OS-344 Assignment-1

Instructions

- **Assignment has to be done by a groups of 4 members.**
- **Group members should ensure that one member submits the completed assignment within the deadline.**
- **Each group should submit a 1-2-page report, with relevant screenshots/ necessary data and findings related to the assignments.**
- **We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through a software and plagiarised submissions will be penalised heavily, irrespective of the source and copy.**
- **There will be a viva associated with each assignment. Attendance of all group members is mandatory.**
- **Assignment code, report and viva all will be considered for grading.**

Introduction

This assignment 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 kernel. Finally, you will add a system call in a toy operating system named xv6, which fills a buffer with an ASCII art drawing of Wolfie.

Part1: PC BootStrap

The purpose of this 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 do 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.

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.

Become familiar with inline assembly by writing a simple program. Modify the program ex1.c (at end of this file) to include inline assembly that increments the value of x by 1.

*You should read the recommended chapters of the PC Assembly book and "The Syntax" section in Brennan's Guide. 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 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 this course, 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, you have to first built a toolchain to use QEMU and GNU debugger and then clone the xv6 OS and as following:

For Linux users:

$ sudo apt-get update *//updates package information*

$ sudo apt-get install build-essential *//installs GNU tools*

$ sudo apt-get install qemu *//installs QEMU*

$ git clone git://github.com/mit-pdos/xv6-public.git *//clones xv6*

For Windows users:

1. Install Linux subsystem for Windows.
2. After successful installation, open a “bash on Ubuntu VM”.
3. Follow the same steps for Linux installation after this.

Now you're ready to boot xv6 OS with QEMU, but it requires an image file of the OS that you want to boot. So you need to issue ***make*** *inside xv6 directory* to compile the OS. This will supply the file **xv6.img**, as the contents of the emulated PC's first "virtual hard disk". This hard disk image contains both our boot loader (bootblock), and our kernel (kernel). Now to boot you can issue following command inside xv6 directory:

$ make qemu *//boot OS*

This executes QEMU with the options required to set the hard disk and direct serial port output to the terminal. (You could also use **make qemu-nox** to run QEMU in the current terminal instead of opening a new one.)

Everything after 'Booting from Hard Disk...' was printed by xv6 kernel; the $ is the prompt printed by the small *shell* included in our OS. These lines printed by the kernel will also appear in the regular shell window from which you ran QEMU. This is because we have set up the xv6 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 outputs to its own standard output because of the -serial argument. Likewise, the xv6 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.

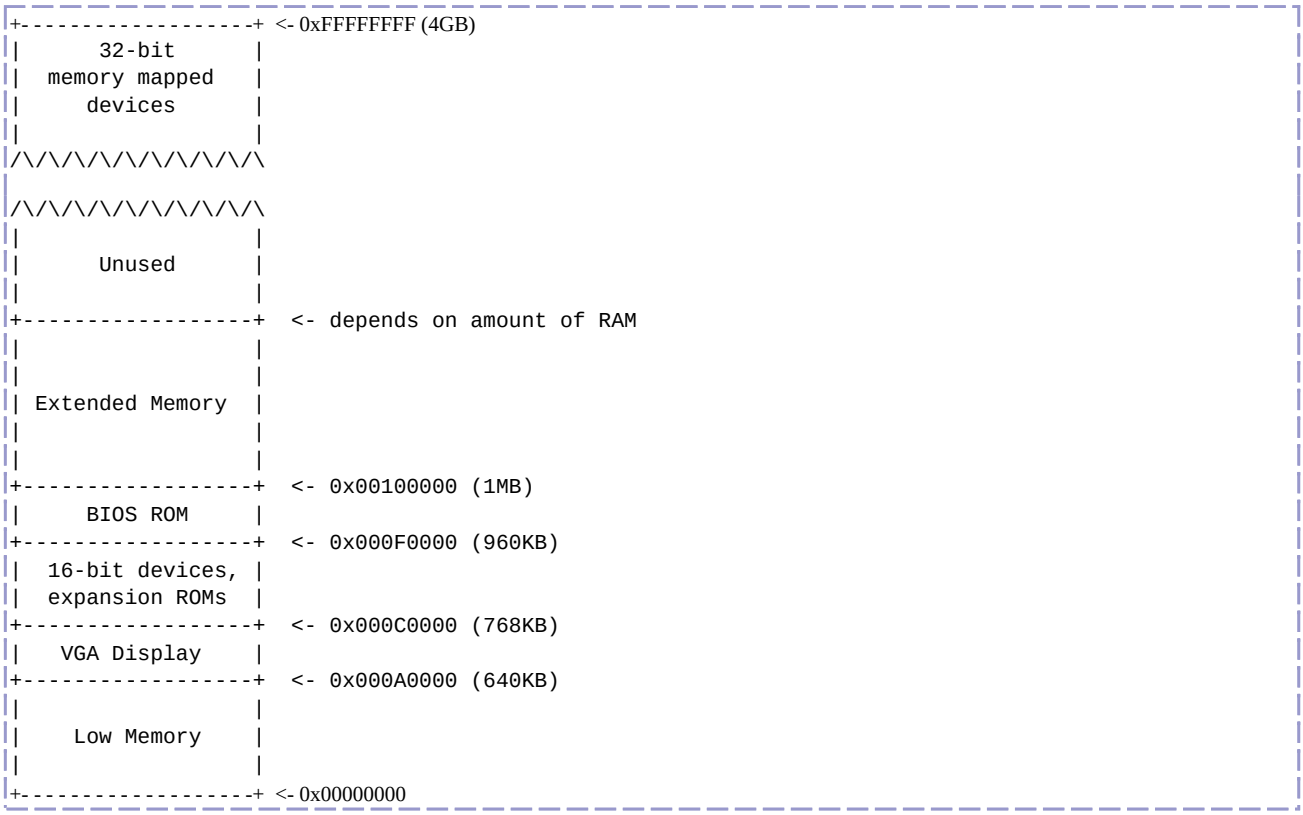
If you type **ls** at the $ prompt, it will list all the files that are included in the file system of our xv6 OS. These files are included in the second virtual hard disk created when you made xv6, namely, **fs.img**. Similarly, you can execute the other programs included in fs.img by typing their names at the command prompt. Of course, we could have combined fs.img and xv6.img into one disk image but we've decided to keep them separate for simplicity.

It is important to note that our OS is running "directly" on the "raw (virtual) hardware" of the simulated PC. This means that you should be able to copy the contents of xv6.img onto the first few sectors of a *real* hard disk, and the contents of fs.img onto another hard disk, insert the disks 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.

NB: We don't recommend you do this on a real machine with useful information on its hard disk, though, because copying the images 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

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. xv6 is a 32-bit operating system so we need not worry about this issue.

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. 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 **gdb**, and type **source .gdbinit** at the prompt. You should see something like this:

```
$ gdb
GNU gdb (GDB) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
...
(gdb) source .gdbinit
+ target remote localhost:26000
...
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
```

A .gdbinit script is provided that sets up GDB to debug the 16-bit code used during early boot and directed it to attach to the listening QEMU. It should be loaded before you start debugging the kernel. You can do this by either using GDB's source command every time you start GDB, or by adding it to the auto load path using its **add-auto-load-safe-pathcommand**.

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.

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

Part2: 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 this course, however, we will use the conventional hard drive boot mechanism, which means that the BIOS will only load the first 512 bytes. The boot loader (bootasm.S and bootmain.c) does the bootstrapping work. Look through these source files carefully and make sure you understand what's going on. The boot loader must perform the following main functions:

1. After some simple initialization, the bootloader has to configure the A20 address line. This is one of the arcane x86 features the x86 architecture. If you are interested to understand it better, skim this article .
2. The boot loader then 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.
3. The boot loader sets up the stack and starts executing the C code in bootmain.c.
4. Finally, 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.

After you understand the boot loader source code, look at the files bootblock.asm. This file is a disassembly of the boot loader that our Makefile 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.

You can set address breakpoints in GDB with the **b** command. You have to start hex numbers with 0x, so say something like **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/*N*i *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(refer last page of assignment) 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 break point. Trace through the code in bootasm.S, using the source code and the disassembly file bootblock.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 bootblock.asm and GDB.

Trace into bootmain() in bootmain.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.

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?
- 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 bootmain.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. 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 lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

We also recommend reading the Ksplice pointer challenge as a way to test that you understand how pointer arithmetic and arrays work in C.

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.

To make sense out of bootmain.c you'll need to know what an ELF binary is. When you compile and link a C program such as the xv6 kernel, the compiler transforms each C source ('.c') file into an *object* ('.o') file containing processor 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 kernel, which in this case is a binary in the ELF format, which stands for "Executable and Linkable Format".

For our purposes, 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 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.

You can display a full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

$ **objdump -h kernel**

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. In the ELF object, this is stored in the ph->p_pa field (in this case, it really is a physical address, though the ELF specification is vague on the actual meaning of this field).

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 this course.)

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

$ **objdump -h bootblock.o**

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 Makefile, 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 Makefile 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 afterwards!**

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 MB), 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:

$ **objdump -f kernel**

You should now be able to understand the minimal ELF loader in bootmain.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/*N*x *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).

Restart 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.)

Link vs Load Address

The *load address* of a binary is the memory address at which a binary is *actually* loaded. For example, the BIOS is loaded by the PC hardware at address 0xf0000. So this is the BIOS's load address. Similarly, the BIOS loads the boot sector at address 0x7c00. So this is the boot sector's load address.

The *link address* of a binary is the memory address for which the binary is linked. Linking a binary for a given link address prepares it to be loaded at that address. The linker encodes the link address in the binary in various ways, for example when the code needs the address of a global variable, with the result that a binary usually won't work if it is not loaded at the address that it is linked for.

In one sentence: the link address is the location where a binary *assumes* it is going to be loaded, while the load address is the location where a binary *is* loaded. It's up to us to make sure that they turn out to be the same.

Look at the -Ttext linker option in Makefile, and at the address mentioned early in the linker script in kernel.ld. These set the link address for the boot loader and kernel respectively.

When object code contains no absolute addresses that encode the link address in this fashion, we say that the code is *position-independent*: it will behave correctly no matter where it is loaded. GCC can generate position-independent code using the -fpic option, and this feature is used extensively in modern shared libraries that use the ELF executable format. Position independence typically has some performance cost, however, because it restricts the ways in which the compiler may choose instructions to access the program's data. We will not use -fpic in this course.

Part3: Adding a System Call

Now that we know how the kernel is compiled and loaded and gained basic familiarity with QEMU and GDB, it's time to get our hands dirty with some kernel coding. Your task is to add a new system call to xv6. It will help to start by reading syscall.c (the kernel side of the system call table), user.h(the user-level header for the system calls), and usys.S (the user-level system call definitions). You may add additional files to xv6 to implement this call.

Exercise 7.

Create a system call int sys_wolfie(void *buf, uint size), which copies an ASCII art image (Use a buffer of a wolf picture, google it for more information) of Wolfie to a user-supplied buffer, provided that the buffer is large enough. You are welcome to use an ASCII art generator, or draw your own by hand.

If the buffer is too small, or not valid, return a negative value. If the call succeeds, return the number of bytes copied.

You may find it helpful to review how other system calls are implemented and compiled into the kernel, such as read.

Exercise 8.

Write a user-level application, called wolfietest.c, that gets the Wolfie image from the kernel, and prints it to the console.

When the OS runs, your program's binary should be included in fs.img and listed if someone runs **ls** at the xv6 shell's command prompt. Study Makefile to figure out how to compile a user-mode program and add it tofs.img.

***This completes the lab assignment.**

ex1.c

```
// Simple inline assembly example
//

#include <stdio.h>

int
main(int argc, char **argv)
{
  int x = 1;
  printf("Hello x = %d\n", x);

  //
  // Put in-line assembly here to increment
  // the value of x by 1 using in-line assembly
  //

  printf("Hello x = %d after increment\n", x);

  if(x == 2){
    printf("OK\n");
  }
  else{
    printf("ERROR\n");
  }
}
```

GDB Tool guide

See the GDB manual for a full guide to GDB commands. Here are some particularly useful commands for this course, some of which don't typically come up outside of OS development.

Ctrl-c
: Halt the machine and break in to GDB at the current instruction. If QEMU has multiple virtual CPUs, this halts all of them.

c (or **continue**)
: Continue execution until the next breakpoint or Ctrl-c.

si (or **stepi**)
: Execute one machine instruction.

b function or **b file:line** (or **breakpoint**)
: Set a breakpoint at the given function or line.

b ****addr* (or **breakpoint)
: Set a breakpoint at the EIP *addr*.

set print pretty
: Enable pretty-printing of arrays and structs.

info registers
: Print the general purpose registers, eip, eflags, and the segment selectors. For a much more thorough dump of the machine register state, see QEMU's own info registers command.

x/*N***x** ***addr***
: Display a hex dump of *N* words starting at virtual address *addr*. If *N* is omitted, it defaults to 1. *addr* can be any expression.

x/*N***i** ***addr***
: Display the *N* assembly instructions starting at *addr*. Using $eip as *addr* will display the instructions at the current instruction pointer.

Symbol-file ***file***
: (Lab 3+) Switch to symbol file f*ile*. When GDB attaches to QEMU, it has no notion of the process boundaries within the virtual machine, so we have to tell it which symbols to use. By default, we configure GDB to use the kernel symbol file, obj/kern/kernel. If the machine is running user code, say hello.c, you can switch to the hello symbol file using symbol-file obj/user/hello.