OS Book

Neil Klingensmith

December 10, 2024

Contents

Appendices

Chapter 1

Introduction

1.1 What is an Operating System?

As we will see throughout this book, computer hardware is complicated and messy. Most programmers would not like to interact directly with the hardware because of all its idiosynchricies. And since every CPU is different, porting software from one platform to another would be nightmarish without some kind of intermediate layer between the application and the hardware.

In fact, in all but the simplest computers, there are many layers of abstraction between an application and the hardware platform that it runs on. Details like how much memory the computer has, which applications are allowed to use which portions of memory, what type of display is available, how the computer is connected to the internet, and many others are managed by these intermediate layers of software.

1.1.1 Managing Hardware with Abstraction

The Android mobile ecosystem is a great example of abstraction between application and hardware. Many mobile devices run the Android operating system, with a huge amount of diversity in hardware capability among them. On the top end of the spectrum live high-performance tablets, mostly with price points in the range of $1,000-$2,000. These devices have big screens, fast WiFi and cellular networking interfaces, large memories, and fast processors. Many of them have detatchable keyboards, styluses, and other Bluetooth accessories. Most high-end tablets also come with facial recognition cameras to authenticate their users without a password. On the low end of the spectrum are small, low-cost and lightweight phones, many of which cost less than $100. They have smaller screens, slower CPUs and network interfaces, less memory, and fewer connectivity options.

Across all of these hardware platforms, even a seemingly simple task of displaying a menu on the screen is implemented completely differently. Once the layout of the menu screen is set by the application software, objects like text, images, and buttons are rendered into a rasterized image in memory. In the rasterization process, each pixel's color is set to an RGB value which is stored in an array in memory. The dimensions of the rasterized screen array depend on the size of the screen, which is different on each device. Once this rasterized array is constructed in memory, it needs to be transmitted from the main CPU to the display unit, which usually contains a separate coprocessor and memory to hold the contents of the rasterized screen buffer. The interface between the main CPU and the display coprocessor is also different on each device.

In the Android operating system, there are many layers of software involved in rendering the display buffer and transmitting the rendered image to the screen. Some of those software modules are hardware-independent and common to all Android systems. These common modules live inside the operating system, and their purpose is mostly to make the application programmer's job easier by automating some common, repetitive tasks like rasterizing text. Those common tasks need to be done by nearly every app that runs on the device, so it makes sense to have one common piece of code within the operating system to serve every app. Doing so reduces bugs (app developers can't introduce bugs in software they don't write) and creates a common interface so application software is easy to write, even by unskilled developers.

Some of the software modules involved in screen rendering are hardware-dependent, such as the communication between the mobile device's main CPU and the display coprocessor. As we will discuss later in the book, we don't trust application developers with any kind of direct access to the system's hardware. Access to network interfaces, display controllers, power management, disks, and every other hardware resource is always mediated by the operating system. With direct access to the hardware, malicious or buggy code could do a lot of damage: erasing files from the disk, starving other applications of CPU time, attacking other computers on the network, shuting the machine down without warning, and any number of other kinds of problematic behavior.

1.1.2 Operating System Abstractions

So the operating system's role is essentially to serve as a trusted mediator between the application and the hardware. In doing so, it provides a lot of higher-level abstractions that free up the application from having to worry about details of the hardware. Android apps don't have to worry about how their screens are rendered, rasterized, and transmitted to the display—the OS does it for them. Apps don't have to worry about how data is organized into files and stored onto the device's SD card—the OS implements a file system that does it for them. And apps don't need to worry about sharing CPU time. The OS does that for them as well. In general, we have a few desired functions for an operating system:

- Allow multiple programs to run simultaneously on the CPU.
- Allocate memory to programs.
- Provide support for files and filesystems.
- Mediate access to displays, input devices, network interfaces, and other I/O devices.
- Provide a user interface with a consistent look and feel.

The common thread among all functions of an OS is that it provides an abstract view of the computer hardware to the applications. People often say that an operating system's job is to present a runtime environment to the computer's applications that appears as if each application was the only one using the computer. Most operating systems provide some version of this vision of solitary execution to their applications, but none of the popular OSes in widespread use fully embrace it.

Why would an application wish to run in an abstracted environment like this? In the early days of computing, programs ran in *batch mode*, in which the computer would run one program at a time from start to finish. The program would have complete access to all of the computer's resources, including its entire memory, its persistent storage[1], and all of its I/O devices. Programmers became accustomed to writing programs that had free rein of the computer, but there was a problem: it was not possible for two or more programs to coexist

[1] Computers in those days used tapes instead of disk drives for long-term storage.

```
top - 19:57:42 up 3 days,  6:39,  2 users,  load average: 0.10, 0.15, 0.14
Tasks: 323 total,   1 running, 320 sleeping,   2 stopped,   0 zombie
%Cpu(s): 0.7 us, 0.3 sy, 0.0 ni, 99.0 id,  0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem :  3891.1 total,    335.3 free,   2181.4 used,   1374.4 buff/cache
MiB Swap:     0.0 total,      0.0 free,      0.0 used.   1357.2 avail Mem

  PID USER  PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
46603 root  20   0       0      0      0 I  1.0  0.0   0:00.35 kworker/0:2+
46541 root  20   0       0      0      0 I  0.7  0.0   0:05.81 kworker/1:0+
46696 neil  20   0   10620   4008   3144 R  0.7  0.1   0:00.10 top
  553 root  rt   0  289324  27108   9076 S  0.3  0.7   0:58.28 multipathd
  640 root  20   0  241204   7024   5340 S  0.3  0.2  10:52.40 vmtoolsd
 1593 neil  20   0  312596   6356   5308 S  0.3  0.2   0:24.96 gvfs-afc-vo+
 1684 neil  20   0 4097088 191444  71664 S  0.3  4.8  75:33.33 gnome-shell
 2152 neil  20   0 4905840 551744 157796 S  0.3 13.8 115:13.77 firefox
27732 neil  20   0 2521696 178500  87048 S  0.3  4.5   3:51.30 Isolated We+
46455 root  20   0       0      0      0 I  0.3  0.0   0:18.64 kworker/0:0+
    1 root  20   0  167968  11588   6400 S  0.0  0.3   0:17.33 systemd
    2 root  20   0       0      0      0 S  0.0  0.0   0:00.18 kthreadd
    3 root   0 -20       0      0      0 I  0.0  0.0   0:00.00 rcu_gp
    4 root   0 -20       0      0      0 I  0.0  0.0   0:00.00 rcu_par_gp
    6 root   0 -20       0      0      0 I  0.0  0.0   0:00.00 kworker/0:0+
    9 root   0 -20       0      0      0 I  0.0  0.0   0:00.00 mm_percpu_wq
   10 root  20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_tasks_r+
   11 root  20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_tasks_t+
   12 root  20   0       0      0      0 S  0.0  0.0   0:30.62 ksoftirqd/0
   13 root  20   0       0      0      0 I  0.0  0.0   0:27.14 rcu_sched
```

Figure 1.1: The output of `top` showing many process running on the system.

on a computer at once. Operating systems were invented in large part to allow multiple programs to "time share" computers.

But by the time operating systems came into wider use, programmers were already accustomed to writing solitary programs that did not need to coexist with one another. So operating system programmers aimed to present a runtime environment that resembled the batch programming their users were comfortable with. Although this illusion of a completely isolated program has deteriorated over the years as programmers and operating systems have evolved, the legacy of the batch-mode programming model lives on in some respects. Isolated memory spaces—the concept that two programs sharing the CPU have no access to one another's code or variables—is a clear vestige of batch-mode programming.

Figure **??** shows the output of the Linux `top` program, which displays the computer's CPU and memory usage along with a sorted list of active processes. When this `top` snapshot was captured, there were 321 processes sharing the CPU, all of which were running under this illusion of isolation provided by the Linux operating system. Each process had its own memory space containing variables and code that only it could access. Each process also had the illusion that the CPU was under its complete control—there is no need for any of the processes to actively relenquish control of the CPU so other programs could run. Although not all 321 process could be actively executing simultaneously, the operating system provides the illusion of concurrency by allowing one to run for a while, then stopping that process and giving access to the CPU to a different process for a while. The Linux operating system is responsible for switching between actively running processes. From the user's perspective, all the processes appear to run simultaneously, even though only one or two of them can be active at a time[2].

In this book, we build a simple operating system that provides basic hardware virtualization: memory isolation, time sharing and file systems. We build this operating system on the Intel 386 PC platform. The PC platform is convenient because it is well-documented and widely available.

1.2 A History of x86 CPUs

Starting in the 1940s and continuing through about 1975, all computers were one-off designs. When a new computer was to be built, a team of 20 or so hardware engineers would spend three to five years laying out its specifications and designing its hardware. In these early days, logic components like AND gates and flip flops were built from discrete transistors wired together on printed circuit boards the size of a sheet of paper. A separate team of software engineers would write a custom operating system, compiler, and other tools specifically for the new design.

All this changed in the mid 1970s with the invention of the integrated circuit, which could pack multiple transistors on a single piece of Silicon. It was possible to fit an entire computer on a single chip and mass produce them at a fraction of the cost of mainframes and supercomputers. Integration as it was called was the key technological advancement that made personal computers economical.

In 1968, a small group of engineers left Fairchild Semiconductor to found Intel. At first they produced memory chips, but in 1971 they released the first fully integrated microprocessor, the 4004. While fairly useless by today's standards, the 4004 enabled miniturized electronics like calculators. It was soon succeed by the 8008 and 8080 the first 8-bit microprocessors.

In 1976, Intel released the 8086, the product that would lead them to market dominance in PCs and servers. The 8086 was a fairly versatile 16-bit computer that made sense as the central component for PCs and minicomputers. However, as we will see later, it has a nonintuitive programmer's model and memory structure that is still being used in modern PCs today.

One important problem that designers of the 8086 had to overcome was memory addressability in a 16-bit computer. 16-bit addresses only would allow the programmer to access $2^{16} = 65536$ bytes of RAM, which was considered too little to be useful at the time. Programmers expected to have at least one megabyte of addressable memory, but that would require 20 address bits.

The solution they devised, which even at the time was widely seen as a kludge, was to use segmented addressing in which complete 20-bit addresses would be formed as the sum of a segment register plus a 16-bit pointer. Although crude, programmers learned to work with segmented addressing in x86 processors.

A marketing campaign by Intel targeted at manufacturers of PCs, minicomputers, and electronics manufacturers resulted in wide adoption of the 8086 and its successors. Despite its shortcomings, the 8086 was a success in the marketplace. Intel's price and first-mover advantages, not the technical merits of their product, made them successful.

A few years later, in 1979, Motorola released a competitive CPU called the 68000, so named because of the number of transistors on its dye. Their design improved upon the 8086 in many ways. First, all registers in the 68000 were 32 bits long, making it possible to address significantly more memory without using segmentation. Second, the instruction set and programmer's model of the 68000 was much more intuitive than that of the 8086.

[2] Each CPU core can actively execute one process. A computer with four CPU cores can actively execute four processes simultaneously, even though many more processes may be running on the system. The remaining processes are waiting for the operating system to give them access to one of the cores.

What happened next—part cost saving measure and part logistics solution—ultimately shaped the PC and server markets for decades. IBM, planning the new release of a home computer, chose the 8086 to be the central component of its new product. This decision was made because of the lower cost and wider availability of the 8086 and its peripheral components relative to the 68000. Apple, which was in process of designing the first Macintosh, chose to work with the 68000 for performance reasons.

When IBM shipped its first PCs, it also published a full set of schematics inside the user manual. Apple, a much more secretive organization, published very little information about the hardware and software inside the Macintosh. This had two effects. First, the availability of documentation about hardware and software on the PC made it much easier for third-party developers to write software for it. Even though the Macintosh had a much sleeker user interface, the lack of third party software support through the 80s and early 90s nearly lead to the company's demise. Second, low-cost clones of the IBM PC—made by Compaq and others—were widely available. As a result, the cost of ownership of the IBM PC was much lower and its utility was much higher. Riding on the success of the IBM PC, Intel's 8086 and successors became the most popular CPUs in desktop computers. Motorola eventually sold off and shut down its semiconductor division, including the 68000.

This book discusses operating system concepts and implementation techniques, with a focus on implementation on 8086-family CPUs. In particular, we work directly with hardware and software details of ihe i386 variant—the first to offer full support for most modern operating system features. Later variants, including the Pentium family and the AMD64 architecture offer architectural enhancements that give us performance gains—at the expense of quite a bit of additional complexity for the programmer. But the core concepts can all be implemented on the i386 CPU.

1.3 The i386 Programmer's Model

Before we begin discussing core operating systems concepts, it is important to have some base understanding of the low-level programmer's model of our CPU. Many of the operating system features we will introduce later in the book—virtual memory, time sharing, etc—will require us to manipulate the CPU at the assembly level. Although most of our operating system will be programmed in C, we will need to make use of a few assembly language instructions, and even many of the features of the OS implemented in C are directly manipulating some low-level features of the CPU.

We will start by discussing the i386's 16-bit real mode, which is a compatibility mode that allows the processor to run programs and operating systems written for the older 8086 CPU. Most of the concepts and features of the 8086 such as interrupts, registers, and stack frames should be familiar. The 8086 family CPUs do present a bit of a curve ball in the way they manage memory, which we will also discuss in the following sections.

1.3.1 16-Bit Real Mode

Figure 1.2 shows the programmer's view of the 8086 registers, all of which are 16 bits long. The 8086 has two distinct types of registers: integer registers `AX`, `BX`, `CX`, `DX`, and pointer registers `SI` and `DI`. The integer registers can be used for local storage of variables, and the pointer registers store addresses in memory. The 8086 instruction pointer `IP` stores the address of the instruction that is currently being executed.

All 8086 assembly instructions operate on two operands: one source and one destination. **The destination operand is always on the left** in 8086 assembly. For example, to set the value of register `AX` to 5:

```
mov ax,5
```

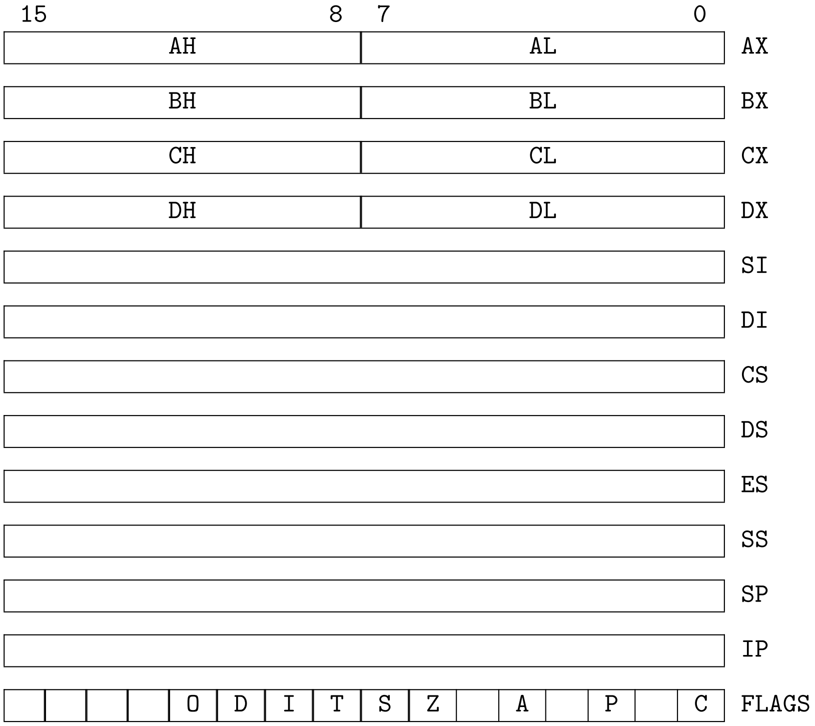


Figure 1.2: Registers of the 8086.

In this `mov` instruction, register `AX` is the destination, and `5` is the source. Putting the destination on the left is similar in notation to the assignment operator (`=`) in higher level languages, which would represent the same operation as follows:

```
ax = 5;
```

Example: Adding Two Numbers Together in 8086 Assembly The following example illustrates how to add two numbers together in 8086 assembly.

```
int i = 2; // in reg ax
int j = 3; // in reg bx
i += j;
```

```
[BITS 16]

boot: ; This file starts executing here.

    mov ax,2 ; ax := 2
    mov bx,3 ; bx := 3
    add ax,bx ; ax := ax + bx
    hlt

times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

In the above example, we use integer registers `AX` and `BX` to hold the values of `i` and `j` respectively. The `ADD` instruction adds the two together, storing the result in destination register `AX`. Source register `BX` is not modified by the `add`.

Assembling and Running Real-Mode Programs This book assumes you are using a Linux system to build and test your code. Like all the examples in this chapter, you can enter the assembly listing into a text editor and compile it with `nasm` to produce a runnable 16-bit 8086 program:

```
user@system ~ $ nasm -f elf32 -F dwarf mbr.asm -o mbr.o
user@system ~ $ ld -Ttext=0x7c00 -melf_i386 mbr.o -o mbr.elf
user@system ~ $ objcopy -O binary mbr.elf mbr.img
```

The first command assembles the source file `mbr.asm` to object file `mbr.o`, which is a binary object file that contains binary instruction encodings of the assembly instructions. The second command uses the `ld` linker to locate the object code to address `0x7C00`, creating output file `mbr.elf`. `mbr.elf` has all the same instruction encodings as `mbr.o` in addition to some extra metadata that says where the program should live in the computer's memory. The third command converts the ELF file to a flat binary file `mbr.img`, which can be booted directly by `qemu`. The flat binary file is a stripped down version of the ELF file with all the metadata removed. The flat binary contains only the instruction encodings we specified in the assembly source file. The flat binary file `mbr.img` is a mini disk image with only one sector–the master boot record–that can be loaded and emulated in 16-bit real mode using `qemu`:

```
user@system ~ $ qemu-system-i386 -hda mbr.img
```

The `qemu` command emulates your `mbr.img` file on a 386 computer. Your `mbr.img` file is treated as the main boot hard disk on the emulated machine. Of course it is a very small hard disk, with only one 512-byte sector.

Loops in 8086 Assembly

As with most assembly languages, building a loop in 8086 consists of three steps: initialization, loop body, and conditional branch instructions. Let's start with a simple example that clears the contents of a 10-byte buffer with zeros.

Example: Filling an Array with Zeros

```
char buf[10];
for(int k = 0; k < 10; k++) {
    buf[k] = 0;
}
```

```
[BITS 16]
  lea di,buf // di points to buf
  mov bx,0 // bx is index reg
loop:
  mov byte [di,bx],0 // buf[bx] := 0
  add bx,1 // increment bx
  cmp bx,10 // bx < 10 ?
  jl loop // loop if bx < 10
  hlt
buf:
  db 10 dup 0xff // 10 bytes initialized to 0xff

times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

Instruction	Condition
`ja`	Jump if Above (Unsigned Greater Than)
`jae`	Jump if Above or Equal (Unsigned Greater or Equal)
`jb`	Jump if Below (Unsigned Less Than)
`jbe`	Jump if Below or Equal (Unsigned Less or Equal)
`jl`	Jump if Less Than (signed)
`jle`	Jump if Less or Equal (signed)
`jg`	Jump if Greater Than (signed)
`jge`	Jump if Greater or Equal (signed)
`je`	Jump if Equal
`jne`	Jump if Not Equal

Table 1.1: Commonly used 8086 jump instructions.

In the above example, we begin with a 10-byte buffer initialized with `0xff`. Lines 2 and 3 are the initialization in which we point the `DI` register to the beginning of the buffer and initialize the index register `BX` to zero. Line 4 has a label that represents the beginning of the body of our loop. On each round of the loop, we will jump to the `loop` label and execute the body of the loop. On lines 7 and 8 we have a compare and conditional jump. The compare instruction on line 7 checks to see if we are finished with all iterations of the loop by comparing the value in register `BX` to 10. The conditional jump on line 8 jumps back to the beginning of the loop if `BX` is less than 10. The most commonly used 8086 jump instructions are shown in Table 1.1.

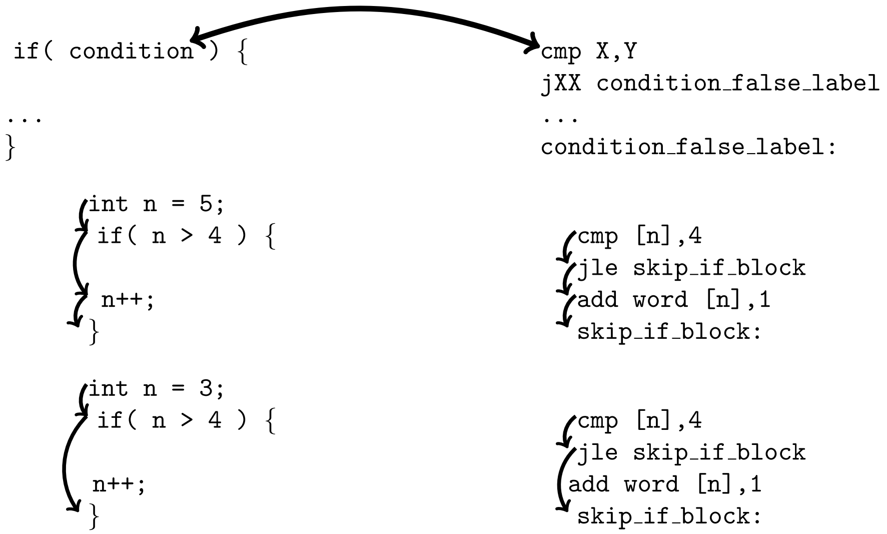
Example: `if` Blocks

```
int n = 5;
if(n > 4) {
    n++;
}
```

```
[BITS 16]
boot:
  cmp word [n],4 ; n > 4 ?
  jle skip_if_block ; Skip following instruction if n <= 4
  add word [n],1 ; n++
skip_if_block:
  hlt
n:
  dw 5
times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

In this example, we are using a conditional jump instruction to implement an `if` block. The canonical form for implementing an `if` block in assembly is shown in the code snippet below. First, the `if` condition is evaluated with a `cmp` instruction. After the `cmp`, we use a conditional jump to skip past the body of the `if` block if the condition evaluates to false. After the `jmp`, we write the body of the `if` block that will be executedd if the condition evaluates to true.

When we convert to assembly language, **we invert the sense of the `if` condition**. In our example, the code inside the `if` block will be executed if `n > 4`, and it will be skipped if `n <= 4`. In assembly language, we tell the computer to jump past the `if` block in the event that the `if` condition is false (i.e. if `n <= 4`). When the `if` condition is true (i.e. if `n > 4`), the computer will not take the jump, and it will continue executing code immediately after the jump instruction. Conditional jumps that are not taken have no effect on the program: instruction continue to execute in order after the untaken conditional jump.



```
if( condition ) {                    cmp X,Y
                                     jXX condition_false_label
...                                  ...
}                                    condition_false_label:

   int n = 5;
    if( n > 4 ) {                     cmp [n],4
                                      jle skip_if_block
    n++;                              add word [n],1
   }                                  skip_if_block:

   int n = 3;
    if( n > 4 ) {                     cmp [n],4
                                      jle skip_if_block
    n++;                              add word [n],1
   }                                  skip_if_block:
```

The 8086 Stack and Calling Conventions

8086 systems use the stack to pass parameters to functions and store local variables. The `SP` register points to the top of the stack, which grows downward in memory (toward lower memory addresses) as new items are pushed. The `push` and `pop` instructions can be used to push and pop data to the stack. The `call` and `ret` instructions are used for function calls and returns: `call` pushes the return address to the stack before jumping to a function, and `ret` pops the return address into the instruction pointer (`IP`).

Example: Calling Functions In 8086, we use the `call` instruction to call functions. `call` pushes the return address onto the stack and jumps to the specified function. When a function is finished executing, it uses the `ret` instruction to return. `ret` pops the return address off the stack into the `IP`. The example below assumes that `char *s` is passed by the caller in the `SI` register, and the result is returned in `AX`.

```
int strlen(char *s) {
  int k = 0;

  while(s[k] != '\0') {
    k++;
  }
  return k;
}
```

Passing the parameter to `strlen` works well in this example, but what about functions that take more than one or two parameters? The 8086 only has four integer registers and two pointer registers, leaving a total of at most six registers for us to use at one time. And some of those will be taken up by local variables and scratch storage. Clearly we need to find a different place for local temporary storage. Main memory is the obvious choice. Next, we discuss a data structure called a *stack frame* that can be used for organizing local storage in memory.

```
[BITS 16]
boot:
  lea si,string ; Get addr of string in SI
  call strlen ; Call strlen function
  hlt ; Halt CPU
strlen:
  mov bx,-1 ; Initialize bx
loop:
  add bx,1 ; increment bx
  cmp [si,bx],0 ; check for NULL term
  jne loop ; loop
  mov ax,bx ; put ret val in AX
  ret
str:
  db 'this is a string',0 ; The string we pass to strlen

times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

Listing 1: A simple implementation of `strlen` in 8086 assembly.

Stack Frames In addition to tracking a function's return address, functions also allocate their local variables on the stack. Like most processors, the 8086 provides some primitave instructions for managing data on its stack: the `push` instruction adds a new element to the top of the stack and `pop` removes an element from the top of the stack. The CPU's stack pointer (`SP`) holds the address in main memory of the last element that was added to the stack.

Let's rewrite the strlen function, this time using the stack to pass parameters from `boot` to `strlen`. In `boot`, just before we call `strlen`, we will push the address of `string` onto the stack instead of placing it in `SI`. Then, in `strlen`, we will retrieve the address of the string from the stack, placing it into the `SI` register before we begin computing the string's length. Other than how we communicate `strlen`'s parameter from the caller to the callee, everything else is the same. We still use the `SI` register to hold the address of the string inside `strlen`, and we still use `BX` to index into the string.

The function call in this example is simple: one line 3, we push the parameter we want to pass to `strlen` onto the stack. On line 4 we call `strlen`. Then, on line 5 after `strlen` returns, we remove the parameter that we pushed from the stack. The pointer argument that we pass to `strlen` is 2 bytes long, so we add 2 to the `SP` to remove it from the stack. If we don't remove parameters from the stack after function calls, the stack will just keep growing, taking up more and more memory.

Inside `strlen`, we need to get the function's parameter from the stack into `SI` before we begin computing the string's length. But the 8086 CPU doesn't allow us to use the `SP` register directly to read from the stack. We need to use a different register instead. On line 9, we copy the address from the `SP` into the `BP` register, which is used to read values from the stack. Then, on line 10, we read the value of the parameter from the stack into `SI`.

	`str`	2 bytes
`SP`,`BP` →	Return Address	2 bytes

Figure 1.3: Stack frame for `strlen` in Listing 2.

Figure 1.3 shows `strlen`'s stack frame. Inside of `strlen`, the `SP` and `BP` both point to the same place on the stack because on Line 9 we copy the value of `SP` to the `BP`. That slot on the stack contains the return address to the caller, which was pushed by the `call` instruction. Just above the return address on the stack is the parameter we passed to `strlen`. That

```
 1 [BITS 16]
 2 boot:
 3   push str ; Push the address of string onto the stack
 4   call strlen ; Call strlen function
 5   add sp,2 ; Remove the address of string from the stack
 6   hlt ; Halt CPU
 7 strlen:
 8   mov bx,-1 ; Initialize bx
 9   mov bp,sp ; Copy SP to BP
10   mov si,[bp+2] ; Read the parameter from the stack
11 loop:
12   add bx,1 ; increment bx
13   cmp byte [si,bx],0 ; check for NULL term
14   jne loop ; loop
15   mov ax,bx ; put ret val in AX
16   ret
17 str:
18   db 'this is a string',0 ; The string we pass to strlen
19 times 510-($-$$) db 0 ; Skip to end of boot sector
20 db 0x55 ; Magic Numbers
21 db 0xaa ; To make disk bootable
```

Listing 2: XXX

parameter lives at SP+2 because the return address is 2 bytes long.

The code in Listing 2 has one problem that needs to be fixed: the strlen function overwrites the BP, SI and BX registers on lines 8, 9 and 10 without saving their values. If there were important values in BP, SI or BX, they will be overwritten. We need to save the value of the both registers before we clobber them. That is simple enough to do by just pushing the BP, SI and BX onto the stack before we clobber their values. Then we can use all three registers in strlen to refer to values on the stack and index into our string. Immediately before strlen returns, we will pop BP, SI and BX off the stack, restoring their original values. On a side note, we also clobber whatever value was in AX, but since that register is used to store strlen's return value, there is no need to save its old value on the stack before we clobber it. The final implementation of the strlen function is shown in Listing 12, with an updated stack frame shown in Figure 1.4.

	str	2 bytes
	Return Address	2 bytes
BP →	Caller's BP	2 bytes
	Caller's SI	2 bytes
SP →	Caller's BX	2 bytes

Figure 1.4: Stack frame for strlen in Listing 12 with registers saved on the stack.

The process of saving copies of the registers before we overwrite them in a function is called the function's *prologue*. strlen's prologue is on lines 8-1 of Listing 12. Restoring the registers is called the function's *epilogue*, shown on lines 19-21 of Listing lst:strlenfinal.

To make our lives easier, the 8086 implements the pusha and popa instructions that respectively push all registers and pop all registers. Using these instructions saves both typing and thinking. It saves typing by reducing the number of push and pop instructions we need to write in the prologue and epilogue. It saves thinking by saving all registers on the stack, freeing us from accounting for which registers our function is using.

Example: Passing Parameters to Functions In the 8086, parameters are passed to a function on the stack. To call the strlen function in the previous example, we need to pass a pointer to a string on the stack.

```
[BITS 16]
boot:
  push str ; Push the address of string onto the stack
  call strlen ; Call strlen function
  add sp,2 ; Remove the address of string from the stack
  hlt ; Halt CPU
strlen:
  push bp ; Stash caller’s BP on the stack
  mov bp,sp ; Copy SP to BP
  push si ; Stash caller’s SI on the stack
  push bx ; Stash caller’s BX on the stack
  mov bx,-1 ; Initialize bx
  mov si,[bp+4] ; Read the parameter from the stack
loop:
  add bx,1 ; increment bx
  cmp byte [si,bx],0 ; check for NULL term
  jne loop ; loop
  mov ax,bx ; put ret val in AX
  pop bx ; Restore BX, SI, and BP before return
  pop si
  pop bp
  ret
str:
  db ’this is a string’,0 ; 10 bytes initialized to 0xff
times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

Listing 3: XXX

Local Variables

The stack allows us to stash and restore the contents of all the registers, freeing the registers up for local storage within a function without needing to worry about overwriting important values. But what about functions with lots of local variables? The 8086 only provides six registers—four integer registers and two pointer registers—enough for trivial functions like `strlen`, but lots of functions need more than six local variables.

There is also a way to use the stack for local variable storage, which allows us to have more local variables than registers. Local variables are customarily allocated on a function’s stack frame. Consider the `max` function below, which finds the maximum value in an integer array:

```
int max(int *buf, unsigned int len) {
    int maxval = -32767;
    int k;
    for(k = 0; k < len; k++) {
        if(buf[k] > maxval) {
            maxval = buf[k];
        }
    }
    return maxval;
}
```

Listing 4: XXX

This function has two local variables: `maxval` and `k` along with two parameters passed by the caller (buf and len). Although this function is not at risk of exhausting the 8086’s registers, we can use it to demonstrate how to allocate local variables on the stack. We will allocate the `maxval` variable on the stack, and we will keep the array index `k` in register `BX` as before. The stack frame for our implementation of `max()` is drawn in Figure 1.5. We can access the value of the `maxval` variable on the stack at location `BP-2`. The prologue we will use to create this stack frame is as follows:

1. `push bp`
2. `mov bp,sp`
3. `sub sp,2`
4. `push bx`

Instruction 1 saves the value of the CPU `BP` register on the stack so we can overwrite it. Instruction 2 copies the `SP` register to the `BP`. The `BP` register is a pointer from which all accesses to the stack frame are made—if we want to read or write the value of a local variable or a parameter passed to the function, we use the `BP` register. After instruction 2, the `SP` and `BP` registers both point to the same location on the stack. Instruction 3 subtracts 2 bytes from the `SP`, allocating space for the 2-byte integer `maxval`. Instruction 4 pushes the value of the `BX` register, which allows us to overwrite it in the function without losing its contents.

Instructions 3 and 4 cause the `SP` to change, but the `BP` continues to point to the same slot on the stack for the remainder of the function. This is a convenient feature of the `BP`: it never moves during the course of a function's execution. The `SP`, on the other hand, does move during a function's execution. For example, if we call a function, we will push the function's arguments onto the stack, causing the `SP` register to change. If we tried to refer to variables relative to the `SP`, their locations relative to the `SP` would change every time we push or pop something to the stack. Since we reference all of our variables relative to the `BP`, the variable's locations relative to the `BP` never change.

In summary, the procedure for function calls is as follows:

1. The caller pushes the function's arguments onto the stack in reverse order.
2. The caller uses a `call` instruction to jump to the function. The `call` instruction pushes the return address onto the stack before jumping.
3. The function being called (the *callee*) pushes the `BP` register and copies the `SP` to the `BP`. `BP` now points to the middle of the stack frame, just below the return address.
4. The callee creates space for its own local variables by subtracting from the `SP` the number of bytes needed to store its locals. For example, if callee has two local `int` variables, it would subtract $2 \times 2 = 4$ bytes from the `SP`.

Interrupts

There are many hardware events that occur in a system that need to be dealt with immediately. Urgent events are usually triggered by I/O devices like the keyboard, mouse, network interface, etc., and if they're not dealt with immediately, the computer will appear laggy and unresponsive to the user. Most CPUs offer a mechanism called *interrupts* to temporarily transfer control from the application that is currently running to a function called an *interrupt service routine* that can process data from the I/O device that caused the interrupt. An interrupt service routine is a regular function in every sense except that it is called by hardware, not by software. Figure 1.6 shows how the keyboard ISR is started in response to a keypress event.

```
[BITS 16]
boot:
  push 10 ; Push the length of buf onto the stack
  push buf ; Push the address of buf onto the stack
  call max ; Call max function
  add sp,4 ; Remove the address of buf and length of buf from the stack
  hlt ; Halt CPU
max:
  push bp ; Prologue
  mov bp,sp
  sub sp,2
  pusha
  mov word [bp-2],-32768 ; Initialize maxval
  mov bx,0 ; Initialize k = 0
  mov si,[bp+4] ; Get pointer to buf in SI
  shl word [bp+6],1 ; Multiply length of buf by 2 since each element of buf is 2 bytes long
max_loop:
  mov ax,[si+bx]
  cmp ax,[bp-2] ; Check if maxval < buf[k]
  jl not_greater ; If maxval > buf[k], do not update maxval
  mov [bp-2],ax ; maxval := buf[k]
not_greater:
  add bx,2 ; Add 2 to index register because each element of buf is 2 bytes long
  cmp bx,[bp+6] ; Check if we have reached end of buf
  jl max_loop ; If not, go back to beginning of loop
  popa ; Epilogue
  mov sp,bp
  pop bp
  ret
buf: dw 167,99,10000,-2598,31000,32000,-31000,-6000,2000,0 ; 10 an array with 10 integers
times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 0xaa ; Magic Numbers to make disk bootable
```

Listing 5: XXX

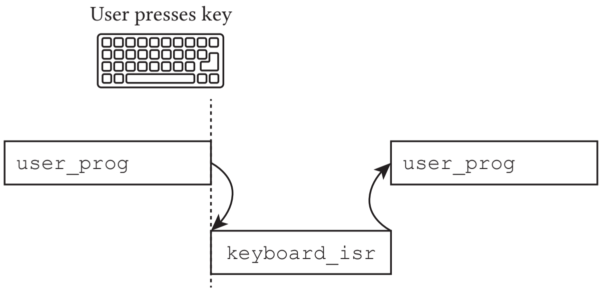


Figure 1.6: The hardware pauses the currently running program and executes an interrupt service routine in response to a keypress event.

Most functions are called by software during the normal course of execution of a program. That is, a programmer plans to call a function by saving important values from the registers into memory and pushing the function's parameters onto the stack.

Interrupt service routines are by definition unplanned. Since we don't know when an interrupt event will occur, we do not know when during a program's execution the CPU will invoke an interrupt service routine.

What if the application is executing some critical piece of code when the CPU takes an interrupt? We do not want the application's state—the register contents, the stack contents, etc—to be disrupted by the interrupt service routine. So when an interrupt occurs, the interrupt service routine must not modify the CPU state.

To avoid modifying the CPU state, the interrupt service routine pushes the contents of all registers onto the stack before responding to the hardware event. Preserving the CPU state is actually done in two phases: first, the CPU automatically pushes the return address and the `FLAGS` register onto the stack before calling the ISR. Then, in software, the ISR pushes the contents of all other registers onto the stack.

Since the 8086 has this weird segmented addressing where each 16-bit register can only represent 64k of the 1M address space, we need to save both the `CS` and `IP` registers to record the return address. Together, they take up a total of four bytes on the interrupt stack frame. Figure 1.7 shows a diagram of the 8086 interrupt stack frame.

	`unsigned int len`	`BP+6`
	`int *buf`	`BP+4`
	Return Address	`BP+2`
`BP` →	Caller's `BP`	`BP+0`
	`int maxval`	`BP-2`
`SP` →	Caller's `BX`	`BP-4`

Figure 1.5: Stack frame for the `max` function

Memory Segmentation

One problem the 8086 had was its ability to support large memories. Its 16-bit registers could only address 64kbytes of memory ($2^{16} = 65536$), which isn't really enough to hold large programs. The 8086 architects wanted to support 1 mbyte of memory, but for that they would need 20 address bits—4 more than they had.

To solve this problem, they created segment registers—extra registers that were appended to the 16-bit address registers to hold the high-order 4 bits of a 20-bit address. Table 1.2 illustrates how segment regsters are combined with address registers to form a physical address. The value in the segment register is shifted left by 4 bits and added to the address register. The sum becomes the 20-bit physical address that is presented to the memory system.

Segment registers are named `CS`, `DS`, `ES`, `SS` (see Table ??). The code segment register `CS` is appended to the instruction pointer `IP` when fetching instructions from memory. The data segment register `DS` is appended to `SI` and `DI` by default when

	`FLAGS`
	`CS`
`SP` →	`IP`

Figure 1.7: Interrupt stack frame created by the 8086 CPU.

fetching data with source index and destination index (`SI` and `DI`) registers. The stack segment register `SS` is appended to the stack pointer `SP` when reading and writing from the stack. The extra segment register `ES` is not used by default, but it can be used as an additional segment register.

The main problem with memory segmentation is address aliasing—many combinations of `SEGMENT:OFFSET` can refer to the same physical address. For example, segment `0x07C0` offset `0x0800` refers to the same address as `0x07E0` offset `0x0000` (both generate physical address `0x7E00`). Programmers of 16-bit 8086 systems had to be very careful when generating pointers!

Memory segmentation makes programming for the 8086 processor uniquely weird, and it is widely regarded by programmers as a blunder. But from a practical perspective, memory segmentation was a success because it allowed the 8086 to support large memories with a relatively simple 16-bit processor. The result was a cheap and versatile CPU (although difficult to program).

	A	0	0	0		DS
+		0	1	0	0	SI
	A	0	1	0	0	Effective Address

Table 1.2: Memory segmentation in the 8086.

Example: 8086 Hardcoded Terminal Output One of the cool things about the x86 PC platform is it allows us to write directly to the display with very little setup. If you're lazy, you can use the BIOS's built-in terminal driver (which we talk about later in the book). Or you can write characters directly to the screen at any location—no `printf` call needed. Writing to the screen is done using *video memory*.

Video memory is basically just a big array in the PC's memory map that is dedicated to video output. Any characters you write to that array is displayed on the screen. By default, the screen is divided into 80 columns × 24 rows of characters. Each character is represented in video memory as 2 bytes: one byte of ASCII character and one byte of color.

	Column 1 ↓		Column 2 ↓		Column 3 ↓		. . .	Column 80 ↓	
B8000	ascii	color	ascii	color	ascii	color	...	ascii	color
B8050	ascii	color	ascii	color	ascii	color	...	ascii	color
B80A0	ascii	color	ascii	color	ascii	color	...	ascii	color

The character at position $(1, 1)$ is located in the upper left corner of the screen, and row numbers increase toward the bottom of the screen. The character at position $(1, 1)$ lives at address `0xB8000`. The next character on the first row—at position $(1, 2)$ lives at address `0xB8002` and so on.

Let's write a program to draw the character `a` in the upper-left corner of the screen. To do that, we'll be writing two bytes starting at address `0xB8000`. The first byte to write will be the ASCII character (`a`), and the second byte will be the color to display. The color code for gray text on black background is `0x07`.

The only problem is that the address of video memory (`0xB8000`) is 20 bits long: bigger than the 16-bit registers in the 8086. We'll need to use segment registers to get the right address. In this code, we are going to use the Extra Segment (`ES`) register to store the beginning of video memory. We will put the value `0xB800` into register `ES`

	B	8	0	0		ES
+		0	0	0	0	SI
	B	8	0	0	0	Effective Address

Table 1.3: Using the `ES` register to address video memory.

and use `SI` as the offset into video memory (depicted in Table 1.3). When we write to `ES:[SI]`, the store will be done to address `0xB8000+SI`. The assembly code listing below shows how to do this in 8086 assembly.

```
[BITS 16]
boot:
    mov ax,0xb800 ; Load ES with base of video memory
    mov es,ax
    mov al,'a' ; ASCII code for the character to display
    mov ah,0x07 ; Color code 7 = gray text on black background
    mov word es:[si],ax
    hlt
times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 0xaa ; Magic Numbers to make disk bootable
```

Listing 6: Writing an `a` to the top left of the screen.

Even a simple operation like writing a single character to the screen is extremely clunky in 8086 assembly. Memory segmentation makes programming very painful. If only the CPU's registers were long enough to support pointers to any location in memory, we wouldn't need memory segmentaiton. Next we discuss the i386's protected mode, which does away with memory segmentation and makes programming much easier.

1.3.2 32-Bit Protected Mode

The x86 family's 32-bit protected mode is an extension of 16-bit real mode that provides wider registers and support for virtual memory. The programmer's model of the i386 CPU is shown in Figure 1.8. All of the 8086 registers are defined in the same way in the i386: `AX` is a 16-bit integer register with upper and lower halves `AH` and `AL`. The four integer registers have extended versions `EAX`, `EBX`, `ECX`, `EDX` in the i386, each 32 bits long. Assembly instruction opcodes (`mov`, `add`, `push`, etc) in protected mode are all the same as real (16-bit) mode. Since C compilers for 32-bit protected mode are widely available, we won't discuss the assembly language in much detail. But in order to write an operating system for the i386, you do need to understand a bit of the CPU's architecture.

Global Descriptor Tables

Memory segmentation lives on in a weird way on the i386's protected mode. But segments in the 32-bit protected mode aren't just simple integer offsets that get added to address references as they were in 16-bit real mode.

1.4 Exercises

Exercise 1.1 Write a program that clears the screen by writing the space character to every position on the terminal.

Exercise 1.2 Write a function that prints one character to an arbitrary (x, y) location on the screen. Your function should take three inputs: (1) the character to print, (2) the x location, and (3) the y location. *Hint: you may need to use the 8086* `mul` *instruction to compute the offset into video memory where the character should be written.*

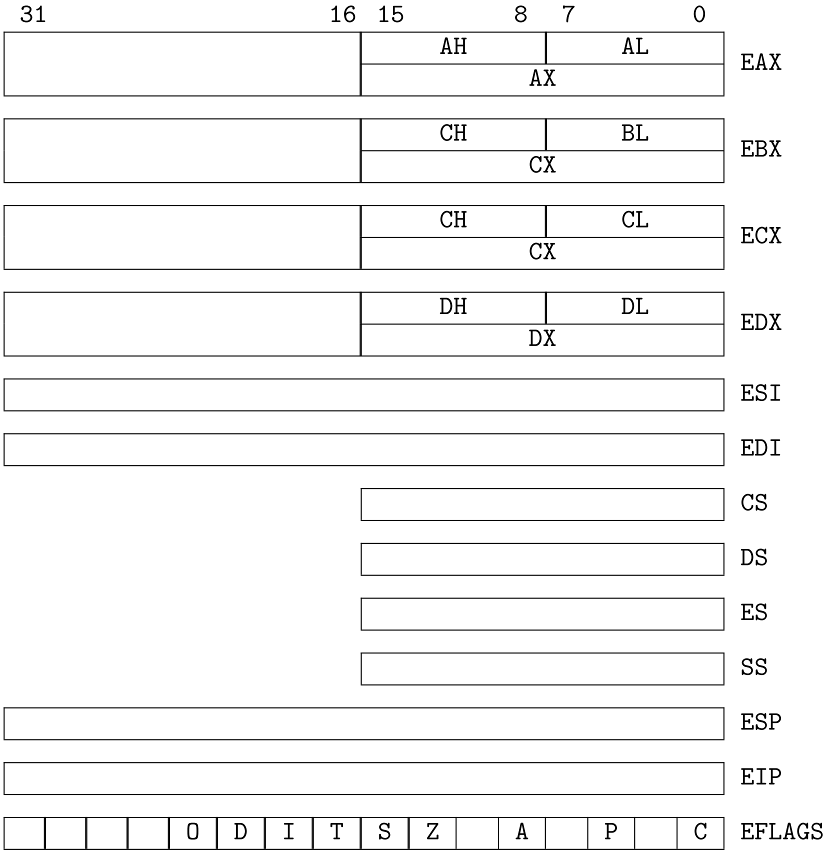


Figure 1.8: Registers of the i386.

Chapter 2

x86 Boot Process

The process by which a computer loads its operating system after it powers up is called booting—a reference to "pulling itself up by its bootstraps." After the computer powers on, its memory and peripherals are uninitialized. We need some software in place to set up the peripherals—particularly the hard disk—and load the operating system from the hard disk into memory.

The IBM PC has been produced in many variants: different CPUs, different disk controllers, different display adapters, all made by different manufacturers. We cannot expect every program that runs on the PC platform to have internal driver support for every possible hardware device—particularly not the simple low-level bootloader. Instead, PC manufacturers provide drivers for their hardware burned into ROM that all present a common interface to the user's software. These drivers, called the BIOS, allow programmers to access basic hardware features without integrating complex driver functionality into their programs.

The BIOS is also responsible for reading the bootloader from the first sector of the hard drive and passing control to it.

2.1 BIOS

The bootloader and other user code acces BIOS functions using software interrupt instructions that are handled by the BIOS. Applications call the BIOS by placing commands into the CPU registers (`AX`, `BX`, etc) and then executing n `INT` instruction, which causes control to jump to BIOS code. The BIOS reads the commands in the CPU registers and executes the corresponding function.

The secret behind 8086 variant's enduring success in the marketplace is compatibility. Even today, a brand-new Intel Core i7 system with all its modern features can run programs and operating systems that were written for IBM PCs manufactured in 1982. This might not seem impressive, but it is extremely valuable for businesses. Companies who invest in new PC-based hardware have never had to worry that hardware upgrades would make their software obsolete. They can always invest in new hardware and be certain that their existing software will continue to work. But how can Intel CPUs continue to support old software as the hardware architecture evolves over time? The BIOS is an important part of that compatability.

On powerup, all x86 CPU variants—even modern 64-bit variants—begin executing code in 16-bit real mode starting at address `0xFFFF0`. This is where the manufacturer-provided BIOS lives. 16-bit real mode is a compatability mode that is supported in modern x86 CPUs to ensure that older software will continue to run on new machines. The instruction set architecture of real mode is the same as the original 8086 CPU. This is the instruction

Address Region	Contents
0x00000 - 0x003FF	Real Mode Interrupt Vector Table
0x00400 - 0x004FF	BIOS Data Area
0x00500 - 0x07BFF	Unused
0x07C00 - 0x07DFF	Bootloader
0x07E00 - 0x9FFFF	Unused
0xA0000 - 0xBFFFF	Video RAM (VRAM) Memory
0xB0000 - 0xB7777	Monochrome Video Memory
0xB8000 - 0xBFFFF	Color Video Memory
0xC0000 - 0xC7FFF	Video ROM BIOS
0xC8000 - 0xEFFFF	BIOS Shadow Area
0xF0000 - 0xFFFFF	System BIOS

Table 2.1: BIOS memory map.

set that the BIOS and bootloader must use to load the operating system.

The BIOS is burned into ROM on the motherboard, and its first instruction lives at address `0xFFFF0` (see Table 2.1 for the BIOS's memory map). On powerup, the BIOS initializes the hardware and the DRAM controller. Then it loads the first 512-byte sector from the boot disk into RAM at address `0x7C00`. This sector is called the master boot record, and it is the first piece of user-writable code that runs.

2.2 Master Boot Record

The Master Boot Record (MBR) contains two critical components for the boot process: the bootloader and the partition table. The bootloader is responsible for loading the operating system from the disk into memory. The partition table tells us where on disk we can find the filesystems. Filesystems and partitions are discussed in Chapter 5, and we will focus on the the bootloader here.

The MBR is small—only 512 bytes. This doesn't leave much room to write a complicated bootloader program. Pretty much the only thing it can do is to find a real program that we actually want to run somewhere else on the boot disk and load that program into memory. But actually the MBR is so small that there isn't really even enough space to implement a proper hard disk driver, which of course is needed to read data from the disk. Fortunately, there is a disk driver available inside the BIOS—the MBR just needs to call that driver with the right parameters to load a real program from the disk. The MBR bootloader is a small program that calls the BIOS to load a larger second-stage bootloader from the hard drive.

2.2.1 BIOS Driver Interface

The BIOS supports low-level drivers that can be called by programs to access hardware. The interface between applications and the BIOS happens through software interrupts—assembly language instructions that cause interrupts to occur. When the CPU encounters an `INT` instruction, it stops executing the application code and pushes the `IP` and `FLAGS` registers onto the stack. the CPU then reads the address of the corresponding interrupt handler from the interrupt vector table and jumps to that address. The interrupt vector table and corresponding interrupt handlers are installed at boot time by the BIOS, so the handler is part of the BIOS code.

Each software interrupt handler supported by the BIOS handles a different hardware

Offset from Start of Sector	Memory Load Address	Contents
`0x000`	`0x7c00`	Bootloader code
`0x1be`	`0x7dbe`	Partition entry 1
`0x1ce`	`0x7dce`	Partition entry 2
`0x1de`	`0x7dde`	Partition entry 3
`0x1ee`	`0x7dee`	Partition entry 4
`0x1fe`	`0x7dfe`	Boot signature `0x55`
`0x1ff`	`0x7dff`	Boot signature `0xaa`

Table 2.2: Layout of the 512-byte Master Boot Record

```
user@system ~ $ nasm -f elf32 -F dwarf mbr.asm -o mbr.o
user@system ~ $ ld -Ttext=0x7c00 -melf_i386 mbr.o -o mbr.elf
user@system ~ $ objcopy -O binary mbr.elf mbr.img
```

driver. Generally, before the application executes a software interrupt, it will place some command code along with additional parameters in the CPU registers that tell the BIOS what to do. If the application needs to read data from a disk, for example, it will populate the CPU registers with information about where on the disk it wants to read and how much data needs to be read. When the BIOS finishes executing its interrupt handler, it uses an `RETI` instruction to pop the `FLAGS` register and return address of the stack and return to the application.

Below we discuss a few useful BIOS functions and give example code for calling them. For a more complete list of all BIOS functions, search Google for *Ralph Brown's Interrupt List*.

Printing Characters to the Screen

To print a character to the terminal, you have to pass a bit of information to the BIOS. The operation code `0x0E` in register AH tells the BIOS that we want to print a character. The ASCII code for the character to print goes in register AL. Registers BH and BL get the BIOS page number and foreground color respectively. Listing 7 shows a complete 16-bit real mode assembly code to create an MBR image that prints one `a` to the screen.

The assembly file in Listing 7 can be compiled with the commands below[1].

```
user@system ~ $ qemu-system-i386 -hda mbr.img

SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCAF60+06F0AF60 CA00



Booting from Hard Disk...
a
```

[1] A complete explanation of these compilation commands is given in §1.3.1

Register	Meaning
AH	Command Code 0x0E
AL	Character to write
BH	Page number (usually 0)
BL	Foreground color (7 for gray text)

Table 2.3: Parameters passed to the BIOS print command INT 0x10

```
[BITS 16]

boot: ; This file starts executing here.
    mov al,’a’
    mov ah, $0xe
    mov bh,0 ; Page 0
    mov bl,7 ; Foreground color 7 (gray)
    int 0x10 ; Call the BIOS, print char in AL
done:
    jmp done ; Spin loop after character prints

times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

Listing 7: Assembly code to create a master boot record image that prints the character ’a’ to the screen.

Getting a Keystroke from the Keyboard

Probably the simplest BIOS call is requesting a keystroke from the keyboard. This call just hangs until the user presses a key, then it returns with the ASCII code for the keypress in register AL. To call the Get Keystroke function in the BIOS, just put the value 0 in register AH then execute an INT 0x16 instruction:

```
[BITS 16]

boot: ; This file starts executing here.

    mov ah,0
    int 0x16  ; This instruction will hang until you press a key.
              ; On key press, BIOS returns with AL = ASCII code
    hlt

times 510-($-$$) db 0 ; Skip to end of boot sector
db 0x55 ; Magic Numbers
db 0xaa ; To make disk bootable
```

You can compile and run the program above using nasm and qemu as discussed in Section 1.3.1. This program will not make any kind of visual display because we haven’t told it to print anything. In the next example, we discuss how to print characters to the screen by calling the BIOS.

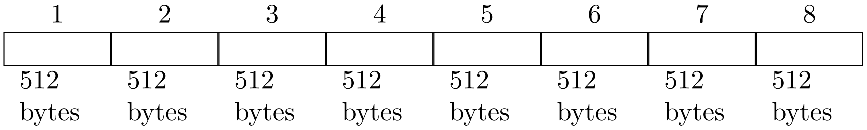


Figure 2.2: Layout of sectors on a disk.

Reading from the Disk

Before we talk about how to read from the disk, we need to discuss how data is organized on disks.

Sectors, Cylinders and Heads You're probably used to storing data on disks in files, but those are higher level abstractions provided by the operating system. At the most basic level, a disk is a *block device* that allows us to store information in blocks of 512 bytes. That's it.

Figure **??** shows the layout of 512 byte sectors on a disk. In each sector, we can write any 512-byte block of data we want. If we want to store larger blocks of data, they have to be split across multiple sectors. Later, when we want to read the data back, we just need to remember what sectors our data lives in.

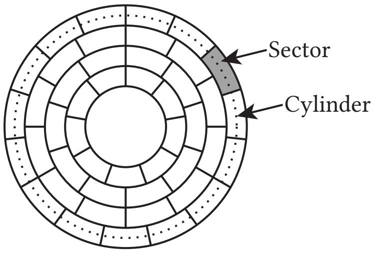


Figure 2.1: Geometry of a magnetic hard disk. Sectors are arranged in concentric circles called *cylinders*.

Reading from the disk can be a little challenging. We need to tell the BIOS (1) the disk read command, (2) where we want to read from on the disk, (3) where we want the data to go in memory, and (4) how much data to read. For (1), the disk read command 0x02 goes in register AH. For (2), we need to specify a sector on the disk to read. The disk is addressed in cylinder, head, and sector numbers. Confusingly, cylinder and head indexes start at 0, and sector indexes start at 1. So the first sector on the disk in C:H:S notation is 0:0:1. The first hard disk number is 0x80. For (3), we need to give a complete 20-bit address (segment and ofset) where we want the data from disk to be stored in memory. This is given to the BIOS in registers `ES:BX`. For (4), we need to tell the BIOS how many 512-byte sectors it should read from disk (passed in `AL`). Finally, call the BIOS with INT 0x13 to initiate the disk read.

Register	**Meaning**
`AH`	Command Code `0x02`
`AL`	Number of sectors to read
`ES`	Segment address where data should be written
`BX`	16-bit offset into `ES` to write data to
`CH`	Low 8 bits of cylinder number
`CL`	Sector number
`DH`	Head number
`DL`	Drive number (0x80 is the first hard disk)

Table 2.4: Parameters passed to the BIOS disk read command `INT 0x13`.

The BIOS disk read functionality only allows us to read individual sectors from a disk into memory. It has no support for filesystems—that's usually handled by the operating system.

2.2.2 DOS

In the early days of PCs, the commonly-used “operating system” was called *Disk Operating System* (DOS). DOS was not really an operating system in the usual sense. It was more like a set of extensions to the PC BIOS. The most useful support it provided above the BIOS was support for file systems and a rudimentary terminal interface. The BIOS, after all, can only read and write individual sectors from a disk—it doesn’t have support for creating files with names and locating those files on disk.

In those days, the most common form of storage was floppy disks, which could hold 720 kbytes or 1.44 mbytes of data depending on the disk type. Programs had to be small to fit on a disk, and computers could only run one program at a time. A program would be loaded off of its floppy disk into main memory, and it would use `INT` instructions to call BIOS or DOS functions.

For example, suppose a program calls the C library’s `fopen` to open a file on disk. The C library would parse the file name and generate a *file control block* data structure from the file path passed to `fopen`. It would then call DOS’s `INT 0x21` function, which is a DOS-specific extension to the BIOS. DOS would examine the file control block and use its file system driver to locate the data sectors of the file on the disk. The DOS file system driver would make calls to the BIOS disk read function using calls to `INT 0x13`.

2.3 GRUB

We only have one disk sector—512 bytes—for the MBR. There is not enough space in the MBR to store a complete bootloader that would be capable of loading a full kernel into memory. The bootloader usually needs lots of software to read the kernel from the disk: disk drivers, filesystem drivers, terminal drivers at least. PCs use a larger stage 1 bootloader to load the kernel from the disk. For the most part, each operating system has a custom bootloader. Linux uses a bootloader called GRUB (the GRand Unified Bootloader) which we will also use to load our kernel.

GRUB does a lot of dirty work for us: it configures hardware, loads our kernel from disk into memory, and most importantly it puts the CPU into protected mode. Putting the CPU into protected mode is not a fun job: it requires setting up a bunch of messy tables in memory

2.3.1 Partitions

Physical hard disks are often divided into smaller sections called partitions which can be used to separate different kinds of information. Commercial operating systems like Windows and macOS generally create several partitions on the computer’s main hard disk, the largest of which is used to store OS and user files. One or more smaller partitions store system and backup information that is generally hidden from the user’s view. A data structure inside the MBR stores the location and size of the disk’s partitions.

2.3.2 Binary File Formats

2.3.3 GRUB Hello World

Let’s write a *Hello World* program that can be loaded and run by GRUB. The overall procedure we will follow is to create a disk image with an empty filesystem, install GRUB on the image, then copy our program to the filesystem. A disk image is a file that contains

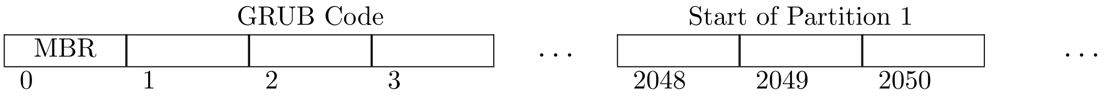


Figure 2.3: Layout of the disk image we will create for the GRUB hello world.

a byte-for-byte copy of the contents of some disk. Disk images are useful for replicating OS installations—fresh installations of operating systems are usually done from disk images. Virtual machines also use disk images to store the contents of their virtual disks. Throughout this book, we will be creating disk images from scratch which we will use to boot our operating system.

We said before that hard disks are laid out as sectors of 512 bytes. The disk image that we create will also be organized in 512 byte blocks. A diagram of the disk image is shown below. The first 2048 sectors of the disk image $(2048 \times 512 bytes = 1 Mbyte)$will be assigned to the MBR and bootloader (GRUB). The first 2048 sectors are not part of any filesystem, so they can't store files in the way we are used to. Instead, they store the raw code of the bootloader loaded into memory by GRUB at boot time.

The first partition starts at sector 2048 and contains a FAT filesystem. FAT is a simple filesystem format developed for MS-DOS in the 1980s. We will be using it later in the book to store files for our operating system. For now, we will use the FAT filesystem to store the binary file for our hello world program.

The binary file that contains our program needs to have some special format to be recognized by GRUB. Its header is different from the standard ELF header used by Linux programs. GRUB expects our binary file to have a *multiboot* header which has fields that are similar to the ELF header. Other than the *multiboot* header, the rest of the binary file is the same. Fields in the Multiboot 2 header are explained in Table **??**.

Field Name	**Meaning**
Magic Number	The integer `0xe85250d6` that tells GRUB that our binary is valid.
Flags	Binary-valued flags that tell GRUB what kind of data is contained in this multiboot2 entry.
Length	Length in bytes of this multiboot2 entry
Checksum	Integer to ensure that this multiboot2 entry has not been corrupted. The sum of all entries in the ent

Table 2.5: Fields in the Multiboot 2 header.

First, we will compile and link the C program. If you are compiling on an Intel or AMD x86 system, use the native compiler called `gcc`. If you are cross-compiling on a non-x86 system like an Apple M1 or other ARM device, use an x86 cross-compiler.

The compiler switches we use to build the kernel in this book tell the compiler to generate code for the i386 CPU using no external libraries. Most of the time when we compile programs, we want the compiler to bring in code from external libraries, which allow us to make calls to functions like `printf`. But those libraries assume that the program is running in an operating system. Since we are writing the operating system ourselves, we don't want to include those external functions. If we need them, we will have to write our own versions that don't depend on Linux syscalls.

```
user@system ~ $ gcc -c -ffreestanding -mgeneral-regs-only -mno-mmx -m32 -march=i386 -fno-pie -fno-sta
user@system ~ $ ld --section-start=.text=100000 --section-start=.rodata=0  -e main -melf_i386  hello.
```

Before we get started, let's install the `mtools` package, which contains tools for editing filesystems that are inside disk images without directly mounting them.

```
#define MULTIBOOT2_HEADER_MAGIC 0xe85250d6

const unsigned int multiboot_header[] = {MULTIBOOT2_HEADER_MAGIC, // Magic number
                                      0, // Flags
                                      16, // Length
                                      -(16+MULTIBOOT2_HEADER_MAGIC), // Checksum
                                      0, // Type
                                      12}; // Size

void main() {
    unsigned short *vram = (unsigned short*)0xb8000; // Base address of video mem
    const unsigned char color = 7; // gray text on black background
    vram[0] = (((unsigned short)color)<<8) | (char)’H’;
    vram[1] = (((unsigned short)color)<<8) | (char)’E’;
    vram[2] = (((unsigned short)color)<<8) | (char)’L’;
    vram[3] = (((unsigned short)color)<<8) | (char)’L’;
    vram[4] = (((unsigned short)color)<<8) | (char)’O’;

    while(1);
}
```

Listing 8: The code for GRUB Hello World.

```
user@system ~ $ sudo apt install mtools
```

Now we will create a 32 mbyte disk image called `rootfs.img` filled with zeros. The `dd` tool copies data in blocks from one file to another. It's kind of like `cat` for binary files. Here we are using it to copy from `/dev/zero`, which is a fake file that just reads the binary value 0. The output will be a new disk image file that we are creating called `rootfs.img` filled with 32 mbytes of binary 0.

```
user@system ~ $ dd if=/dev/zero of=rootfs.img bs=1M count=32
```

```
set timeout=5
set default=0 # Set the default menu entry

menuentry "Hello Grub" {
   set root=(hd0,msdos1)
   multiboot2 /hello   # Load our hello grub program into memory
   boot                # Run hello grub
}
```

Next we will create the GRUB image, a file that contains the GRUB code to be copied to the first megabyte of our disk image. In Figure ??, we drew the GRUB code as starting at sector 1 and occupying up to sector 2047. The GRUB image does not include the MBR (that's in a separate file). The command below creates `grub.img`, which will be copied to our disk image starting at sector 1.

```
user@system ~ $ grub-mkimage -p "(hd0,msdos1)/boot" -o grub.img -O i386-pc normal biosdisk mu
```

Now that we have a GRUB image, we can copy it to our disk image using `dd`. The first command below copies the MBR to sector 0 of our disk image. The MBR is stored in a 512-byte binary file on our Linux system called `boot.img`. The second command copies the grub image, which we created in the previous setp, to the disk image starting at sector 1.

```
user@system ~ $ dd if=/usr/lib/grub/i386-pc/boot.img of=rootfs.img conv=notrunc
user@system ~ $ dd if=grub.img of=rootfs.img conv=notrunc seek=1
```

Before we install our Hello Grub program onto our disk image, we need to format the image. Formatting is the process of setting up a partition table and a filesystem on a disk or disk image. We will only create one partition on the disk image: starting at sector 2048 and occupying all sectors to the end of the disk .

```
user@system ~ $ echo ’start=2048, type=83, bootable’ | sfdisk rootfs.img
```

Create a FAT16 filesystem on the first partition (starting at sector 2048) and copy our hello world binary to that filesystem.

```
user@system ~ $ mkfs.vfat --offset 2048 -F16 rootfs.img
user@system ~ $ mcopy -i rootfs.img@@1M hello ::/
user@system ~ $ mmd -i rootfs.img@@1M boot
user@system ~ $ mcopy -i rootfs.img@@1M grub.cfg ::/boot
```

If all these commands worked, you should have a bootable disk image that can be used as a virtual disk for the `qemu` emulator. Use the command below to start the emulator and run your program in GRUB.

```
user@system ~ $ qemu-system-i386 -hda rootfs.img
```

2.3.4 Linker Scripts

In the previous section, we designed Grub Hello World to be a simple single-file C program which contained all of the components that Grub needs to boot an OS. We used a cheap hack to locate the multiboot header at address 0: we declared it as a `const unsigned int` array, and we told the linker to place all `const` variables (which live in the `rodata` section of the ELF file) at address 0. Our trick is fine for a small kernel, but we need a more scalable solution that will work for larger codebases.

Linker scripts allow us to specify how our binary is assembled and what address is assigned to its various components. As we discussed, every ELF file is composed of sections: the `.text` segment stores code, the `.data` segment stores initialized values of global variables, the `.rodata` segment stores constant data, and so on. Table **??** lists common default section names and their uses.

Compiling Hello Grub was clunky because we had to specify the address of each segment on the command line. Linker scripts let us specify the address of each section in a separate file, which gives us more control over the compilation. Listing 10 shows a simple linker script that is equivalent to the command line switches we used to compile Hello Grub.

Section Name	**Contents**
`.text`	Program code
`.bss`	Uninitialized global variables
`.data`	Initialized global variables
`.rodata`	Constant data

Custom Sections In addition to the standard ELF sections from Table **??**, we can define custom ELF sections for certain variables. For example, suppose we want to create a separate ELF section for the multiboot header:

```
const unsigned int multiboot_header[] __attribute__((section(".multiboot"))) =
                                     {MULTIBOOT2_HEADER_MAGIC, // Magic number
                                      0, // Flags
```

```
ENTRY(main) /* Name of the first function to run */
OUTPUT_FORMAT(elf32-i386) /* Specify ELF file format */

SECTIONS
{
    . = 0; /* Set current location to address 0 */
    .rodata : { *(.rodata) } /* Place .rodata section at address 0 */
    /* Begin putting sections at 1 MiB, a conventional place for kernels to be
      loaded at by the bootloader. */
    . = 1M;
    .text : { *(.text) } /* Place .text starting at 1M (0x100000) */
    .data : { *(.data) } /* Place .data immediately after .text */
    .bss : { *(.bss) } /* Place .bss section immediately after .data */
}
```

Listing 9: Simple linker script for GRUB Hello World.

```
                                      16, // Length
                                      -(16+MULTIBOOT2_HEADER_MAGIC), // Checksum
                                      0, // Type
                                      12}; // Size
```

The __attribute__((section(".multiboot"))) instructs the compiler to place the multiboot_header array in a separate section called .multiboot. This is a nonstandard extension to the C language provided by gcc. We can modify our linker script to place the .multiboot section at address 0:

```
ENTRY(main) /* Name of the first function to run */
OUTPUT_FORMAT(elf32-i386) /* Specify ELF file format */

SECTIONS
{
    . = 0; /* Set current location to address 0 */
    .multiboot : { *(.multiboot) } /* Place only the multiboot header at address 0 */
    /* Begin putting sections at 1 MiB, a conventional place for kernels to be
      loaded at by the bootloader. */
    . = 1M;
    .text : { *(.text) } /* Place .text starting at 1M (0x100000) */
    .data : { *(.data) } /* Place .data immediately after .text */
    .rodata : { *(.rodata) } /* Place .rodata immediately after .data */
    .bss : { *(.bss) } /* Place .bss section immediately after .rodata */
}
```

Listing 10: A better linker script for GRUB Hello World that places only the multiboot header at address 0.

2.4 Exercises

Exercise 2.1 Write a Makefile that builds your Hello Grub program.

Exercise 2.2 Write a program that prints the character a to the screen repetitively. Fill up the whole screen with as.

Exercise 2.3 Write a delay function that causes your program to hang for a short amount of time. You can use a `for` loop. Modify your program from Exercise 2.2 to call your delay function and hang each time it prints the character `a`.

Exercise 2.4 Write a function that can print a string to the screen by writing to video memory. The inputs to your function should be the string to print and the (x, y) location on the screen to start printing:

```
void print_string(char *string, unsigned int x, unsigned int y)
```

Exercise 2.5 Modify your `print_string` function from Exercise 2.4 to implement a complete terminal output driver. Your driver should keep track of the screen position where it last wrote a character. Each successive call to `print_string` should display characters to the terminal immediately after the last character that was printed without the need for the programmer to specify a screen location.

```
void print_string(char *string)
```

Exercise 2.6 Modify your terminal driver from 2.5 to scroll the text on screen once the cursor goes past the end of the screen.

Chapter 3

Input and Output

Computers use input and output ports to interact with the outside world:

- Disk drives
- USB ports
- Network interfaces
- Display adapters
- Audio interfaces
- Timers

In this chapter, we discuss ways in which the computer communicates with its I/O devices and how it presents information from those devices to user programs.

3.1 Hardware I/O Connections

The x86 PC has two mechanisms for connecting IO devices to the CPU. Legacy IO connects directly to the CPU through an IO-specific hardware bus. Modern IO connects to the CPU through the memory system. Figure 3.1 shows a diagram of the interconnections of the IO on an x86 PC.

3.2 Communicating with I/O Devices

There are several mechanisms that the kernel can use to communicate with hardware I/O devices.

3.2.1 Memory-Mapped I/O

The concept of memory-mapped I/O is that the hardware assigns a particular address in memory to a hardware device. Reads and writes to a memory-mapped I/O address do not

Video Output

As we discussed in §2.2.1, the PC's video RAM is located at address `0xB8000`. We can display characters on the screen by writing their ASCII values to the video RAM. Each character position on the screen is controlled by two bytes: one ASCII byte that controls the character to be displayed, and a second byte that specifies what color the character should be.

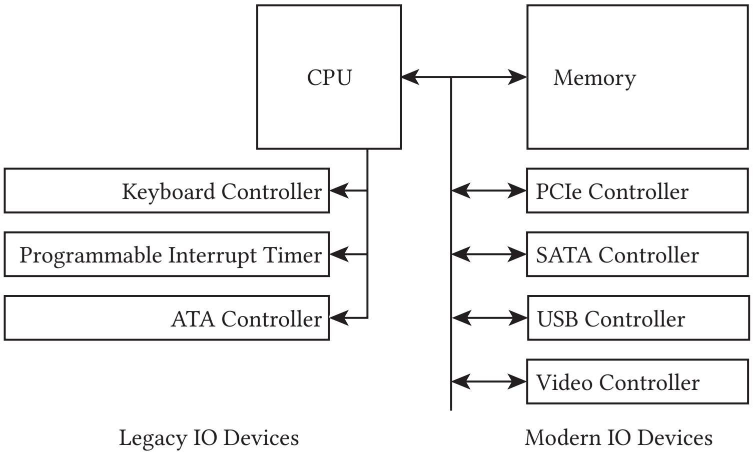


Figure 3.1: Interconnections between the CPU, memory, and IO devices on an x86 PC system.

	Column 1		Column 2		Column 3		. . .	Column 80	
	↓		↓		↓			↓	
B8000	ascii	color	ascii	color	ascii	color	...	ascii	color
B8050	ascii	color	ascii	color	ascii	color	...	ascii	color
B80A0	ascii	color	ascii	color	ascii	color	...	ascii	color

x86 PS/2 Keyboard Controller

I/O Port	Access Type	Function
0x60	R/W	Data Port
0x64	R	Status Register
0x64	W	Command Register

PS/2 Status Register The PS/2 status register contains information about the state of the PS/2 controller. A detailed description is shown in Table 3.2.1. The important bit for taking input from the keyboard is OS, the output status bit. That bit will be set to 1 if there is data available from the keyboard and cleared to 0 of no data is available. If OS is set (1), we can read the data port to get a scancode from the keyboard.

7	6	5	4	3	2	1	0
PE	TOE	UNK	UNK	CD	SF	IS	OS

PE	**Parity Error**	0 indicates no error, 1 indicates error
TOE	**Timeout Error**	0 indicates no error, 1 indicates error
UNK	**Chipset Specific**	
UNK	**Chipset Specific**	
CD	**Command/Data**	0 indicates data written to input buffer is data for PS/2 device, 1 indicates data written to input buffer is data fro PS/2 Controller
SF	**System Flag**	Set by BIOS if system passes self test (POST)
IS	**Input Buffer Status**	0 means empty, 1 means full
OS	**Output Buffer Status**	0 means empty, 1 means full

Reading and Writing to IO Ports on x86 On x86, we have two special instructions to communicate with IO ports: `in` and `out`.

To determine if the user pressed a key, we want to read the PS/2 status register and check the LSB. If the LSB is 1, then that means that means that the PS/2 controller's output buffer contains a scancode. To get the scancode, read from IO port `0x60`.

```
.lp:
    in ax,0x64          ; Get keyboard status reg
    and ax,1
    je .lp              ; If LSB in status is clear, no key
    in al,0x60          ; Get scancode
    mov ah,0
    push ax
    call print_hex_16   ; Print the scancode
    add sp,2
    jmp .lp             ; Check for another keypress
```

Instead of implementing our keyboard echo program in assembly, we can use inline assembly inside of a C program to use an `in` instruction to read from the PS/2 controller.

```
/*
 * outb
 *
 * Performs a one-byte write to an x86 port.
 */
void outb(uint16_t _port, uint8_t _data) {
    __asm__ __volatile__ ("out %1, %0" : : "dN" (_port), "a" (_data));
}
/*
 * inb
 *
 * Performs a one-byte read from an x86 port.
 */
uint8_t inb (uint16_t _port) {
    uint8_t rv;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
    return rv;
}
```

Listing 11: An inline assembly impelmentation of the `in` and `out` instructions.

x86 Programmable Interval Timer (PIT)

```
#define PIC_PIT_CHANNEL_0_DATA 0x40
#define PIC_PIT_CHANNEL_1_DATA 0x41
#define PIC_PIT_CHANNEL_2_DATA 0x42
#define PIC_PIT_MODE_CMD_REG 0x43

void programPIT(uint16_t reload_val) {
    outb(0x34, PIC_PIT_MODE_CMD_REG);
    outb(reload_val & 0xff, PIC_PIT_CHANNEL_0_DATA); // Set reload val low byte
    outb((reload_val>>8) & 0xff, PIC_PIT_CHANNEL_0_DATA); // Set reload val hi byte
}

uint16_t readPitCount() {
    uint16_t count;
    outb(0, PIC_PIT_MODE_CMD_REG); // Send latch count value command for channel 0
    count = inb(0x40); // Read latched value (Lo byte)
    count |= inb(0x40)<<8; // Read latched value (Hi byte)
    return count;
}
```

Listing 12: A simple interface to the x86 PIT that writes its reload value and reads the current count.

3.3 Polling I/O

Like the x86 PIT, most I/O devices have some status register that allows low-level software to request information about the status of the hardware device. The simplest way to interact with I/O devices is by repeatedly polling that status register, checking if some event has taken place. For example, we might write a delay function that continuously calls `readPitCount()` until the count reaches some threshold value:

Register Name	**I/O Address**
PIT Channel 0 Data	0x40
PIT Channel 1 Data	0x41
PIT Channel 2 Data	0x42
PIT Mode & Command Register	0x43

Table 3.1: Registers of the x86 PIT

```
void delay(int n) {
    while(readPitCount() < n);
}
```

This kind of function was pervasive in antique video games to control the screen refresh rate. It was usually part of a larger *game loop* such as the following:

```
void game_loop() {
  while(1) {
    int scancode = read_keyboard(); // Get keyboard input
    int mouse_position = read_mouse(); // Get mouse input
    update_sprite(scancode, mouse_position); // Move sprites according to keyboard input
    draw_screen_buffer();
    delay(2000);
  }
}
```

On each round of the game loop, it (1) reads the input from the keyboard and mouse, (2) updates the state of the game's characters, and (3) redraws the screen buffer. The delay function serves to slow down the game loop and prevent the gameplay from moving too quickly. The keyboard and mouse are polled on each round of the loop, and the I/O polling burns a substantial number of CPU cycles. Polling one or two I/O devices might be manageable, but it does not scale well for a large number of I/O devices.

Exercise 3.1 Augment your Hello Grub program with an infinite loop that polls the PS/2 keyboard controller and displays the scancodes on the screen. On each round of the loop, it should (1) read scancodes from the keyboard and (2) print the scancode out to the terminal in decimal.

Exercise 3.2 Modify the program from Exercise 3.1 to print the ASCII characters from the keyboard. The keyboard map below can be used to translate a keyboard scancode into a corresponding ASCII character. For example, suppose the user presses the *3* key on the keyboard. This will send scancode `4` to the PS/2 controller. We can look up the corresponding ASCII character: `keyboard_map[4] = '3'`.

```
unsigned char keyboard_map[128] =
{
    0, 27, '1', '2', '3', '4', '5', '6', '7', '8', /* 9 */
  '9', '0', '-', '=', '\b', /* Backspace */
  '\t', /* Tab */
  'q', 'w', 'e', 'r', /* 19 */
  't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n', /* Enter key */
    0, /* 29 - Control */
  'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', /* 39 */
  '\'', '`', 0, /* Left shift */
  '\\', 'z', 'x', 'c', 'v', 'b', 'n', /* 49 */
  'm', ',', '.', '/', 0, /* Right shift */
  '*',
    0, /* Alt */
  ' ', /* Space bar */
    0, /* Caps lock */
    0, /* 59 - F1 key ... > */
    0, 0, 0, 0, 0, 0, 0, 0,
    0, /* < ... F10 */
    0, /* 69 - Num lock*/
    0, /* Scroll Lock */
    0, /* Home key */
    0, /* Up Arrow */
    0, /* Page Up */
  '-',
    0, /* Left Arrow */
    0,
    0, /* Right Arrow */
  '+',
    0, /* 79 - End key*/
    0, /* Down Arrow */
    0, /* Page Down */
    0, /* Insert Key */
    0, /* Delete Key */
    0, 0, 0,
    0, /* F11 Key */
    0, /* F12 Key */
    0, /* All other keys are undefined */
};
```

3.4 Interrupt-Driven I/O

Programs that use polling to interact with I/O devices waste a lot of time checking the status registers of various input devices. Interrupts are a mechanism that the hardware can use to notify the program when an input event occurs so the software doesn't have to continuously poll status registers.

When an input event occurs, the CPU interrupts the software that is currently running, saving the contents of some of its registers on the stack. The CPU then automatically jumps to an *interrupt service routine* (ISR)—a special function that handles the hardware event. Once the interrupt service routine finishes, it can return to the original code that was running before the interrupt occured.

Each hardware device on a computer has a designated interrupt service routine. The ISR that handles a keyboard interrupt is different from the one that handles events from the system timer or the hard disk.

When an interrupt occurs, the hardware saves the running program's *context* on the stack. The context is basically just the set of critical information that the CPU will need to save so it can correctly resume executing the program after the ISR completes. At a minimum, the CPU context consists of (1) the program's instruction pointer (`IP`) and (2) the program's `FLAGS` register.

3.4.1 The Interrupt Descriptor Table

3.4.2 Masking Interrupts

3.5 Direct Memory Access

3.6 Interfacing to User Programs

A clean interface from user program to I/O device is one of the most important features an operating system can provide. Operating systems provide a runtime environment that makes each program think that it has complete and exclusive access to the entire system. Of course this cannot be so for I/O devices: most need to be shared by many programs on the computer. So it is a challenge for the OS designer to present users with an easy-to-use virtual interface to all the system's hardware devices while allowing multiple running programs to share hardware resources.

Consider your laptop's video display interface, which obviously must be shared among multiple programs at once. Operating systems generally implement access to the display by isolating each application within its own window. How should the application communicate to the operating system what to draw inside its window? There are generally two ways in which operating systems support graphical interfaces:

- **Operation-specific APIs**: the OS provides a specific API for all common tasks, including drawing text, drawing clickable buttons, displaying lists and menus, and so forth. Each API has an operation-specific interface defined by the OS. An API to draw text requires a specific set of inputs and outputs. Different inputs and outputs might be used to place a clickable button.

- **Virtualized screen buffer**: the OS allows the app to write directly to a virtual screen buffer. Any data drawn to the screen buffer will appear in the app's window. Drawing a button requires the programmer to set the color in the virtual screen buffer of every pixel on the button. There are no operation-specific APIs—those must be implemented by the app or in an external library.

Which technique is better? It is hard to find anything positive to say about operation-specific APIs, although most (all?) operating systems use them. They are not regular: since each type of visual primitive (button, text, etc.) needs different information to draw, there must be a different interface for every primitive. And because most OSes provide lots of options to app developers, their APIs are in practice not simple—developers need to know a lot of details about how to call each API, which means reading lots of documentation.

- **Regularity**: if you do something one way for one kind of I/O device, you should do it the same way for a different kind of I/O device. Predictability in the interface makes the operating system easy to use.
- **Simplicity**: Don't make your user read thousands of pages of documentation in order to interact with your driver.

3.6.1 Privilege

Since many programs share I/O devices on the computer, we don't want one particular program to be allowed to directly control the I/O hardware. Imagine if only one program was allowed to control your computer's network interface—only that program would be capable of sending and receiving packets while all other programs would be cut off. In that scenario, you could have a web browser open or a video conferencing app, but never both at the same time.

Worse yet, imagine if multiple programs were allowed to directly control hardware interfaces simultaneously. When the web browser received a packet intended for the video conferencing app, it would have no idea how to handle that packet.

To avoid problems, programs running on a computer are not allowed to directly access the hardware I/O devices. Dealing with the hardware I/O is the job of the operating system. Access to hardware resources on the computer is mediated by the notion of privilege. Privileged software (like the operating system) is allowed to directly read and write from just about any hardware device on the computer. Unprivileged software (like user programs) is only allowed to read and write to a small portion of the computer's memory that is designated for use by that program. If a user program wants to access a hardware device, it needs to call the operating system's API for that device. The OS—acting as an intermediary between the program and the hardware device—will access the hardware I/O device on behalf of the requesting program and return the desired information.

A program's privilege level is set by the state of the CPU. While the CPU is in privileged mode, it can execute any instruction and access any address in memory—privileged or unprivileged. When the CPU is in unprivileged mode, it can only execute unprivileged instructions and access unprivileged memory. The CPU privilege level is determined by the `IOPL` bits in the `FLAGS` register.

3.7 Exercises

Exercise 3.3 Write a `Makefile` that builds your Hello Grub program.

Chapter 4

Memory Management

4.1 Memory Allocation

The goal of a memory allocator is to keep track of which memory is free and which memory is in use. When some program (either the kernel or a userland program) needs memory, the memory allocator finds a block of free memory that is large enough to satisfy the request, marks that memory as in-use, and returns the address of the beginning of that memory block.

You will need to implement one of the memory allocators described in this chapter for your operating system. The first fit and buddy allocators are preferable because they are the most flexible designs.

Why do we Need Memory Allocators? Simple programs that we write well-defined tasks generally know what their memory requirements will be at compile time and therefore don't need memory allocators. For example, consider a trivial program that counts the number of characters in a pre-defined block of text:

```c
#include <stdio.h>
char text[] = "the quick brown fox jumped over the lazy dog";

int main() {
  int count = 0;
  while(text[count] != 0) {
    count++;
  }
  printf("Character count is %d\n", count);
  return 0;
}
```

Listing 13: A simple program that counts the number of characters in a pre-defined block of text

In this program, we know the dataset we are operating on *at compile time*. In other words, the input dataset (`char text[]`) is integrated into the program's binary. Since the compiler knows what the input datset will be, it can build the dataset into the program's binary. When the operating system runs the program, it will load the program's binary image from disk into memory, and that image will contain the program's input dataset—no memory allocation needed.

Now suppose we want to make our program more flexible by enabling it to count the number of characters in a file on disk. The program will take a filename as input from the user, open that file, read the file into memory, and count the number of characters. In this case, we don't know the size of the input dataset until we run the program. Since we need to read the input dataset from a file and we won't know how big the file is until we open it, we can't reserve memory for the dataset at compile time.

One simple workaround might be to restrict the maximum size of the input dataset. We could allocate a large array at compile time, then insist that the dataset is never larger than our buffer, as in Listing 14. This approach works for small files, but not for files larger than 4kbytes.

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
char text[4096]; // Large buffer

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  int n = read(fd, text, sizeof(text));
  int count = 0;
  while(text[count] != 0) {
    count++;
  }
  printf("Character count is %d\n", count);
  return 0;
}
```

Listing 14: A simple program that counts the number of characters in a file, limited to 4096 characters.

To make our program flexible so it works with files of any size, we need dynamic memory allocation. Our strategy will be to (1) open the file, (2) find out how large it is, then (3) allocate a buffer that will hold the entire file. A demonstration of this approach is shown in Listing 15.

The first call to `lseek()` returns the length of the file in bytes. To read the entire file into memory, we need to have an array at least of that length. The `malloc()` call requests a block of memory that is large enough to accomodate the whole file. `malloc()` returns the memory address of the beginning of the allocated block. This block is then filled up by `read()` with the file contents.

In an operating system, there are many scenarios where we need to dynamically allocate memory, and we need some function like `malloc` that can keep track of what memory on the computer is available for allocation. Our operating system will need to have a custom standalone memory allocator—`malloc()` isn't an option for us because it relies on underlying OS functionality that we haven't built yet. This section discusses techniques for implementing standalone memory allocators.

4.1.1 The Brain Dead Memory Allocator

The brain dead memory allocator starts with a large block of free memory called the heap. All memory requests will be satisfied from the heap. The brain dead allocator maintains a pointer to the beginning of free memory, which initailly is set to the base of the heap as in Figure 4.1a. The allocator satisfies memory requests starting at the beginning of the heap by incrementing the `freeptr` by the amount of memory requested then returning the starting address of the newly allocated block.

```
#include <stdio.h>
#include <fcntl.h> // needed for open()
#include <unistd.h> // needed for read()
#include <stdlib.h> // needed for malloc()

int main(int argc, char **argv) {
  int fd = open(argv[1], O_RDONLY);
  int fsize = lseek(fd, 0, SEEK_END); // Seek to end of file.
                                      // Returns the offset from beginning of file

  lseek(fd, 0, SEEK_SET); // Reset the seek pointer to beginning of file

  char *text = malloc(fsize); // Allocate a buffer big enough to hold file
  int n = read(fd, text, fsize);
  int count = 0;
  while(text[count] != 0) {
    count++;
  }
  printf("Character count is %d\n", count);

  free(text); // Free the buffer
  return 0;
}
```

Listing 15: A simple program that counts the number of characters in a file of any size.

We call this the brain dead allocator because of an obvious flaw: it will eventually run out of memory. There is no provision for this allocator to release unused memory back to the heap when we are finished with it. After enough allocation requests, the `freeptr` will reach the top of the heap and be unable to satisfy subsequent requests.

To avoid running out of memory, we need some way of returning memory blocks to the free heap once we are done using them. The rest of the allocation strategies discussed here enable us to return unused memory to the heap.

Exercise 4.1 Implement `bd_malloc()`, the brain-dead memory allocator in C.

4.1.2 Block Memory Allocator

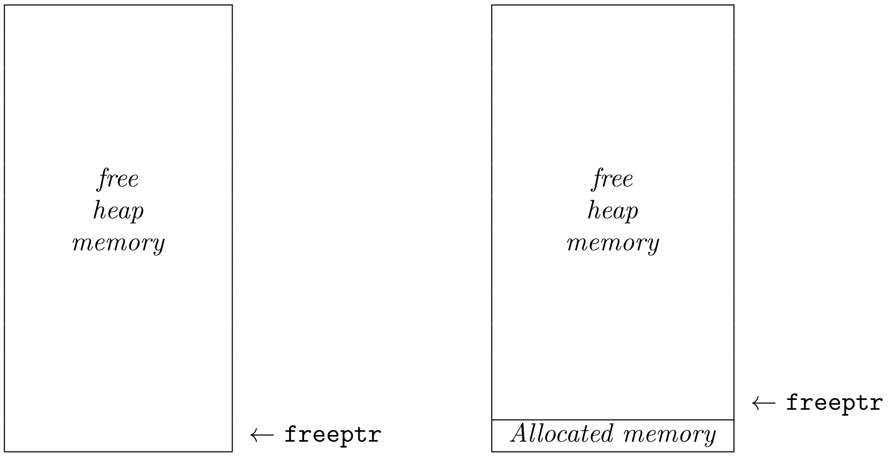
The block memory allocator divides free memory into fixed-size blocks that can be used to satisfy allocation requests. The block allocator keeps track of which blocks are free and which are used. Conventionally, free memory is split into pools of power-of-two sized blocks. For example, the allocator may create pools of 128, 256, 512, and 1024-byte blocks. These blocks are usually configured as arrays at compile time.

Block allocators maintain some metadata structure that tracks whether each block is free or used. This can be implemented as a bitmap, linked list or hash map. When a new memory request arrives, the block allocator searches for a free block that is large enough to satisfy the request. It marks that block as used and returns the base address of the block.

When the calling program is finished using a memory block, it frees the block by calling an allocator function that returns the block to the free pool.

Characteristics of the Block Allocator

- Low external fragmentation, high internal fragmentation
- Fast



(a) Free heap before any memory allocations

(b) Heap after one allocation.

Figure 4.1: Heap in the brain dead allocator. In 4.1a, all memory in the heap is initally free. After one allocation request, a block of memory at the beginning of the heap is allocated, and the rest is free.

- Somewhat inflexible to different workloads: must know about the size of blocks that will be requested at compile time.

The block memory allocator has historically been preferred for long-running unattended software like embedded systems and servers because it does not cause external fragmentation. In fact, the original implementation of the `bash` UNIX shell used a block allocator to avoid fragmentation. As physical memories have gotten larger, programmers have become less concerned with memory fragmentation because we can easily expand the heap. The next two memory allocators are much more flexible than the brain dead allocator or the block allocator because they can expand the heap under high memory pressure.

Exercise 4.2 Implement `block_malloc()` and `block_free()`, in C.

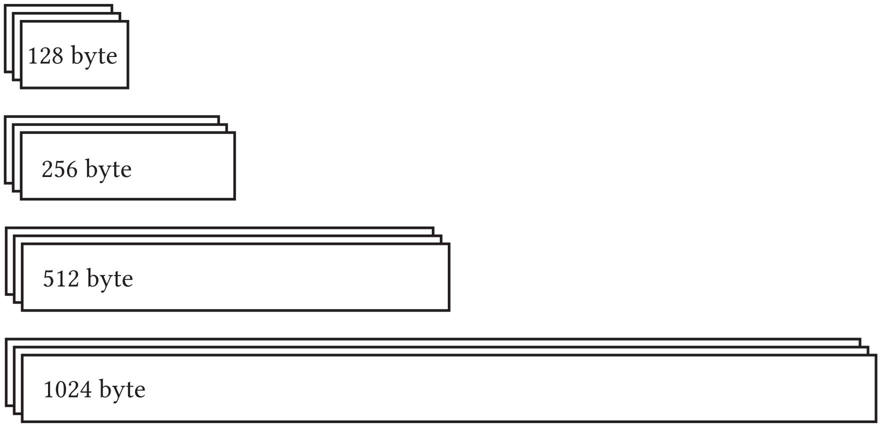


Figure 4.2: Blocks of memory used by the block memory allocator.

4.1.3 The First Fit Allocator

4.1.4 The Buddy Allocator

4.2 Virtual Memory

4.2.1 The Page Table

4.2.2 The Translation Lookaside Buffer

Chapter 5

Filesystems

Appendices

Appendix A

Building a Cross Compiler

If building on a non-x86 system, you will need a cross compiler to generate code for the x86. A cross compiler is a compiler that runs on one system but generates code for a different system.

Some Linux distributions such as Ubuntu have prepackaged cross compilers to cross-compile code for popular platforms. On other Linux distributions, you will need to build your own cross compiler from source. `crosstool`, explained in **??** is a tool to automatically build a complete cross compiler plus necessary extras like libraries.

A.1 A Practical Guide to Using Cross Compilers

As an advanced user of compilers, it's helpful to understand some of the jargon that compiler developers use when they talk about the systems they work on.

A.1.1 Toolchains vs Compilers

Strictly speaking, the compiler is just the program that converts source code into machine code. There are many other ancillary tools that we use to Table A.1 lists some important tools from a package called *Binutils* that come in handy when working on an operating system. We use the binutils tools throughout this book to work with binary files.

Binutils is also used by the toolchain to generate binary files. The binutils tools are architecture-dependent: tools built for one architecture cannot be used to work with binary files for a different architecture.

A.1.2 Compiler Tuples

To specify the platform that a compiler generates code for, `gcc` uses a *compiler tuple*. The tuple tells (i) the CPU the the compiler targets, (ii) the operating system the generated binaries will run in and (iii) the C library the binaries will use when they run. For our purposes in this book, the most important part of the compiler tuple is the CPU that the compiler generates code for. In general, when a compiler compiles a program, it needs to know what OS and library that program will use at runtime. The OS and libraries are not part of the core program code, so the compiler needs to know how to call external functions. We will be writing out own C library and OS, so there will be no external library or OS code to be called. The compiler does not need to know what those are in order to generate code, so the library and OS parts of the tuple are not important for us.

Table A.1: Key tools in GNU Binutils used for manipulating object files.

Tool	Function	Example
`objcopy`	Convert binary file format	`objcopy -O binary obj/mbr.elf mbr.img` Convert `mbr.elf` to flat binary file `mbr.img`
`objdump`	Reverse-engineer object files	`objdump --source file.elf` Disassemble `file.elf`
`size`	Lists the sizes of the sections of a binary file	`size test.elf` Display size in bytes of the `text`, `data` and `bss` segments of `test.elf`
`reaadelf`	Examine metadata in an ELF file	`readelf -h file.elf` Print ELF file header from `file.elf`

For example, you might run a cross compiler on an ARM-based Mac to generate code for an x86 PC. In Ubuntu, you can install with the following command:

```
user@system ~ $ sudo apt install gcc-i686-linux-gnu
```

This command will install a compiler and associated tools called `i686-linux-gnu-gcc`. This is confusing—the package name in Ubuntu's `apt` package manager is called `gcc-i686-linux-gnu`, but the compiler is called `i686-linux-gnu-gcc`. Other tools for the cross compiler like `objdump`, `size`, etc. all have the `i686-linux-gnu-` prefix.

A.2 Crosstool

`crosstool-ng` is a tool for building cross compilers. It is useful on systems where pre-packaged cross compilers are not available. On Gentoo, run:

```
user@system ~ $ sudo bash -c "’echo sys-devel/crosstool-ng **’ \
                  >> /etc/portage/package.accept_keywords/crosstool"
user@system ~ $ sudo emerge sys-devel/crosstool-ng
```

Create an empty directory to hold the crosstool configuration, then

```
user@system ~ $ mkdir i386-crosstool
user@system ~ $ cd i386-crosstool
user@system ~/i386/crosstool $ ct-ng menuconfig
```

In the crosstool menuconfig:

```
  Target options --->
    Target architecture (x86)
    Bitness: (32-bit)
    (i386) Emit assembly for CPU
  C-library --->
    C library (none)
  Debug Facilities --->
    [*] gdb
```

Then build the toolchain:

```
user@system ~/i386/crosstool $ ct-ng build
```

Crosstool with then download the source needed to build `gcc` and `binutils` as a cross compiler. When it finishes, the toolchain will be located in `/x-tools/i386-unknown-elf/bin`. You can leave it in that location or copy it to a `/usr/local` to avoid polluting your home directory:

```
user@system ~/i386/crosstool $ cp -r ~/x-tools/i386-unknown-elf /usr/local
```

Finally, add the location of your toolchain to your shell's path. Place the following line at the end of your `/.bashrc` file.

```
export PATH=$PATH:/usr/local/i386-unknown-elf/bin
```

Appendix B

Building Grub

This chapter explains how to build GRUB from source for the i386 platform. You might need to build GRUB if your development system doesn't offer an i386-PC installation package.

```
user@system ~ $ sudo apt install autoconf gettext autopoint pkg-config
user@system ~ $ git clone https://git.savannah.gnu.org/git/grub.git
user@system ~ $ cd grub
user@system ~ $ ./bootstrap
```

If building on a non-x86 system, you need to specify the name of the cross compiler. In the following command, specify the `--target=` option that matches the name of the cross compiler on your system. For example, if you are using Ubuntu and installed the `i686-linux-gnu-gcc` package, the following command will work without modification. If you have a compiler with a different name (like if you built with Crosstool), you may need to use a different targeet name that matches your toolchain.

```
user@system ~ $ ./configure --with-platform=pc --target=i686-linux-gnu --prefix=/usr/local/grub/
user@system ~ $ make
user@system ~ $ sudo make install
```

Index