.

7.6.2 Read and load sectors from a floppy disk

First, we need to a sample program for writing into the 2nd sector, so we can experiment with floppy disk reading:

sample.asm

```
1  ;*****************************************
2  ; sample.asm
3  ; A Sample Program
```

```
4   ;******************************************
5   mov eax, 1
6   add eax, 1
```

Such a program is good enough. To simplify and for the purpose of demonstration, we will use the same floppy disk that holds the bootloader to hold our operating system. The operating system image starts from the 2nd sector, as the 1st sector is already in use by the bootloader. We compile and write it to the 2nd sector with `dd`:

```
$ nasm -f bin sample.asm -o sample
$ dd if=sample of=disk.img bs=512 count=1 seek=1
```

1st sector	2nd sector	.....	30th sector
bootloader	sample	....	(empty)

Figure 7.6.3: The bootloader and the sample program on floppy disk.

Next, we need to fix the bootloader for reading from the floppy disk and load a number of arbitrary sectors. Before doing so, a basic understanding of floppy disk is required. To read data from disk, interrupt `13` with `AH = 02` is a routine for reading sectors from disk into memory:

```
AH = 02
AL = number of sectors to read (1-128 dec.)
CH = track/cylinder number (0-1023 dec., see below)
CL = sector number (1-17 dec.)
DH = head number (0-15 dec.)
DL = drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)
ES:BX = pointer to buffer
Return:
    AH = status (see INT 13,STATUS)
    AL = number of sectors read
    CF = 0 if successful
       = 1 if error
```

Apply the above routine, the bootloader can read the 2nd sector:

bootloader.asm

```asm
1   ;*******************************************
2   ; Bootloader.asm
3   ; A Simple Bootloader
4   ;*******************************************
5   org 0x7c00
6   bits 16
7   start: jmp boot
8
9   ;; constant and variable definitions
10    msg db "Welcome to My Operating System!", 0ah, 0dh, 0h
11
12  boot:
13    cli ; no interrupts
14    cld ; all that we need to init
15
16    mov ax, 0x50
17
18    ;; set the buffer
19    mov es, ax
20    xor bx, bx
21
22    mov al, 2     ; read 2 sector
23    mov ch, 0     ; track 0
24    mov cl, 2     ; sector to read (The second sector)
25    mov dh, 0     ; head number
26    mov dl, 0     ; drive number
27
28    mov ah, 0x02   ; read sectors from disk
29    int 0x13      ; call the BIOS routine
30    jmp 0x50:0x0    ; jump and execute the sector!
31
32    hlt ; halt the system
33
34    ; We have to be 512 bytes. Clear the rest of the bytes
```

```
          with O
35   times 510 - ($-$$) db 0
36   dw 0xAA55      ; Boot Signiture
```

The above code jumps to the address `0x50:00` (which is `0x500`). To test the code, load it on a QEMU virtual machine and connect through `gdb`, then place a breakpoint at `0x500`. If `gdb` stops at the address, with the assembly listing is the same code as in `sample.asm`, then the bootloader successfully loaded the program. This is an important milestone, as we ensure that our operating system are loaded and ran properly.

7.7   Improve productivity with scripts

7.7.1 Automate build with GNU Make

Up to this point, the whole development process felt repetitive: whenever a change is made, the same commands are entered again. The commands are also complex. `Ctrl+r` helps, but it still feels tedious.

GNU Make is a program that controls and automates the process of building a complex software. For a small program, like a single C source file, invoking *gcc* is quick and easy. However, soon your software will be more complex, with multiples spanning multiple directories, it is a chore to manually build and link files. To solve such problem, a tool was created to automate away this problem and is called a *build system*. GNU Make is one such of tools. There are various build systems out there, but GNU Make is the most popular in Linux world, as it is used for building the Linux kernel.

For a comprehensive introduction to make, please refer to the official Introduction to Make: `https://www.gnu.org/software/make/manual/html_node/Introduction.html#Introduction`. And that's enough for our project. You can also download the manual in different formats e.g. PDF from the official manual page: `https://www.gnu.org/software/make/manual/` .

With Makefile, we can build simpler commands and save time:

```
                         Makefile
 1  all: bootloader bootdisk

 2

 3  bootloader:

 4      nasm -f bin bootloader.asm -o bootloader.o

 5

 6  kernel:

 7      nasm -f bin sample.asm -o sample.o

 8

 9  bootdisk: bootloader.o kernel.o

10      dd if=/dev/zero of=disk.img bs=512 count=2880

11      dd conv=notrunc if=bootloader.o of=disk.img bs=512
            count=1 seek=0

12      dd conv=notrunc if=sample.o of=disk.img bs=512 count=1
            seek=1
```

Now, with a single command, we can build from start to finish a disk
image with a bootloader at 1st sector and the sample program at 2nd sector:

```
$ make bootdisk
```

```
Output
    nasm -f bin bootloader.asm -o bootloader.o

    nasm -f bin sample.asm -o bootloader.o

    dd if=/dev/zero of=disk.img bs=512 count=2880

    2880+0 records in

    2880+0 records out

    1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.00482188 s, 306 MB/s

    dd conv=notrunc if=bootloader.o of=disk.img bs=512 count=1 seek=0

    0+1 records in

    0+1 records out

    10 bytes copied, 7.0316e-05 s, 142 kB/s

    dd conv=notrunc if=sample.o of=disk.img bs=512 count=1 seek=1

    0+1 records in

    0+1 records out
```

```
10 bytes copied, 0.000208375 s, 48.0 kB/s
```

Looking at the Makefile, we can see a few problems:

First, the name `disk.img` are all over the place. When we want to change the disk image name e.g. `floppy_disk.img`, all the places with the name `disk.img` must be changed manually. To solve this problem, we use a variable, and every appearance of `disk.img` is replaced with the reference to the variable. This way, only one place that is changed - the variable definition - all other places are updated automatically. The following variables are added:

```
BOOTLOADER=bootloader.o
OS=sample.o
DISK_IMG=disk.img.o
```

The second problem is, the name `bootloader` and `sample` appears as part of the filenames of the source files e.g. `bootloader.asm` and `sample.asm`, as well as the filenames of the binary files e.g. `bootloader` and `sample`. Similar to `disk.img`, when a name changed, every reference of that name must also be changed manually for both the names of the source files and the names of the binary files e.g. if we change `bootloader.asm` to `loader.asm`, then the object file `bootloader.o` needs changing to `loader.o`. To solve this problem, instead of changing filenames manually, we create a rule that automatically generate the filenames of one extension to another. In this case, we want any source file that starts with `.asm` to have its equivalent binary files, without any extension e.g. `bootloader.asm` → `bootloader.o`. Such transformation is common, so GNU Make provides built-in functions: `wildcard` and `patsubst` for solving such problems:

```
BOOTLOADER_SRCS := $(wildcard *.asm)
BOOTLOADER_OBJS := $(patsubst %.asm, %.o, $(BOOTLOADER_SRCS
    ))
```

`wildcard` matches any `.asm` file in the current directory, then assigned the list of matched files into the variable `BOOTLOADER_SRCS`. In this case, `BOOTLOADER_SRCS` is assigned the value:

```
    bootloader.asm sample.asm
```

`patsubst` substitutes any filename starts with `.asm` into a filename `.o` e.g. `bootloader.asm` → `bootloader.o`. After `patsubsts` runs, we get a list of object files in `BOOTLOADER_OBJS`:

```
    bootloader.o sample.o
```

Finally, a recipe for building from `.asm` to `.o` are needed:

```
%.o: %.asm
    nasm -f bin $< -o $@
```

▷ `$<` is a special variable that refers to the input of the recipe: `%.asm`.

▷ `$@` is a special variable that refers to the output of the recipe: `%.o`.

When the recipe is executed, the variables are replaced with the actual values. For example, if a transformation is `bootloader.asm` → `bootloader.o`, then the actual command executed when replace the placeholders in the recipe is:

```
    nasm -f bin bootloader.asm -o bootloader.o
```

With the recipe, all the `.asm` files are built automatically with the `nasm` command into `.o` files and we no longer need a separate recipe for each object files. Putting it all together with the new variables, we get a better Makefile:

```
                         Makefile
1  BOOTLOADER=bootloader.o
2  OS=sample.o
3  DISK_IMG=disk.img
4
5  BOOTLOADER_SRCS := $(wildcard *.asm)
6  BOOTLOADER_OBJS := $(patsubst %.asm, %.o, $(BOOTLOADER_SRCS
       ))
```

```
 7
 8   all: bootdisk
 9
10   %.o: %.asm
11       nasm -f bin $< -o $@
12
13   bootdisk: $(BOOTLOADER_OBJS)
14       dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
15       dd conv=notrunc if=$(BOOTLOADER) of=$(DISK_IMG) bs=512
             count=1 seek=0
16       dd conv=notrunc if=$(OS) of=$(DISK_IMG) bs=512 count=1
             seek=1
```

From here on, any `.asm` file is compiled automatically, without an explicit recipe for each file.

The object files are in the same directory as the source files, making it more difficult when working with the source tree. Ideally, object files and source files should live in different directories. We want a better organized directory layout like Figure 7.7.1.

bootloader/ directory holds bootloader source files; os/ holds operating system source files that we are going to write later; build/ holds the object files for both the bootloader, the os and the final disk image disk.img. Notice that bootloader/ directory also has its own Makefile. This Makefile will be responsible for building everything in bootloader/ directory, while the top-level Makefile is released from the burden of building the bootloader, but only the disk image. The content of the Makefile in bootloader/ directory should be:

Figure 7.7.1: A better project layout

```
.
├── bootloader
│   └── bootloader.asm
│   └── Makefile
├── build
│   ├── bootloader
│   │   └── bootloader.o
│   ├── disk.img
│   └── os
│       └── sample.o
├── Makefile
└── os
    ├── Makefile
    └── sample.asm
```
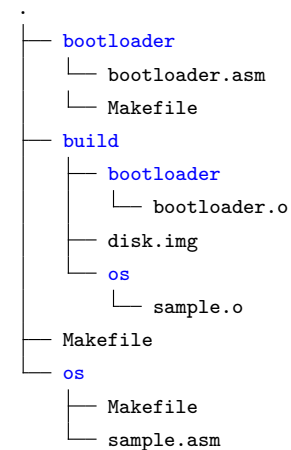
The layout can be displayed with `tree` command:

```
$ tree
```

bootloader/Makefile

```
1   BUILD_DIR=../build/bootloader
2
3   BOOTLOADER_SRCS := $(wildcard *.asm)
4   BOOTLOADER_OBJS := $(patsubst %.asm, $(BUILD_DIR)/%.o, $(
        BOOTLOADER_SRCS))
5
```

```
6   all: $(BOOTLOADER_OBJS)
7
8   $(BUILD_DIR)/%.o: %.asm
9       nasm -f bin $< -o $@
```

Figure 7.7.2: Makefile in `bootloader/`

```
.
├── bootloader
│   └── bootloader.asm
│   └── Makefile
├── build
│   ├── bootloader
│   │   └── bootloader.o
│   ├── disk.img
│   └── os
│       └── sample.o
├── Makefile
└── os
    ├── Makefile
    └── sample.asm
```

Basically everything related to the bootloader in the top-level Makefile are extracted into this Makefile. When `make` runs this Makefile, `bootloader.o` should be built and put into `../build/` directory. As a good practice, all references to `../build/` go through `BUILD_DIR` variable. The recipe for transforming from `.asm → .o` is also updated with proper paths, else it will not work.

▷ `%.asm` refers to the assembly source files in the current directory.

▷ `$(BUILD_DIR)/%.o` refers to the output object files in the build directory in the path `../build/`.

The entire recipe implements the transformation from `<source_file.asm> → ../build/<object_file.o>`. Note that all paths must be correct. If we try to build object files in a different directory e.g. current directory, it will not work since there is no such recipe exists to build objects at such a path.

We also create a similar Makefile for `os/` directory:

Figure 7.7.3: Makefile in `os/`

```
.
├── bootloader
│   └── bootloader.asm
│   └── Makefile
├── build
│   ├── bootloader
│   │   └── bootloader.o
│   ├── disk.img
│   └── os
│       └── sample.o
├── Makefile
└── os
    ├── Makefile
    └── sample.asm
```

os/Makefile

```
1   BUILD_DIR=../build/os
2
3   OS_SRCS := $(wildcard *.asm)
4   OS_OBJS := $(patsubst %.asm, $(BUILD_DIR)/%.o, $(OS_SRCS))
5
6   all: $(OS_OBJS)
7
8   $(BUILD_DIR)/%.o: %.asm
9       nasm -f bin $< -o $@
```
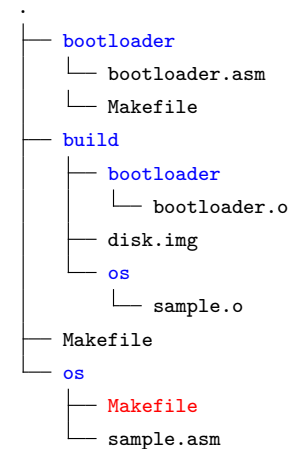
For now, it looks almost identical to the Makefile for bootloader. In the next chapter, we will update it for C code. Then, we update the top-level Makefile:

```
.
├── bootloader
│   └── bootloader.asm
│   └── Makefile
├── build
│   ├── bootloader
│   │   └── bootloader.o
│   ├── disk.img
│   └── os
│       └── sample.o
├── Makefile
└── os
    ├── Makefile
    └── sample.asm
```

Makefile

```
1  BUILD_DIR=build
2  BOOTLOADER=$(BUILD_DIR)/bootloader/bootloader.o
3  OS=$(BUILD_DIR)/os/sample.o
4  DISK_IMG=disk.img
5
6  all: bootdisk
7
8  .PHONY: bootdisk bootloader os
9
10 bootloader:
11     make -C bootloader
12
13 os:
14     make -C os
15
16 bootdisk: bootloader os
17     dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
18     dd conv=notrunc if=$(BOOTLOADER) of=$(DISK_IMG) bs=512
           count=1 seek=0
19     dd conv=notrunc if=$(OS) of=$(DISK_IMG) bs=512 count=1
           seek=1
```
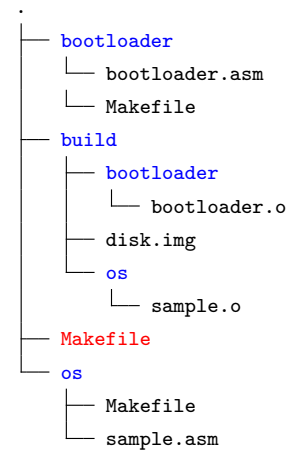
The build process is now truly modularized:

▷ bootloader and os builds are now delegated to child Makefile of respective components. -C option tells make to execute with a Makefile in a supplied directory. In this case, the directories are bootloader/ and os/.

▷ The target all of the top-level Makefile is only responsible for bootdisk target, which is the primary target of this Makefile.

In many cases, a target is not always a filename, but is just a name for a recipe to be always executed when requested. If a filename is of the same name as a target and the file is up-to-date, `make` does not execute the target. To solve this problem, `.PHONY` specifies that some targets are not files. All phony targets will then run when requested, regardless of files of the same names.

To save time entering the command for starting up a QEMU virtual machine, we also add a target to the top-level Makefile:

```
qemu:
    qemu-system-i386 -machine q35 -fda $(DISK_IMG) -gdb tcp
        ::26000 -S
```

One last problem is project cleaning. At the moment, object files need removing manually and this is a repetitive process. Instead, let the Makefile of each component takes care of cleaning its object files, then top-level Makefile performs project cleaning by calling the component Makefile to do the jobs. Each Makefile is added with a `clean` target at the end:

▷ Bootloader Makefile:

```
clean:
    rm $(BUILD_DIR)/*
```

▷ OS Makefile:

```
clean:
    rm $(BUILD_DIR)/*
```

▷ Top-level Makefile:

```
clean:
    make -C bootloader clean
    make -C os clean
```

Simply invoking `make clean` at the project root, all object files the are removed.

7.7.2 GNU Make Syntax summary

GNU Make, at its core, is a domain-specific language for build automation. As any programming language, it needs a way to define data and code. In a Makefile, variables carry data. A variable value is either hard coded or evaluated from invoking a shell such as Bash. All variable values in Make has the same type: a string of text. Number 3 is not a number, but textual representation of the symbol 3. Here are common ways how to define data in a Makefile:

Syntax	Description
| ```
A = 1
B = 2
C = $$(expr $(A) + $(B))
``` ⇒ `A` is 1, `B` is 2, `C` is 3. | Declare a variable and assign a textual value to it. the double dollar sign `$$` means the enclosing expression evaluating by a shell, defined by `/bin/sh`. In this case, the enclosing expression is `(expr $(A) + $(B))` and is evaluated by Bash. |
| ```
PATH = /bin
PATH := $PATH:/usr/bin
``` ⇒ `PATH` is `/bin/:/usr/bin` | Declare a variable and assign to it. However, the difference is that the `=` syntax does not allow refer to a variable to use itself as a value in the right hand side, while this syntax does. |
| ```
PATH = /bin
PATH += /usr/bin
``` ⇒ `PATH` is `/bin/:/usr/bin` | Append a new value at the end of a variable. Equivalent to: `PATH := $PATH:/usr/bin` |
| `CFLAGS ?= -o` ⇒ `CFLAGS` is assigned the value `-o` if it was not defined. | This syntax is called **conditional reference**. Set a variable to a value if it is undefined. This is useful if a user wants to supply different value for a variable from the command line e.g. add debugging option to `CFLAGS`. Otherwise, Make uses the default defined by `?=`. |

```SRCS = lib1.c lib2.c main.c```
```OBJS := $(SRC:.o=.c)```
$\Rightarrow$ OBJS has the value `lib1.o lib2.o`
`main.o`
	This syntax is called **substitution reference**. A part of referenced variable is replaced with something else. In this case, all the `.c` extension is replaced by `.o` extension, thus creating a list of object files for `OBJS` variable from the list of source files from `SRCS` variable.

Code in GNU Make is a collection of recipes that it can run. Each recipe is analogous to a function in a programming language, and can be called like a regular function. Each recipe carries a series of shell commands to be executed by a shell e.g. Bash. A recipe has the following format:

```
target: prerequisites
        command
```

Each `target` is analogous to a function name. Each `prerequisite` is a call another target. Each command is one of Make's built-in commands or a command that is executable by a shell. All prerequisites must be satisfied before entering main body of `target`; that is, each prerequisite must not return any error. If any error is returned, Make terminates the whole build process and prints an error on the command line.

Each time `make` runs, by default if no target is supplied, it starts with `all` target, go through every prerequisites and finally the body of `all`. `all` is analogous to `main` in other programming languages. However, if `make` is given a target, it will start from that target instead of `main`. This feature is useful to automate multiple aspects in a project. For example, one target is for building the project, one target is for generating the documents e.g. test reports, another target for running the whole test suite and `all` runs every main targets.

7.7.3 Automate debugging steps with GDB script

For the convenience, we save GDB configuration to `.gdbinit` file at the project root directory. This configuration is just a collection of GDB commands and a few extra commands. When `gdb` runs, it first loads the `.gdbinit`

file at home directory, then the `.gdbinit` file at the current directory. Why shouldn't we put commands in `~/.gdbinit`? Because these commands are specific to only this project e.g. not all programs are required a remote connection.

Our first configuration:

.gdbinit

```
1  define hook-stop
2      # Translate the segment:offset into a physical address
3      printf "[%4x:%4x] ", $cs, $eip
4      x/i $cs*16+$eip
5  end
```

The above script displays the memory address in `[segment:offset]` format, which is necessary for debugging our bootloader and operating system code.

It is better to use Intel syntax:

```
set disassembly-flavor intel
```

The following commands set a more convenient layout for debugging assembly code:

```
layout asm
layout reg
```

We are currently debugging bootloader code, so it is a good idea to first set it to 16-bit:

```
set architecture i8086
```

Every time the QEMU virtual machine starts, `gdb` must always connect to port `26000`. To avoid the trouble of manually connecting to the virtual machine, add the command:

```
target remote localhost:26000
```

Debugging the bootloader needs a breakpoint at 0x7c00, where our boot-loader code starts:

```
b *0x7c00
```

Now, whenever `gdb` starts, it automatically set correct architecture based on code, automatically connects to the virtual machine[4], displays output in a convenient layout and set a necessary breakpoint. All that need to do is run the program.

[4] The QEMU virtual machine should have already been started before starting `gdb`.

8

Linking and loading on bare metal

Relocation is the process of replacing symbol references with its actual   ***Relocation***
symbolic definitions in an object file. A symbol reference is the memory
address of a symbol.

If the definition is hard to understand, consider a similar analogy: house
relocation. Suppose that a programmer bought a new house and the new
house is empty. He must buy furnitures and appliances to fulfill daily needs
and thus, he made a list of items to buy, and where to place them. To
visualize the placements of new items, he draws a blueprint of the house
and the respective places of all items. He then travels to the shops to
buy goods. Whenever he visit a shop and sees matched items, he tells
the shop owner to note them down. After done selecting, he tells the shop
owner to pick up a brand new item instead of the objects on display, then
give the address for delivering the goods to his new house. Finally, when
the goods arrive, he places the items where he planned at the beginning.

Now that house relocation is clear, object relocation is similar:

▷ The list of items represents the relocation table, where the memory
location for each symbol (item) is predetermined.

▷ Each item represents a pair of *symbol definition* and its *symbol address.*

▷ Each shop represents a compiled object file.

▷ Each item on display represents a symbol definition and references in the object file.

▷ The new address, where all the goods are delivered, represents the final executable binary or the final object file. Since the items on display are not for sale, the shop owner delivers brand new goods instead. Similarly, the object files are not merged together, but copied all over a new file, the object/executable file.

▷ Finally, the goods are placed in the positions according to the shopping list made from the beginning. Similarly, the symbol definitions are placed appropriately in its respective section and the symbol references of the final object/executable file are replaced with the actual memory addresses of the symbol definitions.

8.1   Understand relocations with readelf

Earlier, when we explore object sections, there exists sections that begins with .rel. These sections are relocation tables that maps between a symbol and its location in the final object file or the final executable binary[1].

[1] A .rel section is equivalent to a list of items in the house analogy.

Suppose that a function foo is defined in another object file, so main.c declares it as extern:

main.c

```
int i;
void foo();
int main(int argc, char *argv[])
{
    i = 5;
    foo();
    return 0;
}


void foo() {}
```

When we compile main.c as object file with this command:

```
$ gcc -m32 -masm=intel -c main.c
```

Then, we can inspect the relocation tables with this command:

```
$ readelf -r main.o
```

The output:

Output
```
Relocation section '.rel.text' at offset 0x1cc contains 2 entries:
 Offset     Info    Type              Sym.Value  Sym. Name
00000013  00000801 R_386_32            00000004   i
0000001c  00000a02 R_386_PC32          0000002e   foo
Relocation section '.rel.eh_frame' at offset 0x1dc contains 2 entries:
 Offset     Info    Type              Sym.Value  Sym. Name
00000020  00000202 R_386_PC32          00000000   .text
0000004c  00000202 R_386_PC32          00000000   .text
```

8.1.1 Offset

An *offset* is the location into a section of a binary file, where the actual   ***offset***
memory address of a symbol definition is replaced. The section with `.rel`
prefix determines which section to offset into. For example, `.rel.text`
is the relocation *table* of symbols whose address needs correcting in `.text`
section, at a specific offset into `.text` section. In the example output:

Output
```
0000001c  00000a02 R_386_PC32          0000002e   foo
```

The blue number indicates there exists a reference of symbol `foo` that
is `1c` bytes into `.text` section. To see it clearer, we recompile main.c with
option -g into the file `main_debug.o`, then run `objdump` on it and got:

Output
```
Disassembly of section .text:
00000000 <main>:
int i;
void foo();
```

```
    int main(int argc, char *argv[])
    {
        0:    8d 4c 24 04             lea    ecx,[esp+0x4]
        4:    83 e4 f0                and    esp,0xfffffff0
        7:    ff 71 fc                push   DWORD PTR [ecx-0x4]
        a:    55                      push   ebp
        b:    89 e5                   mov    ebp,esp
        d:    51                      push   ecx
        e:    83 ec 04                sub    esp,0x4
       i = 5;
       11:   c7 05 00 00 00 00 05     mov    DWORD PTR ds:0x0,0x5
       18:   00 00 00
        foo();
       1b:   e8 fc ff ff ff           call   1c <main+0x1c>
        return 0;
       20:   b8 00 00 00 00           mov    eax,0x0
    }
       25:   83 c4 04                 add    esp,0x4
       28:   59                       pop    ecx
       29:   5d                       pop    ebp
       2a:   8d 61 fc                 lea    esp,[ecx-0x4]
       2d:   c3                       ret
    ....irrelevant content omitted....
```

The byte at `1b` is the opcode `e8`, the `call` instruction; byte at `1c` is the value `fc`. Why is the operand value for `e8` is `0xfffffffc`, which is equivalent to -4, but the translated instruction `call 1c`? It will be explained after a few more sections, but you should pause and think a bit about the reason why.

8.1.2 Info

Info specifies index of a symbol in the symbol table and the type of relocation to perform.

Output

```
0000001c  00000a02 R_386_PC32        0000002e   foo
```

The pink number is the index of symbol `foo` in the symbol table, and the green number is the relocation type. The numbers are written in hex format. In the example, `0a` means `10` in decimal, and symbol `foo` is indeed at index `10`:

Output

```
10: 0000002e     6 FUNC    GLOBAL DEFAULT    1 foo
```

8.1.3 Type

Type represents the type value in textual form. Looking at the type of foo:

Output

```
0000001c  00000a02 R_386_PC32        0000002e   foo
```

The green number is type in its numeric form, and `R_386_PC32` is the name assigned to that value. Each value represents a relocation method of calculation. For example, with the type `R_386_PC32`, the following formula is applied for relocation (Inteli386 psABI):

$$Relocated\,Offset = S + A - P$$

To understand the formula, it is necessary to understand symbol values.

8.1.4 Sym.Value

This field shows the *symbol value*. A symbol value is a value assigned to a symbol, whose meaning depends on the `Ndx` field:

A SYMBOL WHOSE SECTION INDEX IS `COMMON`, its symbol value holds alignment constraints.

Example 8.1.1. In the symbol table, the variable i is identified as `COM` (uninitialized variable):[2]

[2] The command for listing symbol table is (assume the object file is `hello.o`):

```
readelf -s hello.o
```

```
Symbol table '.symtab' contains 16 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello2.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT   10
     9: 00000000     0 SECTION LOCAL  DEFAULT   12
    10: 00000000     0 SECTION LOCAL  DEFAULT   14
    11: 00000000     0 SECTION LOCAL  DEFAULT   15
    12: 00000000     0 SECTION LOCAL  DEFAULT   13
    13: 00000004     4 OBJECT  GLOBAL DEFAULT  COM i
    14: 00000000    46 FUNC    GLOBAL DEFAULT    1 main
    15: 0000002e     6 FUNC    GLOBAL DEFAULT    1 foo
```

so its symbol value is a memory alignment for assigning a proper memory address that conforms to the alignment in the final memory address. In the case of `i`, the value is 4, so the starting memory address of `i` in the final binary file will be a multiple of 4.

A SYMBOL WHOSE NDX IDENTIFIES A SPECIFIC SECTION, its symbol value holds a section offset.

Example 8.1.2. In the symbol table, `main` and `foo` belong to section 1:

```
14: 00000000    46 FUNC    GLOBAL DEFAULT    1 main
15: 0000002e     6 FUNC    GLOBAL DEFAULT    1 foo
```

which is `.text`[3] section[4]:

[3] `.text` holds program code and read-only data.

[4] The command for listing sections is (assume the object file is `hello.o`):

```
readelf -S hello.o
```

```
 Output     There are 20 section headers, starting at offset 0x558:

            Section Headers:
              [Nr] Name              Type          Addr     Off    Size   ES Flg Lk Inf Al
              [ 0]                    NULL          00000000 000000 000000 00     0   0  0
              [ 1] .text             PROGBITS      00000000 000034 000034 00  AX 0   0  1
              [ 2] .rel.text         REL           00000000 000414 000010 08   I 18  1  4
              [ 3] .data             PROGBITS      00000000 000068 000000 00  WA 0   0  1
              [ 4] .bss              NOBITS        00000000 000068 000000 00  WA 0   0  1
              [ 5] .debug_info       PROGBITS      00000000 000068 000096 00     0   0  1
            ..... remaining output omitted for clarity....
```

IN THE FINAL EXECUTABLE AND SHARED OBJECT FILES, instead
of the above values, a symbol value holds a memory address.

Example 8.1.3. After compiling hello.o into the final executable hello,
the symbol table now contains the memory address for each symbol[5]:

[5] The command to compile the object
file `hello.o` into the executable `hello`:

`-o hello`

```
 Output     Symbol table '.symtab' contains 75 entries:
               Num:    Value  Size Type    Bind     Vis      Ndx Name
                 0: 00000000     0 NOTYPE  LOCAL    DEFAULT  UND
                 1: 08048154     0 SECTION LOCAL    DEFAULT    1
                 2: 08048168     0 SECTION LOCAL    DEFAULT    2
                 3: 08048188     0 SECTION LOCAL    DEFAULT    3
            ....output omitted...
                64: 08048409     6 FUNC    GLOBAL DEFAULT    14 foo
                65: 0804a020     0 NOTYPE  GLOBAL DEFAULT    26 _end
                66: 080482e0     0 FUNC    GLOBAL DEFAULT    14 _start
                67: 08048488     4 OBJECT  GLOBAL DEFAULT    16 _fp_hw
                68: 0804a01c     4 OBJECT  GLOBAL DEFAULT    26 i
                69: 0804a018     0 NOTYPE  GLOBAL DEFAULT    26 __bss_start
                70: 080483db    46 FUNC    GLOBAL DEFAULT    14 main
            ...ouput omitted...
```

Unlike the values of the symbols `foo`, `i` and `main` as in the hello.o ob-
ject file, the complete memory addresses are in place.

Now it suffices to understand relocation types. Previously, we mentioned the type `R_386_PC32`. The following formula is applied for relocation (Inteli386 psABI):

$$Relocated\,Offset = S + A - P$$

where

S represents the value of the symbol. In the final executable binary, it is the address of the symbol.

A represents the addend, an extra value added to the value of a symbol.

P Represents the memory address to be fixed.

Relocate Offset is the distance between a relocating location[6] and the actual memory location of a symbol definition, or a memory address.

[6] where the referenced memory address is to be fixed.

But why do we waste time in calculating a distance instead of replacing with a direct memory address? The reason is that x86 architecture does not use employ any addressing mode that uses an absolute memory address, as listed in table 4.5.2. All addressing modes in x86 are relative. In some assembly language, an absolute address can be used simply because it is a syntactic sugar that is later transformed into one of the relative addressing mode provided by the x86 hardware by the assembler.

Example 8.1.4. For the `foo` symbol:

```
Output        0000001c  00000a02 R_386_PC32        0000002e   foo
```

The distance between the usage of `foo` in `main.o` and its definition, applying the formula $S + A - P$ is: `2e + 0 - 1c = 12`. That is, the place where memory fixing starts is `0x12` or 18 bytes away from *the definition* of the symbol `foo`. However, to make an instruction works properly, we must also subtract 4 from `0x12` and results in `0xe`. Why the extra `-4`? Because the relative address starts at *the end* of an instruction, *not the*

address where memory fixing starts. For that reason, we must also exclude the 4 bytes of the overwritten address.

Indeed, looking at the `objdump` output of the object file `hello.o`:

Output

```
Disassembly of section .text:
00000000 <main>:
   0:   8d 4c 24 04             lea     ecx,[esp+0x4]
   4:   83 e4 f0                and     esp,0xfffffff0
   7:   ff 71 fc                push    DWORD PTR [ecx-0x4]
   a:   55                      push    ebp
   b:   89 e5                   mov     ebp,esp
   d:   51                      push    ecx
   e:   83 ec 04                sub     esp,0x4
  11:   c7 05 00 00 00 00 05    mov     DWORD PTR ds:0x0,0x5
  18:   00 00 00
  1b:   e8 fc ff ff ff          call    1c <main+0x1c>
  20:   b8 00 00 00 00          mov     eax,0x0
  25:   83 c4 04                add     esp,0x4
  28:   59                      pop     ecx
  29:   5d                      pop     ebp
  2a:   8d 61 fc                lea     esp,[ecx-0x4]
  2d:   c3                      ret
0000002e <foo>:
  2e:   55                      push    ebp
  2f:   89 e5                   mov     ebp,esp
  31:   90                      nop
  32:   5d                      pop     ebp
  33:   c3                      ret
```

The place where memory fixing starts is after the opcode `e8`, with the mock value `fc ff ff ff`, which is `-4` in decimal. However, the assembly code, the value is displayed as `1c`. The memory address right after `e8`. The reason is that the instruction `e8` starts at `1b` and ends at `20`[7]. `-4` means 4 bytes backward from the end of instruction, that is: $20 - 4 = 1c$. After linking, the output of the final executable file is displayed with the actual memory fixing:

[7] The end of an instruction is the memory address right after its last operand. The whole instruction e8 spans from the address 1b to the address 1f.

```
080483db <main>:
  80483db:        8d 4c 24 04              lea    ecx,[esp+0x4]
  80483df:        83 e4 f0                 and    esp,0xfffffff0
  80483e2:        ff 71 fc                 push   DWORD PTR [ecx-0x4]
  80483e5:        55                       push   ebp
  80483e6:        89 e5                    mov    ebp,esp
  80483e8:        51                       push   ecx
  80483e9:        83 ec 04                 sub    esp,0x4
  80483ec:        c7 05 1c a0 04 08 05     mov    DWORD PTR ds:0x804a01c,0x5
  80483f3:        00 00 00
  80483f6:        e8 0e 00 00 00           call   8048409 <foo>
  80483fb:        b8 00 00 00 00           mov    eax,0x0
  8048400:        83 c4 04                 add    esp,0x4
  8048403:        59                       pop    ecx
  8048404:        5d                       pop    ebp
  8048405:        8d 61 fc                 lea    esp,[ecx-0x4]
  8048408:        c3                       ret
08048409 <foo>:
  8048409:        55                       push   ebp
  804840a:        89 e5                    mov    ebp,esp
  804840c:        90                       nop
  804840d:        5d                       pop    ebp
  804840e:        c3                       ret
  804840f:        90                       nop
```

In the final output, the opcode `e8` previously at `1b` now starts at the
address 80483f6. The mock value `fc ff ff ff` is replaced with the ac-
tual value `0e 00 00 00` using the same calculating method from its ob-
ject file: opcode e8 is at 80483f6. The definition of `foo` is at 8048409.
The offset from the next address after e8 is $8048409 + 0 - 80483f7 - 4 = 0e$.
However, for readability, the assembly is displayed as `call 8048409 <foo>`,
since `GNU as`[8] assembler allows specifying the actual memory address
of a symbol definition. Such address is later translated into relative ad-
dressing mode, saving the programmer the trouble of calculating offset

[8] Or any current assembler in use to-
day.

manually.

This field displays the name of a symbol to be relocated. The named symbol is the same as written in a high level language such as C.

8.2 Crafting ELF binary with linker scripts

A *linker* is a program that combines separated object files into a final binary file. When `gcc` is invoked, it runs `ld` underneath to turn object files into the final executable file..

    linker

  A *linker script* is a text file that instructs how a linker should combine object files. When `gcc` runs, it uses its default linker script to build the memory layout of a compiled binary file. Standardized memory layout is called *object file format* e.g. ELF includes program headers, section headers and their attributes. The default linker script is made for running in the current operating system environment[9]. Running on bare metal, the default script cannot be used as it is not designed for such environment. For that reason, a programmer needs to supply his own linker script for such environments.

    linker script

[9] To view the default script use `--verbose` option:

```
ld --verbose
```

  Every linker script consists of a series of commands with the following format:

```
COMMAND
{
    sub-command 1
    sub-command 2
    .... more sub-command....
}
```

Each sub-command is specific to only the top-level command. The simplest linker script needs only one command: `SECTION`, that consumes input sections from object files and produces output sections of the final binary file[10].

[10] Recall that sections are chunks of code or data, or both.

8.2.1 Example linker script

Here is a minimal example of a linker script:

<div align="center">main.lds</div>

```
SECTIONS /* Command */
{
  . = 0x10000; /* sub-command 1 */
  .text : { *(.text) } /* sub-command 2 */
  . = 0x8000000; /* sub-command 3 */
  .data : { *(.data) } /* sub-command 4 */
  .bss : { *(.bss) } /* sub-command 5 */
}
```

Code Dissection:

Code	Description
SECTION	Top-level command that declares a list of custom program sections. `ld` provides a set of such commands.
. = 0x10000;	Set location counter to the address `0x10000`. Location counter specifies the base address for subsequent commands. In this example, subsequent commands will use `0x10000` onward.
.text : { *(.text) }	Since location counter is set to `0x10000`, the output `.text` in the final binary file will starts at the address `0x10000`. This command combines all `.text` sections from all object files with `*(.text)` syntax into a final `.text` section. The `*` is the wildcard which matches any file name.
. = 0x8000000;	Again, the location counter is set to `0x8000000`. Subsequent commands will use this address for working with sections.
.data : { *(.data) }	All `.data` section are combined into one `.data` section in the final binary file.
.bss : { *(.bss) }	All `.bss` section are combined into one `.bss` section in the final binary file.

The addresses `0x10000` and `0x8000000` are called *Virtual Memory Address*.

A *virtual memory address* is the address where a section is loaded in memory when a program runs. To use the linker script, we save it as a file e.g. `main.lds`[11]; then, we need a sample program in a file, e.g. `main.c`:

virtual memory address

[11] `.lds` is the extension for linker script.

<div align="center">main.c</div>

```
void test() {}
```

```
int main(int argc, char *argv[])
{

    return 0;
}
```

Then, we compile the file and explicitly invoke `ld` with the linker script:

```
$ gcc -m32 -g -c main.c
$ ld -m elf_i386 -o main -T main.lds main.o
```

In the `ld` command, the options are similar to `gcc`:

Option	Description
-m	Specify object file format that `ld` produces. In the example, elf_i386 means a 32-bit ELF is to be produced.
-o	Specify the name of the final executable binary.
-T	Specify the linker script to use. In the example, it is `main.lds`.

The remaining input is a list of object files for linking. After the command `ld` is executed, the final executable binary - `main` - is produced. If we try running it:

```
$ ./main
Segmentation fault
```

The reason is that when linking manually, the entry address must be explicitly set, or else `ld` sets it to the start of `.text` section by default. We can verify from the `readelf` output:

```
$ readelf -h main
```

Output

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
```

```
    OS/ABI:                         UNIX - System V

    ABI Version:                    0

    Type:                           EXEC (Executable file)

    Machine:                        Advanced Micro Devices X86-64

    Version:                        0x1

    Entry point address:            0x10000

    Start of program headers:       64 (bytes into file)

    Start of section headers:       2098144 (bytes into file)

    Flags:                          0x0

    Size of this header:            64 (bytes)

    Size of program headers:        56 (bytes)

    Number of program headers:      3

    Size of section headers:        64 (bytes)

    Number of section headers:      14

    Section header string table index: 11
```

The entry point address is set to 0x10000, which is the beginning of
.text section. Using objdump to examine the address:

```
$ objdump -z -M intel -S -D prog | less
```

we see that the address 0x10000 does not start at main function when
the program runs:

Output

```
Disassembly of section .text:
00010000 <test>:
int a = 5;
int i;
void test(){}
   10000:       55                      push    ebp
   10001:       89 e5                   mov     ebp,esp
   10003:       90                      nop
   10004:       5d                      pop     ebp
   10005:       c3                      ret
00010006 <main>:
```

```
    int main(int argc, char *argv[])
    {
       10006:        55                           push    ebp
       10007:        89 e5                        mov     ebp,esp


        return 0;
       10009:        b8 00 00 00 00               mov     eax,0x0
    }
       1000e:        5d                           pop     ebp
       1000f:        c3                           ret
```

The start of .text section at 0x10000 is the function test, not main!
To enable the program to run at main properly, we need to set the entry
point in the linker script with the following line at the beginning of the
file:

```
ENTRY(main)
```

Recompile the executable binary file main again. This time, the output
from readelf is different:

Output

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x10006
  Start of program headers:          52 (bytes into file)
  Start of section headers:          9168 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
```

```
Size of program headers:          32 (bytes)
Number of program headers:        3
Size of section headers:          40 (bytes)
Number of section headers:        14
Section header string table index: 11
```

The program now executes code at the address 0x10006 when it starts.
0x10006 is where main starts! To make sure we really starts at main, we
run the program with gdb, set two breakpoints at main and test func-
tions:

```
$ gdb ./main
```

Output
```
.... output omitted ....
Reading symbols from ./main...done.
```

```
(gdb) b test
```

Output
```
Breakpoint 1 at 0x10003: file main.c, line 1.
```

```
(gdb) b main
```

Output
```
Breakpoint 2 at 0x10009: file main.c, line 5.
```

```
(gdb) r
```

Output
```
Starting program: /tmp/main
Breakpoint 2, main (argc=-11493, argv=0x0) at main.c:5
5       return 0;
```

As displayed in the output, gdb stopped at the 2nd breakpoint first.
Now, we run the program normally, without gdb:

```
$ ./main
Segmentation fault
```

We still get a segmentation fault. It is to be expected, as we ran a custom binary without C runtime support from the operating system. The last statement in the `main` function: `return 0`, simply returns to a random place[12]. The C runtime ensures that the program exit properly. In Linux, the `_exit()` function is implicitly called when `main` returns. To fix this problem, we simply change the program to exit properly:

[12] Return address is above the current `ebp`. However, when we enter `main`, no return value is pushed on the stack. So, when return is executed, it simply retrieves any value above `ebp` and use as a return address.

hello.c

```
1  void test() {}
2  int main(int argc, char *argv[])
3  {
4      asm("mov eax, 0x1\n"
5          "mov ebx, 0x0\n"
6          "int 0x80");
7  }
```

Inline assembly is required because interrupt `0x80` is defined for system calls in Linux. Since the program uses no library, there is no other way to call system functions, aside from using assembly. However, when writing our operating system, we will not need such code, as there is no environment for exiting properly yet.

Now that we can precisely control where the program runs initially, it is easy to bootstrap the kernel from the bootloader. Before we move on to the next section, note how `readelf` and `objdump` can be applied to debug a program even before it runs.

8.2.2 Understand the custom ELF structure

In the example, we manage to create a runnable ELF executable binary from a custom linker script, as opposed to the default one provided by `gcc`. To make it convenient to look into its structure:

```
$ readelf -e main
```

-e option is the combination of 3 options -h -l -S:

Output

```
....... ELF header output omitted .......
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010000 001000 000010 00  AX  0   0  1
  [ 2] .eh_frame         PROGBITS        00010010 001010 000058 00   A  0   0  4
  [ 3] .debug_info       PROGBITS        00000000 001068 000087 00      0   0  1
  [ 4] .debug_abbrev     PROGBITS        00000000 0010ef 000074 00      0   0  1
  [ 5] .debug_aranges    PROGBITS        00000000 001163 000020 00      0   0  1
  [ 6] .debug_line       PROGBITS        00000000 001183 000038 00      0   0  1
  [ 7] .debug_str        PROGBITS        00000000 0011bb 000078 01  MS  0   0  1
  [ 8] .comment          PROGBITS        00000000 001233 000034 01  MS  0   0  1
  [ 9] .shstrtab         STRTAB          00000000 00133a 000074 00      0   0  1
  [10] .symtab           SYMTAB          00000000 001268 0000c0 10     11  10  4
  [11] .strtab           STRTAB          00000000 001328 000012 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00010000 0x00010000 0x00068 0x00068 R E 0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
 Section to Segment mapping:
  Segment Sections...
   00     .text .eh_frame
   01
```

The structure is incredibly simple. Both the segment and section listings can be contained within one screen. This is not the case with default ELF executable binary. From the output, there are only 11 sections,

and only two are loaded at runtime: `.text` and `.eh_frame` because both section are assigned with an actual memroy addresses, `0x10000` and `0x10010` respectively. The remaining sections are assigned with `0` *in the final executable binary*[13], which mean they are not loaded at runtime. It makes sense, as those sections are related to versioning[14], debugging[15] and linking[16].

The program segment header table is even simpler. It only contains 2 segments: `LOAD` and `GNU_STACK`. By default, if the linker script does not supply the instructions for building program segments, `ld` provides reasonable default segments. As in this case, `.text` should be in the `LOAD` segment. `GNU_STACK` segment is a GNU extension used by the Linux kernel to control the state of the program stack. We will not need this segment, along with `.eh_frame`, which is for exception handling, as we write our own operating system from scratch. To achieve these goals, we will need to create our own program headers instead of letting `ld` handles the task, and instruct ld to remove `.eh_frame`.

8.2.3 Manipulate the program segments

First, we need to craft our own program header table by using the following syntax:

```
PHDRS
{
  <name> <type> [ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
       [ FLAGS ( flags ) ] ;
}
```

PHDRS command, similar to `SECTION` command, but for declaring a list of custom program segments with a predefined syntax.

name is the header name for later referenced by a section declared in `SECTION` command.

type is the ELF segment type, as described in section Section 5.5, with added prefix `PT_`. For example, instead of `NULL` or `LOAD` as displayed by `readelf`, it is `PT_NULL` or `PT_LOAD`.

[13] As opposed to the object files, where memory addresses are always 0 and only assigned with actual values in the linking process.
[14] It is the .comment section. It can be viewed with the comment `readelf -p .comment main`.
[15] The ones starts with `.debug` prefix.
[16] The symbol tables and string table.

Example 8.2.1. With only `name` and `type`, we can create any number of program segments. For example, we can add the `NULL` program segment and remove the `GNU_STACK` segment:

main.lds

```
1   PHDRS
2   {
3       null PT_NULL;
4       code PT_LOAD;
5   }
6
7   SECTIONS
8   {
9       . = 0x10000;
10      .text : { *(.text) } :code
11      . = 0x8000000;
12      .data : { *(.data) }
13      .bss : { *(.bss) }
14  }
```

The content of `PHDRS` command tells that the final executable binary contains 2 program segments: `NULL` and `LOAD`. The `NULL` segment is given the name `null` and `LOAD` segment given the name code to signify this `LOAD` segment contains program code. Then, to put a section into a segment, we use the syntax `:<phdr>`, where `phdr` is the name given to a segment earlier. In this example, `.text` section is put into `code` segment. We compile and see the result (assuming `main.o` compiled earlier remains):

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
```

```
      There are 2 program headers, starting at offset 52
      Program Headers:
        Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
        NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
        LOAD           0x001000 0x00010000 0x00010000 0x00010 0x00010 R E 0x1000
       Section to Segment mapping:
        Segment Sections...
         00
         01      .text .eh_frame
```

Those 2 segments are now `NULL` and `LOAD` instead of `LOAD` and `GNU_STACK`.

Example 8.2.2. We can add as many segments of the same type, as long as they are given different names:

main.lds

```
 1  PHDRS
 2  {
 3      null1 PT_NULL;
 4      null2 PT_NULL;
 5      code1 PT_LOAD;
 6      code2 PT_LOAD;
 7  }
 8
 9  SECTIONS
10  {
11      . = 0x10000;
12      .text : { *(.text) } :code1
13      .eh_frame : { *(.eh_frame) } :code2
14      . = 0x8000000;
15      .data : { *(.data) }
16      .bss : { *(.bss) }
17  }
```

After amending the `PHDRS` content earlier with this new segment listing, we put `.text` into `code1` segment and `.eh_frame` into `code2` segment, we compile and see the new segments:

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 4 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  LOAD           0x001000 0x00010000 0x00010000 0x00010 0x00010 R E 0x1000
  LOAD           0x001010 0x00010010 0x00010010 0x00058 0x00058 R   0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01
   02     .text
   03     .eh_frame
```

Now `.text` and `.eh_frame` are in different segments.

FILEHDR is an optional keyword, when added specifies that a program
segment includes the ELF file header of the executable binary. However,
this attribute should only added for the first program segment, as it
drastically alters the size and starting address of a segment because
the ELF header is always at the beginning of a binary file, recall that
a segment starts at the address of its first content, which is in most
of the cases (except for this case, which is the file header), the first sec-
tion.

Example 8.2.3. Adding the `FILEHDR` keyword changes the size of `NULL`
segment:

```
                          main.lds
PHDRS
{
    null PT_NULL FILEHDR;
    code PT_LOAD;
}
..... content is the same .....
```

We link it again and see the result:

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
    Elf file type is EXEC (Executable file)
    Entry point 0x10000
    There are 2 program headers, starting at offset 52
    Program Headers:
      Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
      NULL           0x000000 0x00000000 0x00000000 0x00034 0x00034 R   0x4
      LOAD           0x001000 0x00010000 0x00010000 0x00068 0x00068 R E 0x1000
     Section to Segment mapping:
      Segment Sections...
       00
       01     .text .eh_frame
```

In previous examples, the file size and memory size of the `NULL` section are always 0, now they are both `34` bytes, which is the size of an ELF header.

Example 8.2.4. If we assign `FILEHDR` to a non-starting segment, its size and starting address changes significantly:

```
                          main.lds
PHDRS
{
    null PT_NULL;
```

```
    code PT_LOAD FILEHDR;
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  LOAD           0x000000 0x0000f000 0x0000f000 0x01068 0x01068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

The size of the `LOAD` segment in the previous example is only `0x68`, the same size as the total sizes of `.text` and `.eh_frame` sections in it. But now, it is `0x01068`, got `0x1000` bytes larger. What is the reason for these extra bytes? A simple answer: segment alignment. From the output, the alignment of this segment is `0x1000`; it means that regardless of which address is the start of this segment, it must be divisible by `0x1000`. For that reason, the starting address of `LOAD` is `0xf000` because it is divisible by `0x1000`.

Another question arises: why is the starting address `0xf000` instead of `0x10000`? `.text` is the first section, which starts at `0x10000`, so the segment should start at `0x10000`. The reason is that we include `FILEHDR` as part of the segment, it must expand to include the ELF file header, which is at the very start of an ELF executable binary. To satisfy this constraint and the alignment constraint, `0xf000` is the closest address. Note that the virtual and physical memory addresses are the addresses

at runtime, not the locations of the segment in the file on disk. As the
`FileSiz` field shows, the segment only consumes `0x1068` bytes on disk.
Figure 8.2.1 illustrates the difference between the memory layouts with
and without `FILEHDR` keyword.
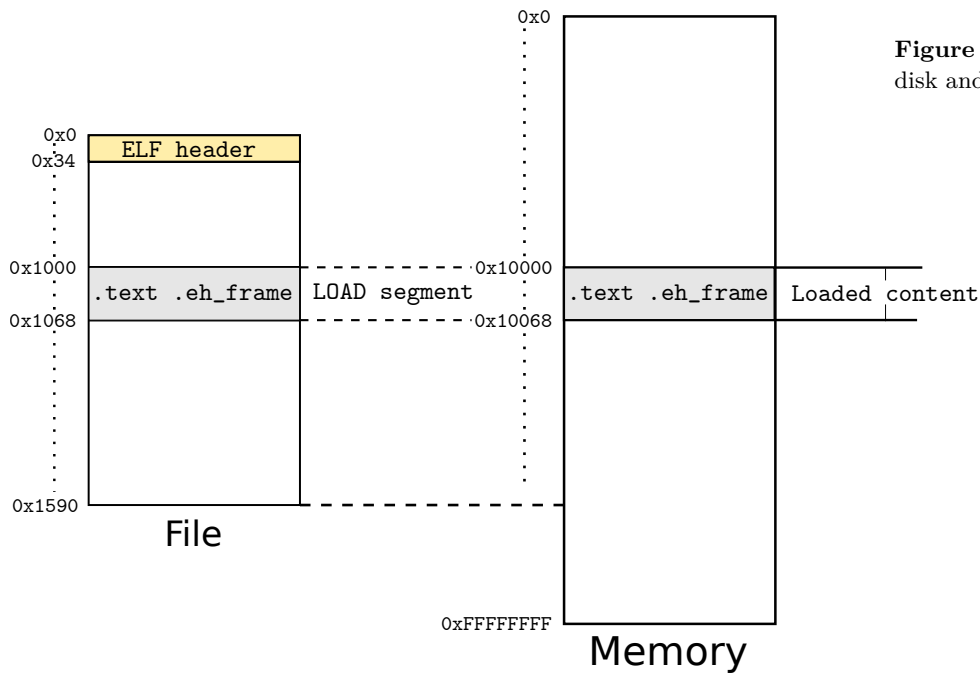
Figure 8.2.1: `LOAD` segment   on disk and in memory.

(a) Without `FILEHDR`.
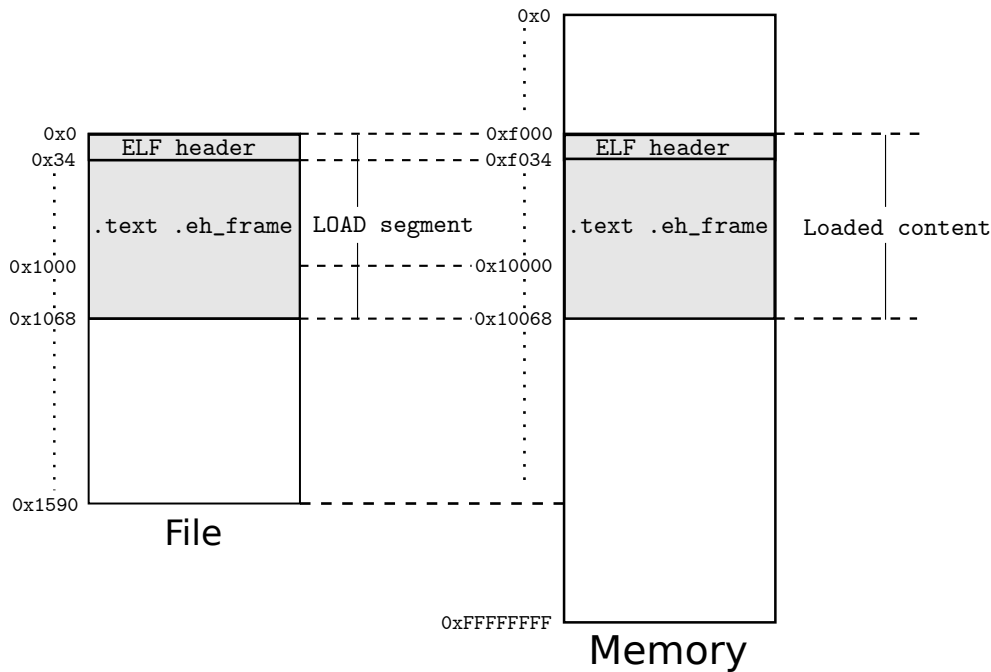
(b) With `FILEHDR`.

PHDRS is an optional keyword, when added specifies that a program segment is a program segment header table.

Example 8.2.5. The first segment of the default executable binary generated by `gcc` is a PHDR since the program segment header table appears right after the ELF header. It is also a convenient segment to put the ELF header into using the `FILEHDR` keyword. We replace the unused `NULL` segment earlier with a `PHDR` segment:

main.lds

```
PHDRS
{
    headers PT_PHDR FILEHDR PHDRS;
    code PT_LOAD FILEHDR;
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x001000 0x00010000 0x00010000 0x00068 0x00068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

As shown in the output, the first segment is of type PHDR. Its size is 0x74, which includes:

▷ `0x34` bytes for ELF header.

▷ `0x40` bytes for the program segment header table, with 2 entries, each
   is `0x20` bytes (32 bytes) in length.

The above number is consistent with ELF header output:

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
....... output omitted ......
  Size of this header:               52 (bytes)   --> 0x34 bytes
  Size of program headers:           32 (bytes)   --> 0x20 bytes each program header
  Number of program headers:         2            --> 0x40 bytes in total
  Size of section headers:           40 (bytes)
  Number of section headers:         12
  Section header string table index: 9
```

AT (address) specifies the load memory address where the segment
is placed. Every segment or section has a *virtual memory address* and
a *load memory address*:

▷ A *virtual memory address* is a starting address of a segment or a   *virtual memory address*
   section when a program is in memory and running. The memory
   address is called virtual because it does not map to the actual mem-
   ory cell that corresponds to the address number, but any random
   memory cell, which depends on how the underlying operating sys-
   tem translates the address. For example, the virtual memory ad-
   dress `0x1` might map to the memory cell with the physical address
   `0x1000`.

▷ A *load memory address* is the physical memory address, where a   *load memory address*
   program is loaded but not yet running.

The load memory address is specified by `AT` syntax. Normally both
types of addresses are the same, and the physical address can be ig-

nored. They differ when loading and running are purposely divided
into two distinct phases that require different address regions.

For example, a program can be designed to load into a ROM[17] at a
fixed address. But when loading into RAM for a bare-metal applica-
tion or an operating system to use, the program needs a load address
that accommodates the addressing scheme of the target application
or operating system.

[17] Read-Only Memory

Example 8.2.6. We can specify a load memory address for the segment
LOAD with AT syntax:

main.lds

```
PHDRS
{
    headers PT_PHDR FILEHDR PHDRS AT(0x500);
    code PT_LOAD;
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x4000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000500 0x00074 0x00074 R   0x4
  LOAD           0x001000 0x00004000 0x00002000 0x00068 0x00068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

It depends on an operating system whether to use the address or not. For our operating system, the virtual memory address and load are the same, so an explicit load address is none of our concern.

FLAGS (flags) assigns permissions to a segment. Each flag is an integer that represents a permission and can be combined with OR operations. Possible values:

Permission	Value	Description
R	1	Readable
W	2	Writable
E	4	Executable

Example 8.2.7. We can create a LOAD segment with Read, Write and Execute permissions enabled:

main.lds

```
PHDRS
{
    headers PT_PHDR FILEHDR PHDRS AT(0x500);
    code PT_LOAD FILEHDR FLAGS(0x1 | 0x2 | 0x4);
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

```
Output
    Elf file type is EXEC (Executable file)
    Entry point 0x0
    There are 2 program headers, starting at offset 52
    Program Headers:
      Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
      PHDR           0x000000 0x00000000 0x00000500 0x00074 0x00074 R   0x4
```

```
   LOAD                0x001000 0x00000000 0x00000000 0x00010 0x00010 RWE 0x1000
  Section to Segment mapping:
   Segment Sections...
    00
    01     .text .eh_frame
```

`LOAD` segment now gets all the `RWE` permissions, as shown above.

Finally, we want to remove the `.eh_frame` or any unwanted section, we add a special section called `/DISCARD/`:

main.lds

```
... program segment header table remains the same ...


SECTIONS
{
    /* . = 0x10000; */
    .text : { *(.text) } :code
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
    /DISCARD/ : { *(.eh_frame) }
}
```

Any section putting in `/DISCARD/` disappears in the final executable binary:

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

```
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52
Program Headers:
```

```
    Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
    PHDR           0x000000 0x00000000 0x00000500 0x00074 0x00074 R    0x4
    LOAD           0x001000 0x00000000 0x00000000 0x00010 0x00010 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01      .text
```

As can be seen, `.eh_frame` is nowhere to be found.

8.3   C Runtime: Hosted vs Freestanding

The purpose of `.init`, `.init_array`, `.fini_array` and `.preinit_array` section is to initialize a C Runtime environment that supports the C standard libraries. Why does C need a runtime environment, when it is supposed to be a compiled language? The reason is that many of the standard functions depend on the underlying operating system, which is of itself a big runtime environment. For example, I/O related functions such as reading from keyboard with `gets()`, reading from file with `open()`, printing on screen with `printf()`, managing system memory with `malloc()`, `free()`, etc.

A C implementation cannot provide such routines without a running operating system, which is a *hosted environment*. A *hosted environment* is a runtime environment that:

▷ provides a default implementation of C libraries that includes system-dependent data and routines.

▷ perform resource allocations to prepare an environment for a program to run.

This process is similar to the hardware initialization process:

▷ When first powered up, a desktop computer loads its basic system routines from a read-only memory stored on the motherboard.

▷ Then, it starts initializing an environment, such as setting default values for various registers in CPU and devices, before executing the any

code.

In contrast, a *freestanding environment* is an environment that does not provide system-dependent data and routines. As a consequence, almost no C library exists and the environment can run code compiled written from pure C syntax. For a free standing environment to become a host environment, it must implement standard C system routines. But for a *conforming* freestanding environment, it only needs these header files available: `<float.h>`, `<limits.h>`, `<stadarg.h>` and `<stddef.h>` (according to GCC manual).

For a typical desktop x86 program, C runtime environment is initialized by a compiler so a program runs normal. However, for an embedded platform where a program runs directly on it, this is not the case. The typical C runtime environment used in desktop operating systems cannot be used on the embedded platforms, because architectural differences and resource constraints. As such, the software writer must implement a custom C runtime environment suitable for the targeted platform. For the embedded platform,

In writing our operating system, the first step is to create a freestanding environment before creating a hosted one.

8.4 Debuggable bootloader on bare metal

Currently, the bootloader is compiled as a flat binary file. Although `gdb` can display the assembly code, it is not always the same as the source code. In the assembly source code, there exists variable names and labels. These symbols are lost when compiled as a flat binary file, making debugging more difficult. Another issue is the mismatch between the written assembly source code and the displayed assembly source code. The written code might contain higher level syntax that is assembler-specific and is generated into lower-level assembly code as displayed by `gdb`. Finally, with debug information available, the command `next/n` and `prev/p` can be used instead of `ni` and `si`.

To enable debug information, we modify the bootloader Makefile:

1. The bootloader must be compiled as a ELF binary. Open the Makefile

in bootloader/ directory and change this line under `$(BUILD_DIR)/%.o:` `%.asm` recipe:

```
nasm -f bin $< -o $@
```

to this line:

```
nasm -f elf $< -F dwarf -g -o $@
```

In the updated recipe, `bin` format is replaced with `elf` format to enable debugging information to be properly produced.`-F` option specifies the debug information format, which is `dwarf` in this case. Finally, `-g` option causes `nasm` to actually generate debug information in selected format.

2. Then, `ld` consumes the ELF bootloader binary and produces another ELF bootloader binary, with proper starting memory address of `.text` section that match the actual address of the bootloader at runtime, when QEMU virtual machine loads it at `0x7c00`. We need `ld` because when compiled by `nasm`, the starting address is assumed to be `0`, not `0x7c00`.

3. Finally, we use `objcopy` to separate extract only the flat binary content as the original bootloader by adding this line to `$(BUILD_DIR)/%.o:` `%.asm`:

```
objcopy -O binary $(BUILD_DIR)/bootloader.o.elf $@
```

`objcopy`, as its name implies, is a program that copies and translates object files. Here, we copy the original ELF bootloader and translate it into a flat binary file.

The updated recipe should look like:

```
$(BUILD_DIR)/%.o: %.asm
    nasm -f elf $< -F dwarf -g -o $@
    ld -m elf_i386 -T bootloader.lds $@ -o $@.elf
    objcopy -O binary $(BUILD_DIR)/bootloader.o.elf $@
```

Now we test the bootloader with debug information available:

1. Start the QEMU machine:

```
$ make qemu
```

2. Start `gdb` with the debug information stored in `bootloader.o.elf`:

```
$ gdb build/bootloader/bootloader.o.elf
```

After getting into `gdb`, press the `Enter` key and if the sample `.gdbinit` section 7.7.3 is used, the output should look like:

Output
```
---Type <return> to continue, or q <return> to quit---
[f000:fff0] 0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00: file bootloader.asm, line 6.
(gdb)
```

`gdb` now understand where the instruction at address `0x7c00` is in the assembly source file, thanks to the debug information.

8.5   Debuggable program on bare metal

The process of building a debug-ready executable binary is similar to that of a bootloader, except more involved. Recall that for a debugger to work properly, its debugging information must contain correct address mappings between memory addresses and the source code. `gcc` stores such mapping information in DIE entries, in which it tells `gdb` at which code address corresponds to a line in a source file, so that breakpoints work properly.

But first, we need a sample C source file, a very simple one:

os.c
```
void main() {}
```

Because this is a free standing environment, standard libraries that involve system functions such as `printf()` would not work, because a C runtime does not exist. At this stage, the goal is to correctly jump to main with source code displayed properly in `gdb`, so no fancy C code is needed yet.

The next step is updating `os/Makefile`:

```
BUILD_DIR=../build
OS=$(BUILD_DIR)/os


CFLAGS+=-ffreestanding -nostdlib -gdwarf-4 -m32 -ggdb3


OS_SRCS := $(wildcard *.c)
OS_OBJS := $(patsubst %.c, $(BUILD_DIR)/%.o, $(OS_SRCS))


all: $(OS)


$(BUILD_DIR)/%.o: %.c
    gcc $(CFLAGS) -c $< -o $@


$(OS): $(OS_OBJS)
    ld -m elf_i386 -Tos.lds $(OS_OBJS) -o $@


clean:
    rm $(OS_OBJS)
```

We updated the Makefile with the following changes:

▷ Add a `CFLAGS` variable for passing options to `gcc`.

▷ Instead of the rule to build assembly source code earlier, it is replaced with a C version with a recipe to build C source files. The `CFLAGS` variable makes the `gcc` command in the recipe looks cleaner regardless how many options are added.

▷ Add a linking command for building the final executable binary of the operating system with a custom linker script `os.lds`.

Everything looks good, except for the linker script part. Why is it needed? The linker script is required for controlling at which physical memory address the operating system binary appears in the memory, so the linker can jump to the operating system code and execute it. To complete this requirement, the default linker script used by `gcc` would not work as it assumes the compiled executable runs inside an existing operating system, while we are writing an operating system itself.

The next question is, what will be the content in the linker script? To answer this question, we must understand what goals to achieve with the linker script:

▷ For the bootloader to correctly jump to and execute the operating system code.

▷ For `gdb` to debug correctly with the operating system source code.

To achieve the goals, we must devise a design of a suitable memory layout for the operating system. Recall that the bootloader developed in chapter 7 can already load a simple binary compiled from the sample Assembly program `sample.asm`. To load the operating system, we can simply throw binary compiled from `sample.asm` with the binary compiled from `os.c` above.

If only it is that simple. The idea is correctly, but not enough. The goals implies the following constraints:

1. The operating system code is written in C and compiled as an ELF executable binary. It means, the bootloader needs to retrieve correct entry address from the ELF header.

2. To debug properly with `gdb`, the debug info must contain correct mappings between instruction addresses and source code.

Thanks to the understanding of ELF and DWARF acquire in the earlier chapters, we can certainly modify the bootloader and create an executable binary that satisfy the above constraint. We will solve these problems one by one.

8.5.1 *Loading an ELF binary from a bootloader*

Earlier we examined that an ELF header contains a entry address of a program. That information is 0x18 bytes away from the beginning of an ELF header, according to `man elf` :

```
typedef struct {
            unsigned char e_ident[EI_NIDENT];
            uint16_t e_type;
            uint16_t e_machine;
            uint32_t e_version;
            ElfN_Addr e_entry;
            ElfN_Off e_phoff;
            ElfN_Off e_shoff;
            uint32_t e_flags;
            uint16_t e_ehsize;
            uint16_t e_phentsize;
            uint16_t e_phnum;
            uint16_t e_shentsize;
            uint16_t e_shnum;
            uint16_t e_shstrndx;
        } ElfN_Ehdr;
```

The offset from the start of the struct to the start of `e_entry` is:

▷ 16 bytes of `e_ident[EI_NIDENT]`:

    #define EI_NIDENT 16

▷ 2 bytes of e_type

▷ 2 bytes of e_machine

▷ 4 bytes of e_version

    Offset = 16 + 2 + 2 + 4 = 24 = 0x18

e_entry is of type `ElfN_Addr`, in which `N` is either `32` or `64`. We are writing 32-bit operating system, in this case $N = 32$ and so `ElfN_Addr` is `Elf32_Addr`, which is 4 bytes long.

Example 8.5.1. With any program, such as this simple one:

hello.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("hello world!\n");
    return 0;
}
```

We can retrieve the entry address with a human-readable presentation using `readelf`:

```
$ gcc hello.c -o hello
$ readelf -h hello
```

Output

```
ELF Header:
   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
   .... output omitted ....
   Entry point address:               0x400430
   .... output omitted ....
```

Or in raw binary with `hd`:

```
$ hd hello | less
```

Output

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  30 04 40 00 00 00 00 00  |..>.....0.@.....|
.........
```

The offset `0x18` is the start of the least-significant byte of `e_entry`,

which is 0x30, followed by 04 40 00, together in reverse makes the address
0x00400430.

Now that we know where the position of the entry address in the ELF
header, it is easy to modify the bootloader made in section 7.6.2 to re-
trieve and jump to the address:

bootloader.asm

```asm
;*****************************************
; Bootloader.asm
; A Simple Bootloader
;*****************************************
bits 16
start: jmp boot


;; constant and variable definitions
msg db "Welcome to My Operating System!", 0ah, 0dh, 0h


boot:
  cli ; no interrupts
  cld ; all that we need to init


  mov  ax, 50h


  ;; set the buffer
  mov es, ax
  xor bx, bx


  mov al, 2     ; read 2 sector
  mov ch, 0 ; we are reading the second sector past us,
                             ; so its still on track
                                    0
  mov cl, 2     ; sector to read (The second sector)
  mov dh, 0     ; head number
  mov dl, 0     ; drive number. Remember Drive 0 is floppy
       drive.
```

```asm
  mov ah, 0x02    ; read floppy sector function
  int 0x13      ; call BIOS - Read the sector
  jmp [500h + 18h]    ; jump and execute the sector!


  hlt ; halt the system


  ; We have to be 512 bytes. Clear the rest of the bytes
      with 0
  times 510 - ($-$$) db 0
  dw 0xAA55     ; Boot Signiture
```

It is as simple as that! First, we load the operating system binary at
0x500, then we retrieve the entry address at the offset 0x18 from 0x500,
by first calculating the expression $500h + 18h = 518h$ to get the actual
in-memory address, then retrieve the content by dereference it.

The first part is done. For the next part, we need to build an ELF op-
erating system image for the bootloader to load. The first step is to cre-
ate a linker script:

main.lds

```
ENTRY(main);


PHDRS
{
  headers PT_PHDR FILEHDR PHDRS;
  code PT_LOAD;
}


SECTIONS
{
  .text 0x500: { *(.text) } :code
  .data : { *(.data) }
  .bss : { *(.bss) }
  /DISCARD/ : { *(.eh_frame) }
```

```
}
```

The script is straight-forward and remains almost the same as before. The only differences are:

▷ `main` are explicitly specified as the entry point by specifying `ENTRY(main)`.

▷ `.text` is explicitly specified with `0x500` as its *virtual memory address* since we load the operating system image at `0x500`.

After putting the script, we compile with `make` and it should work smoothly:

```
$ make clean; make
$ readelf -l build/os/os
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x500
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000500 0x00000500 0x00000500 0x00040 0x00040 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

All looks good, until we run it. We begin by starting the QEMU virtual machine:

```
$ make qemu
```

Then, start `gdb` and load the debug info (which is also in the same binary file) and set a breakpoint at `main`:

```
(gdb) symbol-file build/os/os
Reading symbols from build/os/os...done.
(gdb) b main
Breakpoint 2 at 0x500
```

Then we start the program:

```
(gdb) symbol-file build/os/os
Reading symbols from build/os/os...done.
(gdb) b main
Breakpoint 2 at 0x500
```

Keep the programming running until it stops at `main`:

```
(gdb) c
Continuing.
[ 0:7c00]
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[ 0: 500]
Breakpoint 2, main () at main.c:1
```

At this point, we switch the layout to the C source code instead of the registers:

```
(gdb) layout split
```

`layout split` creates a layout that consists of 3 smaller windows:

▷ Source window at the top.

▷ Assembly window in the middle.

▷ Command window at the bottom.

After the command, the layout should look like this:

Output

```
      ┌main.c────────────────────────────────────────────
B+> │1        void main(){}
    │2
    │3
    │4
    │5
    │6
    │7
    │8
    │9
    │10
    │11
    │12
    │13
    │14
    │15
    │16
      └───────────────────────────────────────────────────
```

```
B+> 0x500 <main>     jg      0x547
    0x502 <main+2>   dec     sp
    0x503 <main+3>   inc     si
    0x504 <main+4>   add     WORD PTR [bx+di],ax
    0x506            add     WORD PTR [bx+si],ax
    0x508            add     BYTE PTR [bx+si],al
    0x50a            add     BYTE PTR [bx+si],al
    0x50c            add     BYTE PTR [bx+si],al
    0x50e            add     BYTE PTR [bx+si],al
    0x510            add     al,BYTE PTR [bx+si]
    0x512            add     ax,WORD PTR [bx+si]
    0x514            add     WORD PTR [bx+si],ax
    0x516            add     BYTE PTR [bx+si],al
    0x518            add     BYTE PTR [di],al
    0x51a            add     BYTE PTR [bx+si],al
```

```
0x51c              xor     al,0x0
0x51e              add     BYTE PTR [bx+si],al
```

```
remote Thread 1 In: main                           L1    PC: 0x500
[f000:fff0] 0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00
(gdb) symbol-file build/os/os
Reading symbols from build/os/os...done.
(gdb) b main
Breakpoint 2 at 0x500: file main.c, line 1.
(gdb) c
Continuing.
[   0:7c00]
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0: 500]
Breakpoint 2, main () at main.c:1
(gdb) layout split
(gdb)
```

Something wrong is going on here. It is not the generated assembly
code for function call as it is known in section 4.9.5. It is definitely wrong,
verified with `objdump`:

```
$ objdump -D build/os/os | less
```

Output

```
/home/tuhdo/workspace/os/build/os/os:      file format elf32-i386
Disassembly of section .text:
00000500 <main>:
 500:   55                      push    %ebp
 501:   89 e5                   mov     %esp,%ebp
 503:   90                      nop
```

```
504:    5d                      pop     %ebp
505:    c3                      ret
.... remaining output omitted ....
```

The assembly code of `main` is completely different. This is why understanding assembly code and its relation to high-level languages are important. Without the knowledge, we would have used `gdb` as a simple source-level debugger without bother looking at the assembly code from the split layout. As a consequence, the true cause of the non-working code could never been discovered.

8.5.2 Debugging the memory layout

What is the reason for the incorrect Assembly code in `main` displayed by `gdb`? There can only be one cause: the bootloader jumped to the wrong addresses. But why was the address wrong? We made the `.text` section at address `0x500`, in which `main` code is in the first byte for executing, and instructed the bootloader to retrieve the address at the offset `0x18`, then jump to the entry address.

Then, it might be possible for the bootloader to load the operating system address at the wrong address. But then, we explicitly set the load address to `50h:00`, which is `0x500`, and so the correct address was used. After the bootloader loas the 2nd sector, the in-memory state should look like the figure 8.5.1:

Here is the problem: `0x500` is the start of the ELF header. The bootloader actually loads the 2nd sector, which stores the executable as a whole, to `0x500`. Clearly, `.text` section, where `main` resides, is far from `0x500`. Since the in-memory entry address of the executable binary is `0x500`, `.text` should be at $0x500 + 0x500 = 0xa00$. However, the entry address recorded in the ELF header remains `0x500` and as a result, the bootloader jumped there instead of `0xa00`. This is one of the issues that must be fixed.

The other issue is the mapping between debug info and the memory address. Because the debug info is compiled with the assumed offset `0x500` that is the start of `.text` section, but due to actual loading, the offset is pushed another `0x500` bytes, making the address actually is at `0xa00`.

Figure 8.5.1: Memory state after loading 2nd sector.

This memory mismatch renders the debug info useless.

Figure 8.5.2: Wrong symbol-memory mappings in debug info.

In summary, we have 2 problems to overcome:

▷ Fix the entry address to account for the extra offset when loading into memory.

▷ Fix the debug info to account for the extra offset when loading into memory.

First, we need to know the actual layout of the compiled executable binary:l

```
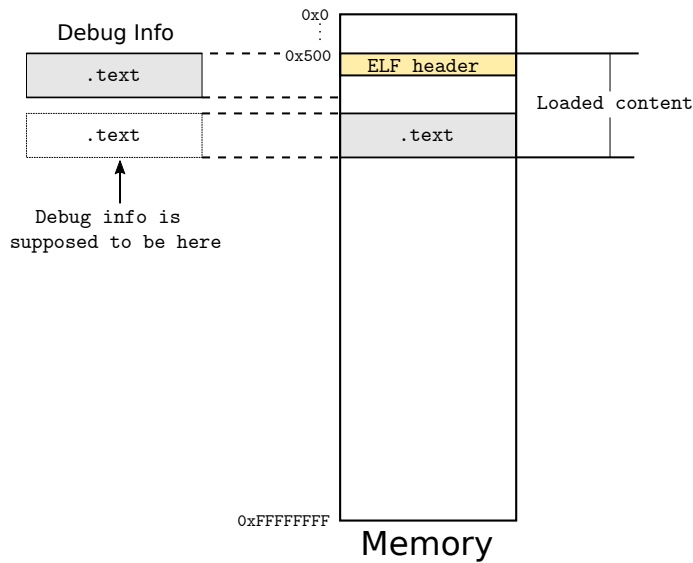$ readelf -l build/os/os
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x500
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000500 0x00000500 0x00000500 0x00040 0x00040 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
```

```
      00

      01      .text
```

Notice the `Offset` and the `VirtAddress` fields: both have the same value. This is problematic, as the entry address and the memory addresses in the debug info depend on `VirtAddr` field, but the `Offset` having the same value destroys the validity of `VirtAddr`[18] because it means that the real in-memory address will always be greater than the `VirtAddr`.

[18] The offset is the distance in bytes between the beginning of the file, the address 0, to the beginning address of a segment or a section.

If we try to adjust the virtual memory address of the `.text` section in the linker script `os.lds`, whatever value we set also sets the `Offset` to the same value, until we set it to some value equal or greater than `0x1074`:

Output

```
Elf file type is EXEC (Executable file)

Entry point 0x1074

There are 2 program headers, starting at offset 52

Program Headers:

  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align

  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R    0x4

  LOAD           0x000074 0x00001074 0x00001074 0x00006 0x00006 R E  0x1000

 Section to Segment mapping:

  Segment Sections...

   00

   01      .text
```

If we adjust the virtual address to `0x1073`, both the `Offset` and `VirtAddr` still share the same value:

Output

```
Elf file type is EXEC (Executable file)

Entry point 0x1073

There are 2 program headers, starting at offset 52

Program Headers:

  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align

  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R    0x4

  LOAD           0x001073 0x00001073 0x00001073 0x00006 0x00006 R E  0x1000

 Section to Segment mapping:
```

```
Segment Sections...
 00
 01     .text
```

The key to answer such phenonemon is in the `Align` field. The value
`0x1000` indicates that the offset address of the segment should be divisi-
ble by `0x1000`, or if the distance between segment is divisible by `0x1000`,
the linker removes such distance to save the binary size. We can do some
experiments to verify this claim[19]:

▷ By setting the virtual address of `.text` to `0x0` to `0x73` (in `os.lds`),
   the offset starts from `0x1000` to `0x1073`, accordingly. For example, by
   setting it to `0x0`:

Output
```
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R    0x4
  LOAD           0x001000 0x00000000 0x00000000 0x00006 0x00006 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

By default, if we do not specify any virtual address, the offset stays at
`0x1000` because `0x1000` is the perfect offset to satisfy the alignment
constraint. Any addition from `0x1` to `0x73` makes the segment mis-
aligned, but the linker keeps it anyway because it is told so.

▷ By setting the virtual address of `.text` to `0x74` (in `os.lds`):

Output
```
Elf file type is EXEC (Executable file)
Entry point 0x74
There are 2 program headers, starting at offset 52
```

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R    0x4
  LOAD           0x000074 0x00000074 0x00000074 0x00006 0x00006 R E 0x1000
 Section to Segment mapping:
 Segment Sections...
  00
  01     .text
```

PHDR is `0x74` bytes in size, so if `LOAD` starts at `0x1074`, the distance between the `PHDR` segment and `LOAD` segment is $0x1074 - 0x74 = 0x1000$ bytes. To save space, it removes that extra `0x1000` bytes.

▷ By setting the virtual address of `.text` to any value between `0x75` and `0x1073` (in `os.lds`), the offset takes the exact values specified, as can be seen in the case of setting to `0x1073` above.

▷ By setting the virtual address of `.text` to any value equal or greater than `0x1074`: it starts all over again at `0x74`, where the distance is equal to `0x1000` bytes.

Now we get a hint how to control the values of `Offset` and `VirtAddr` to produce a desired binary layout. What we need is to change the `Align` field to a value with smaller value for finer grain control. It might work out with a binary layout like this:

Output
```
Elf file type is EXEC (Executable file)
Entry point 0x600
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R    0x4
  LOAD           0x000100 0x00000600 0x00000600 0x00006 0x00006 R E 0x100
 Section to Segment mapping:
 Segment Sections...
  00
  01     .text
```

The binary will look like figure 8.5.3 in memory:

Figure 8.5.3: A good binary layout.

If we set the `Offset` field to `0x100` from the beginning of the file and the `VirtAddr` to `0x600`, when loading in memory, the actual memory of .text is `0x500 + 0x100 = 0x600`; `0x500` is the memory location where the bootloader loads into the physical memory and `0x100` is the offset from the end of ELF header to `.text`. The entry address and the debug info will then take the value `0x600` from the `VirtAddr` field above, which totally matches the actual physical layout. We can do it by changing `os.lds` as follow:

main.lds

```
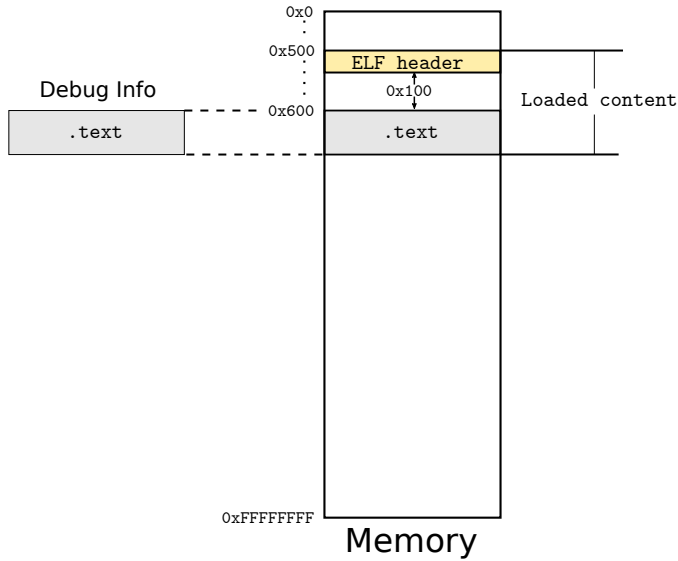ENTRY(main);


PHDRS
{
  headers PT_PHDR FILEHDR PHDRS;
  code PT_LOAD;
}


SECTIONS
{
  .text 0x600: ALIGN(0x100) { *(.text) } :code
```

```
  .data : { *(.data) }

  .bss : { *(.bss) }

  /DISCARD/ : { *(.eh_frame) }

}
```

The `ALIGN` keyword, as it implies, tells the linker to align a section, thus the segment containing it. However, to make the `ALIGN` keyword has any effect, automatic alignment must be disabled. According to `man ld`:

Output

```
    -n
    --nmagic
         Turn off page alignment of sections, and disable linking against shared
         libraries.  If the output format supports Unix style magic numbers, mark the
         output as "NMAGIC"
```

That is, by default, each section is aligned by an operating system page, which is 4096, or `0x1000` bytes in size. The `-n` or `-nmagic` option disables this behavior, which is needed. We amend the `ld` command used in `os/Makefile`:

os/Makefile

```
..... above content omitted ....

$(OS): $(OS_OBJS)

    ld -m elf_i386 -nmagic -Tos.lds $(OS_OBJS) -o $@
```

Finally, we also need to update the top-level Makefile to write more than one sector into the disk image for the operating system binary, as its size exceeds one sector:

```
 $ ls -l build/os/os

 -rwxrwxr-x 1 tuhdo tuhdo 9060 Feb 13 21:37 build/os/os
```

We update the rule so that the sectors are automatically calculated:

os/Makefile

```
..... above content omitted ....

bootdisk: bootloader os
```

```
dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
dd conv=notrunc if=$(BOOTLOADER) of=$(DISK_IMG) bs=512
    count=1 seek=0
dd conv=notrunc if=$(OS) of=$(DISK_IMG) bs=512 count=$$
    (($(shell stat --printf="%s" $(OS))/512)) seek=1
```

After updating the everything, recompiling the executable binary and we get the desired offset and virtual memory at `0x100` and `0x600`, respectively:

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x600
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000100 0x00000600 0x00000600 0x00006 0x00006 R E 0x100
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

8.5.3 Testing the new binary

First, we start the QEMU machine:

```
$ make qemu
```

In another terminal, we start `gdb`, loading the debug info and set a breakpoint at `main`:

```
$ gdb
```

The following output should be produced:

Output

```
---Type <return> to continue, or q <return> to quit---
[f000:fff0] 0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00
Breakpoint 2 at 0x600: file main.c, line 1.
```

Then, let gdb runs until it hits the main function, then we change to the split layout between source and assembly:

```
(gdb) layout split
```

The final terminal output should look like this:

Output

```
     ┌─main.c──────────────────────────────────────┐
B+> │ 1       void main(){}                        │
     │ 2                                            │
     │ 3                                            │
     │ 4                                            │
     │ 5                                            │
     │ 6                                            │
     │ 7                                            │
     │ 8                                            │
     │ 9                                            │
     │ 10                                           │
     │ 11                                           │
     │ 12                                           │
     │ 13                                           │
     │ 14                                           │
     │ 15                                           │
     │ 16                                           │
     └──────────────────────────────────────────────┘
B+> │ 0x600 <main>     push    bp
     │ 0x601 <main+1>   mov     bp,sp
     │ 0x603 <main+3>   nop
```

```
0x604 <main+4>  pop     bp
0x605 <main+5>  ret
0x606           aaa
0x607           add     BYTE PTR [bx+si],al
0x609           add     BYTE PTR [si],al
0x60b           add     BYTE PTR [bx+si],al
0x60d           add     BYTE PTR [bx+si],al
0x60f           add     BYTE PTR [si],al
0x611           add     ax,bp
0x613           push    ss
0x614           add     BYTE PTR [bx+si],al
0x616           or      al,0x67
0x618           adc     al,BYTE PTR [bx+si]
0x61a           add     BYTE PTR [bx+si+0x2],al
```

```
remote Thread 1 In: main                          L1    PC: 0x600
(gdb) c
Continuing.
[   0:7c00]
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0: 600]
Breakpoint 2, main () at main.c:1
(gdb) layout split
```

Now, the displayed assembly is the same as in objdump, except the registers are 16-bit ones. This is normal, as gdb is operating in 16-bit mode, while objdump displays code in 32-bit mode. To make sure, we verify the raw opcode by using x command:

```
(gdb) x/16xb 0x600
```

```
Output    0x600 <main>:    0x55     0x89     0xe5     0x90     0x5d     0xc3     0x37
                  0x00
          0x608:  0x00     0x00     0x04     0x00     0x00     0x00     0x00     0x00
```

From the assembly window, `main` stops at the address `0x605`. As such, the corresponding bytes from `0x600` to `0x605` are highlighted in red from the output of the command `x/16xb 0x600`. Then, the raw opcode from the `objdump` output:

```
$ objdump -z -M intel -S -D build/os/os | less
```

```
Output    build/os/os:      file format elf32-i386
          Disassembly of section .text:
          00000600 <main>:
          void main(){}
           600:   55                     push   ebp
           601:   89 e5                  mov    ebp,esp
           603:   90                     nop
           604:   5d                     pop    ebp
           605:   c3                     ret
          Disassembly of section .debug_info:
          ...... output omitted ......
```

Both raw opcode displayed by the two programs are the same. In this case, it proved that `gdb` correctly jumped to the address in `main` for a proper debugging. This is an extremely important milestone. Being able to debug in bare metal will help tremendously in writing an operating system, as a debugger allows a programmer to inspect the internal state of a running machine at each step to verify his code, step by step, to gradually build up a solid understanding. Some professional programmers do not like debuggers, but it is because they understand their domain deep enough to not need to rely on a debugger to verify their code. When encountering new domains, a debugger is indispensable learning tool because of its verifiability.

However, even with the aid of debugger, writing an operating system is still not a walk in the park. The debugger may give the access to the machine at one point in time, but it does not give the cause. To find out the root cause, is up to the ability of a programmer. Later in the book, we will learn how to use other debugging techniques, such as using QEMU logging facility to debug CPU exceptions.

Part III

Kernel Programming

9

x86 Descriptors

Basic operating system concepts

The first and foremost, OS manages hardware resources. It's easy to see the core features of an OS based on Von Neumann diagram:

CPU management: allows programs to share CPU for multitasking.

Memory management: allocates enough storage for programs to run.

Devices management: detects and communicates with different devices

Any OS should be good at the above fundamentals tasks.

Another important feature of an OS is to provide an software interface layer, that hides away hardware interfaces, to interface with applications that run on top of that OS. The benefits of such a layer:

▷ reusability: that is, the same software API can be reused across programs, thus simplifying software development process

▷ separation of concerns: bugs appear either in application programs, or in the OS; a programmer needs to isolate where the bugs are.

▷ simplify software development process: provides an easier to use software interface layer with a uniform access to hardware resources across

devices, instead of directly using the hardware interface of a particular device.

9.1.1 Hardware Abstraction Layer

There are so many hardware devices out there, so it's best to leave the hardware engineers how the devices talk to an OS. To achieve this goal, the OS only provides a set of agreed software interfaces between itself and the device driver writers and is called *Hardware Abstraction Layer*.

In C, this software interface is implemented through a structure function pointers.

[illustrate with Linux example]

9.1.2 System programming interface

System programming interfaces are standard interfaces that an OS provides application programs to use its services. For example, if a program wishes to read a file on disk, then it must call a function like *open()* and let the OS handle the details of talking to the hard disk for retrieving the file.

9.1.3 The need for an Operating System

In a way, OS is an overhead, but a necessary one, for a user to tell a computer what to do. When resources in a computer system (CPU, GPU, memory, hard drive...) became big and more complicated, it's tedious to manually manage all the resources.

Imagine we have to manually load programs on a computer with $3\,$GB of RAM. We would have to load programs at various fix addresses, and for each program a size must be manually calculated to avoid wasting memory resource, and enough for programs to not overriding each other.

Or, when we want to give computer input through the keyboard, without an OS, an application also has to carry code to facilitate the communication with keyboard hardware; each application then handles such keyboard communication on its own. Why should there be such duplications across applications for such standard feature? If you write an ac-

counting software, why should a programmer concern writing a keyboard driver, totally irrelevant to the problem domain?

That's why a crucial job of an OS is to hide the complexity of hardware devices, so a program is freed from the burden of maintaining its own code for hardware communication by having a standardized set of interfaces and thus, reduce potential bugs along with faster development time.

To write an OS effectively, a programmer need to understand well the underlying computer architecture that programmer are writing an OS for. The first reason is, many OS concepts are supported by the architecture e.g. the concepts of virtual memory are well supported by x86 architecture. If the underlying computer architecture is not well-understood, OS developers are doomed to reinvent it in your OS, and such software-implemented solutions run slower than the hardware version.

9.2   Drivers

Drivers are programs that enable an OS to communicate and use features of hardware devices. For example, a keyboard driver enables an OS to get input from keyboard; or a network driver allows a network card to send and receive data packets to and from the Internet.

If you only write application programs, you may wonder how can software control hardware devices? As mentioned in Chapter 2, through the hardware-software interface: by writing to a device's registers or to write to ports of a device, through the use of CPU's instructions.

9.3   Userspace and kernel space

Kernel space refers to the working environment of an OS that only the kernel can access. Kernel space includes the direct communication with hardware, or manipulate privileged memory regions (such as kernel code and data).

In contrast, *userspace* refers to less privileged processes that run above the OS, and is supervised by the OS. To access the kernel facility, user

program must go through the standardized system programming interfaces provided by the OS.

9.4   Memory Segment

9.5   Segment Descriptor

9.6   Types of Segment Descriptors

9.6.1 Code and Data descriptors

9.6.2 Task Descriptor

9.6.3 Interrupt Descriptor

9.7   Descriptor Scope

9.7.1 Global Descriptor

9.7.2 Local Descriptor

9.8   Segment Selector

9.9   Enhancement: Bootloader with descriptors

10

Process

Concepts

Process

10.2.1 Task

A *task* is a unit of work that an OS needs to do, similar to how human
have tasks to do daily. From a user point of view, a task for a computer
to do can be web browsing, document editing, gaming, sending and receiv-
ing emails, etc. Since a CPU can only execute sequentially, one instruc-
tion after another (fetching from main memory), there must be some way
to do many meaningful tasks at once. For that reason, the computer must
share the resources e.g. registers, stack, memory, etc, between tasks, since
we have many tasks but single and limited resources.

10.2.2 Process

Process is a data structure that keeps track of the execution state of a
task. Task is a general concept, and process is the implementation of a
task. In a general-purpose OS, a task is usually a program. For exam-
ple, when you run Firefox, a process structure is created to keep track
of where the stack and the heap allocated for firefox are, where Firefox's

code area is and which instruction EIP is holding to execute next, etc. The typical process structure looks like this:

[insert process image]

Process is a virtual computer, but much more primitive than the virtual machine in virtualization software like Virtual Box, and that's a good thing. Imagine having to run a full-fledged virtual machine for every task; how wasteful of machine resources that would be.. In the view of a running process, its code executes as if it runs directly on hardware. Each process has its own set of register values, which are kept tracked by the OS, and its own contiguous virtual memory space (which is discontiguous in actual physical memory). The code in a process is given virtual memory addresses to read and write from.

[illustrate: - a process looks like a mini Von Neumann - with contiguous memory, each with a color; each cell of a process mapped to distant memory cell in physical memory]

A process can run so much until the OS tells it to temporary stop for other tasks to use the hardware resources. The suspended process can then wait until further notice from the OS. This whole switching process is so fast that a computer user think his computer actually runs tasks in parallel. The program that does the switching between tasks is called a *scheduler*.

10.2.3 Scheduler

An OS needs to perform a wide range of different functionalities, e.g. web browsing, document editing, gaming, etc. A *scheduler* decides which tasks got to run before the others and, for how long, in an efficient manner. Scheduler enables your computer to become a *time sharing system*, because tasks share CPU execution time and no one process can monopolize the CPU (in practice, it still happens regularly). Without a scheduler, only a single task can be performed at a time.

10.2.4 Context switch

When a process is prepared to be switched out for another process to take its place, certain hardware resources i.e. current open files, current regis-

ter values, etc. must be backed up to later resume that process's execution.

10.2.5 Priority

Priority is an important metric for OS to decide which task is scheduled to run before the others to allocate appropriate CPU execution time for each task.

10.2.6 Preemptive vs Non-preemptive

A *preemptive* OS can interrupt an executing process and switch to another process.

A *non-preemtive* OS, a task runs until its completion.

10.2.7 Process states

State is a particular condition of a process, triggered by an action from the scheduler. A process goes through various states during its life cycle. A process typically has these states:

Run indicating CPU is executing code in this process.

Sleep (or Suspended): indicating CPU is executing some process else.

Destroyed: process is done and waiting to be destroyed completely.

10.2.8 procfs

10.3   Threads

Threads are units of work inside a process that shares the execution environment. A process creates a whole new execution environment with code of its own:

[illustration between process and thread, with each process is a big rectangle box and threads nested boxes point to different code region]

Instead of creating a completely new process structure in memory, OS simply let the thread uses some of the resources of the parent process that created it. A thread has its own registers, program counter, stack

pointer, and its own call stack. Everything else is shared between the threads, such as an address space, heap, static data, and code segments, and file descriptors. Because thread simply reuses existing resources and involve no context switching, it is much faster to create and switch between processes.

However, note that the above scheme is just an implementation of thread concept. You can completely treat thread the same as process (hence you can call all processes threads and vice versa). Or you can just back up some resources, whlie leaving some resources shared. It's up to the OS designer to distinguish between threads and processes. Threads are usually implemented as a component of a process.

On Linux, a thread is simply a process that shares resources with its parent process; for that reason, a Linux thread is also called *light-weight process*. Or put it another way, a thread in Linux is merely an implementation of a single-threaded process that execute its main program code. A multi-threaded program in Linux is just a process with shared with its single-threaded children processes, each points to different code region of its parent process.

[TODO: turn the above table into a diagram]

On Windows, threads and processes are two separated entities, so the above description for Linux does not apply. However, the general idea: a thread shares the execution environment, holds.

10.4   Task: x86 concept of a process

10.5   Task Data Structure

10.5.1 Task State Segment

10.5.2 Task Descriptor

10.6   Process Implementation

10.6.1 Requirements

10.6.2 Major Plan

10.6.3 Stage 1: Switch to a task from bootloader

10.6.4 Stage 2: Switch to a task with one function from kernel

10.6.5 Stage 3: Switch to a task with many functions from kernel

To implement the concept of a process, a kernel needs to be able to save and restore its machine states for different tasks.

Description [Describe task switching mechanism involving LDT and GDT]

  qasdfasdf asd

Constraints

Design

Implementation plan

10.7   Milestone: Code Refactor

11

Interrupt

12

Memory management

12.0.1 Address Space

Address space is the set of all addressable memory locations. There are
2 types of address spaces in physical memory address:

▷ One for memory:

▷ One for I/O:

Each process has its own address space to do whatever it wants, as long
as the physical memory is not exhausted. This address space is called
virtual memory.

12.0.2 Virtual Memory

Physical memory is a contagious memory locations that has a simple map-
ping between a physical memory address and its corresponding location
in memory, decoded by memory controller. On the other hand, *virtual
memory* does not have direct mapping between a memory address and
the corresponding physical memory location, even though it appears con-
tagious from the view of an userspace program. Instead, virtual memory
address is translated by OS into an actual physical memory address. For
that reason, even addresses appear next to each other in virtual memory
space, they are scattered through out the physical memory.

Why virtual memory is needed? Because virtual memory reduces the complexity of programming, by giving each program an illusion that it has its own separate "physical" memory to work with. Without virtual memory, programs must know and agree with each other their own memory regions to not accidentally destroy each other.

[illustration a world without virtual memory]

Virtual memory also enables a more secured OS, as application programs cannot manipulate main memory directly, so malicious programs won't cause havocs by destroying main memory and possibly hardware devices, by gaining access to hardware I/O ports.

Another benefit is that virtual memory can extend beyond physical memory, by storing its data to hard disk. By swapping some of unused memory (i.e. inactive memory of a sleeping process), the system gains some free memory to continue running, so no data is destroyed. Otherwise, the OS is forced to kill a random user process to free up some memory, and you may lose unsaved work that belongs to the killed process. However, this process can significantly slow down the whole system because of Von Neumann bottleneck. In the old days, when memory was scarce, it was useful.

13

File System

File system is a mechanism on how raw bytes in a storage device can be meaningfully managed. That is, a group of bytes at specific locations in a storage device can be allocated for a purpose e.g. storing raw ASCII document, and later the exact chunks of bytes can be retrieved correctly. File system manages many such groups of bytes. It's helpful to think a file system as a database that maps between high level information and specific locations in a hard disk, similar to how business information is mapped to a specific row in a table. The high level information that is relevant to a file system is organized as *files* and *directories*.

[illustration between a file system and a database table to see how they are similar]

File is an entity that includes two components: metadata and the actual raw data. *Metadata* is the information describes the properties of the raw data associated with the file; raw data are real content of a file. *Directory* is a file that holds a group of files and also child directories. Together, they create a file hierarchy system as commonly seen in Windows or Linux.

13.0.1 Example: Ex2 filesystem

Index

Bibliography

G. H. Hardy. *A Mathematician's Apology*, chapter 10, page 13. University of Alberta Mathematical Sciences Society, 2005.

Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel, 2016b.

Benjamin L. Kovitz. *Practical Software Requirements*, chapter 3, page 53. Manning, 1999.

Charles Sanders Peirce. *Collected Papers v. 4*, chapter A Boolean Algebra with One Constant. 1933.

John F. Wakerly. *Digital Design: Principles and Practices*, chapter 3, page 86. Prentice Hall, 1999.