Home

Homework 2: Make QEMU, boot xv6, understand address translation

To start working on this homework follow the xv6 setup instructions. After you're done with them, you'll be 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:

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

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).

```
openlab$ make qemu-nox-gdb
...
```

Now open another terminal (**you do that on your openlab host machine, i.e., andromeda-XX, odin, or tristram, whichever you're using**). In this new terminal change to the folder where you've built xv6, and start GDB:

```
openlab$ cd /vagrant/ics143a/xv6-public
openlab$ 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
```

What you see on the screen is the assembly code of the BIOS that QEMU executes as part of the platform initialization. The BIOS starts at address `0xfff0` (you can read more about it in the How Does an Intel Processor Boot? blog post. You can single step through the BIOS machine code with the `si` (single instruction) GDB command if you like, but it's hard to make sense of what is going on so lets skip it for now and get to the point when QEMU starts executing the xv6 kernel.

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

```
 (gdb) br * 0x0010000c
Breakpoint 1 at 0x10000c
```

The details of what you see may differ slightly from the above output.

Troubleshooting GDB

```
andromeda-1:1001-/16:40>gdb
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-110.el7
Copyright (C) 2013 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-redhat-linux-gnu".
For bug reporting instructions, please see:
.
warning: File "/home/aburtsev/projects/cs143a/xv6-public/.gdbinit" auto-loading has been
declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load:/usr/bin/mono-gdb.py".
To enable execution of this file add
```

```
        add-auto-load-safe-path /home/aburtsev/projects/cs143a/xv6-public/.gdbinit
line to your configuration file "/home/aburtsev/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/aburtsev/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) quit
```

Add the line

```
add-auto-load-safe-path /home/aburtsev/projects/cs143a/xv6-public/.gdbinit
```

to

```
/home/aburtsev/.gdbinit
```

but of course replace "aburtsev" with your user name.

Task 1: Make yourself familiar with GDB

This part of the homework teaches you how to use GDB. If the xv6 and GDB are still running exit them. You can exit xv6 by terminating QEMU with **Ctrl-A X**. You can exit GDB by pressing **Ctrl-C** and then **Ctrl-D**.

Start the xv6 and gdb again as you did before (use two terminals one to start the xv6):

```
make qemu-nox-gdb
```

and another to start gdb

```
gdb
```

Now explore the other ways of setting breakpoints. Instead of

```
(gdb) br * 0x0010000c
```

You can use the name of the function or an assembly label, e.g., to set the breakpoint at the beginning of the `_start` label you can use:

```
(gdb) br _start
```

Similar you can set the breakpoint on the `main()` function.

```
(gdb) br main
```

If you need help with GDB commands, GDB can show you a list of all commands with

```
(gdb) help all
```

Now since you set two breakpoints you can continue execution of the system until one of them gets hit. In gdb enter the "c" (continue) command to run xv6 until it hits the first breakpoint (`_start`).

```
(gdb) c
```

Now use the `si` (step instruction) command to single step your execution (execute it one machine instruction at a time). Remember that the `_start` label is defined in the assembly file, `entry.S` to be the entry point for the kernel. Enter `si` a couple of times. Note, you don't have to enter `si` every time, if you just press "enter" the GDB will execute the last command.

```
(gdb) si
```

Every time you enter `si` it executes one machine instruction and shows you the next machine instruction so you know what is coming next

```
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
```

You can switch between ATT and Intel disassembly syntax with these commands:

```
(gdb) set disassembly-flavor intel
(gdb) set disassembly-flavor att
```

To make sure that you understand what is going on, open the `entry.S` file and look over it. Remember, that the `_start` label is defined in this file. And the instructions that you were executing in GDB come from exactly this file. Make sure that you understand what is happening there (we covered in the lecture), i.e., the kernel works on enabling the 4MB page tables, and create a stack for executing the C code inside `main()`.

You can either continue single stepping until you reach the `main()` function, or you can enter "c" to continue execution until the next breakpoint.

```
(gdb) c
Continuing.
=> 0x80102e60
:       lea    0x4(%esp),%ecx

Thread 1 hit Breakpoint 2, main () at main.c:19
```

Now you've reached the C code, and since we compiled it with the "-g" flag that includes the symbol information into the ELF file we can see the C source code that we're executing. Enter the `l` (list) command.

```
(gdb) l
14      // Bootstrap processor starts running C code here.
15      // Allocate a real stack and switch to it, first
16      // doing some setup required for memory allocator to work.
17      int
18      main(void)
19      {
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
21        kvmalloc();      // kernel page table
22        mpinit();        // detect other processors
23        lapicinit();     // interrupt controller
```

Remember that when you hit the `main` breakpoint the GDB showed you that you're at line 19 in the main.c file (`main.c:19`). This is where you are. You can either step into the functions with the `s` (step) command (note, in

contrast to the `si` step instruction command, this one will execute **one C line at a time**), or step over the functions with the `n` (next) command which will not enter the function, but instead will execute it till completion.

Try stepping into the `kinit1` function.

```
(gdb) s
```

Note, that on my machine when I enter `s` for the first time the GDB believes that I'm executing the `startothers()` function. It's a glitch---the compiler generated an incorrect debug symbol information and GDB is confused. If I hit `s` a couple of times I eventually get to `kinit1()`.

The whole listing of the source code seems a bit inconvenient (entering `l` every time you want to see the source line is a bit annoying). GDB provides a more conventional way of following the program execution with the TUI mechanism. Enable it with the following GDB command

```
(gdb) tui enable
```

Now you see the source code window and the machine instructions at the bottom. You can use the same commands to walk through your program. You can scroll the source with arrow keys, PgUp, and PgDown.

TUI can show you the state of the registers and how they are changing as you execute your code

```
(gdb) tui reg general
```

TUI is a very cute part of GDB and hence it makes sense to read more about various capabilities http://sourceware.org/gdb/onlinedocs/gdb/TUI-Commands.html. For example, you can specify the assembly layout to single step through machine instructions similar to source code:

```
(gdb) layout asm
```

For example, you can switch to the asm layout right after hitting the `_start` breakpoint.

Question 1: What is on the stack?

Restart your xv6 and gdb session. Set a breakpoint at the `_start` label.

```
(gdb) br _start
```

Continue execution until the breakpoint is hit. 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 of the printout explaining what it is. Which part of the 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 (this last file is contains assmbly generated by the compiler from both C and ASM files). A short x86 Assembly Guide and additional resources from the MIT reference page provide 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. Where in bootasm.S the stack pointer is 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='killall qemu-system-i386'
   ```

Add this to your `~/.bash_profile` or `~/.zshrc` file to make it persistant.

If you're still using Vagrant this becomes

```
alias kill-qemu='vagrant ssh -c "killall qemu-system-i386"'
```

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).

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

Exercise 2: Understanding address translation

Question 1: Explain how logical to physical address translation works

This question asks you to illustrate organization of the x86, 4K, 32bit segmentation and paging mechanisms through a simple example. Assume that the hardware translates the logical address '0x803004' that is involved into a memory access in the data segment into the physical address '0x8004'. The physical addresses of the page table directory (Level 1) and the page table (Level 2) involved in the translation of this virtual address are respectively 0x5000 and 0x8000. The entry 1 of the Global Descriptor Table (GDB) contains the base of 0x1000000 and the limit of 2GBs. The DS register contains the value 0x8. Draw a diagram (hand drawn figures are sufficient, but need to be clear) representing the state of the GDT, page table, and hardware CPU registers pointing to the GDT and page table directory 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.

Extra credit (5%): Can you explain the nature of the memory access in the question above?

Question 2: 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()`). 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. 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

```
openlab$make qemu-nox-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.

If you're using Vagrant, shut down your VM

After you're done please halt your VM so it does not run on the cluster forever. All the xv6 files will be there in the cs238p/xv6-vagrant-master folder when you start vagrant the next time.

To shut down your VM, when you close your ssh connection type

```
UCInetID@andromeda-XX$ vagrant halt
```

You will be able to start your Vagrant VM from the same folder (cs238p/xv6-vagrant-master) by typing

```
UCInetID@andromeda-XX$ vagrant up
UCInetID@andromeda-XX$ vagrant ssh
```

Submit

Submit your answers in a single PDF file named `hw2.pdf` on Canvas HW2 Boot xv6.

Updated: October, 2018