Homework 3: Boot into C

This assignment will teach you to build a minimal bootable code that boots on real hardware into C. Technically, you can do this assignment on any operatir that supports the Unix API and can run Qemu (Linux Openlab machines, your laptop that runs Linux or Linux VM, and even MacOS, etc.). **You don't need xv6 for this assignment** Submit your programs and the shell through Gradescope (see instructions at the bottom of this page). For **Mac / OSX** users. The s 32 bit applications is deprecated in the latest version of your system. So if you already updated your system to MacOS Catalina or have updated your XCode recommend you to do the homework at the Openlab machines. This assignment explains how to create a minimal x86 operating system kernel using the standard. In fact, it will just boot and print "Hello, world!" on the screen, and then print "Hello from C!" on the serial line from the C `main()` function. Mc assignment is based on intermezzOS project.

Boot overview

When you turn on a computer, it loads the BIOS from some special flash memory. The BIOS runs self test and initialization routines of the hardware, then it lo bootable devices. If it finds one, the control is transferred to its bootloader, which is a small portion of executable code stored at the device's beginning. The b has to determine the location of the kernel image on the device and load it into memory. It also needs to switch the CPU to the so-called protected mode beca CPUs start in the very limited real mode by default (to be compatible to programs from 1978). We won't write a bootloader because that would be a complex p its own (we partially covered this in class since xv6 implements a simple boot loader with two files: `bootasm.S` and `bootmain.c`). Instead we will use one of the ma tested bootloaders out there to boot our kernel from a CD-ROM.

Multiboot headers

Let's get going! The very first thing we're going to do is create a 'multiboot header'. What's that, you ask? Well, to explain it, let's take a small step back and t how a computer boots up.

One of the amazing and terrible things about the x86 architecture is that it's maintained backwards compatibility throughout the years. This has been a cc advantage, but it's also meant that the boot process is largely a pile of hacks. Each time a new iteration comes out, a new step gets added to the process. Th when your fancy new computer starts up, it thinks it's an 8086 from 1976. And then, through a succession of steps, we transition through more and more architectures until we end at the latest and greatest.

The first mode is called 'real mode'. This is a 16 bit mode that the original x86 chips used. The second is 'protected mode'. This 32 bit mode adds new things real mode. It's called 'protected' because real mode sort of let you do whatever you wanted, even if it was a bad idea. Protected mode was the first time hardware enabled certain kinds of protections that allow us to exercise more control around such things as RAM. We'll talk more about those details later.

The final mode is called 'long mode', and it's 64 bits. Since our OS will only enter 32bit mode we'll not touch 64bit 'long mode'.

> So that's the task ahead of us: make the jump up the ladder and get to 32bit mode. We can do it! Let's talk more details.

Firmware and the BIOS

> So let's begin by turning the power to our computer on.
>
> When we press the power button, a bunch of low-level initialization protocols are executed: Management Engine, BIOS, etc.
>
> With the BIOS we're already in the land of software, but unlike software that you may be used to writing, the BIOS comes bundled with its computer an located in *r*ead-*o*nly *m*emory (ROM).
>
> One of the first things the BIOS does is run a 'POST' or *p*ower-*o*n *s*elf-*t*est which checks for the availability and integrity of all the pieces of hardware the computer needs including the BIOS itself, CPU registers, RAM, etc. If you've ever heard a computer beeping at you as it boots up, that's the PC reporting its findings.
>
> Assuming no problems are found, the BIOS starts the real booting process.
>
> > **By the way...**
> >
> > For a while now most commercial computer manufacturers have hidden their BIOS booting process behind some sort of splash screen. It's usually possible to see the BIOS' logs by pressing some collection of keys when your computer is starting up.
> >
> > The BIOS also has a menu where you can see information about the computer like CPU and memory specs and all the hardware the BIOS detected like hard drives and CD and DVD drives. Typically this menu is accessed by pressing some other weird collection of keyboard keys while the computer is attempting to boot.
>
> The BIOS automatically finds a 'bootable drive' by looking in certain pre-determined places like the computer's hard drive and CD and DVD drives. A d is 'bootable' if it contains software that can finish the booting process. In the BIOS menu you can usually change in what order the BIOS looks for boota drives or tell it to boot from a specific drive.
>
> The BIOS knows it's found a bootable drive by looking at the first few kilobytes of the drive and looking for some magical numbers set in that dri memory. This won't be the last time some magical numbers or hacky sounding things are used on our way to building an OS. Such is life at such a level...
>
> When the BIOS has found its bootable drive, it loads part of the drive into memory and transfers execution to it. With this process, we move away f what comes dictated by the computer manufacturer and move ever closer to getting our OS running.

Bootloaders

The part of our bootable drive that gets executed is called a 'bootloader', since it loads things at boot time. The bootloader's job is to take our kernel, p into memory, and then transition control to it.

Some people start their operating systems journey by writing a bootloader. For example, in class we started by looking at the xv6 bootloader that is loa by the BIOS at the `0x7c00` address. In this assignment we will not be doing that.

In the interest of actually getting around to implementing a kernel, instead, we'll use an existing bootloader: GRUB.

GRUB and Multiboot

GRUB stands for '*gr*and *u*nified *b*ootloader', and it's a common one for GNU/Linux systems. GRUB implements a specification called Multiboot, which set of conventions for how a kernel should get loaded into memory. By following the Multiboot specification, we can let GRUB load our kernel.

The way that we do this is through a 'header'. We'll put some information in a format that multiboot specifies right at the start of our kernel. GRUB will r this information, and follow it to do the right thing.

One other advantage of using GRUB: it will handle the transition from real mode to protected mode for us, skipping the first step. We don't even nee know anything about all of that old stuff. If you're curious about the kinds of things you would have needed to know, put "A20 line" into your favorite sea engine, and get ready to cry yourself to sleep.

Writing our own Multiboot header

I said we were gonna get to the code, and then I went on about more history. Sorry about that! It's code time for real! You can download the entire fo that contains skeletons for the homework files here or save it file by file. Inside your homework folder there is a file called `multiboot_header.asm`. Open your favorite editor. I use `vim`, but you should feel free to use anything you'd like.

```
$ vim multiboot_header.asm
```

This is a `.asm` file, which is short for 'assembly'. That's right, we're going to write some assembly code here. Don't worry! It's not super hard.

An aside about assembly

Have you ever watched Rich Hickey's talk "Simple vs. Easy"? It's a wonderful talk. In it, he draws a distinction between these two words, which commonly used as synonyms.

Assembly coding is simple, but that doesn't mean that it's easy. We'll be doing a little bit of assembly programming to build our operating system, but don't need to know *that much*. It is completely learnable, even for someone coming from a high-level language. You might need to practice a bit, and tal slow, but I believe in you. You've got this. A good manual on NASM assembler is here.

The Magic Number

Our first assembly file will be almost entirely *data*, not code. Here's the first line:

```
dd 0xe85250d6 ; magic number
```

Ugh! Gibberish! Let's start with the semicolon (`;`). It's a comment, that lasts until the end of the line. This particular comment says 'magic number'. As said, you'll be seeing a lot of magic numbers in your operating system work. The idea of a magic number is that it's completely and utterly arbitrar doesn't mean anything. It's just magic. The very first thing that the multiboot specification requires is that we have the magic number `0xe85250d6` right at start.

> **By the way...**
>
> Wondering how a number can have letters inside of it? `0xe85250d6` is written in hexadecimal notation. Hexadecimal is an example of a "numeral system" which is a fancy term for a system for conveying numbers. The numeral system you're probably most familiar with is the decimal system which conveys numbers using a combination of the symbols `0` - `9`. Hexadecimal on the other hand uses a combination of 16 symbols: `0` - `9` and `a` - `f`. Along with its fellow numeral system, binary, hexadecimal is used *a lot* in low level programming. In order to tell if a number is written in hexadecimal, you may be tempted to look for the use of letters in the number, but a more surefire way is to look for a leading `0x`. While `100` isn't a hexadecimal number, `0x100` is.

What's the value in having an arbitrary number there? Well, it's a kind of safeguard against bad things happening. This is one of the ways in which we check that we actually have a real multiboot header. If it doesn't have the magic number, something has gone wrong, and we can throw an error.

I have no idea why it's `0xe85250d6`, and I don't need to care. It just is.

Finally, the `dd`. It's short for 'define double word'. It declares that we're going to stick some 32-bit data at this location. Remember, when x86 first starte was a 16-bit architecture set. That meant that the amount of data that could be held in a CPU register (or one 'word' as it's commonly known) was 16 I To transition to a 32-bit architecture without losing backwards compatibility, x86 got the concept of a 'double word' or double 16 bits.

The mode code

Okay, time to add a second line:

```
dd 0xe85250d6 ; magic number
dd 0          ; protected mode code
```

This is another form of magic number. We want to boot into protected mode, and so we put a zero here, using `dd` again. If we wanted GRUB to something else, we could look up another code, but this is the one that we want.

Header length

The next thing that's required is a header length. We could use `dd` and count out exactly how many bytes that our header is, but there's two reasons we're not doing that:

1. Computers should do math, not people.
2. We're going to add more stuff, and we'd have to recalculate this number each time. Or wait until the end and come back. See #1.

Here's what this looks like:

```
header_start:
    dd 0xe85250d6                ; magic number
    dd 0                         ; protected mode code
    dd header_end - header_start ; header length
header_end:
```

You don't have to align the comments if you don't want to. I usually don't, but it looks nice and after we're done with this file, we're not going to mess wi again, so we won't be constantly re-aligning them in the future.

The `header_start:` and `header_end:` things are called 'labels'. Labels let us use a name to refer to a particular part of our code. Labels also refer to memory occupied by the data and code which directly follows it. So in our code above the label `header_start` points directly to the memory at the beginning of our magic number and thus to the very beginning of our header.

Our third `dd` line uses those two labels to do some math: the header length is the value of `header_end` minus the value of `header_start`. Because `header_s` and `header_end` are just the addresses of places in memory, we can simply subtract to get the distance between those two addresses. When we compile assembly code, the assembler will do this calculation for us. No need to figure out how many bytes there are by hand. Awesome.

You'll also notice that I indented the `dd` statements. Usually, labels go in the first column, and you indent actual instructions. How much you indent is u you; it's a pretty flexible format.

The Checksum

The fourth field multiboot requires is a 'checksum'. The idea is that we sum up some numbers, and then use that number to check that they're all what expected things to be. It's similar to a hash, in this sense: it lets us and GRUB double-check that everything is accurate.

Here's the checksum:

```
header_start:
    dd 0xe85250d6                ; magic number
    dd 0                         ; protected mode code
    dd header_end - header_start ; header length

    ; checksum
    dd 0x100000000 - (0xe85250d6 + 0 + (header_end - header_start))
header_end:
```

Again, we'll use math to let the computer calculate the sum for us. We add up the magic number, the mode code, and the header length, and then subt it from a big number. `dd` then puts that value into this spot in our file.

> **By the way...**
>
> You might wonder why we're subtracting these values from 0x100000000. To answer this we can look at what the multiboot spec says about the checksum value in the header:
>
> > The field `checksum` is a 32-bit unsigned value which, when added to the other magic fields (i.e. `magic`, `architecture` and `header_length`), must have a 32-bit unsigned sum of zero.
>
> In other words:
>
> `checksum` + `magic_number` + `architecture` + `header_length` = 0
>
> We could try and "solve for" `checksum` like so:
>
> `checksum` = -(`magic_number` + `architecture` + `header_length`)
>
> But here's where it gets weird. Computers don't have an innate concept of negative numbers. Normally we get around this by using "signed integers", which is something we cover in an appendix. The point is we have an unsigned integer here, which means we're limited to representing only positive numbers. This means we can't literally represent -(`magic_number` + `architecture` + `header_length`) in our field.

If you look closely at the spec you'll notice it's strangely worded: it's asking for a value that when added to other values has a sum of zero. It's worded this way because integers have a limit to the size of numbers they can represent, and when you go over that size, the values wrap back around to zero. So 0xFFFFFFFF + 1 is.... 0x00000000. This is a hardware limitation: technically it's doing the addition correctly, giving us the 33-bit value 0x100000000, but we only have 32 bits to store things in so it can't actually tell us about that `1` in the most significant digit position! We're left with the rest of the digits, which spell out zero.

So what we can do here is "trick" the computer into giving us zero when we do the addition. Imagine for the sake of argument that `magic_number + architecture + header_length` somehow works out to be 0xFFFFFFFE. The number we'd add to that in order to make 0 would be 0x00000002. This is 0x100000000-0xFFFFFFFE, because 0x100000000 technically maps to 0 when we wrap around. So we replace 0xFFFFFFFE in our contrived example here with `magic_number + architecture + header_length`. This gives us: `dd 0x100000000 - (0xe85250d6 + 0 + (header_end - header_start))`

Ending tag

After the checksum you can list a series of "tags", which is a way for the OS to tell the bootloader to do some extra things before handing control ove the OS, or to give the OS some extra information once started. We don't need any of that yet, though, so we just need to include the required "end t which looks like this:

```
header_start:
    dd 0xe85250d6                ; magic number
    dd 0                         ; protected mode code
    dd header_end - header_start ; header length

    ; checksum
    dd 0x100000000 - (0xe85250d6 + 0 + (header_end - header_start))

    ; required end tag
    dw 0    ; type
    dw 0    ; flags
    dd 8    ; size
header_end:
```

Here we use `dw` to define a 'word' instead of just data. Remember a 'word' is 16 bits or 2 bytes on the x86_64 architecture. The multiboot specifica demands that this be exactly a word. You'll find that this is super common in operating systems: the exact size and amount of everything matters. It's ju side-effect of working at a low level.

The Section

We have one last thing to do: add a 'section' annotation. We'll talk more about sections later, so for now, just put what I tell you at the top of the file.

Here's the final file:

```
section .multiboot_header
header_start:
    dd 0xe85250d6                ; magic number
    dd 0                         ; protected mode code
    dd header_end - header_start ; header length

    ; checksum
    dd 0x100000000 - (0xe85250d6 + 0 + (header_end - header_start))

    ; required end tag
    dw 0    ; type
    dw 0    ; flags
    dd 8    ; size
header_end:
```

That's it! Congrats, you've written a multiboot compliant header. It's a lot of esoterica, but it's pretty straightforward once you've seen it a few times.

Assembling with `nasm`

We can't use this file directly, we need to turn it into binary. We can use a program called an 'assembler' to 'assemble' our assembly code into binary cc It's very similar to using a 'compiler' to 'compile' our source code into binary. But when it's assembly, people often use the more specific name.

We will be using an assembler called `nasm` to do this. You should invoke `nasm` like this:

```
$ nasm -f elf32 multiboot_header.asm
```

The `-f elf32` says that we want to output a file as 32bit ELF.

After you run this command, you should see a `multiboot_header.o` file in the same directory. This is our 'object file', hence the `.o`. Don't let the word 'obj confuse you. It has nothing to do with anything object oriented. 'Object files' are just binary code with some metadata in a particular format - in our c ELF. Later, we'll take this file and use it to build our OS.

You can inspect the bytes of the header with `hexdump`

```
> hexdump -x multiboot_header
0000000    50d6    e852    0000    0000    0018    0000    af12    17ad
0000010    0000    0000    0008    0000
0000018
```

Summary

Congratulations! This is the first step towards building an operating system. We learned about the boot process, the GRUB bootloader, and the Multib specification. We wrote a Multiboot-compliant header file in assembly code, and used `nasm` to create an object file from it.

Next, we'll write the actual code that prints "Hello world" to the screen.

Hello, world!

Now that we've got the headers out of the way, let's do the traditional first program: Hello, world!

The smallest kernel

Our hello world will be just *20* lines of assembly code. Let's begin. Open a file called `boot.asm` and put this in it:

```
start:
    hlt
```

You've seen the `name:` form before: it's a *label*. This lets us name a line of code. We'll call this label `start`, which is the traditional name. GRUB will use convention to know where to begin.

The `hlt` statement is our first bit of 'real' assembly. So far, we had just been declaring data. This is actual, executable code. It's short for 'halt'. In o words, it ends the program.

By giving this line a label, we can call it, sort of like a function. That's what GRUB does: "Call the function named `start`." This function has just one l stop.

Unlike many other languages, you'll notice that there's no way to say if this 'function' takes any arguments or not. We'll talk more about that later.

This code won't quite work on its own though. We need to do a little bit more bookkeeping first. Here's the next few lines:

```
global start

section .text
bits 32
start:
    hlt
```

Three new bits of information. The first:

```
global start
```

This says "I'm going to define a label `start`, and I want it to be available outside of this file." If we don't say this, GRUB won't know where to finc definition. You can kind of think of it like a 'public' annotation in other languages.

```
section .text
```

We saw `section` briefly, but I told you we'd get to it later. The place where we get to it is at the end of this chapter. For the moment, all you need to kno this: code goes into a section named `.text`. Everything that comes after the `section` line is in that section, until another `section` line.

```
bits 32
```

GRUB will boot us into protected mode, aka 32-bit mode (similar to how xv6 bootloader starts in 16bit real mode GRUB will be loaded by the BIOS and switch into protected 32bit mode for us). But we have to specify directly that assembler has to generate 32bit code. Our Hello World will only be in 32 bit

That's it! We could theoretically stop here, but instead, let's actually print the "Hello world" text to the screen. We'll start off with an 'H':

```
global start

section .text
bits 32
start:
    mov word [0xb8000], 0x0248 ; H
    hlt
```

This new line is the most complicated bit of assembly we've seen yet. There's a lot packed into this little line.

The first important bit is `mov`. This is short for `move`, and it sorta looks like this:

```
mov size place, thing
```

Oh, `;` starts a comment, remember? So the `; H` is just for us. I put this comment here because this line prints an `H` to the screen!

Yup, it does. Okay, so here's why: `mov` copies `thing` into `place`. The amount of stuff it copies is determined by `size`.

```
;   size place      thing
;   |    |          |
;   V    V          V
mov word [0xb8000], 0x0248 ; H
```

"Copy one word: the number `0x0248` to ... some place.

The `place` looks like a number just like `0x0248`, but it has square brackets `[]` around it. Those brackets are special. They mean "the address in mem located by this number." In other words, we're copying the number `0x0248` into the specific memory location `0xb8000`. That's what this line does.

Why? Well, we're using the screen as a "memory mapped" device. Specific positions in memory correspond to certain positions on the screen. And position `0xb8000` is one of those positions: the upper-left corner of the screen.

> **By the way...**
>
> "Memory mapping" is one of the fundamental techniques used in computer engineering to help the CPU know how to talk to all the different physical components of a computer. The CPU itself is just a weird little machine that moves numbers around. It's not of any use to humans on its own: it needs to be connected to devices like RAM, hard drives, a monitor, and a keyboard. The way the CPU does this is through a *bus*, which is a huge pipeline of wires connecting the CPU to every single device that might have data the CPU needs. There's one wire per bit (since a wire can store a 1 or a 0 at any given time). A 32-bit bus is literally 32 wires in parallel that run from the CPU to a bunch of devices like Christmas lights around a house.
>
> There are two buses that we really care about in a computer: the address bus and the data bus. There's also a third signal that lets all the devices know whether the CPU is requesting data from an input (reading, like from the keyboard) or sending data to an output (writing, like to the monitor via the video card). The address bus is for the CPU to send location information, and the data bus is for the CPU to either write data to or read data from that location. Every device on the computer has a unique hard coded numerical location, or "address", literally determined by how the thing is wired up at the factory. In the case of an input/read operation, when it sends `0x1001A003` out on the address bus and the control signal notifies every device that it's a read operation, it's asking, "What is the data currently stored at location `0x1001A003`?" If the keyboard happens to be identified by that particular address, and the user is pressing SPACE at this time, the keyboard says, "Oh, you're talking to me!" and sends back the ASCII code `0x00000020` (for "SPACE") on the data bus.
>
> What this means is that memory on a computer isn't just representing things like RAM and your hard drive. Actual human-scale devices like the keyboard and mouse and video card have their own memory locations too. But instead of writing a byte to a hard drive for storage, the CPU might write a byte representing some color and symbol to the monitor for display. There's an industry standard somewhere that says video memory must live in the address range beginning `0xb8000`. In order for computers to be able to work out of the box, this means that the BIOS needs to be manufactured to assume video lives at that location, and the motherboard (which is where the bus is all wired up) has to be manufactured to route a `0xb8000` request to the video card. It's kind of amazing this stuff works at all! Anyway, "memory mapped hardware", or "memory mapping" for short, is the name of this technique.

Now, we are copying `0x0248`. Why this number? Well, it's in three parts:

```
 __ background color
/  __foreground color
| /
V V
0 2 48 <- letter, in ASCII
```

We'll start at the right. First, two numbers are the letter, in ASCII. `H` is 72 in ASCII, and 48 is 72 in hexadecimal: `(4 * 16) + 8 = 72`. So this will write `H`.

The other two numbers are colors. There are 16 colors available, each with a number. Here's the table:

```
| Value | Color          |
|-------|----------------|
| 0x0   | black          |
| 0x1   | blue           |
| 0x2   | green          |
| 0x3   | cyan           |
| 0x4   | red            |
| 0x5   | magenta        |
| 0x6   | brown          |
| 0x7   | gray           |
| 0x8   | dark gray      |
| 0x9   | bright blue    |
| 0xA   | bright green   |
| 0xB   | bright cyan    |
| 0xC   | bright red     |
```

```
| 0xD   | bright magenta |
| 0xE   | yellow         |
| 0xF   | white          |
```

So, `02` is a black background with a green foreground. Classic. Feel free to change this up, use whatever combination of colors you want!

So this gives us a `H` in green, over black. Next letter: `e`.

```
global start

section .text
bits 32
start:
    mov word [0xb8000], 0x0248 ; H
    mov word [0xb8002], 0x0265 ; e
    hlt
```

Lower case `e` is `65` in ASCII, at least, in hexadecimal. And `02` is our same color code. But you'll notice that the memory location is different.

Okay, so we copied four hexadecimal digits into memory, right? For our `H`, `0248`. A hexadecimal digit has sixteen values, which is 4 bits (for example, would be represented in bits as `1111`). Two of them make 8 bits, i.e. one byte. Since we need half a word for the colors (`02`), and half a word for the `H` that's one word in total (or two bytes). Each place that the memory address points to can hold one byte (a.k.a. 8 bits or half a word). Hence, if our memory position is at `0`, the second letter will start at `2`.

> You might be wondering, "If we're in 32 bit mode, isn't a word 32 bits?" since sometimes 'word' is used to talk about native CPU register size. Well, the 'word' keyword in the context of x86_64 assembly specifically refers to 2 bytes, or 16 bits of data. This is for reasons of backwards compatibility.

This math gets easier the more often you do it. And we won't be doing *that* much more of it. There is a lot of working with hex numbers in opera systems work, so you'll get better as we practice.

With this, you should be able to get the rest of Hello, World. Go ahead and try if you want: each letter needs to bump the location twice, and you nee look up the letter's number in hex.

If you don't want to bother with all that, here's the final code:

```
global start

section .text
bits 32
start:
    mov word [0xb8000], 0x0248 ; H
    mov word [0xb8002], 0x0265 ; e
    mov word [0xb8004], 0x026c ; l
    mov word [0xb8006], 0x026c ; l
    mov word [0xb8008], 0x026f ; o
    mov word [0xb800a], 0x022c ; ,
    mov word [0xb800c], 0x0220 ;
    mov word [0xb800e], 0x0277 ; w
    mov word [0xb8010], 0x026f ; o
    mov word [0xb8012], 0x0272 ; r
    mov word [0xb8014], 0x026c ; l
    mov word [0xb8016], 0x0264 ; d
    mov word [0xb8018], 0x0221 ; !
    hlt
```

Finally, now that we've got all of the code working, we can assemble our `boot.asm` file with `nasm`, just like we did with the `multiboot_header.asm` file:

```
$ nasm -f elf32 boot.asm
```

This will produce a `boot.o` file. We're almost ready to go!

Linking it together

Okay! So we have two different `.o` files: `multiboot_header.o` and `boot.o`. But what we need is *one* file with both of them. Our OS doesn't have the ability to anything yet, let alone load itself in two parts somehow. We just want one big binary file.

Enter 'linking'. If you haven't worked in a compiled language before, you probably haven't had to deal with linking before. Linking is how we'll turn these files into a single output: by linking them together.

Open up a file called `linker.ld`and put this in it:

```
ENTRY(start)

SECTIONS {
```

```
  . = 0x100000; /* Tells GRUB to load the kernel starting at the 1MB mark */

  .rodata :
  {
    /* ensure that the multiboot header is at the beginning */
    KEEP(*(.multiboot_header))
    *(.rodata .rodata.*)
    . = ALIGN(4K);
  }

  .text :
  {
    *(.text .text.*)
    . = ALIGN(4K);
  }

  .data :
  {
    *(.data .data.*)
    . = ALIGN(4K);
  }

  .bss :
  {
    *(.bss .bss.*)
    . = ALIGN(4K);
  }
}
```

This is a 'linker script'. It controls how our linker will combine these files into the final output. Let's take it bit-by-bit:

```
ENTRY(start)
```

This sets the 'entry point' for this executable. In our case, we called our entry point by the name people use: `start`. Remember? In `boot.asm`? Same na here.

```
SECTIONS {
```

Okay! I've been promising you that we'd talk about sections. Everything inside of these curly braces is a section. We annotated parts of our code sections earlier, and here, in this part of the linker script, we will describe each section by name and where it goes in the resulting output.

```
. = 0x100000;
```

This line means that we will start putting sections at the one megabyte mark. This is the conventional place to put a kernel, at least to start. Below megabyte is all kinds of memory-mapped stuff. Remember the VGA stuff? It wouldn't work if we mapped our kernel's code to that part of memo garbage on the screen!

```
.boot :
```

This will create a section named `boot`. And inside of it...

```
*(.multiboot_header)
```

... goes every section named `multiboot_header`. Remember how we defined that section in `multiboot_header.asm`? It'll be here, at the start of the `boot` sect That's what we need for GRUB to see it.

```
.text :
```

Next, we define a `text` section. The `text` section is where you put code. And inside of it...

```
*(.text)
```

... goes every section named `.text`. See how this is working? The syntax is a bit weird, but it's not too bad.

We do the same for the `code` and `bss` section.

That's it for our script! We can then use `ld` to link all of this stuff together:

```
$ ld -m elf_i386 -T linker.ld -o kernel.bin multiboot_header.o boot.o
```

Recall that on Mac OS X you will want to use the linker we installed to `~/opt` and not your system linker. For example, if you did not change any of defaults in the installation script, this linker will be located at `$HOME/opt/bin/x86_64-pc-elf-ld`.

By running this command, we do a few things:

```
-m elf_i386
```

ask the linker to generate the 32bit.

```
-T linker.ld
```

This is the linker script we just made, we ask the linker to use it.

```
-o kernel.bin
```

This sets the name of our output file. In our case, that's `kernel.bin`. We'll be using this file in the next step. It's our whole kernel!

```
multiboot_header.o boot.o
```

Finally, we pass all the `.o` files we want to link together.

That's it! We've now got our kernel in the `kernel.bin` file. Next, we're going to make an ISO out of it, so that we can load it up in QEMU.

Making an ISO

Now that we have our `kernel.bin`, the next step is to make an ISO. Remember compact discs? Well, by making an ISO file, we can both test our H World kernel in QEMU, as well as running it on actual hardware!

To do this, we're going to use a GRUB tool called `grub2-mkrescue`. We have to create a certain structure of files on disk, run the tool, and we'll get an `os` file at the end.

Doing so is not very much work, but we need to make the files in the right places. First, we need to make several directories:

```
$ mkdir -p build/isofiles/boot/grub
```

The `-p` flag to `mkdir` will make the directory we specify, as well as any 'parent' directories, hence the `p`. In other words, this will make an `build` directory wi `isofiles` directory inside that has `boot` inside, and finally the `grub` directory inside of that.

Next, create the `grub.cfg` file inside of that `build/isofiles/boot/grub` directory, and put this in it:

```
set timeout=0
set default=0

menuentry "cs143Aos" {
    multiboot2 /boot/kernel.bin
    boot
}
```

This file configures GRUB. Let's talk about the `menuentry` block first. GRUB lets us load up multiple different operating systems, and it usually does thi displaying a menu of OS choices to the user when the machine boots. Each `menuentry` section corresponds to one of these. We give it a name, in this c `cs143Aos`, and then a little script to tell it what to do. First, we use the `multiboot2` command to point at our kernel file. In this case, that locatio `/boot/kernel.bin`. Remember how we made a `boot` directory inside of `isofiles`? Since we're making the ISO out of the `isofiles` directory, everything insid it is at the root of our ISO. Hence `/boot`.

Let's copy our `kernel.bin` file there now:

```
$ cp kernel.bin build/isofiles/boot/
```

Finally, the `boot` command says "that's all the configuration we need to do, boot it up."

But what about those `timeout` and `default` settings? Well, the `default` setting controls which `menuentry` we want to be the default. The numbers start at z and since we only have that one, we set it as the default. When GRUB starts, it will wait for `timeout` seconds, and then choose the `default` option if the u didn't pick a different one. Since we only have one option here, we just set it to zero, so it will start up right away.

The final layout should look like this:

```
build/
```

```
|---isofiles/
    |---boot
        |-- grub
        |   |-- grub.cfg
        |-- kernel.bin
```

Using `grub2-mkrescue` is easy. We run this command:

```
$ grub2-mkrescue -o hello.iso build/isofiles
```

The `-o` flag controls the *o*utput filename, which we choose to be `os.iso`. And then we pass it the directory to make the ISO out of, which is the `build/isof` directory we just set up.

After this, you have an `os.iso` file with our teeny kernel on it. You could burn this to a USB stick or CD and run it on an actual computer if you wanted to! doing so would be really annoying during development. So in the next section, we'll use an emulator, QEMU, to run the ISO file on our developm machine.

Running in QEMU

Let's actually run our kernel! To do this, we'll use QEMU, a full-system emulator. Using QEMU is fairly straightfoward. If you're running on openlab you c really connect to the GUI application, so we'll use `-curses`:

```
$ qemu-system-x86_64 -curses -cdrom hello.iso
```

Type it in, hit Enter, and you should see `Hello, world!` (To exit, hit `Esc+2` and type `quit` in the console.)

If you're running on your own machine you can simply run:

```
$ qemu-system-x86_64 -cdrom hello.iso
```

You should see something what really looks like a screen of the computer with `Hello, world!` (To exit, hit `Alt+2` and type `quit` in the console.)

If it shows up for you too, congrats! If not, something may have gone wrong. Double check that you followed the examples *exactly*. Maybe you mis something, or made a mistake while copying things down.

Note all of this other stuff behind the Hello World message: this part may look different, based on your version of GRUB, and also since we didn't clear screen, everything from GRUB just stays as it is. We'll write a function to do that eventually...

Let's talk about this command before we move on:

```
qemu-system-x86_64
```

We're running the `x86_64` variant of QEMU. While we have a 32-bit kernel the QEMU emulates x86 64bit architecture. And since 32bit code is part everything works.

```
-cdrom hello.iso
```

We're going to start QEMU with a CD-ROM drive, and its contents are the `hello.iso` file we made.

That's it! Here's the thing, though: while that wasn't *too* complicated, it was a lot of steps. Each time we make a change, we have to go through all th steps over again. In the next section, we'll use Make to do all these steps for us.

Automation with Make

Typing all of these commands out every time we want to build the project is tiring and error-prone. It's nice to be able to have a single command that bu our entire project. To do this, we'll use `make`. Download this Makefile and look over it.

The makefile starts by defining several variables `kernel`, `iso`, `linker_script`, and `grub_cfg` that define names of the output files we want to make. `CFLAGS` variable that defines all flags to the GCC compiler.

```
kernel := build/kernel.bin
iso := build/hello.iso

linker_script := linker.ld
grub_cfg := boot/grub.cfg

CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O1 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer -Werror

target ?= hello
```

We then create two lists: a list of assembly files in the folder `assembly_source_files` and a list of C source files, `c_source_files`. We then use the `pats` command to generate another two lists that are the same file names but with `.o` as extension:

```
assembly_source_files := $(wildcard *.asm)
assembly_object_files := $(patsubst %.asm, build/%.o, $(assembly_source_files))
c_source_files := $(wildcard *.c)
c_object_files := $(patsubst %.c, build/%.o, $(c_source_files))
```

```
.PHONY: all clean qemu qemu-nox qemu-gdb qemu-gdb-nox

all: $(kernel)

clean:
	rm -r build

qemu: $(iso)
	qemu-system-x86_64 -cdrom $(iso) -vga std -s -serial file:serial.log

qemu-nox: $(iso)
	qemu-system-x86_64 -m 128 -cdrom $(iso) -vga std -s -no-reboot -nographic

qemu-gdb: $(iso)
	qemu-system-x86_64 -S -m 128 -cdrom $(iso) -vga std -s -serial file:serial.log -no-reboot -no-shutdown -d int,cpu_reset

.PHONY: qemu-gdb-nox
qemu-gdb-nox: $(iso)
	qemu-system-x86_64 -S -m 128 -cdrom $(iso) -vga std -s -serial file:serial.log -no-reboot -no-shutdown -d int,cpu_reset -nograph

iso: $(iso)
	@echo "Done"

$(iso): $(kernel) $(grub_cfg)
	@mkdir -p build/isofiles/boot/grub
	cp $(kernel) build/isofiles/boot/kernel.bin
	cp $(grub_cfg) build/isofiles/boot/grub
	grub2-mkrescue -o $(iso) build/isofiles #2> /dev/null
	@rm -r build/isofiles

$(kernel): $(c_object_files) $(assembly_object_files) $(linker_script)
	ld -m elf_i386  -T $(linker_script) -o $(kernel) $(assembly_object_files) $(c_object_files)

# compile C files
build/%.o: %.c
	@mkdir -p $(shell dirname $@)
	gcc $(CFLAGS) -c $< -o $@

# compile assembly files
build/%.o: %.asm
	@mkdir -p $(shell dirname $@)
	nasm -felf32 $< -o $@
```

Our default action is `all` (it will build the kernel by invoking the linker). Of course before linking the kernel, all object files have to be compiled.

Also it's nice to add targets which describe a specific actions. To run the kernel we add a rule

```
qemu: $(iso)
	qemu-system-x86_64 -cdrom $(iso) -vga std -s -serial file:serial.log
```

Finally, there's another useful common rule: `clean`. The `clean` rule should remove all of the generated files, and allow us to do a full re-build.

Now there's just one more wrinkle. We have four targets that aren't really files on disk, they are just actions: `default`, `build`, `run` and `clean`. Remember said earlier that `make` decides whether or not to execute a command by comparing the last time a target was built with the last-modified-time o prerequisites? Well, it determines the last time a target was built by looking at the last-modified-time of the target file. If the target file doesn't exist, ther definitely out-of-date so the command will be run.

But what if we accidentally create a file called `clean`? It doesn't have any prerequisites so it will always be up-to-date and the commands will never be We need a way to tell `make` that this is a special target, it isn't really a file on disk, it's an action that should always be executed. We can do this with a m built-in target called `.PHONY`:

```
.PHONY: default build run clean
```

Paging

Up until now we did a lot of work that wasn't actually writing kernel code. So let's review what we're up to:

1. GRUB loaded our kernel, and started running it.
2. We're currently running in 'protected mode', a 32-bit environment.
3. We want to enable paging.
4. We want to create a stack and jump into `main`.

We're on step three. More specifically, here's what we have to do:

1. Set up something called 'paging'.
2. Set up something called a 'GDT'.
3. Call into main.

Paging

Paging is actually implemented by a part of the CPU called an 'MMU', for 'memory management unit'. The MMU will translate virtual addresses into t respective physical addresses automatically; we can write all of our software with virtual addresses only. The MMU does this with a data structure calle 'page table'. As an operating system, we load up the page table with a certain data structure, and then tell the CPU to enable paging. This is the t ahead of us; it's required to set up paging before we transition to long mode.

How should we do our mapping of physical to virtual addresses? You can make this easy, or complex, and it depends on exactly what you want your O be good at. Some strategies are better than others, depending on the kinds of programs you expect to be running. We're going to keep it simple, and us strategy called 'identity mapping'. This means that every virtual address will map to a physical address of the same number. Nothing fancy.

Let's talk more about the page table. In 32bit mode, the page table is two levels deep, and each page is 4096 bytes in size. What do I mean by lev Here are the official names:

- Page-Directory Table (PD)
- Page Table (PT)

Creating the page table

So here's the strategy: create a single entry of each of these tables, then point them at each other in the correct way, then tell the CPU that paging sh be enabled.

Creating page table entries

To create space for these page table entries, open up `boot.asm` and add these lines at the bottom:

```
section .bss

align 4096

pt_table:
    resb 4096
ptd_table:
    resb 4096
```

We introduce a new section, 'bss'. It stands for 'block started by symbol', and was introduced in the 1950s. The name doesn't make much sense anym but the reason we use it is because of its behavior: entries in the bss section are automatically set to zero by the linker. This is useful, as we only w certain bits set to 1, and most of them set to zero.

The `resb` directive reserves bytes; we want to reserve space for each entry.

The `align` directive makes sure that we've aligned our tables properly. We haven't talked much about alignment yet: the idea is that the addresses here be set to a multiple of 4096, hence 'aligned' to 4096 byte chunks. We'll eventually talk more about alignment and why it's important, but it doesn't matt ton right now.

After this has been added, we have a single valid entry for each level. However, because our page four entry is all zeroes, we have no valid pages. Th not super useful. Let's set things up properly.

Pointing the entries at each other

In order to do this setup, we need to write some more assembly code! Open up `boot.asm`. You can either leave in printing code, or remove it. If you do le it in, add this code before it: that way, if you see your message print out, you know it ran successfully.

```

global start

section .text
bits 32
start:
    ; map first PTD entry to PT table
    mov eax, pt_table
    or eax, 0b11 ; present + writable
    mov [ptd_table], eax
```

If you recall, ; are comments. Leaving yourself excessive comments in assembly files is a good idea. Let's go over each of these lines:

```
    mov eax, pt_table
```

This copies the contents of the first page table entry into the `eax` register. We need to do this because of the next line:

```
    or eax, 0b11
```

We take the contents of `eax` and `or` it with `0b11`, the result is written in `eax`.

Each entry in a page table contains an address, but it also contains metadata about that page. The first two bits are the 'present bit' and the 'writable bit' By setting the first bit, we say "this page is currently in memory," and by setting the second, we say "this page is allowed to be written to." There are a number of other settings we can change this way, but they're not important for now.

> **By the way...**
>
> You might be wondering, if the entry in the page table is an address, how can we use some of the bits of that address to store metadata without messing up the address? Remember that we used the `align` directive to make sure that the page tables all have addresses that are multiples of 4096. That means that the CPU can assume that the first 12 bits of all the addresses are zero. If they're always implicitly zero, we can use them to store metadata without changing the address.

Now that we have an entry set up properly, the next line is of interest:

```
    mov dword [ptd_table + 0], eax
```

Another `mov` instruction, but this time, copying `eax`, where we've been setting things up, into... something in brackets. `[]` means, "I will be giving you address between the brackets. Please do something at the place this address points." In other words, `[]` is like a dereference operator.

Now, the address we've put is kind of funny looking: `ptd_table + 0`. What's up with that `+ 0`? It's not strictly needed: adding zero to something keeps it same. However, it's intended to convey to the reader that we're accessing the zeroth entry in the page table. We're about to see some more code l where we will do something other than add zero, and so putting it here makes our code look more symmetric overall. If you don't like this style, you d have to put the zero.

These few lines form the core of how we're setting up these page tables. We're going to do the same thing over again, with slight variations.

Here's the full thing again:

```
    ; map first PTD entry to PT table
    mov eax, pt_table
    or eax, 0b11 ; present + writable
    mov [ptd_table], eax
```

We have one last thing to do: set up the level two page table to have valid references to pages. We're going to do something we haven't done ye assembly: write a loop!

Here's the basic outline of loop in assembly:

- Create a counter variable to track how many times we've looped
- make a label to define where the loop starts
- do the body of the loop
- add one to our counter
- check to see if our counter is equal to the number of times we want to loop
- if it's not, jump back to the top of the loop
- if it is, we're done

It's a little more detail-oriented than loops in other languages. Usually, you have curly braces or indentation to indicate that the body of the loop is separ but we don't have any of those things here. We also have to write the code to increment the counter, and check if we're done. Lots of little fiddly bits. that's the nature of what we're doing!

Let's get to it!

```
    ; map each P1 entry to a 4KB page
    mov ecx, 0          ; counter variable

.map_pt_table:
    ; map ecx-th PT entry to a huge page that starts at address 4KB*ecx
    mov eax, 0x1000     ; 4KB
    mul ecx             ; start address of ecx-th page
    or eax, 0b00000011 ; present + writable
    mov [pt_table + ecx * 4], eax ; map ecx-th entry

    inc ecx             ; increase counter
```

```
    cmp ecx, 1024      ; if counter == 1024, the whole P1 table is mapped
    jne .map_pt_table  ; else map the next entry
```

In order to write a loop, we need a counter. `ecx` is the usual loop counter register, that's what the `c` stands for: counter. We also have a comment indica what we're doing in this part of the code.

Next, we need to make a new label:

```
.map_pt_table:
```

As we mentioned above, this is where we will loop back to when the loop continues.

```
    mov eax, 0x1000
```

We're going to store 0x1000 in `eax`, or 4096 which is 4KB. Here's the reason: each page is 4KB in size. So in order to get the right memory location, we multiply the number of the loop counter by 0x1000

Here's that multiplication! `mul` takes just one argument, which in this case is our `ecx` counter, and multiplies that by `eax`, storing the result in `eax`. This wil the location of the next page.

```
    or eax, 0b00000011
```

Next up, our friend `or`. Here, we again set present and writable bits `0b11`.

```
    mov [pt_table + ecx * 4], eax
```

Just like before, we are now writing the value in `eax` to a location. But instead of it being just `pt_table + 0`, we're adding `ecx * 4`. Remember, `ecx` is our l counter. Each entry is eight bytes in size, so we need to multiply the counter by four, and then add it to `pt_table`.

That's the body of the loop! Now we need to see if we need to keep looping or not:

```
    inc ecx
    cmp ecx, 1024
    jne .map_pt_table
```

The `inc` instruction increments the register it's given by one. `ecx` is our loop counter, so we're adding to it. Then, we 'compare' with `cmp`. We're comparing with 1024: we want to map 1024 page entries overall. The page table is 4096 bytes, each entry is 4 bytes, so that means there are 1024 entries. This give us 1024 * 4KB: four megabytes of memory. It's also why we wrote the loop: writing out 1024 entries by hand is possible, theoretically, but is not Let's make the computer do the math for us.

The `jne` instruction is short for 'jump if not equal'. It checks the result of the `cmp`, and if the comparison says 'not equal', it will jump to the label we've defir `map_pt_table` points to the top of the loop.

That's it! We've written our loop and mapped our second-level page table. Here's the full code, all in one place:

```
    ; map first PTD entry to PT table
    mov eax, pt_table
    or eax, 0b11 ; present + writable
    mov [ptd_table + 0], eax

    ; map each P1 entry to a 4KB page
    mov ecx, 0         ; counter variable

.map_pt_table:
    ; map ecx-th PT entry to a huge page that starts at address 4KB*ecx
    mov eax, 0x1000    ; 4KB
    mul ecx            ; start address of ecx-th page
    or eax, 0b00000011 ; present + writable
    mov [pt_table + ecx * 4], eax ; map ecx-th entry

    inc ecx            ; increase counter
    cmp ecx, 1024      ; if counter == 1024, the whole P1 table is mapped
    jne .map_pt_table  ; else map the next entry
```

Now that we've done this, we have a valid initial page table. Time to enable paging!

Enable paging

Now that we have a valid page table, we need to inform the hardware about it. Here's the steps we need to take:

- We have to put the address of the level four page table in a special register

- enable paging

These steps are not particularly interesting, but we have to do them. First, let's do the first step:

```
    ; move page table address to cr3
    mov eax, ptd_table
    mov cr3, eax
```

So, this might seem a bit redundant: if we put `ptd_table` into `eax`, and then put `eax` into `cr3`, why not just put `ptd_table` into `cr3`? As it turns out, `cr3` is a spe register, called a 'control register', hence the `cr`. The `cr` registers are special: they control how the CPU actually works. In our case, the `cr3` register need hold the location of the page table.

Because it's a special register, it has some restrictions, and one of those is that when you `mov` to `cr3`, it has to be from another register. So we need the `mov` to set `p4_table` in a register before we can set `cr3`.

Step one: done!

Finally we are all ready to enable paging!

```
    ; enable paging
    mov eax, cr0
    or eax, 1 << 31
    mov cr0, eax
```

`cr0` is the register we need to modify. We do the usual "move to `eax`, set some bits, move back to the register" pattern. In this case, we set bit 31.

Once we've set these bits, we're done! Here's the full code listing:

```
    ; move page table address to cr3
    mov eax, ptd_table
    mov cr3, eax

    ; enable paging
    mov eax, cr0
    or eax, 1 << 31
    mov cr0, eax
```

Setting up a GDT

The GDT is used for a style of memory handling called 'segmentation', which is in contrast to the paging model that we just set up. Even though we're using segmentation, however, we're still required to have a valid GDT. Such is life.

So let's set up a minimal GDT. Our GDT will have three entries:

- a 'zero entry'
- a 'code segment'
- a 'data segment'

If we were going to be using the GDT for real stuff, it could have a number of code and data segment entries. But we need at least one of each to hav minimum viable table, so let's get to it!

The Zero entry

The first entry in the GDT is special: it needs to be a zero value. Add this to the bottom of `boot.asm`:

```
section .rodata
gdt_start: ; don't remove the labels, they're needed to compute sizes and jumps
    ; the GDT starts with a null 8-byte
    dd 0x0 ; 4 byte
    dd 0x0 ; 4 byte
```

We have a new section: `rodata`. This stands for 'read only data', and since we're not going to modify our GDT, having it be read-only is a good idea.

Next, we have a label: `gdt`. We'll use this label later, to tell the hardware where our GDT is located.

Finally, `dd 0`. This defines a 'doubleword' value, in other words, a 32-bit value. Given that it's a zero entry, it shouldn't be too surprising that the value of entry is zero!

That's all there is to it.

Setting up code and data segments

Next, we need a code segment. Add this below the zero entry that you created above

```
; GDT for code segment. base = 0x00000000, length = 0xfffff
; for flags, refer to os-dev.pdf document, page 36
gdt_code:
    dw 0xffff    ; segment length, bits 0-15
    dw 0x0       ; segment base, bits 0-15
    db 0x0       ; segment base, bits 16-23
    db 10011010b ; flags (8 bits)
    db 11001111b ; flags (4 bits) + segment length, bits 16-19
    db 0x0       ; segment base, bits 24-31
```

Here we carefully follow the layout of the segment descriptor to make sure that all flags are set correctly (the exact layout can be found in the I Developer's Manual, 3.4.5). It would be nice to use a human-friendly macro like xv6 does, but I'm not good with macros in NASM.

Similar, we define the data segment.

```
; GDT for data segment. base and length identical to code segment
; some flags changed, again, refer to os-dev.pdf
gdt_data:
    dw 0xffff
    dw 0x0
    db 0x0
    db 10010010b
    db 11001111b
    db 0x0

gdt_end:
```

Finally, we have to define the GDT descriptor, the special data structure that defines the size of the GDT and has a pointer to it in memory. We compute size as the difference between two labels: `gdt_end` and `gdt_start`. The pointer itself is the value of the label where GDT starts, i.e., `gdt_start`.

```
; GDT descriptor
gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; size (16 bit), always one less of its true size
    dd gdt_start ; address (32 bit)
```

Finally, lets define two constants for the code and data segments that we can load in registers

```
; define some constants for later use
CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

Note that here since each descriptor was 8 bytes long the code segment is defined as 0x8 and the data segment is 0x9. This is exactly what we want si the first 3 bits of the segment selector (3.4.2 in the Intel SDM) are used for flags.

Now we can add this code somewhere at the top to make sure that all segment registers are initialized correctly.

```
; load 0 into all data segment registers
mov ax, DATA_SEG
mov ss, ax
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax
```

Load the GDT

So! We're finally ready to tell the hardware about our GDT. Add this line after all of the paging stuff we did in the last section:

```
    lgdt [gdt_descriptor]
```

We pass `lgdt` the value of our `gdt_descriptor` label. `lgdt` stands for 'load global descriptor table'. That's it!

We have all of the prerequisites done! In the next section, we will complete our boot sequence by entering `main()`.

Setting stack and calling main()

Now we're one step away from calling the `main` function. We need to set up the stack. First, lets reserve 4096 bytes (one page) for the stack in the B section of the `boot.asm` file:

```
stack_bottom:
    resb 4096 * 4 ; Reserve this many bytes
stack_top:
```

Now we can load the address of the stack into the `esp` register and call the main function:

```
    mov esp, stack_top
    call main
```

We need to know the assembler that main will be defined in a different object file (we'll compile it from `main.c`). For that we need to add this line right at tl top of `boot.asm`

```
extern main
```

Compiling main()

Download the skeleton for the main.c, console.c, and console.h files. A minimal `main()` function can look something like this:

```
#include "console.h"

int main(void)
{
    // Initialize the console
    uartinit();

    printk("Hello from C\n");

    return 0;
}
```

It calls the `uartinit()` function to initialize the serial line and then prints "Hello from C" on the serial line.

Serial ports are a legacy communications port common on IBM-PC compatible computers. Use of serial ports for connecting peripherals has largely b deprecated in favor of USB and other modern peripheral interfaces, however it is still commonly used in certain industries for interfacing with indus hardware such as CNC machines or commercial devices such as POS terminals. Historically it was common for many dial-up modems to be connected a computer's serial port, and the design of the underlying UART hardware itself reflects this.

Serial ports are typically controlled by UART hardware. This is the hardware chip responsible for encoding and decoding the data sent over the se interface. Modern serial ports typically implement the RS-232 standard, and can use a variety of different connector interfaces. The DE-9 interface is one most commonly used connector for serial ports in modern systems.

Serial ports are of particular interest to operating-system developers since they are much easier to implement drivers for than USB, and are still comm found in many x86 systems. It is common for operating-system developers to use a system's serial ports for debugging purposes, since they do not req sophisticated hardware setups and are useful for transmitting information in the early stages of an operating-system's initialization. Many emulators such QEMU and Bochs allow the redirection of serial output to either stdio or a file on the host computer.

Why Use a Serial Port?

During the early stages of kernel development, you might wonder why you would bother writing a serial driver. There are several reasons why you might

- GDB debugging You can use the serial port to connect to a host computer, and use the GDB debugger to debug your operating system. This invol writing a stub for GDB within your OS.
- Headless console You can operate the computer without a monitor, keyboard or mouse and instead use the serial port as a console using a protoc such as TTY or VT100.
- External logging When the system itself is in danger of potentially crashing at times, it's nice to get debugging outputs safe to another computer before the test system triple-faults.
- Networking and File transfers Serial ports are useful for transferring information between systems when other more traditional methods are unavailable.

Serial line driver

To print something on the serial line we need to implement a minimal serial line driver. In this homework assignment we provide you a simple serial drive `console.c`. It still makes sense for you to look over the page that describes the details of the serial line protocol Serial Ports @ OSDev.org. At a high l we define which I/O port serial line is connected to:

```
#define COM1    0x3f8
```

We then use a couple of helper functions that provide the interface to assembly `in` and `out` instructions.

```
static inline unsigned char inb(unsigned short port)
{
    unsigned char data;

    asm volatile("in %1,%0" : "=a" (data) : "d" (port));
    return data;
```

```
}

static inline void outb(unsigned short port, unsigned char data)
{
    asm volatile("out %0,%1" : : "a" (data), "d" (port));
}
```

We then use the `uartinit()` function to initialize the serial line interface

```
void uartinit(void)
{

  // Turn off the FIFO
  outb(COM1+2, 0);

  // 9600 baud, 8 data bits, 1 stop bit, parity off.
  outb(COM1+3, 0x80);    // Unlock divisor
  outb(COM1+0, 115200/115200);
  outb(COM1+1, 0);
  outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
  outb(COM1+4, 0);
  outb(COM1+1, 0x01);    // Enable receive interrupts.

  // If status is 0xFF, no serial port.
  if(inb(COM1+5) == 0xFF)
      return;

  uart = 1;

  // Acknowledge pre-existing interrupt conditions;
  // enable interrupts.
  inb(COM1+2);
  inb(COM1+0);
}
```

The `uartputc()` displays an individual character on the screen

```
void uartputc(int c)
{
  int i;

  if(!uart)
      return;

  for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
      microdelay(10);

  outb(COM1+0, c);
}
```

And finally the `printk()` function prints a string on the screen

```
void printk(char *str)
{
    int i, c;

    for(i = 0; (c = str[i]) != 0; i++){
        uartputc(c);
    }
}
```

Booting into C

Now we're finally ready to boot into C. If you put all the files in the correct places, you can run make and get "Hello from C" on the serial line. The serial is configured to be recorded in the `serial.log` file.

```
make qemu
```

Remember if you're using your own code, disable the `-curses` flag in the Makefile. And finally if you want to see only the console (not the VGA) you can r

```
make qemu-nox
```

Implementing the page table

Finally, your assignment is to implement page table that maps the first 8MB of virtual addresses to the first 8MB of physical memory. At the moment have a page table that maps first 4MB, you need to define and construct a new page table once you boot into `main()`.

Once your page table is constructed use the provided `lcr3()` function to load it into the `CR3` register. Note, you have to load the physical address of the p table.

Extra credit: (10% bonus)

Implement support for mapping 256MBs of physical memory with 4KB pages.

Extra credit: (10% bonus)

Implement a simple VGA driver, i.e., when you use the `printk()` it should print on both serial line like now and on the VGA screen.

Extra credit: (5% bonus)

Boot on real hardware. I.e., try booting your code on a real desktop or laptop by either burning a CD-ROM or a USB flash drive. Record a video of y code booting.

Submit your work

Submit your solution through Gradescope Gradescope CS143A Principles of Operating Systems. **Please zip all of your files and submit them. If have done extra credit then place files required for extra credit part into separate folders extra1, extra2 or extra3. The structure of the zip should be the following:**

```
/
  - Makefile
  - console.c
  - console.h
  - main.c
  - ...                          -- any other files required to start
  - /extra1                      -- optional
    - Makefile
    - console.c
    - console.h
    - main.c
    - ...
  - /extra2                      -- optional
    - Makefile
    - console.c
    - console.h
    - main.c
    - ...
  - /extra3                      -- optional
    - Video or a textfile with link to a video (no Rick Roll please)
```

Updated