Lab 1: Booting a PC

Due Thursday, September 13, 2018

Introduction

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.

Software Setup

The files you will need for this and subsequent lab assignments in this course are distributed using the Git version control system. To learn more about Git, take a look at the Git user's manual, or, if you are already familiar with other version control systems, you may find this CS-oriented overview of Git useful.

The URL for the course Git repository is https://pdos.csail.mit.edu/6.828/2018/jos.git. To install the files in your Athena account, you need to *clone* the course repository, by running the commands below. You must use an x86 Athena machine; that is, `uname -a` should mention `i386 GNU/Linux` or `i686 GNU/Linux` or `x86_64 GNU/Linux`. You can log into a public Athena host with `ssh -X athena.dialup.mit.edu`.

```
athena% mkdir ~/6.828
athena% cd ~/6.828
athena% add git
athena% git clone https://pdos.csail.mit.edu/6.828/2018/jos.git lab
Cloning into lab...
athena% cd lab
athena%
```

Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can *commit* your changes by running:

```
athena% git commit -am 'my solution for lab1 exercise 9'
Created commit 60d2135: my solution for lab1 exercise 9
 1 files changed, 1 insertions(+), 0 deletions(-)
athena%
```

You can keep track of your changes by using the `git diff` command. Running `git diff` will display the changes to your code since your last commit, and `git diff origin/lab1` will display the changes relative to the initial code supplied for this lab. Here, `origin/lab1` is the name of the git branch with the initial code you downloaded from our server for this assignment.

We have set up the appropriate compilers and simulators for you on Athena. To use them, run `add -f 6.828`. You must run this command every time you log in (or add it to your `~/.environment` file). If you get obscure errors while compiling or running `qemu`, double check that you added the course locker.

If you are working on a non-Athena machine, you'll need to install `qemu` and possibly `gcc` following the directions on the tools page. We've made several useful debugging changes to `qemu` and some of the later labs depend on these patches, so you must build your own. If your machine uses a native ELF toolchain (such as Linux and most BSD's, but notably *not* OS X), you can simply install `gcc` from your package manager. Otherwise, follow the directions on the tools page.

Hand-In Procedure

You will turn in your assignments using the submission website. You need to request an API key from the submission website before you can turn in any assignments or labs.

The lab code comes with GNU Make rules to make submission easier. After committing your final changes to the lab, type `make handin` to submit your lab.

```
athena% git commit -am "ready to submit my lab"
[lab1 c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

athena% make handin
git archive --prefix=lab1/ --format=tar HEAD | gzip > lab1-handin.tar.gz
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2018/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 50199  100   241  100 49958    414  85824 --:--:-- --:--:-- --:--:-- 85986
athena%
```

`make handin` will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let `make handin` generate it again (*myapi.key* must not include newline characters).

If use `make handin` and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with `??`.

In the case that `make handin` does not work properly, try fixing the problem with the curl or Git commands. Or you can run `make tarball`. This will make a tar file for you, which you can then upload via our web interface.

You can run `make grade` to test your solutions with the grading program. The web interface uses the same grading program to assign your lab submission a grade. You should check the output of the grader (it may take a few minutes since the grader runs periodically) and ensure that you received the grade which you expected. If the grades don't match, your lab submission probably has a bug -- check the output of the grader (resp-lab*.txt) to see which particular test failed.

For Lab 1, you do not need to turn in answers to any of the questions below. (Do answer them for yourself though! They will help with the rest of the lab.)

Part 1: PC Bootstrap

The purpose of the first exercise is to introduce you to x86 assembly language and the PC bootstrap process, and to get you started with QEMU and QEMU/GDB debugging. You will not have to write any code for this part of the lab, but you should go through it anyway for your own understanding and be prepared to answer the questions posed below.

Getting Started with x86 assembly

If you are not already familiar with x86 assembly language, you will quickly become familiar with it during this course! The PC Assembly Language Book is an excellent place to start. Hopefully, the book contains mixture of new and old material for you.

Warning: Unfortunately the examples in the book are written for the NASM assembler, whereas we will be using the GNU assembler. NASM uses the so-called *Intel* syntax while GNU uses the *AT&T* syntax. While semantically equivalent, an assembly file will differ quite a lot, at least superficially, depending on which syntax is used. Luckily the conversion between the two is pretty simple, and is covered in Brennan's Guide to Inline Assembly.

> **Exercise 1.** Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.
>
> We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

Certainly the definitive reference for x86 assembly language programming is Intel's instruction set architecture reference, which you can find on the 6.828 reference page in two flavors: an HTML edition of the old 80386 Programmer's Reference Manual, which is much shorter and easier to navigate than more recent manuals but describes all of the x86 processor features that we will make use of in 6.828; and the full, latest and greatest IA-32 Intel Architecture Software Developer's Manuals from Intel, covering all the features of the most recent processors that we won't need in class but you may be interested in learning about. An equivalent (and often friendlier) set of manuals is available from AMD. Save the Intel/AMD architecture manuals for later or use them for reference when you want to look up the definitive explanation of a particular processor feature or instruction.

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. (It's a little generous to call the code we're running here a "kernel," but we'll flesh it out throughout the semester.)

```
athena% cd lab
athena% make
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 380 bytes (max 510)
+ mk obj/kern/kernel.img
```

(If you get errors like "undefined reference to `__udivdi3'", you probably don't have the 32-bit gcc multilib. If you're running Debian or Ubuntu, try installing the gcc-multilib package.)

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
```

or

```
athena% make qemu-nox
```

This executes QEMU with the options required to set the hard disk and direct serial port output to the terminal. Some text should appear in the QEMU window:

```
Booting from Hard Disk...
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

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`. This may be convenient if you are SSH'd into an Athena dialup. To quit qemu, type `Ctrl+a x`.

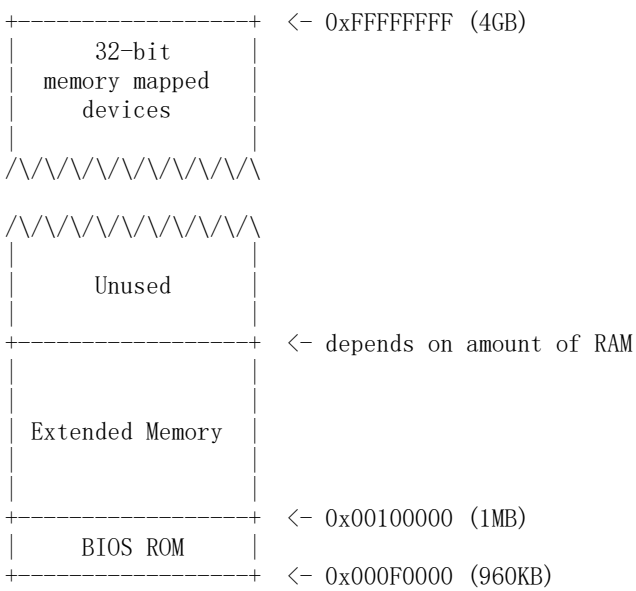
There are only two commands you can give to the kernel monitor, `help` and `kerninfo`.

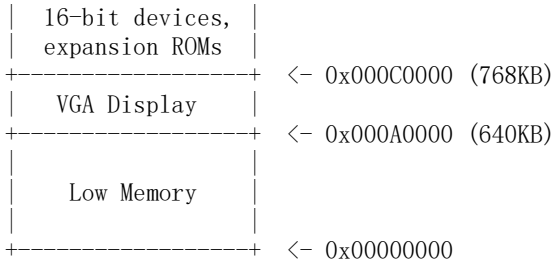
```
K> help
help - display this list of commands
kerninfo - display information about the kernel
K> kerninfo
Special kernel symbols:
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101a75 (virt)  00101a75 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112960 (virt)  00112960 (phys)
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. (We don't recommend you do this on a real machine with useful information on its hard disk, though, because copying `kernel.img` onto the beginning of its hard disk will trash the master boot record and the beginning of the first partition, effectively causing everything previously on the hard disk to be lost!)

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`. You should see something like this,

```
athena% make gdb
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(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; after all how much could it accomplish in just 16 bytes?

> **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.

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.

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 more information, see the "El Torito" Bootable CD-ROM Format Specification.

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 consists of one assembly language source file, `boot/boot.S`, and one C source file, `boot/main.c` Look through these source files carefully and make sure you understand what's going on. 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. Protected mode is described briefly in sections 1.2.7 and 1.2.8 of PC Assembly Language, and in great detail in the Intel architecture manuals. 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 IDE disk device registers via the x86's special I/O instructions. If you would like to understand better what the particular I/O instructions here mean, check out the "IDE hard drive controller" section on the 6.828 reference page. You will not need to learn much about programming specific devices in this class: writing device drivers is in practice a very important part of OS development, but from a conceptual or architectural viewpoint it is also one of the least interesting.

After you understand the boot loader source code, look at the file `obj/boot/boot.asm`. This file is a disassembly of the boot loader that our GNUmakefile creates *after* compiling the boot loader. This disassembly file makes it easy to see exactly where in physical memory all of the boot loader's code resides, and makes it easier to track what's happening while stepping through the boot loader in GDB. Likewise, `obj/kern/kernel.asm` contains a disassembly of the JOS kernel, which can often be useful for debugging.

You can set address breakpoints in GDB with the `b` command. For example, `b *0x7c00` sets a breakpoint at address 0x7C00. Once at a breakpoint, you can continue execution using the `c` and `si` commands: `c` causes QEMU to continue execution until the next breakpoint (or until you press `Ctrl-C` in GDB), and `si N` steps through the instructions *N* at a time.

To examine instructions in memory (besides the immediate next one to be executed, which GDB prints automatically), you use the `x/i` command. This command has the syntax `x/Ni ADDR`, where *N* is the number of consecutive instructions to disassemble and *ADDR* is the memory address at which to start disassembling.

> **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?
- 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?
- 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?

Loading the Kernel

We will now look in further detail at the C language portion of the boot loader, in `boot/main.c`. But before doing so, this is a good time to stop and review some of the basics of C programming.

> **Exercise 4.** Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.
>
> Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in printed lines 1 and 6 come from, how all the values in printed lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.
>
> There are other references on pointers in C (e.g., A tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.
>
> *Warning:* Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

To make sense out of `boot/main.c` you'll need to know what an ELF binary is. 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".

Full information about this format is available in the ELF specification on our reference page, but you will not need to delve very deeply into the details of this format in this class. Although as a whole the format is quite powerful and complex, most of the complex parts are for supporting dynamic loading of shared libraries, which we will not do in this class. The Wikipedia page has a short description.

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. (We will not bother setting up the hardware to prohibit writing, however.)
- `.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
```

```
(If you compiled your own toolchain, you may need to use i386-jos-elf-objdump)
```

You will see many more sections than the ones we listed above, but the others are not important for our purposes. Most of the others are to hold debugging information, which is typically included in the program's executable file but not loaded into memory by the program loader.

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.

The link address of a section is the memory address from which the section expects to execute. The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for. (It is possible to generate *position-independent* code that does not contain any such absolute addresses. This is used extensively by modern shared libraries, but it has performance and complexity costs, so we won't be using it in 6.828.)

Typically, the link and load addresses are the same. For example, look at the `.text` section of the boot loader:

```
athena% objdump -h obj/boot/boot.out
```

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!

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.

> **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.)

Part 3: The Kernel

We will now start to examine the minimal JOS kernel in a bit more detail. (And you will finally get to write some code!). Like the boot loader, the kernel begins with some assembly language code that sets things up so that C language code can execute properly.

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. Go back and check both and make sure you can see what we're talking about. (Linking the kernel is more complicated than the boot loader, so the link and load addresses are at the top of `kern/kernel.ld`.)

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. The reason for this arrangement will become clearer in the next lab.

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 (so that physical address 0x00100000 works), but this is likely to be true of any PC built after about 1990.

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. You should now see why JOS can only use the first 256MB of physical memory.

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 (or endlessly reboot if you aren't using the 6.828-patched version of QEMU).

> **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.

Formatted Printing to the Console

Most people take functions like `printf()` for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, we have to implement all I/O ourselves.

Read through `kern/printf.c`, `lib/printfmt.c`, and `kern/console.c`, and make sure you understand their relationship. It will become clear in later labs why `printfmt.c` is located in the separate `lib` directory.

> **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.

Be able to answer the following questions:

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?
2. Explain the following from `console.c`:

```
1      if (crt_pos >= CRT_SIZE) {
2              int i;
3              memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4              for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                      crt_buf[i] = 0x0700 | ' ';
6              crt_pos -= CRT_COLS;
7      }
```

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

   Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

   - In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?
   - List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.

4. Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

> *Challenge* Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference page and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

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 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.

The above exercise should give you the information you need to implement a stack backtrace function, which you should call `mon_backtrace()`. A prototype for this function is already waiting for you in `kern/monitor.c`. You can do it entirely in C, but you may find the `read_ebp()` function in `inc/x86.h` useful. You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.

The backtrace function should display a listing of function call frames in the following format:

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

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 (why?). 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. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful. (Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)

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.

Here are a few specific points you read about in K&R Chapter 5 that are worth remembering for the following exercise and for future labs.

- 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. The above rule for addition helps this definition work when the objects are larger than one byte.
- `&p[i]` is the same as `(p+i)`, yielding the address of the i'th object in the memory pointed to by p.

Although most C programs never need to cast between pointers and integers, operating systems frequently do. Whenever you see an addition involving a memory address, ask yourself whether it is an integer addition or pointer addition and make sure the value being added is appropriately multiplied or not.

> **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.

At this point, your backtrace function should give you the addresses of the function callers on the stack that lead to `mon_backtrace()` being executed. However, in practice you often want to know the function names corresponding to those addresses. For instance, you may want to know which functions could contain a bug that's causing your kernel to crash.

To help you implement this functionality, we have provided the function `debuginfo_eip()`, which looks up `eip` in the symbol table and returns the debugging information for that address. This function is defined in `kern/kdebug.c`.

> **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).

This completes the lab. In the `lab` directory, commit your changes with `git commit` and type `make handin` to submit your code.

Lab 2: Memory Management

Due Thursday, September 27, 2018

Introduction

In this lab, you will write the memory management code for your operating system. Memory management has two components.

The first component is a physical memory allocator for the kernel, so that the kernel can allocate memory and later free it. Your allocator will operate in units of 4096 bytes, called *pages*. Your task will be to maintain data structures that record which physical pages are free and which are allocated, and how many processes are sharing each allocated page. You will also write the routines to allocate and free pages of memory.

The second component of memory management is *virtual memory*, which maps the virtual addresses used by kernel and user software to addresses in physical memory. The x86 hardware's memory management unit (MMU) performs the mapping when instructions use memory, consulting a set of page tables. You will modify JOS to set up the MMU's page tables according to a specification we provide.

Getting started

In this and future labs you will progressively build up your kernel. We will also provide you with some additional source. To fetch that source, use Git to commit changes you've made since handing in lab 1 (if any), fetch the latest version of the course repository, and then create a local branch called `lab2` based on our lab2 branch, `origin/lab2`:

```
athena% cd ~/6.828/lab
athena% add git
athena% git pull
Already up-to-date.
athena% git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch refs/remotes/origin/lab2.
Switched to a new branch "lab2"
athena%
```

The `git checkout -b` command shown above actually does two things: it first creates a local branch `lab2` that is based on the `origin/lab2` branch provided by the course staff, and second, it changes the contents of your `lab` directory to reflect the files stored on the `lab2` branch. Git allows switching between existing branches using `git checkout` *`branch-name`*, though you should commit any outstanding changes on one branch before switching to a different one.

You will now need to merge the changes you made in your `lab1` branch into the `lab2` branch, as follows:

```
athena% git merge lab1
Merge made by recursive.
 kern/kdebug.c   |    11 +++++++++--
 kern/monitor.c  |    19 +++++++++++++++++++
```

```
 lib/printfmt.c |    7 +++----
 3 files changed, 31 insertions(+), 6 deletions(-)
athena%
```

In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code that is changed in the second lab assignment). In that case, the `git merge` command will tell you which files are *conflicted*, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with `git commit -a`.

Lab 2 contains the following new source files, which you should browse through:

- `inc/memlayout.h`
- `kern/pmap.c`
- `kern/pmap.h`
- `kern/kclock.h`
- `kern/kclock.c`

`memlayout.h` describes the layout of the virtual address space that you must implement by modifying `pmap.c`. `memlayout.h` and `pmap.h` define the `PageInfo` structure that you'll use to keep track of which pages of physical memory are free. `kclock.c` and `kclock.h` manipulate the PC's battery-backed clock and CMOS RAM hardware, in which the BIOS records the amount of physical memory the PC contains, among other things. The code in `pmap.c` needs to read this device hardware in order to figure out how much physical memory there is, but that part of the code is done for you: you do not need to know the details of how the CMOS hardware works.

Pay particular attention to `memlayout.h` and `pmap.h`, since this lab requires you to use and understand many of the definitions they contain. You may want to review `inc/mmu.h`, too, as it also contains a number of definitions that will be useful for this lab.

Before beginning the lab, don't forget to `add -f 6.828` to get the 6.828 version of QEMU.

Lab Requirements

In this lab and subsequent labs, do all of the regular exercises described in the lab and *at least one* challenge problem. (Some challenge problems are more challenging than others, of course!) Additionally, write up brief answers to the questions posed in the lab and a short (e.g., one or two paragraph) description of what you did to solve your chosen challenge problem. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called `answers-lab2.txt` in the top level of your `lab` directory before handing in your work.

Hand-In Procedure

When you are ready to hand in your lab code and write-up, add your `answers-lab2.txt` to the Git repository, commit your changes, and then run `make handin`.

```
athena% git add answers-lab2.txt
athena% git commit -am "my answer to lab2"
[lab2 a823de9] my answer to lab2
```

```
 4 files changed, 87 insertions(+), 10 deletions(-)
athena% make handin
```

As before, we will be grading your solutions with a grading program. You can run `make grade` in the `lab` directory to test your kernel with the grading program. You may change any of the kernel source and header files you need to in order to complete the lab, but needless to say you must not change or otherwise subvert the grading code.

Part 1: Physical Page Management

The operating system must keep track of which parts of physical RAM are free and which are currently in use. JOS manages the PC's physical memory with *page granularity* so that it can use the MMU to map and protect each piece of allocated memory.

You'll now write the physical page allocator. It keeps track of which pages are free with a linked list of `struct PageInfo` objects (which, unlike xv6, are *not* embedded in the free pages themselves), each corresponding to a physical page. You need to write the physical page allocator before you can write the rest of the virtual memory implementation, because your page table management code will need to allocate physical memory in which to store page tables.

> **Exercise 1.** In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given).
>
> `boot_alloc()`
> `mem_init()` (only up to the call to `check_page_free_list(1)`)
> `page_init()`
> `page_alloc()`
> `page_free()`
>
> `check_page_free_list()` and `check_page_alloc()` test your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct.

This lab, and all the 6.828 labs, will require you to do a bit of detective work to figure out exactly what you need to do. This assignment does not describe all the details of the code you'll have to add to JOS. Look for comments in the parts of the JOS source that you have to modify; those comments often contain specifications and hints. You will also need to look at related parts of JOS, at the Intel manuals, and perhaps at your 6.004 or 6.033 notes.

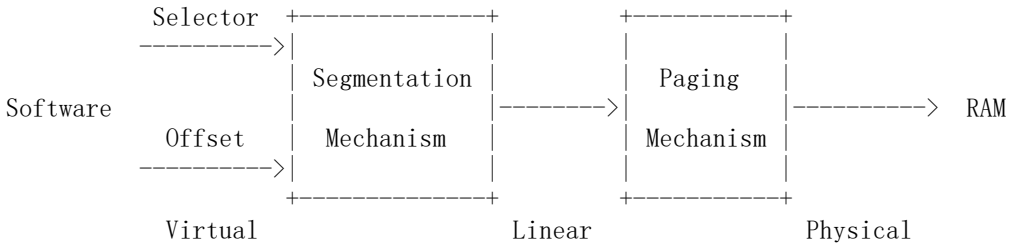
Part 2: Virtual Memory

Before doing anything else, familiarize yourself with the x86's protected-mode memory management architecture: namely *segmentation* and *page translation*.

Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses the paging hardware for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

Virtual, Linear, and Physical Addresses

In x86 terminology, a *virtual address* consists of a segment selector and an offset within the segment. A *linear address* is what you get after segment translation but before page translation. A *physical address* is what you finally get after both segment and page translation and what ultimately goes out on the hardware bus to your RAM.



A C pointer is the "offset" component of the virtual address. In `boot/boot.S`, we installed a Global Descriptor Table (GDT) that effectively disabled segment translation by setting all segment base addresses to 0 and limits to `0xffffffff`. Hence the "selector" has no effect and the linear address always equals the offset of the virtual address. In lab 3, we'll have to interact a little more with segmentation to set up privilege levels, but as for memory translation, we can ignore segmentation throughout the JOS labs and focus solely on page translation.

Recall that in part 3 of lab 1, we installed a simple page table so that the kernel could run at its link address of 0xf0100000, even though it is actually loaded in physical memory just above the ROM BIOS at 0x00100000. This page table mapped only 4MB of memory. In the virtual address space layout you are going to set up for JOS in this lab, we'll expand this to map the first 256MB of physical memory starting at virtual address 0xf0000000 and to map a number of other regions of the virtual address space.

Exercise 3. While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press `Ctrl-a c` in the terminal (the same binding returns to the serial console).

Use the `xp` command in the QEMU monitor and the `x` command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

> Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges of virtual addresses are mapped and with what permissions.

From code executing on the CPU, once we're in protected mode (which we entered first thing in `boot/boot.S`), there's no way to directly use a linear or physical address. *All* memory references are interpreted as virtual addresses and translated by the MMU, which means all pointers in C are virtual addresses.

The JOS kernel often needs to manipulate addresses as opaque values or as integers, without dereferencing them, for example in the physical memory allocator. Sometimes these are virtual addresses, and sometimes they are physical addresses. To help document the code, the JOS source distinguishes the two cases: the type `uintptr_t` represents opaque virtual addresses, and `physaddr_t` represents physical addresses. Both these types are really just synonyms for 32-bit integers (`uint32_t`), so the compiler won't stop you from assigning one type to another! Since they are integer types (not pointers), the compiler *will* complain if you try to dereference them.

The JOS kernel can dereference a `uintptr_t` by first casting it to a pointer type. In contrast, the kernel can't sensibly dereference a physical address, since the MMU translates all memory references. If you cast a `physaddr_t` to a pointer and dereference it, you may be able to load and store to the resulting address (the hardware will interpret it as a virtual address), but you probably won't get the memory location you intended.

To summarize:

C type	Address type
`T*`	Virtual
`uintptr_t`	Virtual
`physaddr_t`	Physical

> **Question**
>
> 1. Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?
>
> ```
> mystery_t x;
> char* value = return_a_pointer();
> *value = 10;
> x = (mystery_t) value;
> ```

The JOS kernel sometimes needs to read or modify memory for which it knows only the physical address. For example, adding a mapping to a page table may require allocating physical memory to store a page directory and then initializing that memory. However, the kernel cannot bypass virtual address translation and thus cannot directly load and

store to physical addresses. One reason JOS remaps all of physical memory starting from physical address 0 at virtual address 0xf0000000 is to help the kernel read and write memory for which it knows just the physical address. In order to translate a physical address into a virtual address that the kernel can actually read and write, the kernel must add 0xf0000000 to the physical address to find its corresponding virtual address in the remapped region. You should use `KADDR(pa)` to do that addition.

The JOS kernel also sometimes needs to be able to find a physical address given the virtual address of the memory in which a kernel data structure is stored. Kernel global variables and memory allocated by `boot_alloc()` are in the region where the kernel was loaded, starting at 0xf0000000, the very region where we mapped all of physical memory. Thus, to turn a virtual address in this region into a physical address, the kernel can simply subtract 0xf0000000. You should use `PADDR(va)` to do that subtraction.

Reference counting

In future labs you will often have the same physical page mapped at multiple virtual addresses simultaneously (or in the address spaces of multiple environments). You will keep a count of the number of references to each physical page in the `pp_ref` field of the `struct PageInfo` corresponding to the physical page. When this count goes to zero for a physical page, that page can be freed because it is no longer used. In general, this count should be equal to the number of times the physical page appears *below* *`UTOP`* in all page tables (the mappings above `UTOP` are mostly set up at boot time by the kernel and should never be freed, so there's no need to reference count them). We'll also use it to keep track of the number of pointers we keep to the page directory pages and, in turn, of the number of references the page directories have to page table pages.

Be careful when using `page_alloc`. The page it returns will always have a reference count of 0, so `pp_ref` should be incremented as soon as you've done something with the returned page (like inserting it into a page table). Sometimes this is handled by other functions (for example, `page_insert`) and sometimes the function calling `page_alloc` must do it directly.

Page Table Management

Now you'll write a set of routines to manage page tables: to insert and remove linear-to-physical mappings, and to create page table pages when needed.

> **Exercise 4.** In the file `kern/pmap.c`, you must implement code for the following functions.
>
> ```
> pgdir_walk()
> boot_map_region()
> page_lookup()
> page_remove()
> page_insert()
> ```
>
> `check_page()`, called from `mem_init()`, tests your page table management routines. You should make sure it reports success before proceeding.

Part 3: Kernel Address Space

JOS divides the processor's 32-bit linear address space into two parts. User environments (processes), which we will begin loading and running in lab 3, will have control over the layout and contents of the lower part, while the kernel always maintains complete control over the upper part. The dividing line is defined somewhat arbitrarily by the symbol `ULIM` in `inc/memlayout.h`, reserving approximately 256MB of virtual address space for the kernel. This explains why we needed to give the kernel such a high link address in lab 1: otherwise there would not be enough room in the kernel's virtual address space to map in a user environment below it at the same time.

You'll find it helpful to refer to the JOS memory layout diagram in `inc/memlayout.h` both for this part and for later labs.

Permissions and Fault Isolation

Since kernel and user memory are both present in each environment's address space, we will have to use permission bits in our x86 page tables to allow user code access only to the user part of the address space. Otherwise bugs in user code might overwrite kernel data, causing a crash or more subtle malfunction; user code might also be able to steal other environments' private data. Note that the writable permission bit (`PTE_W`) affects both user and kernel code!

The user environment will have no permission to any of the memory above `ULIM`, while the kernel will be able to read and write this memory. For the address range `[UTOP,ULIM)`, both the kernel and the user environment have the same permission: they can read but not write this address range. This range of address is used to expose certain kernel data structures read-only to the user environment. Lastly, the address space below `UTOP` is for the user environment to use; the user environment will set permissions for accessing this memory.

Initializing the Kernel Address Space

Now you'll set up the address space above `UTOP`: the kernel part of the address space. `inc/memlayout.h` shows the layout you should use. You'll use the functions you just wrote to set up the appropriate linear to physical mappings.

> **Exercise 5.** Fill in the missing code in `mem_init()` after the call to `check_page()`.
>
> Your code should now pass the `check_kern_pgdir()` and `check_page_installed_pgdir()` checks.

> **Question**

2. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

3. We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
4. What is the maximum amount of physical memory that this operating system can support? Why?
5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?
6. Revisit the page table setup in `kern/entry.S` and `kern/entrypgdir.c`. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

Challenge! We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE_PS ("Page Size") bit in the page directory entries. This bit was *not* supported in the original 80386, but is supported on more recent x86 processors. You will therefore have to refer to Volume 3 of the current Intel manuals. Make sure you design the kernel to use this optimization only on processors that support it!

Challenge! Extend the JOS kernel monitor with commands to:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range

of virtual/linear addresses in the currently active address space. For example, you might enter `'showmappings 0x3000 0x5000'` to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

Address Space Layout Alternatives

The address space layout we use in JOS is not the only one possible. An operating system might map the kernel at low linear addresses while leaving the *upper* part of the linear address space for user processes. x86 kernels generally do not take this approach, however, because one of the x86's backward-compatibility modes, known as *virtual 8086 mode*, is "hard-wired" in the processor to use the bottom part of the linear address space, and thus cannot be used at all if the kernel is mapped there.

It is even possible, though much more difficult, to design the kernel so as not to have to reserve *any* fixed portion of the processor's linear or virtual address space for itself, but instead effectively to allow user-level processes unrestricted use of the *entire* 4GB of virtual address space - while still fully protecting the kernel from these processes and protecting different processes from each other!

Challenge! Each user-level environment maps the kernel. Change JOS so that the kernel has its own page table and so that a user-level environment runs with a minimal number of kernel pages mapped. That is, each user-level environment maps just enough pages mapped so that the user-level environment can enter and leave the kernel correctly. You also have to come up with a plan for the kernel to read/write arguments to system calls.

Challenge! Write up an outline of how a kernel could be designed to allow user environments unrestricted use of the full 4GB virtual and linear address space. Hint: do the previous challenge exercise first, which reduces the kernel to a few mappings in a user environment. Hint: the technique is sometimes known as "*follow the bouncing kernel*." In your design, be sure to address exactly what has to happen when the processor transitions between kernel and user modes, and how the kernel would accomplish such transitions. Also describe how the kernel would access physical memory and I/O devices in this scheme, and how the kernel would access a user environment's virtual

address space during system calls and the like. Finally, think about and describe the advantages and disadvantages of such a scheme in terms of flexibility, performance, kernel complexity, and other factors you can think of.

Challenge! Since our JOS kernel's memory management system only allocates and frees memory on page granularity, we do not have anything comparable to a general-purpose `malloc`/`free` facility that we can use within the kernel. This could be a problem if we want to support certain types of I/O devices that require *physically contiguous* buffers larger than 4KB in size, or if we want user-level environments, and not just the kernel, to be able to allocate and map 4MB *superpages* for maximum processor efficiency. (See the earlier challenge problem about PTE_PS.)

Generalize the kernel's memory allocation system to support pages of a variety of power-of-two allocation unit sizes from 4KB up to some reasonable maximum of your choice. Be sure you have some way to divide larger allocation units into smaller ones on demand, and to coalesce multiple small allocation units back into larger units when possible. Think about the issues that might arise in such a system.

This completes the lab. Make sure you pass all of the `make grade` tests and don't forget to write up your answers to the questions and a description of your challenge exercise solution in `answers-lab2.txt`. Commit your changes (including adding `answers-lab2.txt`) and type `make handin` in the `lab` directory to hand in your lab.

Lab 3: User Environments

Part A due Thursday, October 4, 2018
Part B due Thursday, October 11, 2018

Introduction

In this lab you will implement the basic kernel facilities required to get a protected user-mode environment (i.e., "process") running. You will enhance the JOS kernel to set up the data structures to keep track of user environments, create a single user environment, load a program image into it, and start it running. You will also make the JOS kernel capable of handling any system calls the user environment makes and handling any other exceptions it causes.

Note: In this lab, the terms *environment* and *process* are interchangeable - both refer to an abstraction that allows you to run a program. We introduce the term "environment" instead of the traditional term "process" in order to stress the point that JOS environments and UNIX processes provide different interfaces, and do not provide the same semantics.

Getting Started

Use Git to commit your changes after your Lab 2 submission (if any), fetch the latest version of the course repository, and then create a local branch called `lab3` based on our lab3 branch, `origin/lab3`:

```
athena% cd ~/6.828/lab
athena% add git
athena% git commit -am 'changes to lab2 after handin'
Created commit 734fab7: changes to lab2 after handin
 4 files changed, 42 insertions(+), 9 deletions(-)
athena% git pull
Already up-to-date.
athena% git checkout -b lab3 origin/lab3
Branch lab3 set up to track remote branch refs/remotes/origin/lab3.
Switched to a new branch "lab3"
athena% git merge lab2
Merge made by recursive.
 kern/pmap.c |   42 ++++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
athena%
```

Lab 3 contains a number of new source files, which you should browse:

inc/	env.h	Public definitions for user-mode environments
	trap.h	Public definitions for trap handling
	syscall.h	Public definitions for system calls from user environments to the kernel
	lib.h	Public definitions for the user-mode support library
kern/	env.h	Kernel-private definitions for user-mode environments
	env.c	Kernel code implementing user-mode environments
	trap.h	

		Kernel-private trap handling definitions
	`trap.c`	Trap handling code
	`trapentry.S`	Assembly-language trap handler entry-points
	`syscall.h`	Kernel-private definitions for system call handling
	`syscall.c`	System call implementation code
`lib/`	`Makefrag`	Makefile fragment to build user-mode library, `obj/lib/libjos.a`
	`entry.S`	Assembly-language entry-point for user environments
	`libmain.c`	User-mode library setup code called from `entry.S`
	`syscall.c`	User-mode system call stub functions
	`console.c`	User-mode implementations of `putchar` and `getchar`, providing console I/O
	`exit.c`	User-mode implementation of `exit`
	`panic.c`	User-mode implementation of `panic`
`user/*`		Various test programs to check kernel lab 3 code

In addition, a number of the source files we handed out for lab2 are modified in lab3. To see the differences, you can type:

```
$ git diff lab2
```

You may also want to take another look at the lab tools guide, as it includes information on debugging user code that becomes relevant in this lab.

Lab Requirements

This lab is divided into two parts, A and B. Part A is due a week after this lab was assigned; you should commit your changes and `make handin` your lab before the Part A deadline, making sure your code passes all of the Part A tests (it is okay if your code does not pass the Part B tests yet). You only need to have the Part B tests passing by the Part B deadline at the end of the second week.

As in lab 2, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem (for the entire lab, not for each part). Write up brief answers to the questions posed in the lab and a one or two paragraph description of what you did to solve your chosen challenge problem in a file called `answers-lab3.txt` in the top level of your `lab` directory. (If you implement more than one challenge problem, you only need to describe one of them in the write-up.) Do not forget to include the answer file in your submission with `git add answers-lab3.txt`.

Inline Assembly

In this lab you may find GCC's inline assembly language feature useful, although it is also possible to complete the lab without using it. At the very least, you will need to be able to understand the fragments of inline assembly language ("`asm`" statements) that already exist in the source code we gave you. You can find several sources of information on GCC inline assembly language on the class reference materials page.

Part A: User Environments and Exception Handling

The new include file `inc/env.h` contains basic definitions for user environments in JOS. Read it now. The kernel uses the `Env` data structure to keep track of each user environment. In this lab you will initially create just one environment, but you will need to design the JOS kernel to support multiple environments; lab 4 will take advantage of this feature by allowing a user environment to `fork` other environments.

As you can see in `kern/env.c`, the kernel maintains three main global variables pertaining to environments:

```
struct Env *envs = NULL;		// All environments
struct Env *curenv = NULL;		// The current env
static struct Env *env_free_list;	// Free environment list
```

Once JOS gets up and running, the `envs` pointer points to an array of `Env` structures representing all the environments in the system. In our design, the JOS kernel will support a maximum of `NENV` simultaneously active environments, although there will typically be far fewer running environments at any given time. (`NENV` is a constant `#define`'d in `inc/env.h`.) Once it is allocated, the `envs` array will contain a single instance of the `Env` data structure for each of the `NENV` possible environments.

The JOS kernel keeps all of the inactive `Env` structures on the `env_free_list`. This design allows easy allocation and deallocation of environments, as they merely have to be added to or removed from the free list.

The kernel uses the `curenv` symbol to keep track of the *currently executing* environment at any given time. During boot up, before the first environment is run, `curenv` is initially set to `NULL`.

Environment State

The `Env` structure is defined in `inc/env.h` as follows (although more fields will be added in future labs):

```
struct Env {
	struct Trapframe env_tf;	// Saved registers
	struct Env *env_link;		// Next free Env
	envid_t env_id;			// Unique environment identifier
	envid_t env_parent_id;		// env_id of this env's parent
	enum EnvType env_type;		// Indicates special system environments
	unsigned env_status;		// Status of the environment
	uint32_t env_runs;		// Number of times environment has run

	// Address space
	pde_t *env_pgdir;		// Kernel virtual address of page dir
};
```

Here's what the `Env` fields are for:

env_tf:
This structure, defined in `inc/trap.h`, holds the saved register values for the environment while that environment is *not* running: i.e., when the kernel or a

different environment is running. The kernel saves these when switching from user to kernel mode, so that the environment can later be resumed where it left off.

env_link:
This is a link to the next `Env` on the `env_free_list`. `env_free_list` points to the first free environment on the list.

env_id:
The kernel stores here a value that uniquely identifiers the environment currently using this `Env` structure (i.e., using this particular slot in the `envs` array). After a user environment terminates, the kernel may re-allocate the same `Env` structure to a different environment - but the new environment will have a different `env_id` from the old one even though the new environment is re-using the same slot in the `envs` array.

env_parent_id:
The kernel stores here the `env_id` of the environment that created this environment. In this way the environments can form a "family tree," which will be useful for making security decisions about which environments are allowed to do what to whom.

env_type:
This is used to distinguish special environments. For most environments, it will be `ENV_TYPE_USER`. We'll introduce a few more types for special system service environments in later labs.

env_status:
This variable holds one of the following values:

`ENV_FREE`:
Indicates that the `Env` structure is inactive, and therefore on the `env_free_list`.

`ENV_RUNNABLE`:
Indicates that the `Env` structure represents an environment that is waiting to run on the processor.

`ENV_RUNNING`:
Indicates that the `Env` structure represents the currently running environment.

`ENV_NOT_RUNNABLE`:
Indicates that the `Env` structure represents a currently active environment, but it is not currently ready to run: for example, because it is waiting for an interprocess communication (IPC) from another environment.

`ENV_DYING`:
Indicates that the `Env` structure represents a zombie environment. A zombie environment will be freed the next time it traps to the kernel. We will not use this flag until Lab 4.

env_pgdir:
This variable holds the kernel *virtual address* of this environment's page directory.

Like a Unix process, a JOS environment couples the concepts of "thread" and "address space". The thread is defined primarily by the saved registers (the `env_tf` field), and the address space is defined by the page directory and page tables pointed to by `env_pgdir`. To run an environment, the kernel must set up the CPU with *both* the saved registers and the appropriate address space.

Our `struct Env` is analogous to `struct proc` in xv6. Both structures hold the environment's (i.e., process's) user-mode register state in a `Trapframe` structure. In JOS, individual environments do not have their own kernel stacks as processes do in xv6. There can be only one JOS environment active in the kernel at a time, so JOS needs only a *single* kernel stack.

Allocating the Environments Array

In lab 2, you allocated memory in `mem_init()` for the `pages[]` array, which is a table the kernel uses to keep track of which pages are free and which are not. You will now need to modify `mem_init()` further to allocate a similar array of `Env` structures, called `envs`.

> **Exercise 1.** Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user read-only at `UENVS` (defined in `inc/memlayout.h`) so user processes can read from this array.
>
> You should run your code and make sure `check_kern_pgdir()` succeeds.

Creating and Running Environments

You will now write the code in `kern/env.c` necessary to run a user environment. Because we do not yet have a filesystem, we will set up the kernel to load a static binary image that is *embedded within the kernel itself*. JOS embeds this binary in the kernel as a ELF executable image.

The Lab 3 `GNUmakefile` generates a number of binary images in the `obj/user/` directory. If you look at `kern/Makefrag`, you will notice some magic that "links" these binaries directly into the kernel executable as if they were `.o` files. The `-b binary` option on the linker command line causes these files to be linked in as "raw" uninterpreted binary files rather than as regular `.o` files produced by the compiler. (As far as the linker is concerned, these files do not have to be ELF images at all - they could be anything, such as text files or pictures!) If you look at `obj/kern/kernel.sym` after building the kernel, you will notice that the linker has "magically" produced a number of funny symbols with obscure names like `_binary_obj_user_hello_start`, `_binary_obj_user_hello_end`, and `_binary_obj_user_hello_size`. The linker generates these symbol names by mangling the file names of the binary files; the symbols provide the regular kernel code with a way to reference the embedded binary files.

In `i386_init()` in `kern/init.c` you'll see code to run one of these binary images in an environment. However, the critical functions to set up user environments are not complete; you will need to fill them in.

> **Exercise 2.** In the file `env.c`, finish coding the following functions:
>
> `env_init()`

Initialize all of the `Env` structures in the `envs` array and add them to the `env_free_list`. Also calls `env_init_percpu`, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).

`env_setup_vm()`
Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

`region_alloc()`
Allocates and maps physical memory for an environment

`load_icode()`
You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.

`env_create()`
Allocate an environment with `env_alloc` and call `load_icode` to load an ELF binary into it.

`env_run()`
Start a given environment running in user mode.

As you write these functions, you might find the new cprintf verb `%e` useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

Below is a call graph of the code up to the point where the user code is invoked. Make sure you understand the purpose of each step.

- `start` (`kern/entry.S`)
- `i386_init` (`kern/init.c`)
  - `cons_init`
  - `mem_init`
  - `env_init`
  - `trap_init` (still incomplete at this point)
  - `env_create`
  - `env_run`
    - `env_pop_tf`

Once you are done you should compile your kernel and run it under QEMU. If all goes well, your system should enter user space and execute the `hello` binary until it makes a system call with the `int` instruction. At that point there will be trouble, since JOS has not set up the hardware to allow any kind of transition from user space into the kernel. When the CPU discovers that it is not set up to handle this system call interrupt, it will generate a general protection exception, find that it can't handle that, generate a double fault exception, find that it can't handle that either, and finally give up with what's known as a "triple fault". Usually, you would then see the CPU reset and the system reboot. While this is important for legacy applications (see this blog post for an

explanation of why), it's a pain for kernel development, so with the 6.828 patched QEMU you'll instead see a register dump and a "Triple fault." message.

We'll address this problem shortly, but for now we can use the debugger to check that we're entering user mode. Use `make qemu-gdb` and set a GDB breakpoint at `env_pop_tf`, which should be the last function you hit before actually entering user mode. Single step through this function using `si`; the processor should enter user mode after the `iret` instruction. You should then see the first instruction in the user environment's executable, which is the `cmpl` instruction at the label `start` in `lib/entry.S`. Now use `b *0x...` to set a breakpoint at the `int $0x30` in `sys_cputs()` in `hello` (see `obj/user/hello.asm` for the user-space address). This `int` is the system call to display a character to the console. If you cannot execute as far as the `int`, then something is wrong with your address space setup or program loading code; go back and fix it before continuing.

Handling Interrupts and Exceptions

At this point, the first `int $0x30` system call instruction in user space is a dead end: once the processor gets into user mode, there is no way to get back out. You will now need to implement basic exception and system call handling, so that it is possible for the kernel to recover control of the processor from user-mode code. The first thing you should do is thoroughly familiarize yourself with the x86 interrupt and exception mechanism.

> **Exercise 3.** Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

In this lab we generally follow Intel's terminology for interrupts, exceptions, and the like. However, terms such as exception, trap, interrupt, fault and abort have no standard meaning across architectures or operating systems, and are often used without regard to the subtle distinctions between them on a particular architecture such as the x86. When you see these terms outside of this lab, the meanings might be slightly different.

Basics of Protected Control Transfer

Exceptions and interrupts are both "protected control transfers," which cause the processor to switch from user to kernel mode (CPL=0) without giving the user-mode code any opportunity to interfere with the functioning of the kernel or other environments. In Intel's terminology, an *interrupt* is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity. An *exception*, in contrast, is a protected control transfer caused synchronously by the currently running code, for example due to a divide by zero or an invalid memory access.

In order to ensure that these protected control transfers are actually *protected*, the processor's interrupt/exception mechanism is designed so that the code currently running when the interrupt or exception occurs *does not get to choose arbitrarily where the kernel is entered or how*. Instead, the processor ensures that the kernel can be

entered only under carefully controlled conditions. On the x86, two mechanisms work together to provide this protection:

1. **The Interrupt Descriptor Table.** The processor ensures that interrupts and exceptions can only cause the kernel to be entered at a few specific, well-defined entry-points *determined by the kernel itself*, and not by the code running when the interrupt or exception is taken.

   The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 255. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's *interrupt descriptor table* (IDT), which the kernel sets up in kernel-private memory, much like the GDT. From the appropriate entry in this table the processor loads:

   - the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
   - the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. (In JOS, all exceptions are handled in kernel mode, privilege level 0.)

2. **The Task State Segment.** The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of `EIP` and `CS` before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel.

   For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.

   Although the TSS is large and can potentially serve a variety of purposes, JOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in JOS is privilege level 0 on the x86, the processor uses the `ESP0` and `SS0` fields of the TSS to define the kernel stack when entering kernel mode. JOS doesn't use any other TSS fields.

Types of Exceptions and Interrupts

All of the synchronous exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31. For example, a page fault always causes an exception through vector 14. Interrupt vectors greater than 31 are only used by *software interrupts*, which can be generated by the `int`

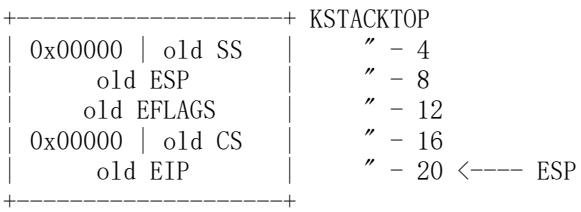
instruction, or asynchronous *hardware interrupts*, caused by external devices when they need attention.

In this section we will extend JOS to handle the internally generated x86 exceptions in vectors 0-31. In the next section we will make JOS handle software interrupt vector 48 (0x30), which JOS (fairly arbitrarily) uses as its system call interrupt vector. In Lab 4 we will extend JOS to handle externally generated hardware interrupts such as the clock interrupt.

An Example

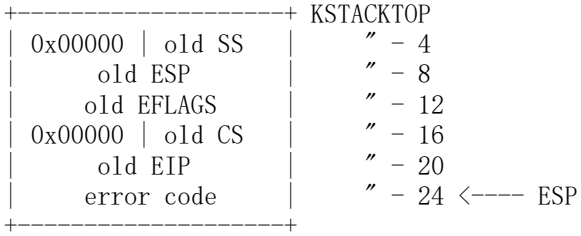
Let's put these pieces together and trace through an example. Let's say the processor is executing code in a user environment and encounters a divide instruction that attempts to divide by zero.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the TSS, which in JOS will hold the values `GD_KD` and `KSTACKTOP`, respectively.
2. The processor pushes the exception parameters on the kernel stack, starting at address `KSTACKTOP`:



3. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets `CS:EIP` to point to the handler function described by the entry.
4. The handler function takes control and handles the exception, for example by terminating the user environment.

For certain types of x86 exceptions, in addition to the "standard" five words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the 80386 manual to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the exception handler when coming in from user mode:



Nested Exceptions and Interrupts

The processor can take exceptions and interrupts both from kernel and user mode. It is only when entering the kernel from user mode, however, that the x86 processor automatically switches stacks before pushing its old register state onto the stack and invoking the appropriate exception handler through the IDT. If the processor is *already* in kernel mode when the interrupt or exception occurs (the low 2 bits of the `CS` register are already zero), then the CPU just pushes more values on the same kernel stack. In this way, the kernel can gracefully handle *nested exceptions* caused by code within the kernel itself. This capability is an important tool in implementing protection, as we will see later in the section on system calls.

If the processor is already in kernel mode and takes a nested exception, since it does not need to switch stacks, it does not save the old `SS` or `ESP` registers. For exception types that do not push an error code, the kernel stack therefore looks like the following on entry to the exception handler:

For exception types that push an error code, the processor pushes the error code immediately after the old `EIP`, as before.

There is one important caveat to the processor's nested exception capability. If the processor takes an exception while already in kernel mode, and *cannot push its old state onto the kernel stack* for any reason such as lack of stack space, then there is nothing the processor can do to recover, so it simply resets itself. Needless to say, the kernel should be designed so that this can't happen.

Setting Up the IDT

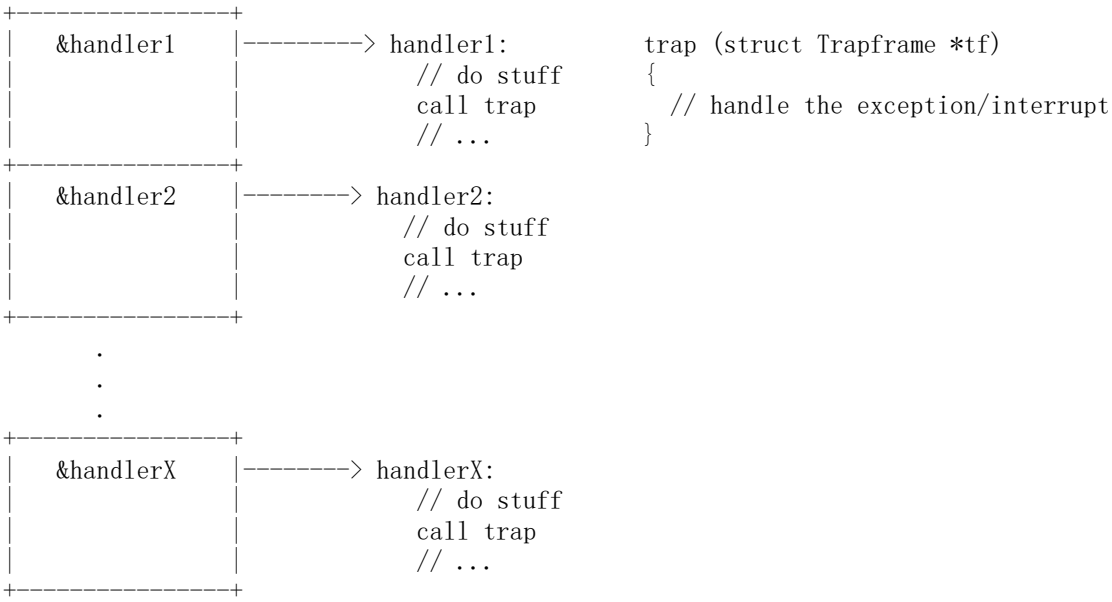
You should now have the basic information you need in order to set up the IDT and handle exceptions in JOS. For now, you will set up the IDT to handle interrupt vectors 0-31 (the processor exceptions). We'll handle system call interrupts later in this lab and add interrupts 32-47 (the device IRQs) in a later lab.

The header files `inc/trap.h` and `kern/trap.h` contain important definitions related to interrupts and exceptions that you will need to become familiar with. The file `kern/trap.h` contains definitions that are strictly private to the kernel, while `inc/trap.h` contains definitions that may also be useful to user-level programs and libraries.

Note: Some of the exceptions in the range 0-31 are defined by Intel to be reserved. Since they will never be generated by the processor, it doesn't really matter how you handle them. Do whatever you think is cleanest.

The overall flow of control that you should achieve is depicted below:

```
IDT                   trapentry.S         trap.c
```



Each exception or interrupt should have its own handler in `trapentry.S` and `trap_init()` should initialize the IDT with the addresses of these handlers. Each of the handlers should build a `struct Trapframe` (see `inc/trap.h`) on the stack and call `trap()` (in `trap.c`) with a pointer to the Trapframe. `trap()` then handles the exception/interrupt or dispatches to a specific handler function.

Exercise 4. Edit `trapentry.S` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.S` should help you, as well as the T_* defines in `inc/trap.h`. You will need to add an entry point in `trapentry.S` (using those macros) for each trap defined in `inc/trap.h`, and you'll have to provide `_alltraps` which the `TRAPHANDLER` macros refer to. You will also need to modify `trap_init()` to initialize the `idt` to point to each of these entry points defined in `trapentry.S`; the `SETGATE` macro will be helpful here.

Your `_alltraps` should:

1. push values to make the stack look like a struct Trapframe
2. load `GD_KD` into `%ds` and `%es`
3. `pushl %esp` to pass a pointer to the Trapframe as an argument to trap()
4. `call trap` (can `trap` ever return?)

Consider using the `pushal` instruction; it fits nicely with the layout of the `struct Trapframe`.

Test your trap handling code using some of the test programs in the `user` directory that cause exceptions before making any system calls, such as `user/divzero`. You should be able to get `make grade` to succeed on the `divzero`, `softint`, and `badsegment` tests at this point.

Challenge! You probably have a lot of very similar code right now, between the lists of `TRAPHANDLER` in `trapentry.S` and their installations in `trap.c`. Clean this up. Change the macros in `trapentry.S` to automatically generate a table for `trap.c` to use. Note that you can switch between laying down code and data in the assembler by using the directives `.text` and `.data`.

Questions

Answer the following questions in your `answers-lab3.txt`:

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)
2. Did you have to do anything to make the `user/softint` program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but `softint`'s code says `int $14`. *Why* should this produce interrupt vector 13? What happens if the kernel actually allows `softint`'s `int $14` instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

This concludes part A of the lab. Don't forget to add `answers-lab3.txt`, commit your changes, and run `make handin` before the part A deadline.

Part B: Page Faults, Breakpoints Exceptions, and System Calls

Now that your kernel has basic exception handling capabilities, you will refine it to provide important operating system primitives that depend on exception handling.

Handling Page Faults

The page fault exception, interrupt vector 14 (`T_PGFLT`), is a particularly important one that we will exercise heavily throughout this lab and the next. When the processor takes a page fault, it stores the linear (i.e., virtual) address that caused the fault in a special processor control register, `CR2`. In `trap.c` we have provided the beginnings of a special function, `page_fault_handler()`, to handle page fault exceptions.

Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get `make grade` to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them. Remember that you

can boot JOS into a particular user program using `make run-x` or `make run-x-nox`. For instance, `make run-hello-nox` runs the *hello* user program.

You will further refine the kernel's page fault handling below, as you implement system calls.

The Breakpoint Exception

The breakpoint exception, interrupt vector 3 (`T_BRKPT`), is normally used to allow debuggers to insert breakpoints in a program's code by temporarily replacing the relevant program instruction with the special 1-byte `int3` software interrupt instruction. In JOS we will abuse this exception slightly by turning it into a primitive pseudo-system call that any user environment can use to invoke the JOS kernel monitor. This usage is actually somewhat appropriate if we think of the JOS kernel monitor as a primitive debugger. The user-mode implementation of `panic()` in `lib/panic.c`, for example, performs an `int3` after displaying its panic message.

Exercise 6. Modify `trap_dispatch()` to make breakpoint exceptions invoke the kernel monitor. You should now be able to get `make grade` to succeed on the `breakpoint` test.

Challenge! Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the `int3`, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the `EFLAGS` register in order to implement single-stepping.

Optional: If you're feeling really adventurous, find some x86 disassembler source code - e.g., by ripping it out of QEMU, or out of GNU binutils, or just write it yourself - and extend the JOS kernel monitor to be able to disassemble and display instructions as you are stepping through them. Combined with the symbol table loading from lab 1, this is the stuff of which real kernel debuggers are made.

Questions

3. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to `SETGATE` from `trap_init`). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?
4. What do you think is the point of these mechanisms, particularly in light of what the `user/softint` test program does?

System calls

User processes ask the kernel to do things for them by invoking system calls. When the user process invokes a system call, the processor enters kernel mode, the processor and the kernel cooperate to save the user process's state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process. The exact details of how the user process gets the kernel's attention and how it specifies which call it wants to execute vary from system to system.

In the JOS kernel, we will use the `int` instruction, which causes a processor interrupt. In particular, we will use `int $0x30` as the system call interrupt. We have defined the constant `T_SYSCALL` to 48 (0x30) for you. You will have to set up the interrupt descriptor to allow user processes to cause that interrupt. Note that interrupt 0x30 cannot be generated by hardware, so there is no ambiguity caused by allowing user code to generate it.

The application will pass the system call number and the system call arguments in registers. This way, the kernel won't need to grub around in the user environment's stack or instruction stream. The system call number will go in `%eax`, and the arguments (up to five of them) will go in `%edx`, `%ecx`, `%ebx`, `%edi`, and `%esi`, respectively. The kernel passes the return value back in `%eax`. The assembly code to invoke a system call has been written for you, in `syscall()` in `lib/syscall.c`. You should read through it and make sure you understand what is going on.

> **Exercise 7.** Add a handler in the kernel for interrupt vector `T_SYSCALL`. You will have to edit `kern/trapentry.S` and `kern/trap.c`'s `trap_init()`. You also need to change `trap_dispatch()` to handle the system call interrupt by calling `syscall()` (defined in `kern/syscall.c`) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall()` in `kern/syscall.c`. Make sure `syscall()` returns `-E_INVAL` if the system call number is invalid. You should read and understand `lib/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in `inc/syscall.h` by invoking the corresponding kernel function for each call.
>
> Run the `user/hello` program under your kernel (`make run-hello`). It should print "`hello, world`" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get `make grade` to succeed on the `testbss` test.

> *Challenge!* Implement system calls using the `sysenter` and `sysexit` instructions instead of using `int 0x30` and `iret`.

The `sysenter/sysexit` instructions were designed by Intel to be faster than `int/iret`. They do this by using registers instead of the stack and by making assumptions about how the segmentation registers are used. The exact details of these instructions can be found in Volume 2B of the Intel reference manuals.

The easiest way to add support for these instructions in JOS is to add a `sysenter_handler` in `kern/trapentry.S` that saves enough information about the user environment to return to it, sets up the kernel environment, pushes the arguments to `syscall()` and calls `syscall()` directly. Once `syscall()` returns, set everything up for and execute the `sysexit` instruction. You will also need to add code to `kern/init.c` to set up the necessary model specific registers (MSRs). Section 6.1.2 in Volume 2 of the AMD Architecture Programmer's Manual and the reference on SYSENTER in Volume 2B of the Intel reference manuals give good descriptions of the relevant MSRs. You can find an implementation of `wrmsr` to add to `inc/x86.h` for writing to these MSRs here.

Finally, `lib/syscall.c` must be changed to support making a system call with `sysenter`. Here is a possible register layout for the `sysenter` instruction:

```
eax                - syscall number
edx, ecx, ebx, edi - arg1, arg2, arg3, arg4
esi                - return pc
ebp                - return esp
esp                - trashed by sysenter
```

GCC's inline assembler will automatically save registers that you tell it to load values directly into. Don't forget to either save (push) and restore (pop) other registers that you clobber, or tell the inline assembler that you're clobbering them. The inline assembler doesn't support saving `%ebp`, so you will need to add code to save and restore it yourself. The return address can be put into `%esi` by using an instruction like `leal after_sysenter_label, %%esi`.

Note that this only supports 4 arguments, so you will need to leave the old method of doing system calls around to support 5 argument system calls. Furthermore, because this fast path doesn't update the current environment's trap frame, it won't be suitable for some of the system calls we add in later labs.

You may have to revisit your code once we enable asynchronous interrupts in the next lab. Specifically, you'll need to enable interrupts when returning to the user process, which `sysexit` doesn't do for you.

User-mode startup

A user program starts running at the top of `lib/entry.S`. After some setup, this code calls `libmain()`, in `lib/libmain.c`. You should modify `libmain()` to initialize the global pointer `thisenv` to point at this environment's `struct Env` in the `envs[]` array. (Note that `lib/entry.S` has already defined `envs` to point at the `UENVS` mapping you set up in Part A.) Hint: look in `inc/env.h` and use `sys_getenvid`.

`libmain()` then calls `umain`, which, in the case of the hello program, is in `user/hello.c`. Note that after printing "`hello, world`", it tries to access `thisenv->env_id`. This is why it faulted earlier. Now that you've initialized `thisenv` properly, it should not fault. If it still faults, you probably haven't mapped the `UENVS` area user-readable (back in Part A in `pmap.c`; this is the first time we've actually used the `UENVS` area).

> **Exercise 8.** Add the required code to the user library, then boot your kernel. You should see `user/hello` print "`hello, world`" and then print "`i am environment 00001000`". `user/hello` then attempts to "exit" by calling `sys_env_destroy()` (see `lib/libmain.c` and `lib/exit.c`). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get `make grade` to succeed on the `hello` test.

Page faults and memory protection

Memory protection is a crucial feature of an operating system, ensuring that bugs in one program cannot corrupt other programs or corrupt the operating system itself.

Operating systems usually rely on hardware support to implement memory protection. The OS keeps the hardware informed about which virtual addresses are valid and which are not. When a program tries to access an invalid address or one for which it has no permissions, the processor stops the program at the instruction causing the fault and then traps into the kernel with information about the attempted operation. If the fault is fixable, the kernel can fix it and let the program continue running. If the fault is not fixable, then the program cannot continue, since it will never get past the instruction causing the fault.

As an example of a fixable fault, consider an automatically extended stack. In many systems the kernel initially allocates a single stack page, and then if a program faults accessing pages further down the stack, the kernel will allocate those pages automatically and let the program continue. By doing this, the kernel only allocates as much stack memory as the program needs, but the program can work under the illusion that it has an arbitrarily large stack.

System calls present an interesting problem for memory protection. Most system call interfaces let user programs pass pointers to the kernel. These pointers point at user buffers to be read or written. The kernel then dereferences these pointers while carrying out the system call. There are two problems with this:

1. A page fault in the kernel is potentially a lot more serious than a page fault in a user program. If the kernel page-faults while manipulating its own data structures, that's a kernel bug, and the fault handler should panic the kernel (and hence the whole system). But when the kernel is dereferencing pointers given to it by the user program, it needs a way to remember that any page faults these dereferences cause are actually on behalf of the user program.
2. The kernel typically has more memory permissions than the user program. The user program might pass a pointer to a system call that points to memory that the kernel can read or write but that the program cannot. The kernel must be careful not to be tricked into dereferencing such a pointer, since that might reveal private information or destroy the integrity of the kernel.

For both of these reasons the kernel must be extremely careful when handling pointers presented by user programs.

You will now solve these two problems with a single mechanism that scrutinizes all pointers passed from userspace into the kernel. When a program passes the kernel a pointer, the kernel will check that the address is in the user part of the address space, and that the page table would allow the memory operation.

Thus, the kernel will never suffer a page fault due to dereferencing a user-supplied pointer. If the kernel does page fault, it should panic and terminate.

Exercise 9. Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change `kern/syscall.c` to sanity check arguments to system calls.

Boot your kernel, running `user/buggyhello`. The environment should be destroyed, and the kernel should *not* panic. You should see:

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

Finally, change `debuginfo_eip` in `kern/kdebug.c` to call `user_mem_check` on `usd`, `stabs`, and `stabstr`. If you now run `user/breakpoint`, you should be able to run `backtrace` from the kernel monitor and see the backtrace traverse into `lib/libmain.c` before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens.

Note that the same mechanism you just implemented also works for malicious user applications (such as `user/evilhello`).

> **Exercise 10.** Boot your kernel, running `user/evilhello`. The environment should be destroyed, and the kernel should not panic. You should see:
>
> ```
> [00000000] new env 00001000
> ...
> [00001000] user_mem_check assertion failure for va f010000c
> [00001000] free env 00001000
> ```

This completes the lab. Make sure you pass all of the `make grade` tests and don't forget to write up your answers to the questions and a description of your challenge exercise solution in `answers-lab3.txt`. Commit your changes and type `make handin` in the `lab` directory to submit your work.

Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab3.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 3'`, then `make handin` and follow the directions.

Lab 4: Preemptive Multitasking

Part A due Thursday, October 18, 2018
Part B due Thursday, October 25, 2018
Part C due Thursday, November 1, 2018

Introduction

In this lab you will implement preemptive multitasking among multiple simultaneously active user-mode environments.

In part A you will add multiprocessor support to JOS, implement round-robin scheduling, and add basic environment management system calls (calls that create and destroy environments, and allocate/map memory).

In part B, you will implement a Unix-like `fork()`, which allows a user-mode environment to create copies of itself.

Finally, in part C you will add support for inter-process communication (IPC), allowing different user-mode environments to communicate and synchronize with each other explicitly. You will also add support for hardware clock interrupts and preemption.

Getting Started

Use Git to commit your Lab 3 source, fetch the latest version of the course repository, and then create a local branch called `lab4` based on our lab4 branch, `origin/lab4`:

```
athena% cd ~/6.828/lab
athena% add git
athena% git pull
Already up-to-date.
athena% git checkout -b lab4 origin/lab4
Branch lab4 set up to track remote branch refs/remotes/origin/lab4.
Switched to a new branch "lab4"
athena% git merge lab3
Merge made by recursive.
...
athena%
```

Lab 4 contains a number of new source files, some of which you should browse before you start:

`kern/cpu.h`	Kernel-private definitions for multiprocessor support
`kern/mpconfig.c`	Code to read the multiprocessor configuration
`kern/lapic.c`	Kernel code driving the local APIC unit in each processor
`kern/mpentry.S`	Assembly-language entry code for non-boot CPUs
`kern/spinlock.h`	Kernel-private definitions for spin locks, including the big kernel lock
`kern/spinlock.c`	Kernel code implementing spin locks
`kern/sched.c`	Code skeleton of the scheduler that you are about to implement

Lab Requirements

This lab is divided into three parts, A, B, and C. We have allocated one week in the schedule for each part.

As before, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem. (You do not need to do one challenge problem per part, just one for the whole lab.) Additionally, you will need to write up a brief description of the challenge problem that you implemented. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called `answers-lab4.txt` in the top level of your `lab` directory before handing in your work.

Part A: Multiprocessor Support and Cooperative Multitasking

In the first part of this lab, you will first extend JOS to run on a multiprocessor system, and then implement some new JOS kernel system calls to allow user-level environments to create additional new environments. You will also implement *cooperative* round-robin scheduling, allowing the kernel to switch from one environment to another when the current environment voluntarily relinquishes the CPU (or exits). Later in part C you will implement *preemptive* scheduling, which allows the kernel to re-take control of the CPU from an environment after a certain time has passed even if the environment does not cooperate.

Multiprocessor Support

We are going to make JOS support "symmetric multiprocessing" (SMP), a multiprocessor model in which all CPUs have equivalent access to system resources such as memory and I/O buses. While all CPUs are functionally identical in SMP, during the boot process they can be classified into two types: the bootstrap processor (BSP) is responsible for initializing the system and for booting the operating system; and the application processors (APs) are activated by the BSP only after the operating system is up and running. Which processor is the BSP is determined by the hardware and the BIOS. Up to this point, all your existing JOS code has been running on the BSP.

In an SMP system, each CPU has an accompanying local APIC (LAPIC) unit. The LAPIC units are responsible for delivering interrupts throughout the system. The LAPIC also provides its connected CPU with a unique identifier. In this lab, we make use of the following basic functionality of the LAPIC unit (in `kern/lapic.c`):

- Reading the LAPIC identifier (APIC ID) to tell which CPU our code is currently running on (see `cpunum()`).
- Sending the `STARTUP` interprocessor interrupt (IPI) from the BSP to the APs to bring up other CPUs (see `lapic_startap()`).
- In part C, we program LAPIC's built-in timer to trigger clock interrupts to support preemptive multitasking (see `apic_init()`).

A processor accesses its LAPIC using memory-mapped I/O (MMIO). In MMIO, a portion of *physical* memory is hardwired to the registers of some I/O devices, so the same load/store instructions typically used to access memory can be used to access device registers. You've already seen one IO hole at physical address `0xA0000` (we use this to write to the VGA display buffer). The LAPIC lives in a hole starting at physical address `0xFE000000` (32MB short of 4GB), so it's too high for us to access using our usual direct map at KERNBASE. The JOS virtual memory map leaves a 4MB gap at `MMIOBASE` so we have a place to map devices like this. Since later labs introduce more MMIO regions, you'll write a simple function to allocate space from this region and map device memory to it.

> **Exercise 1.** Implement `mmio_map_region` in `kern/pmap.c`. To see how this is used, look at the beginning of `lapic_init` in `kern/lapic.c`. You'll have to do the next exercise, too, before the tests for `mmio_map_region` will run.

Application Processor Bootstrap

Before booting up APs, the BSP should first collect information about the multiprocessor system, such as the total number of CPUs, their APIC IDs and the MMIO address of the LAPIC unit. The `mp_init()` function in `kern/mpconfig.c` retrieves this information by reading the MP configuration table that resides in the BIOS's region of memory.

The `boot_aps()` function (in `kern/init.c`) drives the AP bootstrap process. APs start in real mode, much like how the bootloader started in `boot/boot.S`, so `boot_aps()` copies the AP entry code (`kern/mpentry.S`) to a memory location that is addressable in the real mode. Unlike with the bootloader, we have some control over where the AP will start executing code; we copy the entry code to `0x7000` (`MPENTRY_PADDR`), but any unused, page-aligned physical address below 640KB would work.

After that, `boot_aps()` activates APs one after another, by sending `STARTUP` IPIs to the LAPIC unit of the corresponding AP, along with an initial `CS:IP` address at which the AP should start running its entry code (`MPENTRY_PADDR` in our case). The entry code in `kern/mpentry.S` is quite similar to that of `boot/boot.S`. After some brief setup, it puts the AP into protected mode with paging enabled, and then calls the C setup routine `mp_main()` (also in `kern/init.c`). `boot_aps()` waits for the AP to signal a `CPU_STARTED` flag in `cpu_status` field of its `struct CpuInfo` before going on to wake up the next one.

> **Exercise 2.** Read `boot_aps()` and `mp_main()` in `kern/init.c`, and the assembly code in `kern/mpentry.S`. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of `page_init()` in `kern/pmap.c` to avoid adding the page at `MPENTRY_PADDR` to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated `check_page_free_list()` test (but might fail the updated `check_kern_pgdir()` test, which we will fix soon).

Question

1. Compare `kern/mpentry.S` side by side with `boot/boot.S`. Bearing in mind that `kern/mpentry.S` is compiled and linked to run above `KERNBASE` just like everything else in the kernel, what is the purpose of macro `MPBOOTPHYS`? Why is it necessary in `kern/mpentry.S` but not in `boot/boot.S`? In other words, what could go wrong if it were omitted in `kern/mpentry.S`?
   Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

Per-CPU State and Initialization

When writing a multiprocessor OS, it is important to distinguish between per-CPU state that is private to each processor, and global state that the whole system shares. `kern/cpu.h` defines most of the per-CPU state, including `struct CpuInfo`, which stores per-CPU variables. `cpunum()` always returns the ID of the CPU that calls it, which can be used as an index into arrays like `cpus`. Alternatively, the macro `thiscpu` is shorthand for the current CPU's `struct CpuInfo`.

Here is the per-CPU state you should be aware of:

- **Per-CPU kernel stack**.
  Because multiple CPUs can trap into the kernel simultaneously, we need a separate kernel stack for each processor to prevent them from interfering with each other's execution. The array `percpu_kstacks[NCPU][KSTKSIZE]` reserves space for NCPU's worth of kernel stacks.

  In Lab 2, you mapped the physical memory that `bootstack` refers to as the BSP's kernel stack just below `KSTACKTOP`. Similarly, in this lab, you will map each CPU's kernel stack into this region with guard pages acting as a buffer between them. CPU 0's stack will still grow down from `KSTACKTOP`; CPU 1's stack will start `KSTKGAP` bytes below the bottom of CPU 0's stack, and so on. `inc/memlayout.h` shows the mapping layout.

- **Per-CPU TSS and TSS descriptor**.
  A per-CPU task state segment (TSS) is also needed in order to specify where each CPU's kernel stack lives. The TSS for CPU *i* is stored in `cpus[i].cpu_ts`, and the corresponding TSS descriptor is defined in the GDT entry `gdt[(GD_TSS0 >> 3) + i]`. The global `ts` variable defined in `kern/trap.c` will no longer be useful.

- **Per-CPU current environment pointer**.
  Since each CPU can run different user process simultaneously, we redefined the symbol `curenv` to refer to `cpus[cpunum()].cpu_env` (or `thiscpu->cpu_env`), which points to the environment *currently* executing on the *current* CPU (the CPU on which the code is running).

- **Per-CPU system registers**.
  All registers, including system registers, are private to a CPU. Therefore, instructions that initialize these registers, such as `lcr3()`, `ltr()`, `lgdt()`, `lidt()`, etc., must be executed once on each CPU. Functions `env_init_percpu()` and `trap_init_percpu()` are defined for this purpose.

  In addition to this, if you have added any extra per-CPU state or performed any additional CPU-specific initialization (by say, setting new bits in the CPU registers) in your solutions to challenge problems in earlier labs, be sure to replicate them on each CPU here!

> **Exercise 3.** Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in `inc/memlayout.h`. The size of each stack is `KSTKSIZE` bytes plus `KSTKGAP` bytes of unmapped guard pages. Your code should pass the new check in `check_kern_pgdir()`.

> **Exercise 4.** The code in `trap_init_percpu()` (`kern/trap.c`) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global `ts` variable any more.)

When you finish the above exercises, run JOS in QEMU with 4 CPUs using `make qemu CPUS=4` (or `make qemu-nox CPUS=4`), you should see output like this:

```
...
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
```

Locking

Our current code spins after initializing the AP in `mp_main()`. Before letting the AP get any further, we need to first address race conditions when multiple CPUs run kernel code simultaneously. The simplest way to achieve this is to use a *big kernel lock*. The big kernel lock is a single global lock that is held whenever an environment enters kernel mode, and is released when the environment returns to user mode. In this model, environments in user mode can run concurrently on any available CPUs, but no more than one environment can run in kernel mode; any other environments that try to enter kernel mode are forced to wait.

`kern/spinlock.h` declares the big kernel lock, namely `kernel_lock`. It also provides `lock_kernel()` and `unlock_kernel()`, shortcuts to acquire and release the lock. You should apply the big kernel lock at four locations:

- In `i386_init()`, acquire the lock before the BSP wakes up the other CPUs.
- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

> **Exercise 5.** Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

How to test if your locking is correct? You can't at this moment! But you will be able to after you implement the scheduler in the next exercise.

> **Question**
>
> 2. It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

> *Challenge!* The big kernel lock is simple and easy to use. Nevertheless, it eliminates all concurrency in kernel mode. Most modern operating systems use different locks to protect different parts of their shared state, an approach called *fine-grained locking*. Fine-grained locking can increase performance significantly, but is more difficult to implement and error-prone. If you are brave enough, drop the big kernel lock and embrace concurrency in JOS!
>
> It is up to you to decide the locking granularity (the amount of data that a lock protects). As a hint, you may consider using spin locks to ensure exclusive access to these shared components in the JOS kernel:
>
> - The page allocator.
> - The console driver.
> - The scheduler.
> - The inter-process communication (IPC) state that you will implement in the part C.

Round-Robin Scheduling

Your next task in this lab is to change the JOS kernel so that it can alternate between multiple environments in "round-robin" fashion. Round-robin scheduling in JOS works as follows:

- The function `sched_yield()` in the new `kern/sched.c` is responsible for selecting a new environment to run. It searches sequentially through the `envs[]` array in circular fashion, starting just after the previously running environment (or at the beginning of the array if there was no previously running environment), picks the first environment it finds with a status of `ENV_RUNNABLE` (see `inc/env.h`), and calls `env_run()` to jump into that environment.
- `sched_yield()` must never run the same environment on two CPUs at the same time. It can tell that an environment is currently running on some CPU (possibly the current CPU) because that environment's status will be `ENV_RUNNING`.
- We have implemented a new system call for you, `sys_yield()`, which user environments can call to invoke the kernel's `sched_yield()` function and thereby voluntarily give up the CPU to a different environment.

Exercise 6. Implement round-robin scheduling in `sched_yield()` as described above. Don't forget to modify `syscall()` to dispatch `sys_yield()`.

Make sure to invoke `sched_yield()` in `mp_main`.

Modify `kern/init.c` to create three (or more!) environments that all run the program `user/yield.c`.

Run `make qemu`. You should see the environments switch back and forth between each other five times before terminating, like below.

Test also with several CPUS: `make qemu CPUS=2`.

```
...
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

After the `yield` programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

Question

3. In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?
4. Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

Challenge! Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.)

Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented `fork()` and IPC in parts B and C of this lab.

Challenge! The JOS kernel currently does not allow applications to use the x86 processor's x87 floating-point unit (FPU), MMX instructions, or Streaming SIMD Extensions (SSE). Extend the `Env` structure to provide a save area for the processor's floating point state, and extend the context switching code to save and restore this state properly when switching from one environment to another. The `FXSAVE` and `FXRSTOR` instructions may be useful, but note that these are not in the old i386 user's manual because they were introduced in more recent processors. Write a user-level test program that does something cool with floating-point.

System Calls for Environment Creation

Although your kernel is now capable of running and switching between multiple user-level environments, it is still limited to running environments that the *kernel* initially set up. You will now implement the necessary JOS system calls to allow *user* environments to create and start other new user environments.

Unix provides the `fork()` system call as its process creation primitive. Unix `fork()` copies the entire address space of calling process (the parent) to create a new process (the child). The only differences between the two observable from user space are their process IDs and parent process IDs (as returned by `getpid` and `getppid`). In the parent, `fork()` returns the child's process ID, while in the child, `fork()` returns 0. By default, each process gets its own private address space, and neither process's modifications to memory are visible to the other.

You will provide a different, more primitive set of JOS system calls for creating new user-mode environments. With these system calls you will be able to implement a Unix-like `fork()` entirely in user space, in addition to other styles of environment creation. The new system calls you will write for JOS are as follows:

`sys_exofork`:
: This system call creates a new environment with an almost blank slate: nothing is mapped in the user portion of its address space, and it is not runnable. The new environment will have the same register state as the parent environment at the time of the `sys_exofork` call. In the parent, `sys_exofork` will return the `envid_t` of the newly created environment (or a negative error code if the environment allocation failed). In the child, however, it will return 0. (Since the child starts out marked as not runnable, `sys_exofork` will not actually return in the child until the parent has explicitly allowed this by marking the child runnable using....)

`sys_env_set_status`:
: Sets the status of a specified environment to `ENV_RUNNABLE` or `ENV_NOT_RUNNABLE`. This system call is typically used to mark a new environment ready to run, once its address space and register state has been fully initialized.

`sys_page_alloc`:
: Allocates a page of physical memory and maps it at a given virtual address in a given environment's address space.

`sys_page_map`:
: Copy a page mapping (*not* the contents of a page!) from one environment's address space to another, leaving a memory sharing arrangement in place so that the new and the old mappings both refer to the same page of physical memory.

`sys_page_unmap`:
: Unmap a page mapped at a given virtual address in a given environment.

For all of the system calls above that accept environment IDs, the JOS kernel supports the convention that a value of 0 means "the current environment." This convention is implemented by `envid2env()` in `kern/env.c`.

We have provided a very primitive implementation of a Unix-like `fork()` in the test program `user/dumbfork.c`. This test program uses the above system calls to create and run a child environment with a copy of its own address space. The two environments then switch back and forth using `sys_yield` as in the previous exercise. The parent exits after 10 iterations, whereas the child exits after 20.

> **Exercise 7.** Implement the system calls described above in `kern/syscall.c` and make sure `syscall()` calls them. You will need to use

various functions in `kern/pmap.c` and `kern/env.c`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

Challenge! Add the additional system calls necessary to *read* all of the vital state of an existing environment as well as set it up. Then implement a user mode program that forks off a child environment, runs it for a while (e.g., a few iterations of `sys_yield()`), then takes a complete snapshot or *checkpoint* of the child environment, runs the child for a while longer, and finally restores the child environment to the state it was in at the checkpoint and continues it from there. Thus, you are effectively "replaying" the execution of the child environment from an intermediate state. Make the child environment perform some interaction with the user using `sys_cgetc()` or `readline()` so that the user can view and mutate its internal state, and verify that with your checkpoint/restart you can give the child environment a case of selective amnesia, making it "forget" everything that happened beyond a certain point.

This completes Part A of the lab; make sure it passes all of the Part A tests when you run `make grade`, and hand it in using `make handin` as usual. If you are trying to figure out why a particular test case is failing, run `./grade-lab4 -v`, which will show you the output of the kernel builds and QEMU runs for each test, until a test fails. When a test fails, the script will stop, and then you can inspect `jos.out` to see what the kernel actually printed.

Part B: Copy-on-Write Fork

As mentioned earlier, Unix provides the `fork()` system call as its primary process creation primitive. The `fork()` system call copies the address space of the calling process (the parent) to create a new process (the child).

xv6 Unix implements `fork()` by copying all data from the parent's pages into new pages allocated for the child. This is essentially the same approach that `dumbfork()` takes. The copying of the parent's address space into the child is the most expensive part of the `fork()` operation.

However, a call to `fork()` is frequently followed almost immediately by a call to `exec()` in the child process, which replaces the child's memory with a new program. This is what the the shell typically does, for example. In this case, the time spent copying the parent's address space is largely wasted, because the child process will use very little of its memory before calling `exec()`.

For this reason, later versions of Unix took advantage of virtual memory hardware to allow the parent and child to *share* the memory mapped into their respective address spaces until one of the processes actually modifies it. This technique is known as *copy-*

on-write. To do this, on `fork()` the kernel would copy the address space *mappings* from the parent to the child instead of the contents of the mapped pages, and at the same time mark the now-shared pages read-only. When one of the two processes tries to write to one of these shared pages, the process takes a page fault. At this point, the Unix kernel realizes that the page was really a "virtual" or "copy-on-write" copy, and so it makes a new, private, writable copy of the page for the faulting process. In this way, the contents of individual pages aren't actually copied until they are actually written to. This optimization makes a `fork()` followed by an `exec()` in the child much cheaper: the child will probably only need to copy one page (the current page of its stack) before it calls `exec()`.

In the next piece of this lab, you will implement a "proper" Unix-like `fork()` with copy-on-write, as a user space library routine. Implementing `fork()` and copy-on-write support in user space has the benefit that the kernel remains much simpler and thus more likely to be correct. It also lets individual user-mode programs define their own semantics for `fork()`. A program that wants a slightly different implementation (for example, the expensive always-copy version like `dumbfork()`, or one in which the parent and child actually share memory afterward) can easily provide its own.

User-level page fault handling

A user-level copy-on-write `fork()` needs to know about page faults on write-protected pages, so that's what you'll implement first. Copy-on-write is only one of many possible uses for user-level page fault handling.

It's common to set up an address space so that page faults indicate when some action needs to take place. For example, most Unix kernels initially map only a single page in a new process's stack region, and allocate and map additional stack pages later "on demand" as the process's stack consumption increases and causes page faults on stack addresses that are not yet mapped. A typical Unix kernel must keep track of what action to take when a page fault occurs in each region of a process's space. For example, a fault in the stack region will typically allocate and map new page of physical memory. A fault in the program's BSS region will typically allocate a new page, fill it with zeroes, and map it. In systems with demand-paged executables, a fault in the text region will read the corresponding page of the binary off of disk and then map it.

This is a lot of information for the kernel to keep track of. Instead of taking the traditional Unix approach, you will decide what to do about each page fault in user space, where bugs are less damaging. This design has the added benefit of allowing programs great flexibility in defining their memory regions; you'll use user-level page fault handling later for mapping and accessing files on a disk-based file system.

Setting the Page Fault Handler

In order to handle its own page faults, a user environment will need to register a *page fault handler entrypoint* with the JOS kernel. The user environment registers its page fault entrypoint via the new `sys_env_set_pgfault_upcall` system call. We have added a new member to the `Env` structure, `env_pgfault_upcall`, to record this information.

> **Exercise 8.** Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

Normal and Exception Stacks in User Environments

During normal execution, a user environment in JOS will run on the *normal* user stack: its `ESP` register starts out pointing at `USTACKTOP`, and the stack data it pushes resides on the page between `USTACKTOP-PGSIZE` and `USTACKTOP-1` inclusive. When a page fault occurs in user mode, however, the kernel will restart the user environment running a designated user-level page fault handler on a different stack, namely the *user exception* stack. In essence, we will make the JOS kernel implement automatic "stack switching" on behalf of the user environment, in much the same way that the x86 *processor* already implements stack switching on behalf of JOS when transferring from user mode to kernel mode!

The JOS user exception stack is also one page in size, and its top is defined to be at virtual address `UXSTACKTOP`, so the valid bytes of the user exception stack are from `UXSTACKTOP-PGSIZE` through `UXSTACKTOP-1` inclusive. While running on this exception stack, the user-level page fault handler can use JOS's regular system calls to map new pages or adjust mappings so as to fix whatever problem originally caused the page fault. Then the user-level page fault handler returns, via an assembly language stub, to the faulting code on the original stack.

Each user environment that wants to support user-level page fault handling will need to allocate memory for its own exception stack, using the `sys_page_alloc()` system call introduced in part A.

Invoking the User Page Fault Handler

You will now need to change the page fault handling code in `kern/trap.c` to handle page faults from user mode as follows. We will call the state of the user environment at the time of the fault the *trap-time* state.

If there is no page fault handler registered, the JOS kernel destroys the user environment with a message as before. Otherwise, the kernel sets up a trap frame on the exception stack that looks like a `struct UTrapframe` from `inc/trap.h`:

```
                    <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax       start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

The kernel then arranges for the user environment to resume execution with the page fault handler running on the exception stack with this stack frame; you must figure out how to make this happen. The `fault_va` is the virtual address that caused the page fault.

If the user environment is *already* running on the user exception stack when an exception occurs, then the page fault handler itself has faulted. In this case, you should start the new stack frame just under the current `tf->tf_esp` rather than at `UXSTACKTOP`. You should first push an empty 32-bit word, then a `struct UTrapframe`.

To test whether `tf->tf_esp` is already on the user exception stack, check whether it is in the range between `UXSTACKTOP-PGSIZE` and `UXSTACKTOP-1`, inclusive.

> **Exercise 9.** Implement the code in `page_fault_handler` in `kern/trap.c` required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

User-mode Page Fault Entrypoint

Next, you need to implement the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction. This assembly routine is the handler that will be registered with the kernel using `sys_env_set_pgfault_upcall()`.

> **Exercise 10.** Implement the `_pgfault_upcall` routine in `lib/pfentry.S`. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

Finally, you need to implement the C user library side of the user-level page fault handling mechanism.

> **Exercise 11.** Finish `set_pgfault_handler()` in `lib/pgfault.c`.

Testing

Run `user/faultread` (`make run-faultread`). You should see:

```
...
[00000000] new env 00001000
[00001000] user fault va 00000000 ip 0080003a
TRAP frame ...
[00001000] free env 00001000
```

Run `user/faultdie`. You should see:

```
...
[00000000] new env 00001000
i faulted at va deadbeef, err 6
[00001000] exiting gracefully
[00001000] free env 00001000
```

Run `user/faultalloc`. You should see:

```
...
[00000000] new env 00001000
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001000] exiting gracefully
[00001000] free env 00001000
```

If you see only the first "this string" line, it means you are not handling recursive page faults properly.

Run `user/faultallocbad`. You should see:

```
...
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va deadbeef
[00001000] free env 00001000
```

Make sure you understand why `user/faultalloc` and `user/faultallocbad` behave differently.

> *Challenge!* Extend your kernel so that not only page faults, but *all* types of processor exceptions that code running in user space can generate, can be redirected to a user-mode exception handler. Write user-mode test programs to test user-mode handling of various exceptions such as divide-by-zero, general protection fault, and illegal opcode.

Implementing Copy-on-Write Fork

You now have the kernel facilities to implement copy-on-write `fork()` entirely in user space.

We have provided a skeleton for your `fork()` in `lib/fork.c`. Like `dumbfork()`, `fork()` should create a new environment, then scan through the parent environment's entire address space and set up corresponding page mappings in the child. The key difference is that, while `dumbfork()` copied *pages*, `fork()` will initially only copy page *mappings*. `fork()` will copy each page only when one of the environments tries to write it.

The basic control flow for `fork()` is as follows:

1. The parent installs `pgfault()` as the C-level page fault handler, using the `set_pgfault_handler()` function you implemented above.
2. The parent calls `sys_exofork()` to create a child environment.

3. For each writable or copy-on-write page in its address space below UTOP, the parent calls `duppage`, which should map the page copy-on-write into the address space of the child and then *remap* the page copy-on-write in its own address space. [Note: The ordering here (i.e., marking a page as COW in the child before marking it in the parent) actually matters! Can you see why? Try to think of a specific case where reversing the order could cause trouble.] `duppage` sets both PTEs so that the page is not writeable, and to contain `PTE_COW` in the "avail" field to distinguish copy-on-write pages from genuine read-only pages.

   The exception stack is *not* remapped this way, however. Instead you need to allocate a fresh page in the child for the exception stack. Since the page fault handler will be doing the actual copying and the page fault handler runs on the exception stack, the exception stack cannot be made copy-on-write: who would copy it?

   `fork()` also needs to handle pages that are present, but not writable or copy-on-write.

4. The parent sets the user page fault entrypoint for the child to look like its own.
5. The child is now ready to run, so the parent marks it runnable.

Each time one of the environments writes a copy-on-write page that it hasn't yet written, it will take a page fault. Here's the control flow for the user page fault handler:

1. The kernel propagates the page fault to `_pgfault_upcall`, which calls `fork()`'s `pgfault()` handler.
2. `pgfault()` checks that the fault is a write (check for `FEC_WR` in the error code) and that the PTE for the page is marked `PTE_COW`. If not, panic.
3. `pgfault()` allocates a new page mapped at a temporary location and copies the contents of the faulting page into it. Then the fault handler maps the new page at the appropriate address with read/write permissions, in place of the old read-only mapping.

The user-level `lib/fork.c` code must consult the environment's page tables for several of the operations above (e.g., that the PTE for a page is marked `PTE_COW`). The kernel maps the environment's page tables at `UVPT` exactly for this purpose. It uses a clever mapping trick to make it to make it easy to lookup PTEs for user code. `lib/entry.S` sets up `uvpt` and `uvpd` so that you can easily lookup page-table information in `lib/fork.c`.

> **Exercise 12.** Implement `fork`, `duppage` and `pgfault` in `lib/fork.c`.
>
> Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.
>
> ```
> 1000: I am ''
> 1001: I am '0'
> 2000: I am '00'
> 2001: I am '000'
> 1002: I am '1'
> ```

```
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

Challenge! Implement a shared-memory `fork()` called `sfork()`. This version should have the parent and child *share* all their memory pages (so writes in one environment appear in the other) except for pages in the stack area, which should be treated in the usual copy-on-write manner. Modify `user/forktree.c` to use `sfork()` instead of regular `fork()`. Also, once you have finished implementing IPC in part C, use your `sfork()` to run `user/pingpongs`. You will have to find a new way to provide the functionality of the global `thisenv` pointer.

Challenge! Your implementation of `fork` makes a huge number of system calls. On the x86, switching into the kernel using interrupts has non-trivial cost. Augment the system call interface so that it is possible to send a batch of system calls at once. Then change `fork` to use this interface.

How much faster is your new `fork`?

You can answer this (roughly) by using analytical arguments to estimate how much of an improvement batching system calls will make to the performance of your `fork`: How expensive is an `int 0x30` instruction? How many times do you execute `int 0x30` in your `fork`? Is accessing the `TSS` stack switch also expensive? And so on...

Alternatively, you can boot your kernel on real hardware and *really* benchmark your code. See the `RDTSC` (read time-stamp counter) instruction, defined in the IA32 manual, which counts the number of clock cycles that have elapsed since the last processor reset. QEMU doesn't emulate this instruction faithfully (it can either count the number of virtual instructions executed or use the host TSC, neither of which reflects the number of cycles a real CPU would require).

This ends part B. Make sure you pass all of the Part B tests when you run `make grade`. As usual, you can hand in your submission with `make handin`.

Part C: Preemptive Multitasking and Inter-Process communication (IPC)

In the final part of lab 4 you will modify the kernel to preempt uncooperative environments and to allow environments to pass messages to each other explicitly.

Clock Interrupts and Preemption

Run the `user/spin` test program. This test program forks off a child environment, which simply spins forever in a tight loop once it receives control of the CPU. Neither the parent environment nor the kernel ever regains the CPU. This is obviously not an ideal situation in terms of protecting the system from bugs or malicious code in user-mode environments, because any user-mode environment can bring the whole system to a halt simply by getting into an infinite loop and never giving back the CPU. In order to allow the kernel to *preempt* a running environment, forcefully retaking control of the CPU from it, we must extend the JOS kernel to support external hardware interrupts from the clock hardware.

Interrupt discipline

External interrupts (i.e., device interrupts) are referred to as IRQs. There are 16 possible IRQs, numbered 0 through 15. The mapping from IRQ number to IDT entry is not fixed. `pic_init` in `picirq.c` maps IRQs 0-15 to IDT entries `IRQ_OFFSET` through `IRQ_OFFSET+15`.

In `inc/trap.h`, `IRQ_OFFSET` is defined to be decimal 32. Thus the IDT entries 32-47 correspond to the IRQs 0-15. For example, the clock interrupt is IRQ 0. Thus, IDT[IRQ_OFFSET+0] (i.e., IDT[32]) contains the address of the clock's interrupt handler routine in the kernel. This `IRQ_OFFSET` is chosen so that the device interrupts do not overlap with the processor exceptions, which could obviously cause confusion. (In fact, in the early days of PCs running MS-DOS, the `IRQ_OFFSET` effectively *was* zero, which indeed caused massive confusion between handling hardware interrupts and handling processor exceptions!)

In JOS, we make a key simplification compared to xv6 Unix. External device interrupts are *always* disabled when in the kernel (and, like xv6, enabled when in user space). External interrupts are controlled by the `FL_IF` flag bit of the `%eflags` register (see `inc/mmu.h`). When this bit is set, external interrupts are enabled. While the bit can be modified in several ways, because of our simplification, we will handle it solely through the process of saving and restoring `%eflags` register as we enter and leave user mode.

You will have to ensure that the `FL_IF` flag is set in user environments when they run so that when an interrupt arrives, it gets passed through to the processor and handled by your interrupt code. Otherwise, interrupts are *masked*, or ignored until interrupts are re-enabled. We masked interrupts with the very first instruction of the bootloader, and so far we have never gotten around to re-enabling them.

> **Exercise 13.** Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through

15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.

Also uncomment the `sti` instruction in `sched_halt()` so that idle CPUs unmask interrupts.

The processor never pushes an error code when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., `spin`), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor.

Handling Clock Interrupts

In the `user/spin` program, after the child environment was first run, it just spun in a loop, and the kernel never got control back. We need to program the hardware to generate clock interrupts periodically, which will force control back to the kernel where we can switch control to a different user environment.

The calls to `lapic_init` and `pic_init` (from `i386_init` in `init.c`), which we have written for you, set up the clock and the interrupt controller to generate interrupts. You now need to write the code to handle these interrupts.

Exercise 14. Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

This is a great time to do some *regression testing*. Make sure that you haven't broken any earlier part of that lab that used to work (e.g. `forktree`) by enabling interrupts. Also, try running with multiple CPUs using `make CPUS=2 target`. You should also be able to pass `stresssched` now. Run `make grade` to see for sure. You should now get a total score of 65/80 points on this lab.

Inter-Process communication (IPC)

(Technically in JOS this is "inter-environment communication" or "IEC", but everyone else calls it IPC, so we'll use the standard term.)

We've been focusing on the isolation aspects of the operating system, the ways it provides the illusion that each program has a machine all to itself. Another important service of an operating system is to allow programs to communicate with each other when they want to. It can be quite powerful to let programs interact with other programs. The Unix pipe model is the canonical example.

There are many models for interprocess communication. Even today there are still debates about which models are best. We won't get into that debate. Instead, we'll implement a simple IPC mechanism and then try it out.

IPC in JOS

You will implement a few additional JOS kernel system calls that collectively provide a simple interprocess communication mechanism. You will implement two system calls, `sys_ipc_recv` and `sys_ipc_try_send`. Then you will implement two library wrappers `ipc_recv` and `ipc_send`.

The "messages" that user environments can send to each other using JOS's IPC mechanism consist of two components: a single 32-bit value, and optionally a single page mapping. Allowing environments to pass page mappings in messages provides an efficient way to transfer more data than will fit into a single 32-bit integer, and also allows environments to set up shared memory arrangements easily.

Sending and Receiving Messages

To receive a message, an environment calls `sys_ipc_recv`. This system call de-schedules the current environment and does not run it again until a message has been received. When an environment is waiting to receive a message, *any* other environment can send it a message - not just a particular environment, and not just environments that have a parent/child arrangement with the receiving environment. In other words, the permission checking that you implemented in Part A will not apply to IPC, because the IPC system calls are carefully designed so as to be "safe": an environment cannot cause another environment to malfunction simply by sending it messages (unless the target environment is also buggy).

To try to send a value, an environment calls `sys_ipc_try_send` with both the receiver's environment id and the value to be sent. If the named environment is actually receiving (it has called `sys_ipc_recv` and not gotten a value yet), then the send delivers the message and returns 0. Otherwise the send returns `-E_IPC_NOT_RECV` to indicate that the target environment is not currently expecting to receive a value.

A library function `ipc_recv` in user space will take care of calling `sys_ipc_recv` and then looking up the information about the received values in the current environment's `struct Env`.

Similarly, a library function `ipc_send` will take care of repeatedly calling `sys_ipc_try_send` until the send succeeds.

Transferring Pages

When an environment calls `sys_ipc_recv` with a valid `dstva` parameter (below `UTOP`), the environment is stating that it is willing to receive a page mapping. If the sender sends a page, then that page should be mapped at `dstva` in the receiver's address space. If the receiver already had a page mapped at `dstva`, then that previous page is unmapped.

When an environment calls `sys_ipc_try_send` with a valid `srcva` (below `UTOP`), it means the sender wants to send the page currently mapped at `srcva` to the receiver, with permissions `perm`. After a successful IPC, the sender keeps its original mapping for the page at `srcva` in its address space, but the receiver also obtains a mapping for this same physical page at the `dstva` originally specified by the receiver, in the receiver's address space. As a result this page becomes shared between the sender and receiver.

If either the sender or the receiver does not indicate that a page should be transferred, then no page is transferred. After any IPC the kernel sets the new field `env_ipc_perm` in the receiver's `Env` structure to the permissions of the page received, or zero if no page was received.

Implementing IPC

> **Exercise 15.** Implement `sys_ipc_recv` and `sys_ipc_try_send` in `kern/syscall.c`. Read the comments on both before implementing them, since they have to work together. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.
>
> Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c`.
>
> Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. `user/primes` will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

> *Challenge!* Why does `ipc_send` have to loop? Change the system call interface so it doesn't have to. Make sure you can handle multiple environments trying to send to one environment at the same time.

Challenge! The prime sieve is only one neat use of message passing between a large number of concurrent programs. Read C. A. R. Hoare, ``Communicating Sequential Processes,'' *Communications of the ACM* 21(8) (August 1978), 666-667, and implement the matrix multiplication example.

Challenge! One of the most impressive examples of the power of message passing is Doug McIlroy's power series calculator, described in M. Douglas McIlroy, ``Squinting at Power Series,'' *Software--Practice and Experience*, 20(7) (July 1990), 661-683. Implement his power series calculator and compute the power series for *sin*(*x*+*x*^3).

Challenge! Make JOS's IPC mechanism more efficient by applying some of the techniques from Liedtke's paper, Improving IPC by Kernel Design, or any other tricks you may think of. Feel free to modify the kernel's system call API for this purpose, as long as your code is backwards compatible with what our grading scripts expect.

This ends part C. Make sure you pass all of the `make grade` tests and don't forget to write up your answers to the questions and a description of your challenge exercise solution in `answers-lab4.txt`.

Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab4.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 4'`, then `make handin` and follow the directions.

Lab 5: File system, Spawn and Shell

Due Thursday, November 15, 2018

Introduction

In this lab, you will implement `spawn`, a library call that loads and runs on-disk executables. You will then flesh out your kernel and library operating system enough to run a shell on the console. These features need a file system, and this lab introduces a simple read/write file system.

Getting Started

Use Git to fetch the latest version of the course repository, and then create a local branch called `lab5` based on our lab5 branch, `origin/lab5`:

```
athena% cd ~/6.828/lab
athena% add git
athena% git pull
Already up-to-date.
athena% git checkout -b lab5 origin/lab5
Branch lab5 set up to track remote branch refs/remotes/origin/lab5.
Switched to a new branch "lab5"
athena% git merge lab4
Merge made by recursive.
.....
athena%
```

The main new component for this part of the lab is the file system environment, located in the new `fs` directory. Scan through all the files in this directory to get a feel for what all is new. Also, there are some new file system-related source files in the `user` and `lib` directories,

`fs/fs.c`	Code that mainipulates the file system's on-disk structure.
`fs/bc.c`	A simple block cache built on top of our user-level page fault handling facility.
`fs/ide.c`	Minimal PIO-based (non-interrupt-driven) IDE driver code.
`fs/serv.c`	The file system server that interacts with client environments using file system IPCs.
`lib/fd.c`	Code that implements the general UNIX-like file descriptor interface.
`lib/file.c`	The driver for on-disk file type, implemented as a file system IPC client.
`lib/console.c`	The driver for console input/output file type.
`lib/spawn.c`	Code skeleton of the `spawn` library call.

You should run the pingpong, primes, and forktree test cases from lab 4 again after merging in the new lab 5 code. You will need to comment out the `ENV_CREATE(fs_fs)` line in `kern/init.c` because `fs/fs.c` tries to do some I/O, which JOS does not allow yet. Similarly, temporarily comment out the call to `close_all()` in `lib/exit.c`; this function calls subroutines that you will implement later in the lab, and therefore will panic if called. If

your lab 4 code doesn't contain any bugs, the test cases should run fine. Don't proceed until they work. Don't forget to un-comment these lines when you start Exercise 1.

If they don't work, use `git diff lab4` to review all the changes, making sure there isn't any code you wrote for lab4 (or before) missing from lab 5. Make sure that lab 4 still works.

Lab Requirements

As before, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem. Additionally, you will need to write up brief answers to the questions posed in the lab and a short (e.g., one or two paragraph) description of what you did to solve your chosen challenge problem. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called `answers-lab5.txt` in the top level of your `lab5` directory before handing in your work.

File system preliminaries

The file system you will work with is much simpler than most "real" file systems including that of xv6 UNIX, but it is powerful enough to provide the basic features: creating, reading, writing, and deleting files organized in a hierarchical directory structure.

We are (for the moment anyway) developing only a single-user operating system, which provides protection sufficient to catch bugs but not to protect multiple mutually suspicious users from each other. Our file system therefore does not support the UNIX notions of file ownership or permissions. Our file system also currently does not support hard links, symbolic links, time stamps, or special device files like most UNIX file systems do.

On-Disk File System Structure

Most UNIX file systems divide available disk space into two main types of regions: *inode* regions and *data* regions. UNIX file systems assign one *inode* to each file in the file system; a file's inode holds critical meta-data about the file such as its `stat` attributes and pointers to its data blocks. The data regions are divided into much larger (typically 8KB or more) *data blocks*, within which the file system stores file data and directory meta-data. Directory entries contain file names and pointers to inodes; a file is said to be *hard-linked* if multiple directory entries in the file system refer to that file's inode. Since our file system will not support hard links, we do not need this level of indirection and therefore can make a convenient simplification: our file system will not use inodes at all and instead will simply store all of a file's (or sub-directory's) meta-data within the (one and only) directory entry describing that file.

Both files and directories logically consist of a series of data blocks, which may be scattered throughout the disk much like the pages of an environment's virtual address space can be scattered throughout physical memory. The file system environment hides the details of block layout, presenting interfaces for reading and writing sequences of

bytes at arbitrary offsets within files. The file system environment handles all modifications to directories internally as a part of performing actions such as file creation and deletion. Our file system does allow user environments to *read* directory meta-data directly (e.g., with `read`), which means that user environments can perform directory scanning operations themselves (e.g., to implement the `ls` program) rather than having to rely on additional special calls to the file system. The disadvantage of this approach to directory scanning, and the reason most modern UNIX variants discourage it, is that it makes application programs dependent on the format of directory meta-data, making it difficult to change the file system's internal layout without changing or at least recompiling application programs as well.

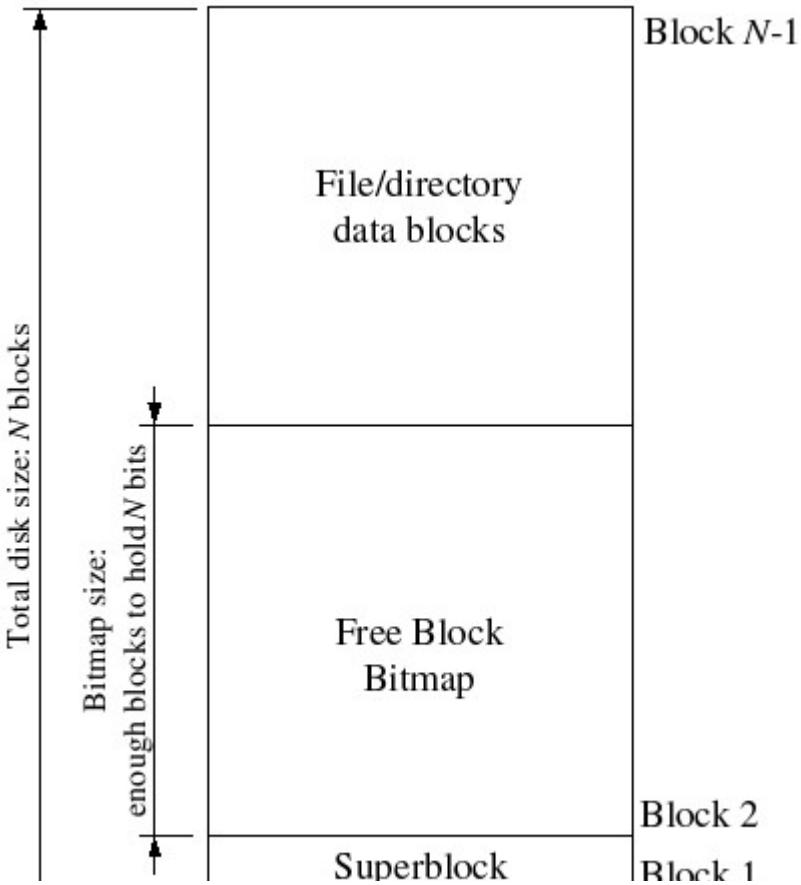
Sectors and Blocks

Most disks cannot perform reads and writes at byte granularity and instead perform reads and writes in units of *sectors*. In JOS, sectors are 512 bytes each. File systems actually allocate and use disk storage in units of *blocks*. Be wary of the distinction between the two terms: *sector size* is a property of the disk hardware, whereas *block size* is an aspect of the operating system using the disk. A file system's block size must be a multiple of the sector size of the underlying disk.

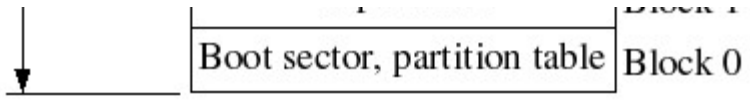
The UNIX xv6 file system uses a block size of 512 bytes, the same as the sector size of the underlying disk. Most modern file systems use a larger block size, however, because storage space has gotten much cheaper and it is more efficient to manage storage at larger granularities. Our file system will use a block size of 4096 bytes, conveniently matching the processor's page size.

Superblocks

File systems typically reserve certain disk blocks at "easy-to-find" locations on the disk (such as the very start or the very end) to hold meta-data describing properties of the file system as a whole, such as the block size, disk size, any meta-data required to find the root directory, the time the file system was last mounted, the time the file system was last checked for errors, and so on. These special blocks are called *superblocks*.

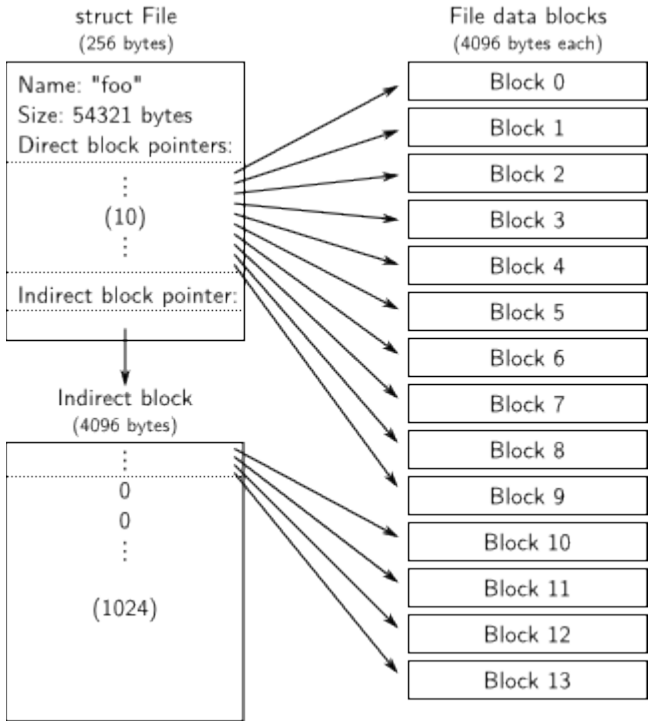


Our file system will have exactly one superblock, which will always be at block 1 on the disk. Its layout is defined by `struct Super` in `inc/fs.h`. Block 0 is typically reserved to hold boot



loaders and partition tables, so file systems generally do not use the very first disk block. Many "real" file systems maintain multiple superblocks, replicated throughout several widely-spaced regions of the disk, so that if one of them is corrupted or the disk develops a media error in that region, the other superblocks can still be found and used to access the file system.

File Meta-data



The layout of the meta-data describing a file in our file system is described by `struct File` in `inc/fs.h`. This meta-data includes the file's name, size, type (regular file or directory), and pointers to the blocks comprising the file. As mentioned above, we do not have inodes, so this meta-data is stored in a directory entry on disk. Unlike in most "real" file systems, for simplicity we will use this one `File` structure to represent file meta-data as it appears *both on disk and in memory*.

The `f_direct` array in `struct File` contains space to store the block numbers of the first 10 (`NDIRECT`) blocks of the file, which we call the file's *direct* blocks. For small files up to 10*4096 = 40KB in size, this means that the block numbers of all of the file's blocks will fit directly within the `File` structure itself. For larger files, however, we need a place to hold the rest of the file's block numbers. For any file greater than 40KB in size, therefore, we allocate an additional disk block, called the file's *indirect block*, to hold up to 4096/4 = 1024 additional block numbers. Our file system therefore allows files to be up to 1034 blocks, or just over four megabytes, in size. To support larger files, "real" file systems typically support *double-* and *triple-indirect blocks* as well.

Directories versus Regular Files

A `File` structure in our file system can represent either a *regular* file or a directory; these two types of "files" are distinguished by the `type` field in the `File` structure. The file system manages regular files and directory-files in exactly the same way, except that it does not interpret the contents of the data blocks associated with regular files at all, whereas the file system interprets the contents of a directory-file as a series of `File` structures describing the files and subdirectories within the directory.

The superblock in our file system contains a `File` structure (the `root` field in `struct Super`) that holds the meta-data for the file system's root directory. The contents of this directory-file is a sequence of `File` structures describing the files and directories located within the root directory of the file system. Any subdirectories in the root directory may in turn contain more `File` structures representing sub-subdirectories, and so on.

The File System

The goal for this lab is not to have you implement the entire file system, but for you to implement only certain key components. In particular, you will be responsible for reading blocks into the block cache and flushing them back to disk; allocating disk blocks; mapping file offsets to disk blocks; and implementing read, write, and open in the IPC interface. Because you will not be implementing all of the file system yourself, it is very important that you familiarize yourself with the provided code and the various file system interfaces.

Disk Access

The file system environment in our operating system needs to be able to access the disk, but we have not yet implemented any disk access functionality in our kernel. Instead of taking the conventional "monolithic" operating system strategy of adding an IDE disk driver to the kernel along with the necessary system calls to allow the file system to access it, we instead implement the IDE disk driver as part of the user-level file system environment. We will still need to modify the kernel slightly, in order to set things up so that the file system environment has the privileges it needs to implement disk access itself.

It is easy to implement disk access in user space this way as long as we rely on polling, "programmed I/O" (PIO)-based disk access and do not use disk interrupts. It is possible to implement interrupt-driven device drivers in user mode as well (the L3 and L4 kernels do this, for example), but it is more difficult since the kernel must field device interrupts and dispatch them to the correct user-mode environment.

The x86 processor uses the IOPL bits in the EFLAGS register to determine whether protected-mode code is allowed to perform special device I/O instructions such as the IN and OUT instructions. Since all of the IDE disk registers we need to access are located in the x86's I/O space rather than being memory-mapped, giving "I/O privilege" to the file system environment is the only thing we need to do in order to allow the file system to access these registers. In effect, the IOPL bits in the EFLAGS register provides the kernel with a simple "all-or-nothing" method of controlling whether user-mode code can access I/O space. In our case, we want the file system environment to be able to access I/O space, but we do not want any other environments to be able to access I/O space at all.

> **Exercise 1.** `i386_init` identifies the file system environment by passing the type `ENV_TYPE_FS` to your environment creation function, `env_create`. Modify `env_create` in `env.c`, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.
>
> Make sure you can start the file environment without causing a General Protection fault. You should pass the "fs i/o" test in `make grade`.

Question

1. Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Why?

Note that the `GNUmakefile` file in this lab sets up QEMU to use the file `obj/kern/kernel.img` as the image for disk 0 (typically "Drive C" under DOS/Windows) as before, and to use the (new) file `obj/fs/fs.img` as the image for disk 1 ("Drive D"). In this lab our file system should only ever touch disk 1; disk 0 is used only to boot the kernel. If you manage to corrupt either disk image in some way, you can reset both of them to their original, "pristine" versions simply by typing:

```
$ rm obj/kern/kernel.img obj/fs/fs.img
$ make
```

or by doing:

```
$ make clean
$ make
```

Challenge! Implement interrupt-driven IDE disk access, with or without DMA. You can decide whether to move the device driver into the kernel, keep it in user space along with the file system, or even (if you really want to get into the micro-kernel spirit) move it into a separate environment of its own.

The Block Cache

In our file system, we will implement a simple "buffer cache" (really just a block cache) with the help of the processor's virtual memory system. The code for the block cache is in `fs/bc.c`.

Our file system will be limited to handling disks of size 3GB or less. We reserve a large, fixed 3GB region of the file system environment's address space, from 0x10000000 (`DISKMAP`) up to 0xD0000000 (`DISKMAP+DISKMAX`), as a "memory mapped" version of the disk. For example, disk block 0 is mapped at virtual address 0x10000000, disk block 1 is mapped at virtual address 0x10001000, and so on. The `diskaddr` function in `fs/bc.c` implements this translation from disk block numbers to virtual addresses (along with some sanity checking).

Since our file system environment has its own virtual address space independent of the virtual address spaces of all other environments in the system, and the only thing the file system environment needs to do is to implement file access, it is reasonable to reserve most of the file system environment's address space in this way. It would be awkward for a real file system implementation on a 32-bit machine to do this since modern disks are larger than 3GB. Such a buffer cache management approach may still be reasonable on a machine with a 64-bit address space.

Of course, it would take a long time to read the entire disk into memory, so instead we'll implement a form of *demand paging*, wherein we only allocate pages in the disk map region and read the corresponding block from the disk in response to a page fault in this region. This way, we can pretend that the entire disk is in memory.

> **Exercise 2.** Implement the `bc_pgfault` and `flush_block` functions in `fs/bc.c`. `bc_pgfault` is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) `addr` may not be aligned to a block boundary and (2) `ide_read` operates in sectors, not blocks.
>
> The `flush_block` function should write a block out to disk *if necessary*. `flush_block` shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the `PTE_D` "dirty" bit is set in the `uvpt` entry. (The `PTE_D` bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, `flush_block` should clear the `PTE_D` bit using `sys_page_map`.
>
> Use `make grade` to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".

The `fs_init` function in `fs/fs.c` is a prime example of how to use the block cache. After initializing the block cache, it simply stores pointers into the disk map region in the `super` global variable. After this point, we can simply read from the `super` structure as if they were in memory and our page fault handler will read them from disk as necessary.

> *Challenge!* The block cache has no eviction policy. Once a block gets faulted in to it, it never gets removed and will remain in memory forevermore. Add eviction to the buffer cache. Using the `PTE_A` "accessed" bits in the page tables, which the hardware sets on any access to a page, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region. Be careful with dirty blocks.

The Block Bitmap

After `fs_init` sets the `bitmap` pointer, we can treat `bitmap` as a packed array of bits, one for each block on the disk. See, for example, `block_is_free`, which simply checks whether a given block is marked free in the bitmap.

Exercise 3. Use `free_block` as a model to implement `alloc_block` in `fs/fs.c`, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with `flush_block`, to help file system consistency.

Use `make grade` to test your code. Your code should now pass "alloc_block".

File Operations

We have provided a variety of functions in `fs/fs.c` to implement the basic facilities you will need to interpret and manage `File` structures, scan and manage the entries of directory-files, and walk the file system from the root to resolve an absolute pathname. Read through *all* of the code in `fs/fs.c` and make sure you understand what each function does before proceeding.

Exercise 4. Implement `file_block_walk` and `file_get_block`. `file_block_walk` maps from a block offset within a file to the pointer for that block in the `struct File` or the indirect block, very much like what `pgdir_walk` did for page tables. `file_get_block` goes one step further and maps to the actual disk block, allocating a new one if necessary.

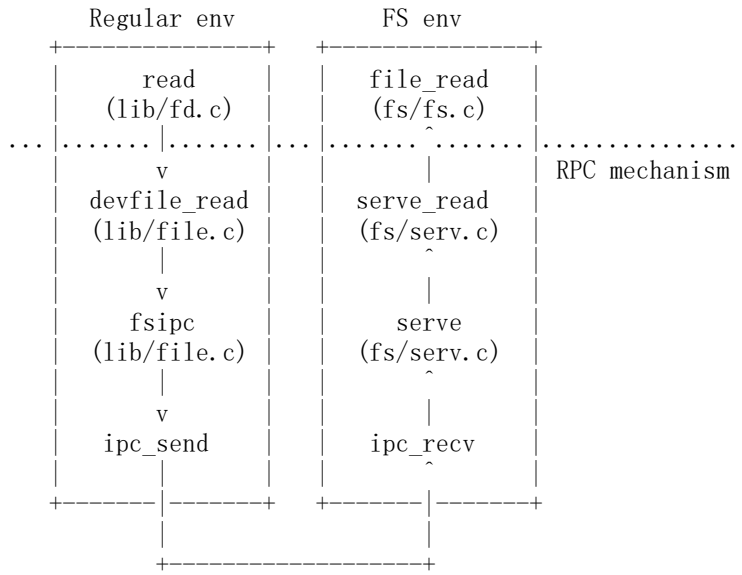
Use `make grade` to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file rewrite", and "testfile".

`file_block_walk` and `file_get_block` are the workhorses of the file system. For example, `file_read` and `file_write` are little more than the bookkeeping atop `file_get_block` necessary to copy bytes between scattered blocks and a sequential buffer.

Challenge! The file system is likely to be corrupted if it gets interrupted in the middle of an operation (for example, by a crash or a reboot). Implement soft updates or journalling to make the file system crash-resilient and demonstrate some situation where the old file system would get corrupted, but yours doesn't.

The file system interface

Now that we have the necessary functionality within the file system environment itself, we must make it accessible to other environments that wish to use the file system. Since other environments can't directly call functions in the file system environment, we'll expose access to the file system environment via a *remote procedure call*, or RPC, abstraction, built atop JOS's IPC mechanism. Graphically, here's what a call to the file system server (say, read) looks like



Everything below the dotted line is simply the mechanics of getting a read request from the regular environment to the file system environment. Starting at the beginning, `read` (which we provide) works on any file descriptor and simply dispatches to the appropriate device read function, in this case `devfile_read` (we can have more device types, like pipes). `devfile_read` implements `read` specifically for on-disk files. This and the other `devfile_*` functions in `lib/file.c` implement the client side of the FS operations and all work in roughly the same way, bundling up arguments in a request structure, calling `fsipc` to send the IPC request, and unpacking and returning the results. The `fsipc` function simply handles the common details of sending a request to the server and receiving the reply.

The file system server code can be found in `fs/serv.c`. It loops in the `serve` function, endlessly receiving a request over IPC, dispatching that request to the appropriate handler function, and sending the result back via IPC. In the read example, `serve` will dispatch to `serve_read`, which will take care of the IPC details specific to read requests such as unpacking the request structure and finally call `file_read` to actually perform the file read.

Recall that JOS's IPC mechanism lets an environment send a single 32-bit number and, optionally, share a page. To send a request from the client to the server, we use the 32-bit number for the request type (the file system server RPCs are numbered, just like how syscalls were numbered) and store the arguments to the request in a `union Fsipc` on the page shared via the IPC. On the client side, we always share the page at `fsipcbuf`; on the server side, we map the incoming request page at `fsreq` (`0x0ffff000`).

The server also sends the response back via IPC. We use the 32-bit number for the function's return code. For most RPCs, this is all they return. `FSREQ_READ` and `FSREQ_STAT` also return data, which they simply write to the page that the client sent its request on. There's no need to send this page in the response IPC, since the client shared it with the file system server in the first place. Also, in its response, `FSREQ_OPEN` shares with the client a new "Fd page". We'll return to the file descriptor page shortly.

> **Exercise 5.** Implement `serve_read` in `fs/serv.c`.

`serve_read`'s heavy lifting will be done by the already-implemented `file_read` in `fs/fs.c` (which, in turn, is just a bunch of calls to `file_get_block`). `serve_read` just has to provide the RPC interface for file reading. Look at the comments and code in `serve_set_size` to get a general idea of how the server functions should be structured.

Use `make grade` to test your code. Your code should pass "serve_open/file_stat/file_close" and "file_read" for a score of 70/150.

Exercise 6. Implement `serve_write` in `fs/serv.c` and `devfile_write` in `lib/file.c`.

Use `make grade` to test your code. Your code should pass "file_write", "file_read after file_write", "open", and "large file" for a score of 90/150.

Spawning Processes

We have given you the code for `spawn` (see `lib/spawn.c`) which creates a new environment, loads a program image from the file system into it, and then starts the child environment running this program. The parent process then continues running independently of the child. The `spawn` function effectively acts like a `fork` in UNIX followed by an immediate `exec` in the child process.

We implemented `spawn` rather than a UNIX-style `exec` because `spawn` is easier to implement from user space in "exokernel fashion", without special help from the kernel. Think about what you would have to do in order to implement `exec` in user space, and be sure you understand why it is harder.

Exercise 7. `spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Implement `sys_env_set_trapframe` in `kern/syscall.c` (don't forget to dispatch the new system call in `syscall()`).

Test your code by running the `user/spawnhello` program from `kern/init.c`, which will attempt to spawn `/hello` from the file system.

Use `make grade` to test your code.

Challenge! Implement Unix-style `exec`.

Challenge! Implement `mmap`-style memory-mapped files and modify `spawn` to map pages directly from the ELF image when possible.

Sharing library state across fork and spawn

The UNIX file descriptors are a general notion that also encompasses pipes, console I/O, etc. In JOS, each of these device types has a corresponding `struct Dev`, with pointers to the functions that implement read/write/etc. for that device type. `lib/fd.c` implements the general UNIX-like file descriptor interface on top of this. Each `struct Fd` indicates its device type, and most of the functions in `lib/fd.c` simply dispatch operations to functions in the appropriate `struct Dev`.

`lib/fd.c` also maintains the *file descriptor table* region in each application environment's address space, starting at `FDTABLE`. This area reserves a page's worth (4KB) of address space for each of the up to `MAXFD` (currently 32) file descriptors the application can have open at once. At any given time, a particular file descriptor table page is mapped if and only if the corresponding file descriptor is in use. Each file descriptor also has an optional "data page" in the region starting at `FILEDATA`, which devices can use if they choose.

We would like to share file descriptor state across `fork` and `spawn`, but file descriptor state is kept in user-space memory. Right now, on `fork`, the memory will be marked copy-on-write, so the state will be duplicated rather than shared. (This means environments won't be able to seek in files they didn't open themselves and that pipes won't work across a fork.) On `spawn`, the memory will be left behind, not copied at all. (Effectively, the spawned environment starts with no open file descriptors.)

We will change `fork` to know that certain regions of memory are used by the "library operating system" and should always be shared. Rather than hard-code a list of regions somewhere, we will set an otherwise-unused bit in the page table entries (just like we did with the `PTE_COW` bit in `fork`).

We have defined a new `PTE_SHARE` bit in `inc/lib.h`. This bit is one of the three PTE bits that are marked "available for software use" in the Intel and AMD manuals. We will establish the convention that if a page table entry has this bit set, the PTE should be copied directly from parent to child in both `fork` and `spawn`. Note that this is different from marking it copy-on-write: as described in the first paragraph, we want to make sure to *share* updates to the page.

> **Exercise 8.** Change `duppage` in `lib/fork.c` to follow the new convention. If the page table entry has the `PTE_SHARE` bit set, just copy the mapping directly. (You should use `PTE_SYSCALL`, not `0xfff`, to mask out the relevant bits from the page table entry. `0xfff` picks up the accessed and dirty bits as well.)
>
> Likewise, implement `copy_shared_pages` in `lib/spawn.c`. It should loop through all page table entries in the current process (just like `fork` did), copying any page mappings that have the `PTE_SHARE` bit set into the child process.

Use `make run-testpteshare` to check that your code is behaving properly. You should see lines that say "`fork handles PTE_SHARE right`" and "`spawn handles PTE_SHARE right`".

Use `make run-testfdsharing` to check that file descriptors are shared properly. You should see lines that say "`read in child succeeded`" and "`read in parent succeeded`".

The keyboard interface

For the shell to work, we need a way to type at it. QEMU has been displaying output we write to the CGA display and the serial port, but so far we've only taken input while in the kernel monitor. In QEMU, input typed in the graphical window appear as input from the keyboard to JOS, while input typed to the console appear as characters on the serial port. `kern/console.c` already contains the keyboard and serial drivers that have been used by the kernel monitor since lab 1, but now you need to attach these to the rest of the system.

> **Exercise 9.** In your `kern/trap.c`, call `kbd_intr` to handle trap `IRQ_OFFSET+IRQ_KBD` and `serial_intr` to handle trap `IRQ_OFFSET+IRQ_SERIAL`.

We implemented the console input/output file type for you, in `lib/console.c`. `kbd_intr` and `serial_intr` fill a buffer with the recently read input while the console file type drains the buffer (the console file type is used for stdin/stdout by default unless the user redirects them).

Test your code by running `make run-testkbd` and type a few lines. The system should echo your lines back to you as you finish them. Try typing in both the console and the graphical window, if you have both available.

The Shell

Run `make run-icode` or `make run-icode-nox`. This will run your kernel and start `user/icode`. `icode` execs `init`, which will set up the console as file descriptors 0 and 1 (standard input and standard output). It will then spawn `sh`, the shell. You should be able to run the following commands:

```
echo hello world | cat
cat lorem |cat
cat lorem |num
cat lorem |num |num |num |num |num
lsfd
```

Note that the user library routine `cprintf` prints straight to the console, without using the file descriptor code. This is great for debugging but not great for piping into other programs. To print output to a particular file descriptor (for example, 1, standard output), use `fprintf(1, "...", ...)`. `printf("...", ...)` is a short-cut for printing to FD 1. See `user/lsfd.c` for examples.

Exercise 10.

The shell doesn't support I/O redirection. It would be nice to run `sh <script` instead of having to type in all the commands in the script by hand, as you did above. Add I/O redirection for < to `user/sh.c`.

Test your implementation by typing `sh <script` into your shell

Run `make run-testshell` to test your shell. `testshell` simply feeds the above commands (also found in `fs/testshell.sh`) into the shell and then checks that the output matches `fs/testshell.key`.

Challenge! Add more features to the shell. Possibilities include (a few require changes to the file system too):

- backgrounding commands (`ls &`)
- multiple commands per line (`ls; echo hi`)
- command grouping (`(ls; echo hi) | cat > out`)
- environment variable expansion (`echo $hello`)
- quoting (`echo "a | b"`)
- command-line history and/or editing
- tab completion
- directories, cd, and a PATH for command-lookup.
- file creation
- ctl-c to kill the running environment

but feel free to do something not on this list.

Your code should pass all tests at this point. As usual, you can grade your submission with `make grade` and hand it in with `make handin`.

This completes the lab. As usual, don't forget to run `make grade` and to write up your answers and a description of your challenge exercise solution. Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab5.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 5'`, then `make handin` to submit your solution.

Lab 6: Network Driver (default final project)

Due on Thursday, December 6, 2018

Introduction

This lab is the default final project that you can do on your own.

Now that you have a file system, no self respecting OS should go without a network stack. In this the lab you are going to write a driver for a network interface card. The card will be based on the Intel 82540EM chip, also known as the E1000.

Getting Started

Use Git to commit your Lab 5 source (if you haven't already), fetch the latest version of the course repository, and then create a local branch called `lab6` based on our lab6 branch, `origin/lab6`:

```
athena% cd ~/6.828/lab
athena% add git
athena% git commit -am 'my solution to lab5'
nothing to commit (working directory clean)
athena% git pull
Already up-to-date.
athena% git checkout -b lab6 origin/lab6
Branch lab6 set up to track remote branch refs/remotes/origin/lab6.
Switched to a new branch "lab6"
athena% git merge lab5
Merge made by recursive.
 fs/fs.c |   42 ++++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
athena%
```

The network card driver, however, will not be enough to get your OS hooked up to the Internet. In the new lab6 code, we have provided you with a network stack and a network server. As in previous labs, use git to grab the code for this lab, merge in your own code, and explore the contents of the new `net/` directory, as well as the new files in `kern/`.

In addition to writing the driver, you will need to create a system call interface to give access to your driver. You will implement missing network server code to transfer packets between the network stack and your driver. You will also tie everything together by finishing a web server. With the new web server you will be able to serve files from your file system.

Much of the kernel device driver code you will have to write yourself from scratch. This lab provides much less guidance than previous labs: there are no skeleton files, no system call interfaces written in stone, and many design decisions are left up to you. For this reason, we recommend that you read the entire assignment write up before starting any individual exercises. Many students find this lab more difficult than previous labs, so please plan your time accordingly.

Lab Requirements

As before, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem. Write up brief answers to the questions posed in the lab and a description of your challenge exercise in `answers-lab6.txt`.

QEMU's virtual network

We will be using QEMU's user mode network stack since it requires no administrative privileges to run. QEMU's documentation has more about user-net here. We've updated the makefile to enable QEMU's user-mode network stack and the virtual E1000 network card.

By default, QEMU provides a virtual router running on IP 10.0.2.2 and will assign JOS the IP address 10.0.2.15. To keep things simple, we hard-code these defaults into the network server in `net/ns.h`.

While QEMU's virtual network allows JOS to make arbitrary connections out to the Internet, JOS's 10.0.2.15 address has no meaning outside the virtual network running inside QEMU (that is, QEMU acts as a NAT), so we can't connect directly to servers running inside JOS, even from the host running QEMU. To address this, we configure QEMU to run a server on some port on the *host* machine that simply connects through to some port in JOS and shuttles data back and forth between your real host and the virtual network.

You will run JOS servers on ports 7 (echo) and 80 (http). To avoid collisions on shared Athena machines, the makefile generates forwarding ports for these based on your user ID. To find out what ports QEMU is forwarding to on your development host, run `make which-ports`. For convenience, the makefile also provides `make nc-7` and `make nc-80`, which allow you to interact directly with servers running on these ports in your terminal. (These targets only connect to a running QEMU instance; you must start QEMU itself separately.)

Packet Inspection

The makefile also configures QEMU's network stack to record all incoming and outgoing packets to `qemu.pcap` in your lab directory.

To get a hex/ASCII dump of captured packets use `tcpdump` like this:

```
tcpdump -XXnr qemu.pcap
```

Alternatively, you can use Wireshark to graphically inspect the pcap file. Wireshark also knows how to decode and inspect hundreds of network protocols. If you're on Athena, you'll have to use Wireshark's predecessor, ethereal, which is in the sipbnet locker.

Debugging the E1000

We are very lucky to be using emulated hardware. Since the E1000 is running in software, the emulated E1000 can report to us, in a user readable format, its internal state and any problems it encounters. Normally, such a luxury would not be available to a driver developer writing with bare metal.

The E1000 can produce a lot of debug output, so you have to enable specific logging channels. Some channels you might find useful are:

Flag	Meaning
tx	Log packet transmit operations
txerr	Log transmit ring errors
rx	Log changes to RCTL
rxfilter	Log filtering of incoming packets
rxerr	Log receive ring errors

unknown	Log reads and writes of unknown registers
eeprom	Log reads from the EEPROM
interrupt	Log interrupts and changes to interrupt registers.

To enable "tx" and "txerr" logging, for example, use `make E1000_DEBUG=tx,txerr ...`.

> *Note:* `E1000_DEBUG` flags only work in the 6.828 version of QEMU.

You can take debugging using software emulated hardware one step further. If you are ever stuck and do not understand why the E1000 is not responding the way you would expect, you can look at QEMU's E1000 implementation in `hw/net/e1000.c`.

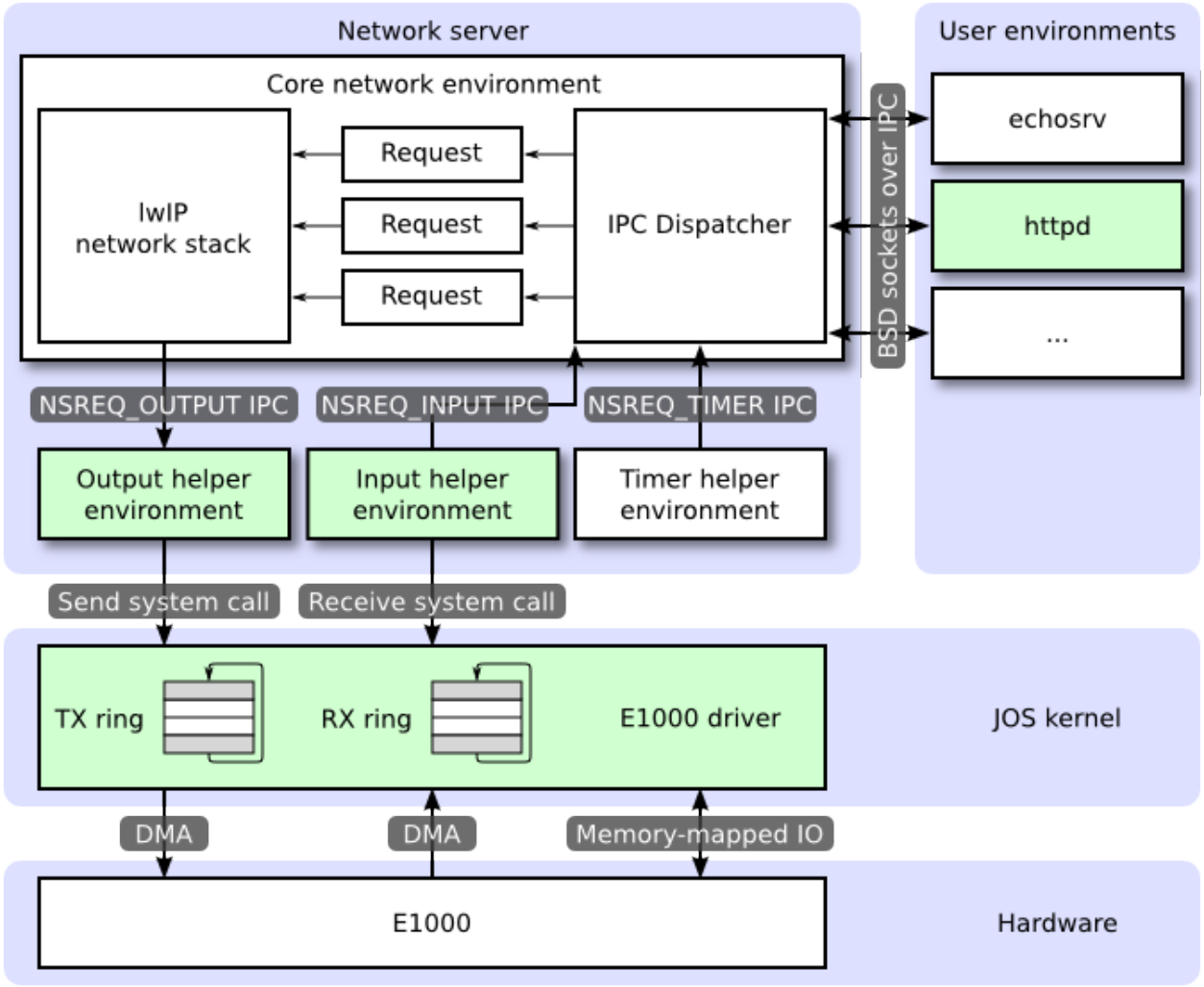
The Network Server

Writing a network stack from scratch is hard work. Instead, we will be using lwIP, an open source lightweight TCP/IP protocol suite that among many things includes a network stack. You can find more information on lwIP here. In this assignment, as far as we are concerned, lwIP is a black box that implements a BSD socket interface and has a packet input port and packet output port.

The network server is actually a combination of four environments:

- core network server environment (includes socket call dispatcher and lwIP)
- input environment
- output environment
- timer environment

The following diagram shows the different environments and their relationships. The diagram shows the entire system including the device driver, which will be covered later. In this lab, you will implement the parts highlighted in green.



The Core Network Server Environment

The core network server environment is composed of the socket call dispatcher and lwIP itself. The socket call dispatcher works exactly like the file server. User environments use stubs (found in `lib/nsipc.c`) to send IPC messages to the core network environment. If you look at `lib/nsipc.c` you will see that we find the core network server the same way we found the file server: `i386_init` created the NS environment with NS_TYPE_NS, so we scan `envs`, looking for this special environment type. For each user environment IPC, the dispatcher in the network server calls the appropriate BSD socket interface function provided by lwIP on behalf of the user.

Regular user environments do not use the `nsipc_*` calls directly. Instead, they use the functions in `lib/sockets.c`, which provides a file descriptor-based sockets API. Thus, user environments refer to sockets via file descriptors, just like how they referred to on-disk files. A number of operations (`connect`, `accept`, etc.) are specific to sockets, but `read`, `write`, and `close` go through the normal file descriptor device-dispatch code in `lib/fd.c`. Much like how the file server maintained internal unique ID's for all open files, lwIP also generates unique ID's for all open sockets. In both the file server and the network server, we use information stored in `struct Fd` to map per-environment file descriptors to these unique ID spaces.

Even though it may seem that the IPC dispatchers of the file server and network server act the same, there is a key difference. BSD socket calls like `accept` and `recv` can block indefinitely. If the dispatcher were to let lwIP execute one of these blocking calls, the dispatcher would also block and there could only be one outstanding network call at a time for the whole system. Since this is unacceptable, the network server uses user-level threading to avoid blocking the entire

server environment. For every incoming IPC message, the dispatcher creates a thread and processes the request in the newly created thread. If the thread blocks, then only that thread is put to sleep while other threads continue to run.

In addition to the core network environment there are three helper environments. Besides accepting messages from user applications, the core network environment's dispatcher also accepts messages from the input and timer environments.

The Output Environment

When servicing user environment socket calls, lwIP will generate packets for the network card to transmit. LwIP will send each packet to be transmitted to the output helper environment using the `NSREQ_OUTPUT` IPC message with the packet attached in the page argument of the IPC message. The output environment is responsible for accepting these messages and forwarding the packet on to the device driver via the system call interface that you will soon create.

The Input Environment

Packets received by the network card need to be injected into lwIP. For every packet received by the device driver, the input environment pulls the packet out of kernel space (using kernel system calls that you will implement) and sends the packet to the core server environment using the `NSREQ_INPUT` IPC message.

The packet input functionality is separated from the core network environment because JOS makes it hard to simultaneously accept IPC messages and poll or wait for a packet from the device driver. We do not have a `select` system call in JOS that would allow environments to monitor multiple input sources to identify which input is ready to be processed.

If you take a look at `net/input.c` and `net/output.c` you will see that both need to be implemented. This is mainly because the implementation depends on your system call interface. You will write the code for the two helper environments after you implement the driver and system call interface.

The Timer Environment

The timer environment periodically sends messages of type `NSREQ_TIMER` to the core network server notifying it that a timer has expired. The timer messages from this thread are used by lwIP to implement various network timeouts.

Part A: Initialization and transmitting packets

Your kernel does not have a notion of time, so we need to add it. There is currently a clock interrupt that is generated by the hardware every 10ms. On every clock interrupt we can increment a variable to indicate that time has advanced by 10ms. This is implemented in `kern/time.c`, but is not yet fully integrated into your kernel.

> **Exercise 1.** Add a call to `time_tick` for every clock interrupt in `kern/trap.c`. Implement `sys_time_msec` and add it to `syscall` in `kern/syscall.c` so that user space has access to the time.

Use `make INIT_CFLAGS=-DTEST_NO_NS run-testtime` to test your time code. You should see the environment count down from 5 in 1 second intervals. The "-DTEST_NO_NS" disables starting the network server environment because it will panic at this point in the lab.

The Network Interface Card

Writing a driver requires knowing in depth the hardware and the interface presented to the software. The lab text will provide a high-level overview of how to interface with the E1000, but you'll need to make extensive use of Intel's manual while writing your driver.

> **Exercise 2.** Browse Intel's Software Developer's Manual for the E1000. This manual covers several closely related Ethernet controllers. QEMU emulates the 82540EM.
>
> You should skim over chapter 2 now to get a feel for the device. To write your driver, you'll need to be familiar with chapters 3 and 14, as well as 4.1 (though not 4.1's subsections). You'll also need to use chapter 13 as reference. The other chapters mostly cover components of the E1000 that your driver won't have to interact with. Don't worry about the details right now; just get a feel for how the document is structured so you can find things later.
>
> While reading the manual, keep in mind that the E1000 is a sophisticated device with many advanced features. A working E1000 driver only needs a fraction of the features and interfaces that the NIC provides. Think carefully about the easiest way to interface with the card. We strongly recommend that you get a basic driver working before taking advantage of the advanced features.

PCI Interface

The E1000 is a PCI device, which means it plugs into the PCI bus on the motherboard. The PCI bus has address, data, and interrupt lines, and allows the CPU to communicate with PCI devices and PCI devices to read and write memory. A PCI device needs to be discovered and initialized before it can be used. Discovery is the process of walking the PCI bus looking for attached devices. Initialization is the process of allocating I/O and memory space as well as negotiating the IRQ line for the device to use.

We have provided you with PCI code in `kern/pci.c`. To perform PCI initialization during boot, the PCI code walks the PCI bus looking for devices. When it finds a device, it reads its vendor ID and device ID and uses these two values as a key to search the `pci_attach_vendor` array. The array is composed of `struct pci_driver` entries like this:

```
struct pci_driver {
    uint32_t key1, key2;
    int (*attachfn) (struct pci_func *pcif);
};
```

If the discovered device's vendor ID and device ID match an entry in the array, the PCI code calls that entry's `attachfn` to perform device initialization. (Devices can also be identified by class, which is what the other driver table in `kern/pci.c` is for.)

The attach function is passed a *PCI function* to initialize. A PCI card can expose multiple functions, though the E1000 exposes only one. Here is how we represent a PCI function in JOS:

```
struct pci_func {
    struct pci_bus *bus;

    uint32_t dev;
    uint32_t func;

    uint32_t dev_id;
    uint32_t dev_class;

    uint32_t reg_base[6];
    uint32_t reg_size[6];
    uint8_t irq_line;
};
```

The above structure reflects some of the entries found in Table 4-1 of Section 4.1 of the developer manual. The last three entries of `struct pci_func` are of particular interest to us, as they record the negotiated memory, I/O, and interrupt resources for the device. The `reg_base` and `reg_size` arrays contain information for up to six Base Address Registers or BARs. `reg_base` stores the base memory addresses for memory-mapped I/O regions (or base I/O ports for I/O port resources), `reg_size` contains the size in bytes or number of I/O ports for the corresponding base values from `reg_base`, and `irq_line` contains the IRQ line assigned to the device for interrupts. The specific meanings of the E1000 BARs are given in the second half of table 4-2.

When the attach function of a device is called, the device has been found but not yet *enabled*. This means that the PCI code has not yet determined the resources allocated to the device, such as address space and an IRQ line, and, thus, the last three elements of the `struct pci_func` structure are not yet filled in. The attach function should call `pci_func_enable`, which will enable the device, negotiate these resources, and fill in the `struct pci_func`.

> **Exercise 3.** Implement an attach function to initialize the E1000. Add an entry to the `pci_attach_vendor` array in `kern/pci.c` to trigger your function if a matching PCI device is found (be sure to put it before the `{0, 0, 0}` entry that mark the end of the table). You can find the vendor ID and device ID of the 82540EM that QEMU emulates in section 5.2. You should also see these listed when JOS scans the PCI bus while booting.
>
> For now, just enable the E1000 device via `pci_func_enable`. We'll add more initialization throughout the lab.
>
> We have provided the `kern/e1000.c` and `kern/e1000.h` files for you so that you do not need to mess with the build system. They are currently blank; you need to fill them in for this exercise. You may also need to include the `e1000.h` file in other places in the kernel.
>
> When you boot your kernel, you should see it print that the PCI function of the E1000 card was enabled. Your code should now pass the `pci attach` test of `make grade`.

Memory-mapped I/O

Software communicates with the E1000 via *memory-mapped I/O* (MMIO). You've seen this twice before in JOS: both the CGA console and the LAPIC are devices that you control and

query by writing to and reading from "memory". But these reads and writes don't go to DRAM; they go directly to these devices.

`pci_func_enable` negotiates an MMIO region with the E1000 and stores its base and size in BAR 0 (that is, `reg_base[0]` and `reg_size[0]`). This is a range of *physical memory addresses* assigned to the device, which means you'll have to do something to access it via virtual addresses. Since MMIO regions are assigned very high physical addresses (typically above 3GB), you can't use `KADDR` to access it because of JOS's 256MB limit. Thus, you'll have to create a new memory mapping. We'll use the area above MMIOBASE (your `mmio_map_region` from lab 4 will make sure we don't overwrite the mapping used by the LAPIC). Since PCI device initialization happens before JOS creates user environments, you can create the mapping in `kern_pgdir` and it will always be available.

> **Exercise 4.** In your attach function, create a virtual memory mapping for the E1000's BAR 0 by calling `mmio_map_region` (which you wrote in lab 4 to support memory-mapping the LAPIC).
>
> You'll want to record the location of this mapping in a variable so you can later access the registers you just mapped. Take a look at the `lapic` variable in `kern/lapic.c` for an example of one way to do this. If you do use a pointer to the device register mapping, be sure to declare it `volatile`; otherwise, the compiler is allowed to cache values and reorder accesses to this memory.
>
> To test your mapping, try printing out the device status register (section 13.4.2). This is a 4 byte register that starts at byte 8 of the register space. You should get `0x80080783`, which indicates a full duplex link is up at 1000 MB/s, among other things.

Hint: You'll need a lot of constants, like the locations of registers and values of bit masks. Trying to copy these out of the developer's manual is error-prone and mistakes can lead to painful debugging sessions. We recommend instead using QEMU's `e1000_hw.h` header as a guideline. We don't recommend copying it in verbatim, because it defines far more than you actually need and may not define things in the way you need, but it's a good starting point.

DMA

You could imagine transmitting and receiving packets by writing and reading from the E1000's registers, but this would be slow and would require the E1000 to buffer packet data internally. Instead, the E1000 uses *Direct Memory Access* or DMA to read and write packet data directly from memory without involving the CPU. The driver is responsible for allocating memory for the transmit and receive queues, setting up DMA descriptors, and configuring the E1000 with the location of these queues, but everything after that is asynchronous. To transmit a packet, the driver copies it into the next DMA descriptor in the transmit queue and informs the E1000 that another packet is available; the E1000 will copy the data out of the descriptor when there is time to send the packet. Likewise, when the E1000 receives a packet, it copies it into the next DMA descriptor in the receive queue, which the driver can read from at its next opportunity.

The receive and transmit queues are very similar at a high level. Both consist of a sequence of *descriptors*. While the exact structure of these descriptors varies, each descriptor contains some flags and the physical address of a buffer containing packet data (either packet data for the card to send, or a buffer allocated by the OS for the card to write a received packet to).

The queues are implemented as circular arrays, meaning that when the card or the driver reach the end of the array, it wraps back around to the beginning. Both have a *head pointer* and a *tail pointer* and the contents of the queue are the descriptors between these two pointers. The hardware always consumes descriptors from the head and moves the head pointer, while the driver always add descriptors to the tail and moves the tail pointer. The descriptors in the transmit queue represent packets waiting to be sent (hence, in the steady state, the transmit queue is empty). For the receive queue, the descriptors in the queue are free descriptors that the card can receive packets into (hence, in the steady state, the receive queue consists of all available receive descriptors). Correctly updating the tail register without confusing the E1000 is tricky; be careful!

The pointers to these arrays as well as the addresses of the packet buffers in the descriptors must all be *physical addresses* because hardware performs DMA directly to and from physical RAM without going through the MMU.

Transmitting Packets

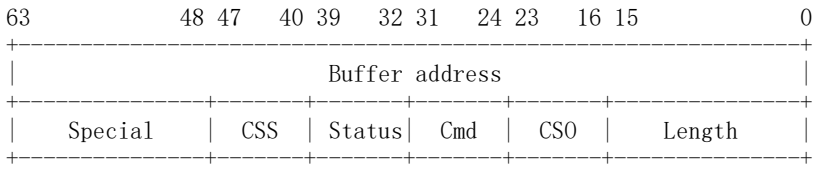
The transmit and receive functions of the E1000 are basically independent of each other, so we can work on one at a time. We'll attack transmitting packets first simply because we can't test receive without transmitting an "I'm here!" packet first.

First, you'll have to initialize the card to transmit, following the steps described in section 14.5 (you don't have to worry about the subsections). The first step of transmit initialization is setting up the transmit queue. The precise structure of the queue is described in section 3.4 and the structure of the descriptors is described in section 3.3.3. We won't be using the TCP offload features of the E1000, so you can focus on the "legacy transmit descriptor format." You should read those sections now and familiarize yourself with these structures.

C Structures

You'll find it convenient to use C `struct`s to describe the E1000's structures. As you've seen with structures like the `struct Trapframe`, C `struct`s let you precisely layout data in memory. C can insert padding between fields, but the E1000's structures are laid out such that this shouldn't be a problem. If you do encounter field alignment problems, look into GCC's "packed" attribute.

As an example, consider the legacy transmit descriptor given in table 3-8 of the manual and reproduced here:



The first byte of the structure starts at the top right, so to convert this into a C struct, read from right to left, top to bottom. If you squint at it right, you'll see that all of the fields even fit nicely into a standard-size types:

```
struct tx_desc
{
        uint64_t addr;
        uint16_t length;
        uint8_t cso;
        uint8_t cmd;
        uint8_t status;
        uint8_t css;
```

```
        uint16_t special;
};
```

Your driver will have to reserve memory for the transmit descriptor array and the packet buffers pointed to by the transmit descriptors. There are several ways to do this, ranging from dynamically allocating pages to simply declaring them in global variables. Whatever you choose, keep in mind that the E1000 accesses physical memory directly, which means any buffer it accesses must be contiguous in physical memory.

There are also multiple ways to handle the packet buffers. The simplest, which we recommend starting with, is to reserve space for a packet buffer for each descriptor during driver initialization and simply copy packet data into and out of these pre-allocated buffers. The maximum size of an Ethernet packet is 1518 bytes, which bounds how big these buffers need to be. More sophisticated drivers could dynamically allocate packet buffers (e.g., to reduce memory overhead when network usage is low) or even pass buffers directly provided by user space (a technique known as "zero copy"), but it's good to start simple.

> **Exercise 5.** Perform the initialization steps described in section 14.5 (but not its subsections). Use section 13 as a reference for the registers the initialization process refers to and sections 3.3.3 and 3.4 for reference to the transmit descriptors and transmit descriptor array.
>
> Be mindful of the alignment requirements on the transmit descriptor array and the restrictions on length of this array. Since TDLEN must be 128-byte aligned and each transmit descriptor is 16 bytes, your transmit descriptor array will need some multiple of 8 transmit descriptors. However, don't use more than 64 descriptors or our tests won't be able to test transmit ring overflow.
>
> For the TCTL.COLD, you can assume full-duplex operation. For TIPG, refer to the default values described in table 13-77 of section 13.4.34 for the IEEE 802.3 standard IPG (don't use the values in the table in section 14.5).

Try running `make E1000_DEBUG=TXERR,TX qemu`. If you are using the course qemu, you should see an "e1000: tx disabled" message when you set the TDT register (since this happens before you set TCTL.EN) and no further "e1000" messages.

Now that transmit is initialized, you'll have to write the code to transmit a packet and make it accessible to user space via a system call. To transmit a packet, you have to add it to the tail of the transmit queue, which means copying the packet data into the next packet buffer and then updating the TDT (transmit descriptor tail) register to inform the card that there's another packet in the transmit queue. (Note that TDT is an *index* into the transmit descriptor array, not a byte offset; the documentation isn't very clear about this.)

However, the transmit queue is only so big. What happens if the card has fallen behind transmitting packets and the transmit queue is full? In order to detect this condition, you'll need some feedback from the E1000. Unfortunately, you can't just use the TDH (transmit descriptor head) register; the documentation explicitly states that reading this register from software is unreliable. However, if you set the RS bit in the command field of a transmit descriptor, then, when the card has transmitted the packet in that descriptor, the card will set the DD bit in the status field of the descriptor. If a descriptor's DD bit is set, you know it's safe to recycle that descriptor and use it to transmit another packet.

What if the user calls your transmit system call, but the DD bit of the next descriptor isn't set, indicating that the transmit queue is full? You'll have to decide what to do in this situation. You could simply drop the packet. Network protocols are resilient to this, but if you drop a large burst of packets, the protocol may not recover. You could instead tell the user environment that it has to retry, much like you did for `sys_ipc_try_send`. This has the advantage of pushing back on the environment generating the data.

> **Exercise 6.** Write a function to transmit a packet by checking that the next descriptor is free, copying the packet data into the next descriptor, and updating TDT. Make sure you handle the transmit queue being full.

Now would be a good time to test your packet transmit code. Try transmitting just a few packets by directly calling your transmit function from the kernel. You don't have to create packets that conform to any particular network protocol in order to test this. Run `make E1000_DEBUG=TXERR,TX qemu` to run your test. You should see something like

```
e1000: index 0: 0x271f00 : 9000002a 0
...
```

as you transmit packets. Each line gives the index in the transmit array, the buffer address of that transmit descriptor, the cmd/CSO/length fields, and the special/CSS/status fields. If QEMU doesn't print the values you expected from your transmit descriptor, check that you're filling in the right descriptor and that you configured TDBAL and TDBAH correctly. If you get "e1000: TDH wraparound @0, TDT x, TDLEN y" messages, that means the E1000 ran all the way through the transmit queue without stopping (if QEMU didn't check this, it would enter an infinite loop), which probably means you aren't manipulating TDT correctly. If you get lots of "e1000: tx disabled" messages, then you didn't set the transmit control register right.

Once QEMU runs, you can then run `tcpdump -XXnr qemu.pcap` to see the packet data that you transmitted. If you saw the expected "e1000: index" messages from QEMU, but your packet capture is empty, double check that you filled in every necessary field and bit in your transmit descriptors (the E1000 probably went through your transmit descriptors, but didn't think it had to send anything).

> **Exercise 7.** Add a system call that lets you transmit packets from user space. The exact interface is up to you. Don't forget to check any pointers passed to the kernel from user space.

Transmitting Packets: Network Server

Now that you have a system call interface to the transmit side of your device driver, it's time to send packets. The output helper environment's goal is to do the following in a loop: accept `NSREQ_OUTPUT` IPC messages from the core network server and send the packets accompanying these IPC message to the network device driver using the system call you added above. The `NSREQ_OUTPUT` IPC's are sent by the `low_level_output` function in `net/lwip/jos/jif/jif.c`, which glues the lwIP stack to JOS's network system. Each IPC will include a page consisting of a `union Nsipc` with the packet in its `struct jif_pkt pkt` field (see `inc/ns.h`). `struct jif_pkt` looks like

```
struct jif_pkt {
    int jp_len;
    char jp_data[0];
};
```

`jp_len` represents the length of the packet. All subsequent bytes on the IPC page are dedicated to the packet contents. Using a zero-length array like `jp_data` at the end of a struct is a common C trick (some would say abomination) for representing buffers without pre-determined lengths. Since C doesn't do array bounds checking, as long as you ensure there's enough unused memory following the struct, you can use `jp_data` as if it were an array of any size.

Be aware of the interaction between the device driver, the output environment and the core network server when there is no more space in the device driver's transmit queue. The core network server sends packets to the output environment using IPC. If the output environment is suspended due to a send packet system call because the driver has no more buffer space for new packets, the core network server will block waiting for the output server to accept the IPC call.

> **Exercise 8.** Implement `net/output.c`.

You can use `net/testoutput.c` to test your output code without involving the whole network server. Try running `make E1000_DEBUG=TXERR,TX run-net_testoutput`. You should see something like

```
Transmitting packet 0
e1000: index 0: 0x271f00 : 9000009 0
Transmitting packet 1
e1000: index 1: 0x2724ee : 9000009 0
...
```

and `tcpdump -XXnr qemu.pcap` should output

```
reading from file qemu.pcap, link-type EN10MB (Ethernet)
-5:00:00.600186 [|ether]
        0x0000:  5061 636b 6574 2030 30                 Packet.00
-5:00:00.610080 [|ether]
        0x0000:  5061 636b 6574 2030 31                 Packet.01
...
```

To test with a larger packet count, try `make E1000_DEBUG=TXERR,TX NET_CFLAGS=-DTESTOUTPUT_COUNT=100 run-net_testoutput`. If this overflows your transmit ring, double check that you're handling the DD status bit correctly and that you've told the hardware to set the DD status bit (using the RS command bit).

Your code should pass the `testoutput` tests of `make grade`.

> **Question**
>
> 1. How did you structure your transmit implementation? In particular, what do you do if the transmit ring is full?

Part B: Receiving packets and the web server

Receiving Packets

Just like you did for transmitting packets, you'll have to configure the E1000 to receive packets and provide a receive descriptor queue and receive descriptors. Section 3.2 describes how packet reception works, including the receive queue structure and receive descriptors, and the initialization process is detailed in section 14.4.

> **Exercise 9.** Read section 3.2. You can ignore anything about interrupts and checksum offloading (you can return to these sections if you decide to use these features later), and you don't have to be concerned with the details of thresholds and how the card's internal caches work.

The receive queue is very similar to the transmit queue, except that it consists of empty packet buffers waiting to be filled with incoming packets. Hence, when the network is idle, the transmit queue is empty (because all packets have been sent), but the receive queue is full (of empty packet buffers).

When the E1000 receives a packet, it first checks if it matches the card's configured filters (for example, to see if the packet is addressed to this E1000's MAC address) and ignores the packet if it doesn't match any filters. Otherwise, the E1000 tries to retrieve the next receive descriptor from the head of the receive queue. If the head (RDH) has caught up with the tail (RDT), then the receive queue is out of free descriptors, so the card drops the packet. If there is a free receive descriptor, it copies the packet data into the buffer pointed to by the descriptor, sets the descriptor's DD (Descriptor Done) and EOP (End of Packet) status bits, and increments the RDH.

If the E1000 receives a packet that is larger than the packet buffer in one receive descriptor, it will retrieve as many descriptors as necessary from the receive queue to store the entire contents of the packet. To indicate that this has happened, it will set the DD status bit on all of these descriptors, but only set the EOP status bit on the last of these descriptors. You can either deal with this possibility in your driver, or simply configure the card to not accept "long packets" (also known as *jumbo frames*) and make sure your receive buffers are large enough to store the largest possible standard Ethernet packet (1518 bytes).

> **Exercise 10.** Set up the receive queue and configure the E1000 by following the process in section 14.4. You don't have to support "long packets" or multicast. For now, don't configure the card to use interrupts; you can change that later if you decide to use receive interrupts. Also, configure the E1000 to strip the Ethernet CRC, since the grade script expects it to be stripped.
>
> By default, the card will filter out *all* packets. You have to configure the Receive Address Registers (RAL and RAH) with the card's own MAC address in order to accept packets addressed to that card. You can simply hard-code QEMU's default MAC address of 52:54:00:12:34:56 (we already hard-code this in lwIP, so doing it here too doesn't make things any worse). Be very careful with the byte order; MAC addresses are written from lowest-order byte to highest-order byte, so 52:54:00:12 are the low-order 32 bits of the MAC address and 34:56 are the high-order 16 bits.
>
> The E1000 only supports a specific set of receive buffer sizes (given in the description of RCTL.BSIZE in 13.4.22). If you make your receive packet buffers large enough and disable long packets, you won't have to worry about packets spanning multiple receive buffers. Also, remember that, just like for transmit, the receive queue and the packet buffers must be contiguous in physical memory.
>
> You should use at least 128 receive descriptors

You can do a basic test of receive functionality now, even without writing the code to receive packets. Run `make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-net_testinput`. `testinput` will transmit an ARP (Address Resolution Protocol) announcement packet (using your packet transmitting system call), which QEMU will automatically reply to. Even though your driver can't receive this reply yet, you should see a "e1000: unicast match[0]: 52:54:00:12:34:56" message, indicating that a packet was received by the E1000 and matched the configured receive filter. If you see a "e1000: unicast mismatch: 52:54:00:12:34:56" message instead, the E1000 filtered out the packet, which means you probably didn't configure RAL and RAH correctly. Make sure you got the byte ordering right and didn't forget to set the "Address Valid" bit in RAH. If you don't get any "e1000" messages, you probably didn't enable receive correctly.

Now you're ready to implement receiving packets. To receive a packet, your driver will have to keep track of which descriptor it expects to hold the next received packet (hint: depending on your design, there's probably already a register in the E1000 keeping track of this). Similar to transmit, the documentation states that the RDH register cannot be reliably read from software, so in order to determine if a packet has been delivered to this descriptor's packet buffer, you'll have to read the DD status bit in the descriptor. If the DD bit is set, you can copy the packet data out of that descriptor's packet buffer and then tell the card that the descriptor is free by updating the queue's tail index, RDT.

If the DD bit isn't set, then no packet has been received. This is the receive-side equivalent of when the transmit queue was full, and there are several things you can do in this situation. You can simply return a "try again" error and require the caller to retry. While this approach works well for full transmit queues because that's a transient condition, it is less justifiable for empty receive queues because the receive queue may remain empty for long stretches of time. A second approach is to suspend the calling environment until there are packets in the receive queue to process. This tactic is very similar to `sys_ipc_recv`. Just like in the IPC case, since we have only one kernel stack per CPU, as soon as we leave the kernel the state on the stack will be lost. We need to set a flag indicating that an environment has been suspended by receive queue underflow and record the system call arguments. The drawback of this approach is complexity: the E1000 must be instructed to generate receive interrupts and the driver must handle them in order to resume the environment blocked waiting for a packet.

> **Exercise 11.** Write a function to receive a packet from the E1000 and expose it to user space by adding a system call. Make sure you handle the receive queue being empty.

> *Challenge!* If the transmit queue is full or the receive queue is empty, the environment and your driver may spend a significant amount of CPU cycles polling, waiting for a descriptor. The E1000 can generate an interrupt once it is finished with a transmit or receive descriptor, avoiding the need for polling. Modify your driver so that processing the both the transmit and receive queues is interrupt driven instead of polling.
>
> Note that, once an interrupt is asserted, it will remain asserted until the driver clears the interrupt. In your interrupt handler make sure to clear the interrupt as soon as you handle it. If you don't, after returning from your interrupt handler, the CPU will jump back into it again. In addition to clearing the interrupts on the E1000 card, interrupts also need to be cleared on the LAPIC. Use `lapic_eoi` to do so.

Receiving Packets: Network Server

In the network server input environment, you will need to use your new receive system call to receive packets and pass them to the core network server environment using the `NSREQ_INPUT` IPC message. These IPC input message should have a page attached with a `union Nsipc` with its `struct jif_pkt pkt` field filled in with the packet received from the network.

> **Exercise 12.** Implement `net/input.c`.

Run `testinput` again with `make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-net_testinput`. You should see

```
Sending ARP announcement...
Waiting for packets...
e1000: index 0: 0x26dea0 : 900002a 0
e1000: unicast match[0]: 52:54:00:12:34:56
input: 0000   5254 0012 3456 5255  0a00 0202 0806 0001
input: 0010   0800 0604 0002 5255  0a00 0202 0a00 0202
input: 0020   5254 0012 3456 0a00  020f 0000 0000 0000
input: 0030   0000 0000 0000 0000  0000 0000 0000 0000
```

The lines beginning with "input:" are a hexdump of QEMU's ARP reply.

Your code should pass the `testinput` tests of `make grade`. Note that there's no way to test packet receiving without sending at least one ARP packet to inform QEMU of JOS' IP address, so bugs in your transmitting code can cause this test to fail.

To more thoroughly test your networking code, we have provided a daemon called `echosrv` that sets up an echo server running on port 7 that will echo back anything sent over a TCP connection. Use `make E1000_DEBUG=TX,TXERR,RX,RXERR,RXFILTER run-echosrv` to start the echo server in one terminal and `make nc-7` in another to connect to it. Every line you type should be echoed back by the server. Every time the emulated E1000 receives a packet, QEMU should print something like the following to the console:

```
e1000: unicast match[0]: 52:54:00:12:34:56
e1000: index 2: 0x26ea7c : 9000036 0
e1000: index 3: 0x26f06a : 9000039 0
e1000: unicast match[0]: 52:54:00:12:34:56
```

At this point, you should also be able to pass the `echosrv` test.

> **Question**
>
> 2. How did you structure your receive implementation? In particular, what do you do if the receive queue is empty and a user environment requests the next incoming packet?

> *Challenge!* Read about the EEPROM in the developer's manual and write the code to load the E1000's MAC address out of the EEPROM. Currently, QEMU's default MAC address is hard-coded into both your receive initialization and lwIP. Fix your initialization to use the MAC address you read from the EEPROM, add a system call to pass the MAC address to lwIP,

and modify lwIP to the MAC address read from the card. Test your change by configuring QEMU to use a different MAC address.

Challenge! Modify your E1000 driver to be "zero copy." Currently, packet data has to be copied from user-space buffers to transmit packet buffers and from receive packet buffers back to user-space buffers. A zero copy driver avoids this by having user space and the E1000 share packet buffer memory directly. There are many different approaches to this, including mapping the kernel-allocated structures into user space or passing user-provided buffers directly to the E1000. Regardless of your approach, be careful how you reuse buffers so that you don't introduce races between user-space code and the E1000.

Challenge! Take the zero copy concept all the way into lwIP.

A typical packet is composed of many headers. The user sends data to be transmitted to lwIP in one buffer. The TCP layer wants to add a TCP header, the IP layer an IP header and the MAC layer an Ethernet header. Even though there are many parts to a packet, right now the parts need to be joined together so that the device driver can send the final packet.

The E1000's transmit descriptor design is well-suited to collecting pieces of a packet scattered throughout memory, like the packet fragments created inside lwIP. If you enqueue multiple transmit descriptors, but only set the EOP command bit on the last one, then the E1000 will internally concatenate the packet buffers from these descriptors and only transmit the concatenated buffer when it reaches the EOP-marked descriptor. As a result, the individual packet pieces never need to be joined together in memory.

Change your driver to be able to send packets composed of many buffers without copying and modify lwIP to avoid merging the packet pieces as it does right now.

Challenge! Augment your system call interface to service more than one user environment. This will prove useful if there are multiple network stacks (and multiple network servers) each with their own IP address running in user mode. The receive system call will need to decide to which environment it needs to forward each incoming packet.

Note that the current interface cannot tell the difference between two packets and if multiple environments call the packet receive system call, each respective environment will get a subset of the incoming packets and that subset may include packets that are not destined to the calling environment.

Sections 2.2 and 3 in this Exokernel paper have an in-depth explanation of the problem and a method of addressing it in a kernel like JOS. Use the paper to help you get a grip on the problem, chances are you do not need a solution as complex as presented in the paper.

The Web Server

A web server in its simplest form sends the contents of a file to the requesting client. We have provided skeleton code for a very simple web server in `user/httpd.c`. The skeleton code deals with incoming connections and parses the headers.

> **Exercise 13.** The web server is missing the code that deals with sending the contents of a file back to the client. Finish the web server by implementing `send_file` and `send_data`.

Once you've finished the web server, start the webserver (`make run-httpd-nox`) and point your favorite browser at http://*host*:*port*/index.html, where *host* is the name of the computer running QEMU (If you're running QEMU on athena use `hostname.mit.edu` (hostname is the output of the `hostname` command on athena, or `localhost` if you're running the web browser and QEMU on the same computer) and *port* is the port number reported for the web server by `make which-ports` . You should see a web page served by the HTTP server running inside JOS.

At this point, you should score 105/105 on `make grade`.

> *Challenge!* Add a simple chat server to JOS, where multiple people can connect to the server and anything that any user types is transmitted to the other users. To do this, you will have to find a way to communicate with multiple sockets at once *and* to send and receive on the same socket at the same time. There are multiple ways to go about this. lwIP provides a MSG_DONTWAIT flag for recv (see `lwip_recvfrom` in `net/lwip/api/sockets.c`), so you could constantly loop through all open sockets, polling them for data. Note that, while `recv` flags are supported by the network server IPC, they aren't accessible via the regular `read` function, so you'll need a way to pass the flags. A more efficient approach is to start one or more environments for each connection and to use IPC to coordinate them. Conveniently, the lwIP socket ID found in the struct Fd for a socket is global (not per-environment), so, for example, the child of a `fork` inherits its parents sockets. Or, an environment can even send on another environment's socket simply by constructing an Fd containing the right socket ID.

> **Question**
>
> 3. What does the web page served by JOS's web server say?
> 4. How long approximately did it take you to do this lab?

This completes the lab. As usual, don't forget to run `make grade` and to write up your answers and a description of your challenge exercise solution. Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab6.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 6'`, then `make handin` and follow the directions.

Lab 7: Final JOS project

Piazza Discussion Due, November 2, 2018
Proposals Due, November 8, 2018
Code repository Due, December 6, 2018
Check-off and in-class demos, Week of December 10, 2018

Introduction

For the final project you have two options:

- Work on your own and do lab 6, including one challenge exercise in lab 6. (You are free, of course, to extend lab 6, or any part of JOS, further in interesting ways, but it isn't required.)
- Work in a team of one, two or three, on a project of your choice that involves your JOS. This project must be of the same scope as lab 6 or larger (if you are working in a team).

The goal is to have fun and explore more advanced O/S topics; you don't have to do novel research.

If you are doing your own project, we'll grade you on how much you got working, how elegant your design is, how well you can explain it, and how interesting and creative your solution is. We do realize that time is limited, so we don't expect you to re-write Linux by the end of the semester. Try to make sure your goals are reasonable; perhaps set a minimum goal that's definitely achievable (e.g., something of the scale of lab 6) and a more ambitious goal if things go well.

If you are doing lab 6, we will grade you on whether you pass the tests and the challenge exercise.

Deliverables

Nov 3: Piazza discussion and form groups of 1, 2, or 3 (depending on which final project option you are choosing). Use the lab7 tag/folder on Piazza. Discuss ideas with others in comments on their Piazza posting. Use these postings to help find other students interested in similar ideas for forming a group. Course staff will provide feedback on project ideas on Piazza; if you'd like more detailed feedback, come chat with us in person.

Nov 9: Submit a proposal at the submission website, just a paragraph or two. The proposal should include your group members list, the problem you want to address, how you plan to address it, and what

are you proposing to specifically design and implement. (If you are doing lab 6, there is nothing to do for this deliverable.)

Dec 7: submit source code along with a brief write-up. Put the write-up under the top-level source directory with the name "`README.pdf`". Since some of you will be working in groups for this lab assignment, you may want to use git to share your project code between group members. You will need to decide on whose source code you will use as a starting point for your group project. Make sure to create a branch for your final project, and name it **lab7**. (If you do lab 6, follow the lab 6 submission instructions.)

Week of Dec 11: short in-class demonstration. Prepare a short in-class demo of your JOS project. We will provide a projector that you can use to demonstrate your project. Depending on the number of groups and the kinds of projects that each group chooses, we may decide to limit the total number of presentations, and some groups might end up not presenting in class.

Week of Dec 11: check-off with TAs. Demo your project to the TAs so that we can ask you some questions and find out in more detail what you did.

Project ideas

If you are not doing lab 6, here's a list of ideas to get you started thinking. But, you should feel free to pursue your own ideas. Some of the ideas are starting points and by themselves not of the scope of lab 6, and others are likely to be much of larger scope.

- Build a virtual machine monitor that can run multiple guests (for example, multiple instances of JOS), using x86 VM support.
- Do something useful with the hardware protection of Intel SGX. Here is a recent paper using Intel SGX.
- Make the JOS file system support writing, file creation, logging for durability, etc., perhaps taking ideas from Linux EXT3.
- Use file system ideas from Soft updates, WAFL, ZFS, or another advanced file system.
- Add snapshots to a file system, so that a user can look at the file system as it appeared at various points in the past. You'll probably want to use some kind of copy-on-write for disk storage to keep space consumption down.
- Build a distributed shared memory (DSM) system, so that you can run multi-threaded shared memory parallel programs on a cluster of machines, using paging to give the appearance of real shared memory. When a thread tries to access a

page that's on another machine, the page fault will give the DSM system a chance to fetch the page over the network from whatever machine currently stores it.
- Allow processes to migrate from one machine to another over the network. You'll need to do something about the various pieces of a process's state, but since much state in JOS is in user-space it may be easier than process migration on Linux.
- Implement paging to disk in JOS, so that processes can be bigger than RAM. Extend your pager with swapping.
- Implement mmap() of files for JOS.
- Use xfi to sandbox code within a process.
- Support x86 2MB or 4MB pages.
- Modify JOS to have kernel-supported threads inside processes. See in-class uthread assignment to get started. Implementing scheduler activations would be one way to do this project.
- Use fine-grained locking or lock-free concurrency in JOS in the kernel or in the file server (after making it multithreaded). The linux kernel uses read copy update to be able to perform read operations without holding locks. Explore RCU by implementing it in JOS and use it to support a name cache with lock-free reads.
- Implement ideas from the Exokernel papers, for example the packet filter.
- Make JOS have soft real-time behavior. You will have to identify some application for which this is useful.
- Make JOS run on 64-bit CPUs. This includes redoing the virtual memory system to use 4-level pages tables. See reference page for some documentation.
- Port JOS to a different microprocessor. The osdev wiki may be helpful.
- A window system for JOS, including graphics driver and mouse. See reference page for some documentation. sqrt(x) is an example JOS window system (and writeup).
- Implement dune to export privileged hardware instructions to user-space applications in JOS.
- Write a user-level debugger; add strace-like functionality; hardware register profiling (e.g. Oprofile); call-traces
- Binary emulation for (static) Linux executables