Homework 2: Make QEMU, boot xv6, understand page tables

Compile and Install QEMU

Xv6 is a real operating system kernel, and hence, it needs real hardware to boot. Fortunatelly, today we can emulate hardware in software. Programs like QEMU can emulate functionality of the real physical CPU in software. I.e., QEMU implements the normal CPU loop similar to the one we discussed in class: fetches an instruction pointed by the instruction pointer register (EIP), decodes it, performs all permission and condition checks, computes the outcome, increments EIP and continues to the next instruction. Like a real PC platform, QEMU emulates hardware boot protocol. QEMU starts by loading the disk sector number 0 into the memory location 0x7c00 and jumping to it. Xv6 takes it from there. At a high level, for xv6 it does not matter if it runs on the real hardware or under QEMU. Of course, emulation is slower than real hardware, but besides that as long as QEMU implements the logic of the CPU correctly we do not see any deviations from a real baremetal execution. Surprisingly, QEMU is reasonably fast, so you as a human barely notice the difference.

To run xv6 we need to compile and install a version of the QEMU emulator. Due to some compatibility issues it is impossible to compile QEMU directly on the openlab machines. Instead, we will work inside yet another virtual machine called Vagrant. Vagrant will start on the openlab machines, and will boot into a version of the Ubuntu Linux system. Currently vagrant is only installed on two openlab machines: **odin.ics.uci.edu** and **tristram.ics.uci.edu**.

To start working on this homework, login to either odin.ics.uci.edu or tristram.ics.uci.edu:

```
$ ssh odin.ics.uci.edu
```

I suggest you create a new folder for your ics143a homeworks, like:

```
odin$mkdir ics143a
```

Change into that directory:

```
odin$cd ics143a
```

Fetch a version of the vagrant environment that explains to vagrant what kind of virtual machine you're planning to run:

```
odin$ wget http://www.ics.uci.edu/~aburtsev/143A/hw/xv6-vagrant-master.tgz
odin$ tar -xzvf xv6-vagrant-master.tgz
```

Change into the new folder

```
odin$ cd xv6-vagrant-master
```

Change the name of the vagrant VM to something unique (otherwise we all end up with the same VM and vagrant is confused). In the `Vagrantfile` file change the following line

```
      vb.name = "xv6_box_anton" # <--- You should change this to make VM names unique
```

Start vagrant VM (this will take several minutes as it is building QEMU inside)

```
odin$ vagrant up
```

If vagrant fails with the following message:

```
==> default: Clearing any previously set forwarded ports...
Vagrant cannot forward the specified ports on this VM, since they
would collide with some other application that is already listening
on these ports. The forwarded port to 20000 is already in use
on the host machine.

To fix this, modify your current project's Vagrantfile to use another
port. Example, where '1234' would be replaced by a unique host port:

  config.vm.network :forwarded_port, guest: 26001, host: 1234
```

Go ahead with the suggested fix. Change the following line in the Vagrantfile setting the host port to something random below 64000:

```
  config.vm.network "forwarded_port", guest: 26001, host: 30000
```

If vagrant VM is up, you're ready to log in inside and start working on your xv6 Linux environment. Log in inside the vagrant VM. From the same folder where Vagrantfile is (i.e., from ics143a/xv6-vagrant-master) type

```
odin$ vagrant ssh
```

Now you're inside the Linux Ubuntu 12.04.5 LTS. Your new vagrant machine should have everything you need to compile and run your xv6 code. Vagrant automatically shares the directory of your host machine (i.e., odin.ics.uci.edu) where the `Vagrantfile` is as the `/vagrant` directory of the vagrant VM.

Note

While I provide instructions for how to use openlab machines (`odin` and `tristram`), you are more than welcome to configure and run xv6 on your own laptop, desktop, or VM. If you decide to use your own environment, see the instructions on the xv6 tools page for how to set up xv6. I've successfully built xv6 on my Ubuntu 14.04 LTS. I had to install the following packages in order to build QEMU: libz-dev, libtool-bin, libtool, libglib2.0-dev, libpixman-1-dev, libfdt-dev.

Boot xv6

From inside your Vagrant VM fetch the xv6 source:

```
vagrant@odin$ cd /vagrant
vagrant@odin$ mkdir ics143a
vagrant@odin$ cd ics143a
vagrant@odin$ git clone git://github.com/mit-pdos/xv6-public.git
Cloning into xv6...
...
```

Build xv6:

```
vagrant@odin$ cd xv6-public
vagrant@odin$ make
...
gcc -O -nostdinc -I. -c bootmain.c
gcc -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
```

```
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
...
vagrant@odin$
```

You're now ready to start working on the assignment.

Exercise 1: Finding and breaking at an address

Find the address of `_start`, the entry point of the kernel:

```
vagrant@odin$ nm kernel | grep _start
8010b50c D _binary_entryother_start
8010b4e0 D _binary_initcode_start
0010000c T _start
vagrant@odin$
```

In this case, the address is `0010000c`.

Run the kernel inside QEMU GDB, setting a breakpoint at `_start` (i.e., the address you just found).

```
vagrant@odin$ make qemu-nox-gdb
...
```

Now open another terminal (**you do that on your openlab host machine, i.e., odin or tristram, whichever you're using**). Change to the folder where Vagrant is configured, and connect to the same Vagrant VM, i.e.:

```
odin$ cd ~/ics143a/xv6-vagrant-master/
odin$ vagrant ssh
```

In this new terminal (**but now inside your vagrant VM**) change to the folder where you've built xv6, and start GDB:

```
vagrant@odin$ cd /vagrant/ics143a/xv6-public
vagrant@odin$ gdb
GNU gdb 6.8-debian
```

```
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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 "x86_64-linux-gnu".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
```

Set a breakpoint at the address of `_start`, e.g.

```
(gdb) br * 0x0010000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Breakpoint 1, 0x0010000c in ?? ()
(gdb)
```

The details of what you see are likely to differ from the above output.

Question 1: What is on the stack?

Look at the registers and the stack contents:

```
(gdb) info reg
...
(gdb) x/24x $esp
...
(gdb)
```

Write a short (3-5 word) comment next to each **zero and non-zero** value on the stack explaining what is. Which part of the stack printout is actually the stack? (Hint: not all of it.)

You might find it convenient to consult the files bootasm.S, bootmain.c, and bootblock.asm (which contains the output of the compiler/assembler). The reference page has pointers to x86 assembly documentation, if you are wondering about the semantics of a particular instruction. Here are some questions to help you along:

- Start by setting a break-point at 0x7c00, the start of the boot block (bootasm.S). Single step through the instructions (type si to the gdb prompt). Where in bootasm.S is the stack pointer initialized?
- Single step through the call to `bootmain`; what is on the stack now?
- What do the first assembly instructions of bootmain do to the stack? Look for bootmain in bootblock.asm.
- Look in bootmain in bootblock.asm for the call that changes `eip` to 0x10000c. What does that call do to the stack?

Tip

Since we're running QEMU in headless mode (`make clean qemu-nox`) you don't have a GUI window to close whenever you want. There are two ways to exit QEMU.

1. First, you can exit the xv6 QEMU session by killing the QEMU process from another terminal. A useful shortcut is to define an `alias` in your local machine as follows:
   ```
   alias kill-qemu='vagrant ssh -c "killall qemu-system-i386"'
   ```
   Add this to your `~/.bash_profile` or `~/.zshrc` file to make it persistant This will send the `killall qemu-system-i386` command over ssh to your vagrant machine. Notice this command will only work if you're running it from somewhere in the directory path of the Vagrantfile running this machine
2. Alternatively you can send a **Ctrl-a x** command to the QEMU emulator forcing it to exit (here is a list of QEMU shortcuts and commands).

Exercise 2: Understanding page tables

Question 1: Explain how virtual to physical address translation works

This question asks you to illustrate organization of the x86, 4K, 32bit page table through a simple example. Assume that the hardware translates the virtual address '0x402003' into the physical address '0x1003'. The physical addresses of the page table directory and the page table (Level 2) involved in the translation of this virtual address are 0x2000 and 0x0. Draw a diagram (hand drawn figures are sufficient, but need to be clear) representing the state of the page table in physical memory and the process of translation (similar to Figure 2-1 in the xv6 book but using concrete physical and virtual addresses and page table entries). Provide a short explanation. Use Chapter 2 of the xv6 book to review how page tables work.

Question 2: Why two-level page tables?

You probably wandered why the hardware uses two-level page table organization versus a simple array for translation of virtual addresses. After all, an array that contains physical addresses for each virtual address is sufficient. A virtual address can be an index into the array, and the translation can be done as `physical = page_table_array[virtual]`. To understand the benefits of two level page tables lets compare the size of two-level page tables versus such an array. First, find the size of the two-level page table that is required to translate 3 distinct pages of virtual memory (e.g., an address space of a simple process that consists of only 3 pages: a page for text/code, a page for data, and a page for stack). Compare the size of the page table with the size of the array implementation described above (assume that array entries are 4 bytes each, and are identical to page table entries (PTEs) of the second level of traditional two-level page tables). How does the size of the page table changes when you need to map the entire 4GB virtual address space?

Exercise 3: What is the state of page tables after xv6 is done initializing the first 4K page table?

During boot, xv6 calls the `kvmalloc()` function to set up the kernel page table (`kvmalloc()` is called from `main()` (lines 1316-1340)). What is the state of this first 4K page table after `kvmalloc()` returns, and what are the details of page table initialization? To understand the state of the page table, we first take a look at the implementation of the `kvmalloc` function and then inspect the actual page tables with the QEMU monitor.

Question 1: Explain implementation of `kvmalloc()` and related functions

Use xv6 source code to understand and explain implementation of the following functions: `setupkvm()`, `walkpgdir()`, `mappages()`, and their role in the implementation of `kvmalloc()`. Use line numbers from the source printout to refer to specific places of code. Provide a reasonably detailed description for individual lines of code to demonstrate your understanding of the code.

Question 2: What is the state of page tables after `kvmalloc()` returns?

Based on your analysis of the source code, and based on the inspection of the page tables with the QEMU monitor, explain the state of the page tables after `kvmalloc()` is done. Draw a figure of a page table, and briefly explain each page table entry.

To aid your analysis, inspect the actual page table with QEMU. QEMU includes a built-in monitor that can inspect and modify the machine state in useful ways. To enter the monitor, press **Ctrl-a c** in the terminal running QEMU. Press **Ctrl-a c** again to switch back to the serial console.

To inspect the page table, run xv6 to the point after it is done with `kvmalloc()`. Similar to Exercise 1, open two terminal windows. In one window start xv6 under control of the GDB debugger

```
vagrant@odin$make qemu-nox-gdb
```

In another window start GDB

```
vagrant@odin$ cd /vagrant/ics143a/xv6-public
vagrant@odin$ gdb
```

In the GDB window set breakpoint on `main` function, run continue (c), and run (n) executing functions inside `main` until you exit `kvmalloc()` (use "n" three times):

```
(gdb) b main
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80102eb0
:       push   %ebp

Breakpoint 1, main () at main.c:19
19      {
(gdb) n
=> 0x80102eba : movl   $0x80400000,0x4(%esp)
```

```
20          kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) n
=> 0x80102ece : call   0x80106790
21          kvmalloc();      // kernel page table
(gdb) n
=> 0x80102ed3 : call   0x80103070
22          mpinit();        // detect other processors
```

At this point switch back to the terminal running the QEMU monitor and type `info pg` to display the state of the active page table. You will see page directory entries and page table entries along with the permissions for each separately. Repeated PTE's and entire page tables are folded up into a single line. For example,

```
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     -------UWP
  [00200-00233]  PTE[200-233] -------U-P 00380 0037e 0037d 0037c 0037b 0037a ..
[00800-00bff]  PDE[002]     ----A--UWP
  [00800-00801]  PTE[000-001] ----A--U-P 0034b 00349
  [00802-00802]  PTE[002]     -------U-P 00348
```

This shows two page directory entries, spanning virtual addresses 0x00000000 to 0x003fffff and 0x00800000 to 0x00bfffff, respectively. Both PDE's are present, writable, and user and the second PDE is also accessed. The second of these page tables maps three pages, spanning virtual addresses 0x00800000 through 0x00802fff, of which the first two are present, user, and accessed and the third is only present and user. The first of these PTE's maps physical page 0x34b.

You can find more information about QEMU monitor and GDB debugger here, feel free to explore them.

Submit

Submit your answers to the Class Dropbox (143A HW 2: Boot xv6) (as a PDF file).

Updated: February, 2017