This lab is split into three parts. The first part concentrates on getting familiarized with x86 assembly language, the QEMU x86 emulator, and the PC's power-on bootstrap procedure. The second part examines the boot loader for our 6.828 kernel, which resides in the boot directory of the lab tree. Finally, the third part delves into the initial template for our 6.828 kernel itself, named JOS, which resides in the kernel directory.

Part 1: PC Bootstrap

Simulating the x86
Instead of developing the operating system on a real, physical personal computer (PC), we use a program that faithfully emulates a complete PC: the code you write for the emulator will boot on a real PC too. Using an emulator simplifies debugging; you can, for example, set break points inside of the emulated x86, which is difficult to do with the silicon version of an x86.

In 6.828 we will use the QEMU Emulator, a modern and relatively fast emulator. While QEMU's built-in monitor provides only limited debugging support, QEMU can act as a remote debugging target for the GNU debugger (GDB), which we'll use in this lab to step through the early boot process.

To get started, extract the Lab 1 files into your own directory on Athena as described above in "Software Setup", then type make (or gmake on BSD systems) in the lab directory to build the minimal 6.828 boot loader and kernel you will start with.

Now you're ready to run QEMU, supplying the file obj/kern/kernel.img, created above, as the contents of the emulated PC's "virtual hard disk." This hard disk image contains both our boot loader (obj/boot/boot) and our kernel (obj/kernel).

athena% make qemu-nox
This executes QEMU with the options required to set the hard disk and direct serial port output to the terminal.

Everything after 'Booting from Hard Disk...' was printed by our skeletal JOS kernel; the K> is the prompt printed by the small monitor, or interactive control program, that we've included in the kernel. If you used make qemu, these lines printed by the kernel will appear in both the regular shell window from which you ran QEMU and the QEMU display window. This is because for testing and lab grading purposes we have set up the JOS kernel to write its console output not only to the virtual VGA display (as seen in the QEMU window), but also to the simulated PC's virtual serial port, which QEMU in turn outputs to its own standard output. Likewise, the JOS kernel will take input from both the keyboard and the serial port, so you can give it commands in either the VGA display window or the terminal running QEMU. Alternatively, you can use the serial console without the virtual VGA by running make qemu-nox.

There are only two commands you can give to the kernel monitor, help and kerninfo.
```
K> help
help - display this list of commands
kerninfo - display information about the kernel
K> kerninfo
Special kernel symbols:
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101a75 (virt)  00101a75 (phys)
```

```
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112960 (virt)  00112960 (phys)
Kernel executable memory footprint: 75KB
K>
```

The help command is obvious, and we will shortly discuss the meaning of what the kerninfo command prints. Although simple, it's important to note that this kernel monitor is running "directly" on the "raw (virtual) hardware" of the simulated PC. This means that you should be able to copy the contents of obj/kern/kernel.img onto the first few sectors of a real hard disk, insert that hard disk into a real PC, turn it on, and see exactly the same thing on the PC's real screen as you did above in the QEMU window.

POWER ON Procedure
The process begins when the power supply is switched on
The power supply performs a self-test:
When all voltages and current levels are acceptable (+5v, +3.0 through +6.0 is generally considered acceptable), the supply indicates that the power is stable and sends the "Power Good" signal to the motherboard.
The "Power Good" signal is received by the microprocessor timer chip, which controls the reset line to the microprocessor. The time between turning on the switch to the generation of the "Power Good" signal is usually between 0.1 and 0.5 seconds. In the absence of the "Power Good" signal, the timer chip continuously resets the microprocessor, which prevents the system from running under bad or unstable power conditions.
In the absence of the "Power Good" signal, the timer chip continuously resets the microprocessor, which prevents the system from running under bad or unstable power conditions
The microprocessor timer chip receives the "Power Good" signal:
After the power supply is switched on, the microprocessor timer chip generates a reset
signal to the processor (the same as if you held the reset button down for a while on
your case) until it receives the "Power Good" signal from the power supply.
After the reset signal turns off, the CPU begins to operate. Code in RAM cannot be executed since the RAM is empty. The CPU manufacturers pre-program the processor to always begin executing code at address "FFFF:0000" (usually the ROM BIOS) of the ROM
The CPU starts executing the ROM BIOS code:
The CPU loads and executes the ROM BIOS code starting at ROM memory address "FFFF:0000" which is only 16 bytes from the top of ROM memory. As such, it contains
only a JMP (jump) instruction that points to the actual address of the ROM BIOS code

The BIOS searches for adapters (usually video adapters) that may need to load their own ROM BIOS routines:
Video adapters provide the most common source of adapter ROM BIOS. The start-up BIOS routines scan memory addresses "C000:0000" through "C780:0000" to find video ROM.
An error loading any adapter ROM generates an error such as:"XXXX ROM Error" where XXXX represents the segment address of the failed module.
The ROM BIOS checks to see if this is a 'cold boot' or a 'warm boot': To determine whether this is a "cold boot" or a "warm boot" the ROM BIOS startup routines check the value of the two bytes located at memory location "0000:0472".
Warm boot - A word value of 1234h in this location is a flag that indicates a

*"warm
boot", which causes the memory test portion of the POST (Power-On Self-Test) to be skipped.
Cold boot - Any other word value in this location indicates a "cold boot" and full POST.*

*POST (Power-On Self-Test):The POST is a series of diagnostic tests that run automatically when you turn your computer on. The actual tests can differ depending on how the BIOS is configured, but usually the POST tests the following:
The Video adapter - It is initialized, the video card and video memory is tested, and
configuration information or any errors are displayed.
The RAM - A read/write test of each memory address is performed and a running sum of installed memory is displayed.
The keyboard - PS/2 ports or USB ports are checked to verify whether the keyboard is connected or not.
The Processor - The cache memory is checked and the CPU type and speed are displayed.
CMOS - Read/write test.
ROM BIOS checksum.
RAM refresh verification.*

*Any errors found during the POST are reported by a combination of beeps and displayed error messages. The errors which occur during the POST can be classified as either 'fatal' or 'non-fatal'. A non-fatal error (e.g. problem in the
extended memory) will typically display an error message on the screen and allow the system to continue the boot process. A fatal error (e.g. problem in the processor), on the other hand, stops the process of booting the computer and is generally signaled by a series of beep-codes. However, successful completion of the POST is indicated by a single beep.*

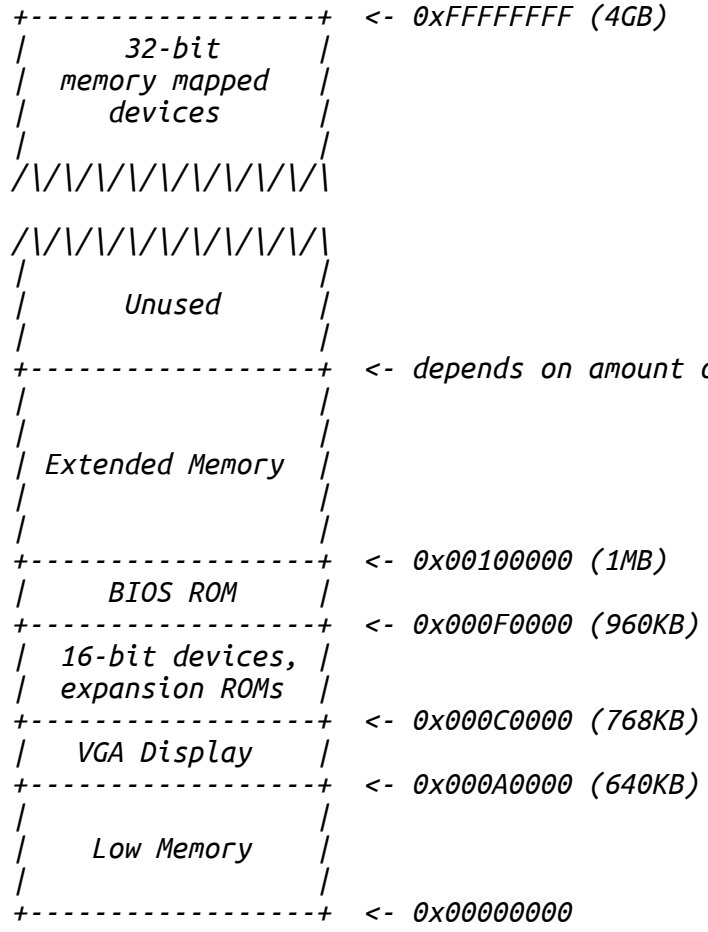
The BIOS locates and reads the configuration information stored in CMOS:CMOS (Complementary Metal-Oxide Semiconductor) is a small area of memory (64 bytes) which is maintained by the current of a small battery attached to the motherboard. Most importantly, for the ROM BIOS startup routines (boot sequence), CMOS determines the order in which drives should be examined for an operating system (floppy disk first, CD-Rom first, or fixed disk first). Furthermore, it holds some essential information such as hard drive size, memory address location, and Date & Time.

Shadow RAM: (Optional, you can turn it off/on using the CMOS settings) Shadow RAM is where a copy of BIOS routines from ROM is stored, it is a special area of RAM, so that the BIOS routines can be accessed more quickly.

Loading the OS (Operating System):The BIOS will attempt booting using the boot sequence determined by the CMOS settings, and examine the MBR (Master Boot Record) of the bootable disk. The MBR is the information in the first sector (512 bytes) of any hard disk or diskette that identifies how and where an operating system is located so that it can be loaded into the RAM (booted).

The PC's Physical Address Space

We will now dive into a bit more detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:



The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an early PC would therefore start at 0x00000000 but end at 0x000FFFFF instead of 0xFFFFFFFF. The 640KB area marked "Low Memory" was the only random-access memory (RAM) that an early PC could use; in fact the very earliest PCs only could be configured with 16KB, 32KB, or 64KB of RAM!

The 384KB area from 0x000A0000 through 0x000FFFFF was reserved by the hardware for special uses such as video display buffers and firmware held in non-volatile memory. The most important part of this reserved area is the Basic Input/Output System (BIOS), which occupies the 64KB region from 0x000F0000 through 0x000FFFFF. In early PCs the BIOS was held in true read-only memory (ROM), but current PCs store the BIOS in updateable flash memory. The BIOS is responsible for performing basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS loads the operating system from some appropriate location such as floppy disk, hard disk, CD-ROM, or the network, and passes control of the machine to the operating system.

When Intel finally "broke the one megabyte barrier" with the 80286 and 80386 processors, which supported 16MB and 4GB physical address spaces respectively, the PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from 0x000A0000 to 0x00100000, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices.

Recent x86 processors can support more than 4GB of physical RAM, so RAM can extend further above 0xFFFFFFFF. In this case the BIOS must arrange to leave a second hole in the system's RAM at the top of the 32-bit addressable region, to leave room for these 32-bit devices to be mapped. Because of design limitations JOS will use only the first 256MB of a PC's physical memory anyway, so for now we will pretend that all PCs have "only" a 32-bit physical address space. But dealing with complicated physical address spaces and other aspects of hardware organization that evolved over many years is one of the important practical challenges of OS development.

The ROM BIOS
In this portion of the lab, you'll use QEMU's debugging facilities to investigate how an IA-32 compatible computer boots.

Open two terminal windows and cd both shells into your lab directory. In one, enter make qemu-gdb (or make qemu-nox-gdb). This starts up QEMU, but QEMU stops just before the processor executes the first instruction and waits for a debugging connection from GDB. In the second terminal, from the same directory you ran make, run make gdb.

We provided a .gdbinit file that set up GDB to debug the 16-bit code used during early boot and directed it to attach to the listening QEMU. (If it doesn't work, you may have to add an add-auto-load-safe-path in your .gdbinit in your home directory to convince gdb to process the .gdbinit we provided. gdb will tell you if you have to do this.)

The following line:

[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
is GDB's disassembly of the first instruction to be executed. From this output you can conclude a few things:

The IBM PC starts executing at physical address 0x000ffff0, which is at the very

top of the 64KB area reserved for the ROM BIOS.
The PC starts executing with CS = 0xf000 and IP = 0xfff0.
The first instruction to be executed is a jmp instruction, which jumps to the segmented address CS = 0xf000 and IP = 0xe05b.
Why does QEMU start like this? This is how Intel designed the 8088 processor, which IBM used in their original PC. Because the BIOS in a PC is "hard-wired" to the physical address range 0x000f0000-0x000fffff, this design ensures that the BIOS always gets control of the machine first after power-up or any system restart - which is crucial because on power-up there is no other software anywhere in the machine's RAM that the processor could execute. The QEMU emulator comes with its own BIOS, which it places at this location in the processor's simulated physical address space. On processor reset, the (simulated) processor enters real mode and sets CS to 0xf000 and the IP to 0xfff0, so that execution begins at that (CS:IP) segment address. How does the segmented address 0xf000:fff0 turn into a physical address?

*To answer that we need to know a bit about real mode addressing. In real mode (the mode that PC starts off in), address translation works according to the formula: physical address = 16 * segment + offset. So, when the PC sets CS to 0xf000 and IP to 0xfff0, the physical address referenced is:*

```
16 * 0xf000 + 0xfff0   # in hex multiplication by 16 is
= 0xf0000 + 0xfff0     # easy--just append a 0.
= 0xffff0
```

0xffff0 is 16 bytes before the end of the BIOS (0x100000). Therefore we shouldn't be surprised that the first thing that the BIOS does is jmp backwards to an earlier location in the BIOS

When the BIOS runs, it sets up an interrupt descriptor table and initializes various devices such as the VGA display. This is where the "Starting SeaBIOS" message you see in the QEMU window comes from.

After initializing the PCI bus and all the important devices the BIOS knows about, it searches for a bootable device such as a floppy, hard drive, or CD-ROM. Eventually, when it finds a bootable disk, the BIOS reads the boot loader from the disk and transfers control to it.
If the disk is bootable, the first sector is called the boot sector, since this is where the boot loader code resides. When the BIOS finds a bootable floppy or hard disk, it loads the 512-byte boot sector into memory at physical addresses 0x7c00 through 0x7dff, and then uses a jmp instruction to set the CS:IP to 0000:7c00, passing control to the boot loader. Like the BIOS load address, these addresses are fairly arbitrary - but they are fixed and standardized for PCs.
REAL MODE
- *The processor is in real mode, in which it simulates an Intel 8088*
- *In real mode there are eight 16-bit general- purpose registers, but the processor sends 20 bits of address to memory*
- *The segment registers %cs, %ds, %es, and %ss provide the additional bits necessary to generate 20-bit memory addresses from 16- bit registers.*

****
*
Exercise 2. Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

Ans:
The BIOS mainly undertakes two important tasks as witnessed through running the si instructions:
1) sets the architecture assumption to i386 i.e. 80386 from i8086
2) transfers control to the boot sector i.e. MBR of the bootable device found.

The initial architecture for JOS is assumed to be i8086 which can address only 1 MB of physical memory according to the memory address provided in the lab. As such, the first instruction's disassembly suggests that a jump is undertaken to a lower address which is the starting of the actual ROM BIOS code.
[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
The BIOS in the aftermath of the jump undertakes actions such as resetting the ax registers and setting the eax register which can be witnessed by running a few 'si' instructions.
Most important among these are as follows:
(gdb) si
[f000:d15f] 0xfd15f: cli
0x0000d15f in ?? ()
(gdb) si
[f000:d160] 0xfd160: cld
While cli will clear the interrupt flags to enable disabling of the interrupts, the cld will clear the direction flag indicating that the address for now will increase from lower to higher.
The out in instructions in the aftermath are used to write or read a byte/word/dword respectively from hardware through port specified in the instructions. As such, the below instructions
(gdb) si
[f000:d161] 0xfd161: mov $0x8f,%eax
0x0000d161 in ?? ()
(gdb) si
[f000:d167] 0xfd167: out %al,$0x70
0x0000d167 in ?? ()
(gdb) si
[f000:d169] 0xfd169: in $0x71,%al
are used to write to the to port 0x70 and read values from ports 0x71 respectively.
The detailed functionalities of the ports can be found at the following link:
http://bochs.sourceforge.net/techspec/PORTS.LST
As is stated in the above link, port 0x70 is the CMOS RAM Index Register in which bit 7 is the NMI bit which is being set to 1 so as to disable the Non Makable Interrupts i.e. Interrupts which cannot be ignored by regular interrupt masking routine.
With the link also stating that any write to 0x70 should be accompanied by a action on 0x71 to avoid putting the RTC in a unknown state value from 0x71 is read.

IMP

IO port 0x70 is the "CMOS/RTC index register", and IO port 0x71 is the "CMOS/RTC data register". To access something in CMOS you're supposed to set the index then read/write to the data register.
For some RTC chips, if you set the index and don't read or write to the data register the chip is left in an undefined state. This means that if you want to set an index for later you have to read from the data register to avoid "undefined state" between now and later.
In other words; the value that was read isn't relevant - reading causes a side-/effect, and it's the side-effect that matters.

The instructions immediately following the read from 0x71 is
(gdb) si
[f000:d16d]   0xfd16d: or $0x2,%al
0x0000d16d in ?? ()
(gdb) si
[f000:d16f]   0xfd16f: out   %al,$0x92

Again referring the same link for the port, 0x92 port is PS/2 system control port A to which a value of 1 is being written to indicate that Address Line A20 is being enabled.

The lidtw and lgdtw instructions in the aftermath are as follows:
Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.
The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.
In 64-bit mode, the instruction’s operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

Lastly, before the architecture is assumed to i386, bit 0 of the cr0 register is set to one so as to enable the protected mode and switch the processor from 16 to 32 bit mode.

Part 2: The Boot Loader

Floppy and hard disks for PCs are divided into 512 byte regions called sectors. A sector is the disk's minimum transfer granularity: each read or write operation must be one or more sectors in size and aligned on a sector boundary. If the disk is bootable, the first sector is called the boot sector, since this is where the boot loader code resides. When the BIOS finds a bootable floppy or hard disk, it loads the 512-byte boot sector into memory at physical addresses 0x7c00 through 0x7dff, and then uses a jmp instruction to set the CS:IP to 0000:7c00, passing

control to the boot loader. Like the BIOS load address, these addresses are fairly arbitrary - but they are fixed and standardized for PCs.

The ability to boot from a CD-ROM came much later during the evolution of the PC, and as a result the PC architects took the opportunity to rethink the boot process slightly. As a result, the way a modern BIOS boots from a CD-ROM is a bit more complicated (and more powerful). CD-ROMs use a sector size of 2048 bytes instead of 512, and the BIOS can load a much larger boot image from the disk into memory (not just one sector) before transferring control to it.

For 6.828, however, we will use the conventional hard drive boot mechanism, which means that our boot loader must fit into a measly 512 bytes.
The boot loader must perform two main functions:

1. *First, the boot loader switches the processor from real mode to 32-bit protected mode, because it is only in this mode that software can access all the memory above 1MB in the processor's physical address space. At this point you only have to understand that translation of segmented addresses (segment:offset pairs) into physical addresses happens differently in protected mode, and that after the transition offsets are 32 bits instead of 16.*

2. *Second, the boot loader reads the kernel from the hard disk by directly accessing the Integrated Drive Electronics (IDE) disk device registers via the x86's special I/O instructions.*

STEPS followed by the Boot Loader:
The BIOS Loads the boot loader from the first sector of the hard disk also known as the boot sector into memory at address 0x7c00 with CS being 0x00 and IP being 0x7c00, thus passing control to the boot Loader. The address of the IP is given the label 'start'.
Firstly, the boot Loader enables the Address Line 20. With backward compatibility in mind, Address Line 20, A20, is disabled by default as 8086 could only address 20 bits of memory due to 1 MB limitation and a virtual segment: offset can yield addresses greater than 20 bits. As such, A20 is disabled as a result of which the top most bit of addresses larger than ones that can be represented by 20 bits are discarded i.e. addresses higher than 1MB wrap around to zero by default.

As such, the boot Loader must enable A20 for the processor to access addresses greater than 20 bits. It does so through the output port of the keyboard controller (port 0x64), the details of which can be sought through the link above.
The output port of the keyboard controller has a number of functions.
Bit 0 is used to reset the CPU (go to real mode) – a reset happens when bit 0 is 0.
Bit 1 is used to control A20 - it is enabled when bit 1 is 1, disabled when bit 1 is 0.
One sets the output port of the keyboard controller by first writing 0xd1 to port 0x64, and the desired value of the output port to port 0x60. One usually sees the values 0xdd and 0xdf used to disable/enable A20.
Hence, the value 0xdf is being written to port 0x60, in the aftermath of writing 0xd1 to port 0x64.

Secondly, the boot Loader switches the processor from 16-bit Real Mode in which

the processor acts like 8086 to 32-bit protected Mode which is essential to access all physical addresses above 1MB.

THE ACTUAL PROCESS FOR SEGMENTATION:

- *The selector address, which is part of the logical address (last 16 bits, the other being offset) provided through one of the instructions specified above is loaded into the appropriate segment register which is also specified in the instruction itself.*

- *The Processor in the aftermath will use the selector address to locate the table i.e. global descriptor or local descriptor tables through Table Indicator bit (TI) of the address. The addresses of the Local and Global descriptor tables will be obtained through the LDTR and GDTR registers respectively. Selecting the appropriate table, The Index bits of the selector address (bits 3 to 15) will be used to offset into the table by multiplying them by 8.*

- *The information present, if any, for the resultant descriptor will be fetched into the invisible portion of segment register to avoid the whole process again, which might be required frequently as data is often referred to in segments by instructions whose selector addresses are already in memory and thus can be easily accessed without any overhead.*

- *The 8-byte descriptor of a particular segments specifies its location within the physical address space, its size, type, DPL, and if the segment is actually present in memory etc.*
- *If paging has been enabled the linear address obtained through the base address fields of the segment descriptor will be further converted through page directories, page tables and indexes within those page tables into physical addresses residing within pages of 4k bytes each.*

The switch from real to protected Mode is undertaken by setting the bit 0 in CR0 register.
The protected mode however, is not fully enabled until a new value is loaded into the segment register (especially code segment). Hence, the far jump is undertaken in boot.S which allows a new segment selector to be specified whose descriptor specifies it to be a 32-bit segment thereby enabling the switch.

The final step undertaken in boot.S is setting up of stack for main.c program. The address of the stack pointer is chosen to be that of boot loader i.e. 0x7c00 from which the stack shall grow downwards.

The function bootmain.c will read the kernel from hard disk in LBA (Logical Block Addressing) Mode.
Through functions readseg and readsect.

In the LBA Mode, 32 bit address will be generated with the head, cylinder high, cylinder low and sector number registers which points to the data block (size of 512 bytes) to be read.

- To read a sector using LBA28:
  - Send a NULL byte to port 0x1F1: `outb(0x1F1, 0x00);`
  - Send a sector count to port 0x1F2: `outb(0x1F2, 0x01);`
  - Send the low 8 bits of the block address to port 0x1F3: `outb(0x1F3, (unsigned char)addr);`
  - Send the next 8 bits of the block address to port 0x1F4: `outb(0x1F4, (unsigned char)(addr >> 8);`
  - Send the next 8 bits of the block address to port 0x1F5: `outb(0x1F5, (unsigned char)(addr >> 16);`
  - Send the drive indicator, some magic bits, and highest 4 bits of the block address to port 0x1F6: `outb(0x1F6, (addr >> 24) | 0xE0 ));`
  - Send the command (0x20) to port 0x1F7: `outb(0x1F7,0x20);`

```
Status Register(0x1F7):
    7      6    5      4    3    2    1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Errot Register (0x1F1): if status.ERROR=1
    7      6    5      4    3    2    1    0
   BBK    UNC   MC    IDNF  MCR  ABRT TONF AMNF
```

The only thing left after the above procedure is for the drive to signal ready which is accomplished in main.c through the waitdisk() function. Reading and writing shall commence from the disk when the disk signals ready.

Exercise 3. Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.

Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

Be able to answer the following questions:

- *At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?*

First, the following loads the GDT:

```
lgdt    gdtdesc
```

where gdtdesc is a region in memory that stores the content of what the GDT should load. the format is [size of gdt][address of gdt]. The GDT contains the null segment, executable and readable code segment, and writable data segments. both segments span from address 0 to address 4GB.

```
movl    %cr0, %eax

orl     $CR0_PE_ON, %eax

movl    %eax, %cr0
```

Enable 32-bit Protected Mode which causes the processor to switch from 16-bit Real Mode to 32-bit protected mode.

```
 # Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

 ljmp    $PROT_MODE_CSEG, $protcseg
```

Finally, a long jump is performed through above instruction to load 32 bit segment address into the CS register which causes the processor to begin executing 32-bit code.

For the ljmp instruction, In Real Address Mode or Virtual 8086 mode, the former pointer provides 16 bits for the CS register. In protected mode, the former 16-bit now works as selector. And PROT_MODE_CSEG(0x8) ensure that we still work in the same segment. The offset $protcseg is exactly the next instruction. Till now,we have switch to 32-bit mode, but we still work in the same program logic segment.

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded? Where is the first instruction of the kernel?

Last Instruction of Boot Loader executed is as follows:

```
// call the entry point from the ELF header
```

// note: does not return!

((void (*)(void)) (ELFHDR→e_entry))();

*7d6b: ff 15 18 00 01 00 call *0x10018*

With the ELFHDR of the kernel being loaded at address 0x10000 in memory and the corresponding call to transfer control to the kernel being to location 0x10018 which is at an offset of 24 bytes from the 0x10000, which is also the offset of the 'e_entry' field in struct elf in inc/elf.h, we use gdb to notice the first instruction of the kernel and its address.

(gdb) x/1w 0x10018

0x10018: 0x0010000c

(gdb) x/1i 0x10000c

0x10000c: movw $0x1234,0x472

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

```
struct Proghdr *ph, *eph;

// read 1st page off disk

readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

// is this a valid ELF?

if (ELFHDR->e_magic != ELF_MAGIC)

     goto bad;

// load each program segment (ignores ph flags)

ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);

eph = ph + ELFHDR->e_phnum;

for (; ph < eph; ph++)

     // p_pa is the load address of this segment (as well as the physical address)
```

```
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

The above instructions read sectors from disk into memory.

Fetching the first sector initially from the disk, the Boot Loader reads the start and count of the program headers in the elf headers and fetches each segment from memory.

Loading the Kernel

When you compile and link a C program such as the JOS kernel, the compiler transforms each C source ('.c') file into an object ('.o') file containing assembly language instructions encoded in the binary format expected by the hardware. The linker then combines all of the compiled object files into a single binary image such as obj/kern/kernel, which in this case is a binary in the ELF format, which stands for "Executable and Linkable Format".

For purposes of 6.828, you can consider an ELF executable to be a header with loading information, followed by several program sections, each of which is a contiguous chunk of code or data intended to be loaded into memory at a specified address. The boot loader does not modify the code or data; it loads it into memory and starts executing it.

An ELF binary starts with a fixed-length ELF header, followed by a variable-length program header listing each of the program sections to be loaded. The C definitions for these ELF headers are in inc/elf.h. The program sections we're interested in are:

- *.text: The program's executable instructions.*
- *.rodata: Read-only data, such as ASCII string constants produced by the C compiler.*
- *.data: The data section holds the program's initialized data, such as global variables declared with initializers like int x = 5;.*

When the linker computes the memory layout of a program, it reserves space for uninitialized global variables, such as int x;, in a section called .bss that immediately follows .data in memory. C requires that "uninitialized" global variables start with a value of zero. Thus there is no need to store contents for .bss in the ELF binary; instead, the linker records just the address and size of the .bss section. The loader or the program itself must arrange to zero the .bss section.

Examine the full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

athena% objdump -h obj/kern/kernel

Take particular note of the "VMA" (or link address) and the "LMA" (or load address) of the .text section. The load address of a section is the memory address at which that section should be loaded into memory i.e. physical address.

The link address of a section is the memory address from which the section expects to execute i.e. virtual address. The linker encodes the link address in the binary in various ways with the result that a binary usually won't work if it is executing from an address that it is not linked for. Typically, the link and load addresses are the same.

The boot loader uses the ELF program headers to decide how to load the sections. The program headers specify which parts of the ELF object to load into memory and the destination address each should occupy. You can inspect the program headers by typing:

athena% objdump -x obj/kern/kernel

The program headers are then listed under "Program Headers" in the output of objdump. The areas of the ELF object that need to be loaded into memory are those that are marked as "LOAD". Other information for each program header is given, such as the virtual address ("vaddr"), the physical address ("paddr"), and the size of the loaded area ("memsz" and "filesz").

Back in boot/main.c, the ph->p_pa field of each program header contains the segment's destination physical address (in this case, it really is a physical address, though the ELF specification is vague on the actual meaning of this field).

The BIOS loads the boot sector into memory starting at address 0x7c00, so this is the boot sector's load address. This is also where the boot sector executes from, so this is also its link address. We set the link address by passing -Ttext 0x7C00 to the linker in boot/Makefrag, so the linker will produce the correct memory addresses in the generated code.

Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in boot/Makefrag to something wrong, run make clean, recompile the lab

with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean again afterward!

Changing the link address to 0x7e11, following is a snippet of the generated boot.asm file

```
 # Switch from real to protected mode, using a bootstrap GDT

 62   # and segment translation that makes virtual addresses

 63   # identical to their physical addresses, so that the

 64   # effective memory map does not change during the switch.

 65   lgdt    gdtdesc

 66     7e1e: 0f 01 16                lgdtl  (%esi)

 67     7e21: 64 7e 0f                fs jle 7e33 <protcseg+0x1>

 68   movl    %cr0, %eax

 69     7e24: 20 c0                   and    %al,%al

 70   orl     $CR0_PE_ON, %eax

 71     7e26: 66 83 c8 01             or     $0x1,%ax

 72   movl    %eax, %cr0

 73     7e2a: 0f 22 c0                mov    %eax,%cr0

 74

 75   # Jump to next instruction, but in 32-bit code segment.

 76   # Switches processor into 32-bit mode.

 77   ljmp    $PROT_MODE_CSEG, $protcseg

 78     7e2d: ea                      .byte 0xea

 79     7e2e: 32 7e 08                xor    0x8(%esi),%bh
```

```
80      …

82 00007e32 <protcseg>:

83

84   .code32                     # Assemble for 32-bit mode

85 protcseg:

86   # Set up the protected-mode data segment registers

87   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector

88     7e32: 66 b8 10 00              mov    $0x10,%ax

89   movw    %ax, %ds                # -> DS: Data Segment

90     7e36: 8e d8                    mov    %eax,%ds

91   movw    %ax, %es                # -> ES: Extra Segment

92     7e38: 8e c0                    mov    %eax,%es

93   movw    %ax, %fs                # -> FS

94     7e3a: 8e e0                    mov    %eax,%fs

95   movw    %ax, %gs                # -> GS

96     7e3c: 8e e8                    mov    %eax,%gs

97   movw    %ax, %ss                # -> SS: Stack Segment

98     7e3e: 8e d0                    mov    %eax,%ss
```

Since the ljmp uses relative address for jump and not offset, it is the instruction that causes the program to break.

```
(gdb) b *0x7e36

Breakpoint 1 at 0x7e36
```

(gdb) c

Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.

[0:7c2d] => 0x7c2d: ljmp $0x8,$0x7e32

0x00007c2d in ?? ()

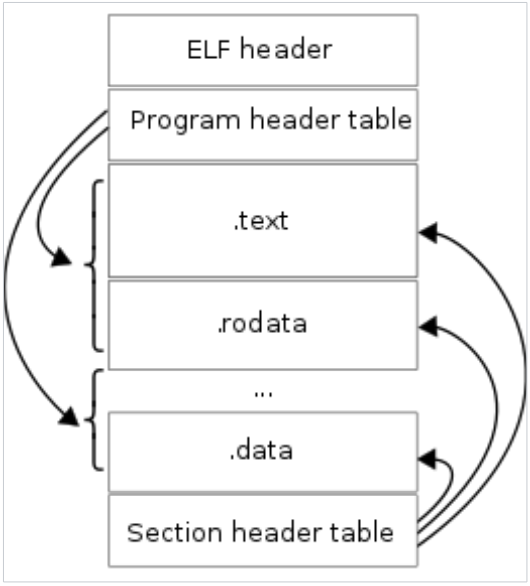
The same is also verified by gdb as shown above.

Look back at the load and link addresses for the kernel. Unlike the boot loader, these two addresses aren't the same: the kernel is telling the boot loader to load it into memory at a low address (1 megabyte), but it expects to execute from a high address. We'll dig in to how we make this work in the next section.

Besides the section information, there is one more field in the ELF header that is important to us, named e_entry. This field holds the link address of the entry point in the program: the memory address in the program's text section at which the program should begin executing. You can see the entry point:

athena% objdump -f obj/kern/kernel

You should now be able to understand the minimal ELF loader in boot/main.c. It reads each section of the kernel from disk into memory at the section's load address and then jumps to the kernel's entry point.



An ELF file has two views: the program header shows the segments used at run time, whereas the section header lists the set of sections of the binary.

Exercise 6. We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command x/Nx ADDR prints N words of memory at ADDR. (Note that both 'x's in the command are lowercase.) Warning: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

```
(gdb) b *0x7c00

Breakpoint 1 at 0x7c00

(gdb) c

Continuing.

[   0:7c00] => 0x7c00:     cli



Breakpoint 1, 0x00007c00 in ?? ()
```

(gdb) x/8x 0x100000

0x100000: 0x00000000 0x00000000 0x00000000 0x00000000

0x100010: 0x00000000 0x00000000 0x00000000 0x00000000

*(gdb) b*0x7d6b*

Breakpoint 2 at 0x7d6b

(gdb) c

Continuing.

The target architecture is assumed to be i386

*=> 0x7d6b: call   *0x10018*

Breakpoint 2, 0x00007d6b in ?? ()

(gdb) x/8x 0x100000

0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766

0x100010: 0x34000004 0x0000b812 0x220f0011 0xc0200fd8

As is evident, when BIOS enters the boot Loader, the memory location 0x100000 is empty as the text section of the kernel is yet to be loaded

With objdump specifying that the load address of the kernel should be 0x100000, the boot loader copies the text segment of the kernel into the very memory location i.e. the kernel's text segment begins at memory location 0x100000 before boot loader passes control over to the kernel.

Part 3: The Kernel

Using virtual memory to work around position dependence

When you inspected the boot loader's link and load addresses above, they matched perfectly, but there was a (rather large) disparity between the kernel's link address (as printed by objdump) and its load address.

Operating system kernels often like to be linked and run at very high virtual address, such as 0xf0100000, in order to leave the lower part of the processor's virtual address space for user programs to use.

Many machines don't have any physical memory at address 0xf0100000, so we can't count on being able to store the kernel there. Instead, we will use the processor's memory management hardware to map virtual address 0xf0100000 (the link address at which the kernel code expects to run) to physical address 0x00100000 (where the boot loader loaded the kernel into physical memory). This way, although the kernel's virtual address is high enough to leave plenty of address space for user processes, it will be loaded in physical memory at the 1MB point in the PC's RAM, just above the BIOS ROM. This approach requires that the PC have at least a few megabytes of physical memory.

In fact, in the next lab, we will map the entire bottom 256MB of the PC's physical address space, from physical addresses 0x00000000 through 0x0fffffff, to virtual addresses 0xf0000000 through 0xffffffff respectively.

For now, we'll just map the first 4MB of physical memory, which will be enough to get us up and running. We do this using the hand-written, statically-initialized page directory and page table in kern/entrypgdir.c. For now, you don't have to understand the details of how this works, just the effect that it accomplishes. Up until kern/entry.S sets the CR0_PG flag, memory references are treated as physical addresses (strictly speaking, they're linear addresses, but boot/boot.S set up an identity mapping from linear addresses to physical addresses and we're never going to change that). Once CR0_PG is set, memory references are virtual addresses that get translated by the virtual memory hardware to physical addresses. entry_pgdir translates virtual addresses in the range 0xf0000000 through 0xf0400000 to physical addresses 0x00000000 through 0x00400000, as well as virtual addresses 0x00000000 through 0x00400000 to physical addresses 0x00000000 through 0x00400000. Any virtual address that is not in one of these two ranges will cause a hardware exception which, since we haven't set up interrupt handling yet, will cause QEMU to dump the machine state and exit.

***Exercise 7.** Use QEMU and GDB to trace into the JOS kernel and stop at the movl %eax, %cr0. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the **stepi** GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.*

What is the first instruction after the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the movl %eax, %cr0 in kern/entry.S, trace into it, and see if you were right.

(gdb) x/5w 0xf0100000

0xf0100000 <_start+4026531828>: 0x00000000 0x00000000 0x00000000 0x00000000

```
0xf0100010 <entry+4>: 0x00000000

(gdb) si

=> 0x10001d:    mov    %cr0,%eax

0x0010001d in ?? ()

(gdb) si

=> 0x100020:    or     $0x80010001,%eax

0x00100020 in ?? ()

(gdb) si

=> 0x100025:    mov    %eax,%cr0

0x00100025 in ?? ()

(gdb) si

=> 0x100028:    mov    $0xf010002f,%eax

0x00100028 in ?? ()

(gdb) x/5w 0xf0100000

0xf0100000 <_start+4026531828>: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766

0xf0100010 <entry+4>: 0x34000004

(gdb) x/5w 0x100000

0x100000:  0x1badb002 0x00000000 0xe4524ffe      0x7205c766

0x100010:  0x34000004
```

As is evident from above, prior to paging being turned ON by setting the bit 31 in CR0 register, the linking memory address of kernel is empty. Such is not the case whence paging has been turned ON as the virtual memory hardware through entrypgdir.c will map the first 4 MB of Processor's memory i.e. from 0x00000000 – 0x00400000 to virtual addresses 0xf0000000 – 0xf0400000.

In the aftermath of the paging being turned ON being disabled, the first instruction to fail would be

movl $0x0,%ebp

as the instruction prior to it makes a jump to a virtual address which in absence of paging will be interpreted as physical address resulting in following error

qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c.

 # Now paging is enabled, but we're still running at a low EIP why is this okay?

As first first 4 MB of virtual addresses has also been mapped to the first 4 MB of physical addresses, it is okay.

Formatted Printing to the Console

***Exercise 8.** We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.*

Lib/printfmt.c:

```
206          // (unsigned) octal
207          case 'o':
208               num = getuint(&ap, lflag);
209               base = 8;
210               goto number;
211               // Replace this with your code.
212          /*   putch('X', putdat);
213               putch('X', putdat);
214               putch('X', putdat);
215          */   break;
```

The Stack

In the final exercise of this lab, we will explore in more detail the way the C language uses the stack on the x86, and in the process write a useful new kernel monitor function that prints a backtrace of the stack: a list of the saved Instruction Pointer (IP) values from the nested call instructions that led to the current point of execution.

Exercise 9. *Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?*

The kernel initializes its stack in file entry.S with the following instruction, before executing any of its associated C Code.

```
76      # Set the stack pointer

 77      movl $(bootstacktop),%esp
```

The exact location of the stack in memory can be viewed in the disassembly file of the kernel obj/kern/kernel.asm

```
 56      # Set the stack pointer

  57      movl $(bootstacktop),%esp

  58 f0100034: bc 00 00 11 f0           mov    $0xf0110000,%esp
```

*The stack pointer has thus been initially been set at memory address of 0xf0110000. The size of the stack as can be viewed in .data section of entry.S file as well in inc/memlayout.h file to be KSTKSIZE = 8 *PGSIZE = 8 *4096 = 32768 = 0x8000.*

Thus, the kernel stack while beginning from 0xf0110000 will grow downwards towards memory location 0xf0108000.

The x86 stack pointer (esp register) points to the lowest location on the stack that is currently in use. Everything below that location in the region reserved for the stack is free. Pushing a value onto the stack involves decreasing the stack pointer and then writing the value to the place the stack pointer points to. Popping a value from the stack involves reading the value the stack pointer points

to and then increasing the stack pointer. In 32-bit mode, the stack can only hold 32-bit values, and esp is always divisible by four. Various x86 instructions, such as call, are "hard-wired" to use the stack pointer register.

The ebp (base pointer) register, in contrast, is associated with the stack primarily by software convention. On entry to a C function, the function's prologue code normally saves the previous function's base pointer by pushing it onto the stack, and then copies the current esp value into ebp for the duration of the function. If all the functions in a program obey this convention, then at any given point during the program's execution, it is possible to trace back through the stack by following the chain of saved ebp pointers and determining exactly what nested sequence of function calls caused this particular point in the program to be reached. This capability can be particularly useful, for example, when a particular function causes an assert failure or panic because bad arguments were passed to it, but you aren't sure who passed the bad arguments. A stack backtrace lets you find the offending function.

Exercise 10. *To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?*

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

```
Breakpoint 3, test_backtrace (x=5) at kern/init.c:14

14   {

(gdb) info registers

eax            0x0    0

ecx            0x3d4  980

edx            0x3d5  981

ebx            0x10094     65684

esp            0xf010ffdc  0xf010ffdc

ebp            0xf010fff8  0xf010fff8
```

```
esi            0x10094     65684
edi            0x0    0
eip            0xf0100040  0xf0100040 <test_backtrace>
eflags         0x46   [ PF ZF ]
cs             0x8    8
ss             0x10   16
ds             0x10   16
es             0x10   16
fs             0x10   16
gs             0x10   16
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 3, test_backtrace (x=4) at kern/init.c:14
14   {
(gdb) info registers
eax            0x4    4
ecx            0x3d4  980
edx            0x3d5  981
ebx            0x5    5
```

```
esp            0xf010ffbc  0xf010ffbc

ebp            0xf010ffd8  0xf010ffd8

esi            0x10094     65684

edi            0x0    0

eip            0xf0100040  0xf0100040 <test_backtrace>

eflags         0x92   [ AF SF ]

cs             0x8    8

ss             0x10   16

ds             0x10   16

es             0x10   16

fs             0x10   16

gs             0x10   16
```

As is evident from from above,the difference between esp registers for two successive calls of test_backtrace is 0x20 which is equivalent of 32 bytes in decimal. Since 80386 is a 32-bit processor, esp can hold only 4 byte values. As such, eight 4-byte words are being pushed on to the stack for each recursive call of test_backtrace.

Thus, each time the funcction is called the return address of the calling function, its frame pointer i.e. ebp, its callee saved save registers i.e. ebx, the argument x for next instruction to use , 0xf0101800 and the eip register are pushed on to the stack. The proof for the same can be obtained through gdb by setting at breakpoint over the starting address of the function test_backtrace.

Each line contains an ebp, eip, and args. The ebp value indicates the base pointer into the stack used by that function: i.e., the position of the stack pointer just after the function was entered and the function prologue code set up the base pointer. The listed eip value is the function's return instruction pointer: the instruction address to which control will return when the function returns. The return instruction pointer typically points to the instruction after the call instruction. Finally, the five hex values listed after args are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called.

The first line printed reflects the currently executing function, namely mon_backtrace itself, the second line reflects the function that called mon_backtrace, the third line reflects the function that called that one, and so on. You should print all the outstanding stack frames. By studying kern/entry.S you'll find that there is an easy way to tell when to stop.

*If int *p = (int*)100, then (int)p + 1 and (int)(p + 1) are different numbers: the first is 101 but the second is 104. When adding an integer to a pointer, as in the second case, the integer is implicitly multiplied by the size of the object the pointer points to.*

*p[i] is defined to be the same as *(p+i), referring to the i'th object in the memory pointed to by p.*

&p[i] is the same as (p+i), yielding the address of the i'th object in the memory pointed to by p.

***Exercise 11.** Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run **make grade** to see if its output conforms to what our grading script expects, and fix it if it doesn't. After you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.*

If you use read_ebp(), note that GCC may generate "optimized" code that calls read_ebp() before mon_backtrace()'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of mon_backtrace() and make sure the call to read_ebp() is happening after the function prologue.

The stack backtrace function is as follows:

```
        int mon_backtrace(int argc, char **argv, struct Trapframe *tf)

{

        // Your code here.

/*
```

The code of read_ebp function in inc/x86.h has been utilized which returns the value of ebp as an unsigned integer to avoid the optimization by the gcc in any form along with the volatile keyword.

```
*/

        uint32_t func_ebp = 0, ueip = 0, arg = 0;

        asm volatile("movl %%ebp,%0" : "=r" (func_ebp));

        cprintf("Stack Backtrace: \n");

/*

equating baseframe to func_ebp below which is the base pointer of the
mon_backtrace function, the former has been cast into a pointer as we are
interested in the values stored at memory location held by ebp. The rest of the
code is self explantory. While eip value will be the one preceding ebp, the values
of arguments to be passed to functions will be preceding eip on the stack as its
grows downwards. With base pointer being set to zero in kern/entry.S code, we use
the same trick for implementing the function by running the while loop in the code
until the value of ebp i.e. baseframe reduces to 0, which will eventually be the
case causing the loop to exit as we traverse up the stack of the kernel.

*/

        uint32_t baseframe = func_ebp;

        while(baseframe != 0)

        {

                ueip = *((uint32_t *)baseframe + 1);

                cprintf("ebp %08x eip %08x args ", baseframe, ueip);

                for (int i = 2; i < 7; i ++)

                {

                        arg = *((uint32_t*) baseframe + i);

                        cprintf(" %08x ", arg);

                }

                cprintf ("\n");
```

```
                baseframe = *(uint32_t *)baseframe;

        }

        return 0;

}
```

Exercise 12. *Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.*

In debuginfo_eip, where do __STAB_* come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- *look in the file kern/kernel.ld for __STAB_**
- *run* ***objdump -h obj/kern/kernel***
- *run* ***objdump -G obj/kern/kernel***
- *run* ***gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c****, and look at init.s.*
- *see if the bootloader loads the symbol table in memory as part of loading the kernel binary*

Complete the implementation of debuginfo_eip by inserting the call to stab_binsearch to find the line number for an address.

Add a backtrace command to the kernel monitor, and extend your implementation of mon_backtrace to call debuginfo_eip and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's eip, followed by the name of the function and the offset of the eip from the first instruction of the function (e.g., monitor+106 means the return eip is 106 bytes past the beginning of monitor).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

*Tip: printf format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. printf("%.*s", length, string) prints at most length characters of string. Take a look at the printf man page to find out why this works.*

You may find that some functions are missing from the backtrace. For example, you will probably see a call to monitor() but not to runcmd(). This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the -O2 from GNUMakefile, the backtraces may make more sense (but your kernel will run more slowly).

The various Stabs in debuginnfo_eip function are actually symbols that have been defined in the linker script of the kernel kern//kernel.ld indicating the start and end of stabs and stab_string sections.

Refer the following link for detailed information on stabs

http://sourceware.org/gdb/onlinedocs/stabs.html#Overview

Following is the mainly the essence of stab and stab strings.

For some object file formats, the debugging information is encapsulated in assembler directives known collectively as stab (symbol table) directives, which are interspersed with the generated code.

Stabs are the native format for debugging information in the a.out and XCOFF objects.

To confirm that symbol table containing stab i.e. debugging information os loaded into memory we can use the objdump instruction as follows:

```
objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
```

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001851  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000077c  f0101860  00101860  00002860  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000038f5  f0101fdc  00101fdc  00002fdc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      000018fa  f01058d1  001058d1  000068d1  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000648  f0112300  00112300  00013300  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  6 .comment      00000035  00000000  00000000  00013948  2**0
                  CONTENTS, READONLy
```

Further verification of the addresses of stabs and stab string which contains the real eseence of the dedbugging information we need, we can use the gdb

```
(gdb) b i386_init
Breakpoint 3 at 0xf0100094: file kern/init.c, line 25.
(gdb) c
Continuing.
=> 0xf0100094 <i386_init>: push   %ebp
```

```
Breakpoint 3, i386_init () at kern/init.c:25

25   {

(gdb) x/10w 0xf01058d1

0xf01058d1:     0x74737b00 0x61646e61 0x69206472 0x7475706e

0xf01058e1:     0x656b007d 0x652f6e72 0x7972746e 0x6b00532e

0xf01058f1:     0x2f6e7265 0x72746e65

(gdb) x/10s 0xf01058d1

0xf01058d1:     ""

0xf01058d2:     "{standard input}"

0xf01058e3:     "kern/entry.S"

0xf01058f0:     "kern/entrypgdir.c"

0xf0105902:     "gcc2_compiled."

0xf0105911:     "int:t(0,1)=r(0,1);-2147483648;2147483647;"

0xf010593b:     "char:t(0,2)=r(0,2);0;127;"

0xf0105955:     "long int:t(0,3)=r(0,3);-2147483648;2147483647;"

0xf0105984:     "unsigned int:t(0,4)=r(0,4);0;4294967295;"

0xf01059ad:     "long unsigned int:t(0,5)=r(0,5);0;4294967295;"
```

With stabs containing debugging information, the very information will be filled into the struct Stab defined in inc/stab.h. The different types associated with stabs are also defined in the very file.

The stab_binsearch function defined in kern/kdebug.c will search the region between left and right for stab entry of the address with matching stab type. The search region in the process will be modified to bracket the address.

The function debuginfo_eip will store the necessary information for a particluar address i.e. the source file, the function name etc. it belongs to and fill in the fields of the struct Eipdebuginfo defined kern/kdebug.h using the stab_binsearch function. The stab_binsearch function will be used repetetively to fill in all the fields of the struct Eipdebuginfo.

In sequence, given the address I.e eip, the entire sgtabs table will be searhced for the source file, then the source file stabs will be searched for the function name for the address and eventually, the function's stabs will be searched for the line number.

As suc, the missing piece of code in kdebug.c is as follows:

```
 // Search within [lline, rline] for the line number stab.

        // If found, set info->eip_line to the right line number.

        // If not found, return -1.

        //

        // Hint:

        //There's a particular stabs type used for line numbers.

        //Look at the STABS documentation and <inc/stab.h> to find

        //which one.

        // Your code here.



        stab_binsearch (stabs, &lline, &rline, N_SLINE, addr);

        if (lline <= rline)

        {

                info -> eip_line = stabs [lline].n_desc;

        } else
```

```
{
        return -1;
}
```

Additionally, certain changes are needed to the mon_backtrace fucntion in kern/monitor.c so as to call the function debuginfo_eip. The entire function is as follows:

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf) {
// Your code here.
uint32_t func_ebp = 0, ueip = 0, arg = 0;
asm volatile("movl %%ebp,%0" : "=r" (func_ebp));
cprintf("Stack Backtrace: \n");
uint32_t baseframe = func_ebp;
while(baseframe != 0)
{
        ueip = *((uint32_t *)baseframe + 1);
        cprintf("ebp %08x eip %08x args ", baseframe, ueip);
        for (int i = 2; i < 7; i ++)
        {
                arg = *((uint32_t*) baseframe + i);
                cprintf(" %08x ", arg);
        }
        cprintf("\n");
        struct Eipdebuginfo information;
```

```
                debuginfo_eip (ueip, &information);

                uintptr_t offset = ueip - information.eip_fn_addr;

                cprintf("\t%s:%d: ", information.eip_file, information.eip_line);

                cprintf("%.*s+%d\n",information.eip_fn_namelen,
information.eip_fn_name, offset);

                baseframe = *(uint32_t *) baseframe;

        }

        return 0;

}
```

Notice that the line numbers are based on the line to which the function returns instead of the line where the call is being made. This makes sense because we derived those from the %eip register which would store the return address.

This Completes the Lab.