

# CS 444/544 Lab

## Lab Setup

You'll use two sets of tools in this class: an x86 emulator, QEMU, for running your kernel; and a compiler toolchain, including assembler, linker, C compiler, and debugger, for compiling and testing your kernel. This page has the information you'll need to do on our OS ([os1, os2, oldos1, oldos2].engr.oregonstate.edu) servers. This class assumes familiarity with Unix commands throughout.

For your convenience, the CS 444/544 staff has prepared a server environment that pre-configured all required toolchain, and you can get access to those toolchain by following the steps listed below.

### Login to the OS servers

Connect to any of OS servers that you wish to use.

On Linux/macOS, running the following command will let you connect to the server:

```
[host] $ ssh -X your_username@os2.engr.oregonstate.edu
```

e.g., my ONID username is songyip, then,

```
[host] $ ssh -X songyip@os2.engr.oregonstate.edu
```

On Windows, you may connect to server through your SSH client, i.e., MobaXterm or PuTTY.

You may also use Windows Subsystem for Linux (WSL), which can run a Linux distribution on top of Windows 10. You can find more information about installing and enabling WSL at [here](#) (We recommend installing Ubuntu).

After typing your password and passing the DUO two-factor authentication, you may get your command line. Alternatively, you may setup ssh public/private key pair to log on to the server w/o typing password. If you would like to do so, please follow the instruction at [here](#) or [here](#).

### Running the SETUP Script

Next, please run the setup script for the lab. The script is located at:

```
/nfs/farm/classes/eecs/spring2024/cs444-001/cs444-setup.py
```

Please run the script by running the following command:

```
[os2] $ /nfs/farm/classes/eecs/fall2023/cs444-010/cs444-setup.py
Cloning into '/nfs/stak/users/[your-username]/.cs444'...
remote: Enumerating objects: 48, done.
remote: Counting objects: 100% (48/48), done.
remote: Compressing objects: 100% (35/35), done.
remote: Total 395 (delta 22), reused 37 (delta 11), pack-reused 347
Receiving objects: 100% (395/395), 9.29 MiB | 0 bytes/s, done.
Resolving deltas: 100% (239/239), done.
```

```
Do you want to install peda to ~/.gdbinit (y/n) ?
```

```
n
```

```
Do you want to install cs444 custom tmux configuration (y/n) ?
```

```
n
```

```
Do you want to install .bashrc (y/n) ?
```

```
n
```

```
Do you want to install .vimrc and vim plugins (y/n) ?
```

```
n
```

The script will clone the prepared environmental scripts (.dotfiles) and setup gdb, tmux, bash, and vim.

If you wish to use the prepared dotfiles by CS 444/544 staff, then please type 'y' at each question from the script. In this case, all your existing dotfiles will be saved as .dotfile\_name.bak, e.g., .vimrc.bak or .bashrc.bak, in your home directory.

If you wish to keep your settings for any of them, please type 'n' for the corresponding question.

## CS 444/544 GitHub Classroom

We will use a GitHub Classroom to collect all your lab assignment submissions.

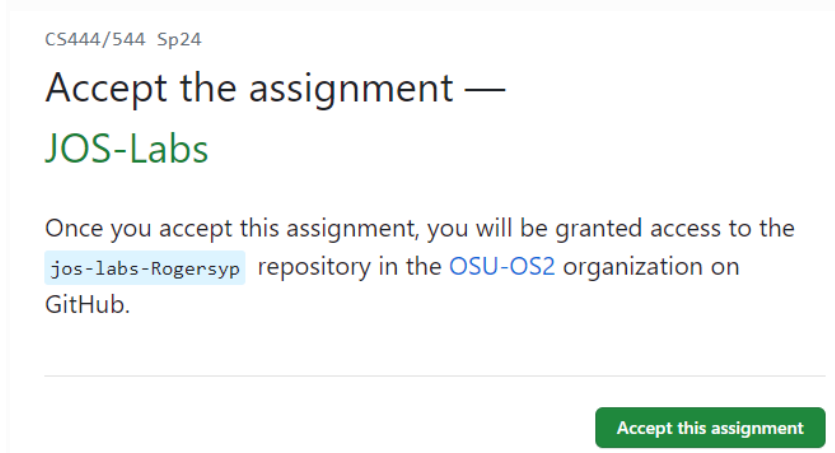
You will need an account for GitHub, if you don't have one, create your GitHub account via <https://github.com/join>.

After creating the GitHub account, please log-on to the GitHub classroom via <https://classroom.github.com/classrooms>

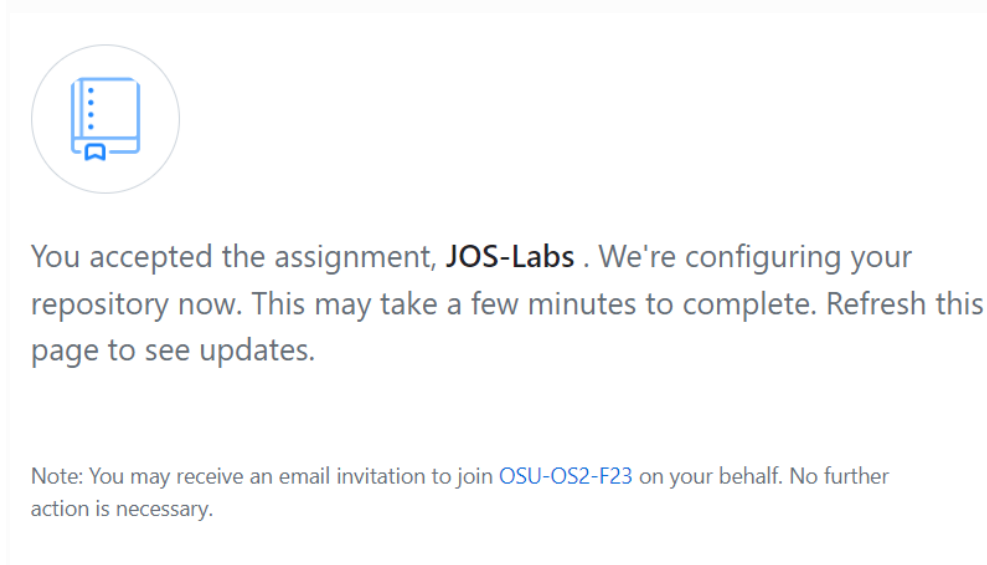


Use the invitation link on Canvas → Labs page, link your account to your OSU email address, then accept the JOS Lab assignment:

<https://classroom.github.com/a/6lknqiou>



Once you Click the “Accept this assignment”, you should see this page.



Refresh the page, you will see this:



## You're ready to go!

You accepted the assignment, **JOS-Labs**.

Your assignment repository has been created:

<https://github.com/OSU-OS2/jos-labs-Rogersyp>

We've configured the repository associated with this assignment ([update](#)).

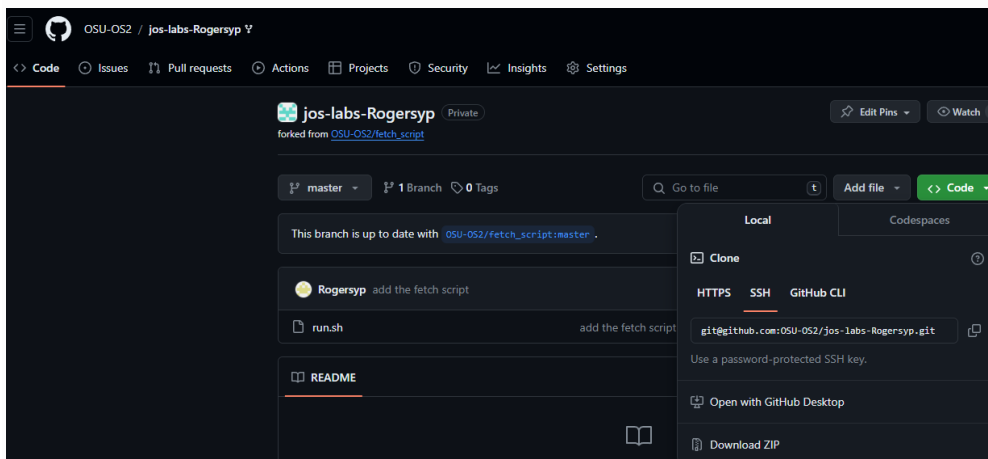
Note: You may receive an email invitation to join **OSU-OS2** on your behalf. No further action is necessary.

Click the link to go to your assignment repository.

For example: `https://github.com/OSU-OS2 /jos-labs-[your_github_username]`

## Cloning the labs

1. As of now, the repository contains nothing but a setup script. Click “Code” to copy the URL



*Note: Depend on your settings, you might choose either HTTPS or SSH (if you have your ssh keys set up) to clone the repo.*

On the OS server, clone the repo.

i.e., `git clone git@github.com:OSU-OS2/jos-labs-[your-username].git`

\*Note: you must change *your-id* part to your GitHub id.

```
[songyip@os2 ~/cs444/s24/test$] git clone git@github.com:OSU-OS2/jos-labs-Rogersyp.git
Cloning into 'jos-labs-Rogersyp'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
[songyip@os2 ~/cs444/s24/test$] ls
jos-labs-Rogersyp
```

2. Change directory into your jos lab repo

i.e., `cd jos-labs-[your-id]`

3. Make the script executable with the command

`chmod +x run.sh`

Then run it with the command

`./run.sh`

Note: if you see an error:

```
bash: ./run.sh: /bin/bash^M: bad interpreter: No such file or directory
```

This means your file has Windows line endings (i.e., has an additional carriage return ^M), which is confusing Linux. To fix it, open the file using vim on the server, and type the following in command mode:

```
:set ff=unix
```

```
:wq
```

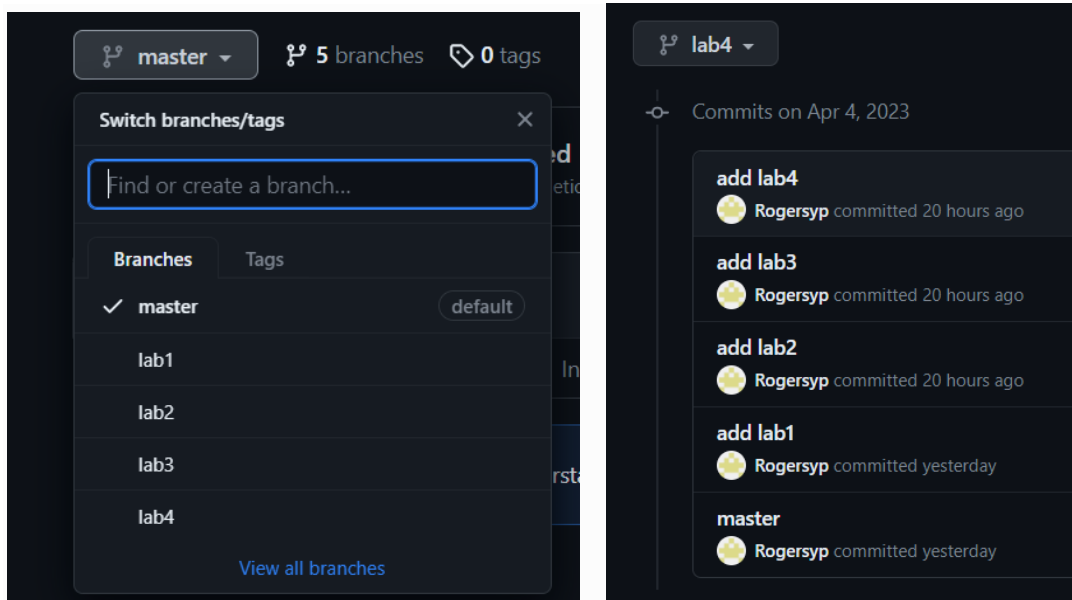
4. You should see something similar to the following if the script runs successfully:

```

[songyip@os2 (master) ~/cs444/s24/test/jos-labs-Rogersyp$] ./run.sh
Success. Remote origin URL: git@github.com:OSU-OS2/jos-labs-Rogersyp.git
warning: no common commits
remote: Enumerating objects: 205, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 205 (delta 2), reused 4 (delta 2), pack-reused 199
Receiving objects: 100% (205/205), 412.19 KiB | 0 bytes/s, done.
Resolving deltas: 100% (56/56), done.
From github.com:OSU-OS2-523/JOS-Labs
+ 7a49c6c...7f12943 master -> origin/master (forced update)
* [new branch] lab1 -> origin/lab1
* [new branch] lab2 -> origin/lab2
* [new branch] lab3 -> origin/lab3
* [new branch] lab4 -> origin/lab4
Already on 'master'
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)
Everything up-to-date
Branch lab4 set up to track remote branch lab4 from origin.
Switched to a new branch 'lab4'
Counting objects: 199, done.
Delta compression using up to 96 threads.
Compressing objects: 100% (137/137), done.
Writing objects: 100% (199/199), 411.07 KiB | 0 bytes/s, done.
Total 199 (delta 52), reused 198 (delta 52)
remote: Resolving deltas: 100% (52/52), done.
remote:
remote: Create a pull request for 'lab4' on GitHub by visiting:
remote: https://github.com/OSU-OS2/jos-labs-Rogersyp/pull/new/lab4
remote:
remote: To git@github.com:OSU-OS2/jos-labs-Rogersyp.git
* [new branch] lab4 -> lab4
Branch lab3 set up to track remote branch lab3 from origin.
Switched to a new branch 'lab3'
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'lab3' on GitHub by visiting:
remote: https://github.com/OSU-OS2/jos-labs-Rogersyp/pull/new/lab3
remote:
remote: To git@github.com:OSU-OS2/jos-labs-Rogersyp.git
* [new branch] lab3 -> lab3
Branch lab2 set up to track remote branch lab2 from origin.
Switched to a new branch 'lab2'
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'lab2' on GitHub by visiting:
remote: https://github.com/OSU-OS2/jos-labs-Rogersyp/pull/new/lab2
remote:
remote: To git@github.com:OSU-OS2/jos-labs-Rogersyp.git
* [new branch] lab2 -> lab2
Branch lab1 set up to track remote branch lab1 from origin.
Switched to a new branch 'lab1'
Counting objects: 8, done.
Delta compression using up to 96 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 1.23 KiB | 0 bytes/s, done.
Total 6 (delta 3), reused 6 (delta 3)
remote: Resolving deltas: 100% (3/3), completed with 2 local objects.
remote:
remote: Create a pull request for 'lab1' on GitHub by visiting:
remote: https://github.com/OSU-OS2/jos-labs-Rogersyp/pull/new/lab1
remote:
remote: To git@github.com:OSU-OS2/jos-labs-Rogersyp.git
* [new branch] lab1 -> lab1
End of the script.

```

To verify it, go back to your repo webpage on GitHub Classroom, refresh the page. It should now contain 5 branches and the whole commit history:



You may also verify it on the terminal by the “git branch” command:

```
[songyip@os2 (lab1) ~/cs444/s24/test/jos-labs-Rogersyp$] git branch
* lab1
  lab2
  lab3
  lab4
  master
```

5. Now your repo is successfully set up. Remember you need to switch to the correct branch to work on each lab assignment. For example, to work on lab 1, type:

```
git checkout lab1
```

To commit and push to the repository, you may want to setup your git information by running the following commands (if you have not done this before...) :

```
[os2] $ git config --global user.email "your@email.address.com"
[os2] $ git config --global user.name your-name
[os2] $ git config --global core.editor /usr/bin/vim
[os2] $ git config --global push.default simple
[os2] $ git config --global core.autocrlf false
```

## Updating your student.info file

After having cloned your repository, your first task to finish is to update the student.info file in your repository. Please fill the information (your OSUID, server username, name, either 444/544, and your lab class section) in the file.

The reason why we collect such information is to match your repository to your account at OSU (to collect and record your scores).

```
OSU ID (xxx-yyy-zzz) : 933456789
ONID ID (e.g., songyip) : songyip
Name : Yipeng Song
CS 444/544 ? : 444
Lab Class # : Lab 1
```

Right now, it has some placeholder information, and please change it to your information. After making changes, you can make it accessible on our GitHub Classroom server by doing add, commit, and push. You can do that by running the following commands:

```
[os2] $ git status
```

```
... see that your student.info is edited
```

```
[os2] $ git add -A
```

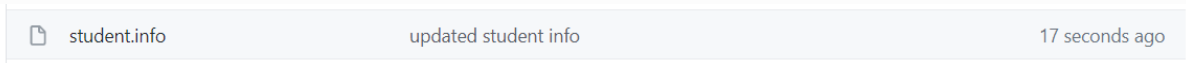
```
[os2] $ git commit
```

```
.. write some commit messages in the editor
```

```
[os2] $ git push
```

```
.. this will push the change in the remote server (our GitHub classroom server)
```

After pushing your changes, you can verify if that is available remotely by visiting your repository website on our GitHub Classroom server.

A notification bar from GitHub Classroom showing a file update. On the left is a document icon followed by the filename "student.info". In the center is the text "updated student info". On the right is the timestamp "17 seconds ago".

student.info

updated student info

17 seconds ago

## Tool Guide (not required for setup, just for FYI)

Familiarity with your environment is crucial for productive development and debugging. This page gives a brief overview of the JOS environment and useful GDB and QEMU commands. Don't take our word for it, though. Read the GDB and QEMU manuals. These are powerful tools that are worth knowing how to use.

### Kernel

GDB is your friend. Use the **qemu-nox-gdb** target to make QEMU wait for GDB to attach. See the **GDB** reference below for some commands that are useful when debugging kernels.



If you're getting unexpected interrupts, exceptions, or triple faults, you can ask QEMU to generate a detailed log of interrupts using the `-d` argument.

To debug virtual memory issues, try the QEMU monitor commands `info mem` (for a high-level overview) or `info pg` (for lots of detail). Note that these commands only display the *current* page table.

**(Lab 4+)** To debug multiple CPUs, use GDB's thread-related commands like `thread` and `info threads`.

## User environments (lab 3+)

GDB also lets you debug user environments, but there are a few things you need to watch out for, since GDB doesn't know that there's a distinction between multiple user environments, or between user and kernel.

You can start JOS with a specific user environment using `make run-*name*` (or you can edit `kern/init.c` directly). To make QEMU wait for GDB to attach, use the `run-*name*-gdb` variant.

You can symbolically debug user code, just like you can kernel code, but you have to tell GDB which symbol table to use with the `symbol-file` command, since it can only use one symbol table at a time. The provided `.gdbinit` loads the kernel symbol table, `obj/kern/kernel`. The symbol table for a user environment is in its ELF binary, so you can load it using `symbol-file obj/user/name`. *Don't* load symbols from any `.o` files, as those haven't been relocated by the linker (libraries are statically linked into JOS user binaries, so those symbols are already included in each user binary). Make sure you get the *right* user binary; library functions will be linked at different EIPs in different binaries and GDB won't know any better!

**(Lab 4+)** Since GDB is attached to the virtual machine as a whole, it sees clock interrupts as just another control transfer. This makes it basically impossible to step through user code because a clock interrupt is virtually guaranteed the moment you let the VM run again. The `stepi` command works because it suppresses interrupts, but it only steps one assembly instruction. Breakpoints generally work, but watch out because you can hit the same EIP in a different environment (indeed, a different binary altogether!).

# Reference

## JOS makefile

The JOS Makefile includes a number of phony targets for running JOS in various ways. All of these targets configure QEMU to listen for GDB connections (the `*-gdb` targets also wait for this connection). To start once QEMU is running, simply run `gdb` from your lab directory. We provide a `.gdbinit` file that automatically points GDB at QEMU, loads the kernel symbol file, and switches between 16-bit and 32-bit mode. Exiting GDB will shut down QEMU.

### make `qemu-nox`

Like `make qemu`, but run with only the serial console. To exit, press `Ctrl-a x`. This is particularly useful over SSH connections to Athena dialups because the VGA window consumes a lot of bandwidth.

### make `qemu-nox-gdb`

A combination of the `qemu-nox` and `qemu-gdb` targets.

### make `run-name`

(Lab 3+) Run user program *name*. For example, `make run-hello` runs `user/hello.c`.

### make `run-name-nox`, `run-name-gdb`, `run-name-gdb-nox`,

(Lab 3+) Variants of `run-name` that correspond to the variants of the `qemu` target.

The makefile also accepts a few useful variables:

### make `V=1` ...

Verbose mode. Print out every command being executed, including arguments.

### make `V=1 grade`

Stop after any failed grade test and leave the QEMU output in `jos.out` for inspection.

### make `QEMUEXTRA='args'` ...

Specify additional arguments to pass to QEMU.

## JOS `obj/`

When building JOS, the makefile also produces some additional output files that may prove useful while debugging:

`obj/boot/boot.asm`, `obj/kern/kernel.asm`, `obj/user/hello.asm`, etc.

Assembly code listings for the bootloader, kernel, and user programs.

`obj/kern/kernel.sym`, `obj/user/hello.sym`, etc.

Symbol tables for the kernel and user programs.

`obj/boot/boot.out`, `obj/kern/kernel`, `obj/user/hello`, etc

Linked ELF images of the kernel and user programs. These contain symbol information that can be used by GDB.

## GDB

See the [GDB manual](#) for a full guide to GDB commands. Here are some particularly useful commands for CS 444/544, 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/Nx 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/Ni 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 *file*. 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`.

## Note

QEMU represents each virtual CPU as a thread in GDB, so you can use all of GDB's thread-related commands to view or manipulate QEMU's virtual CPUs.

### thread *n*

GDB focuses on one thread (i.e., CPU) at a time. This command switches that focus to thread *n*, numbered from zero.

### info threads

List all threads (i.e., CPUs), including their state (active or halted) and what function they're in.

## 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.

For a complete reference to the monitor commands, see the [QEMU manual](#). Here are some particularly useful commands:

### xp/*Nx* *paddr*

Display a hex dump of *N* words starting at *physical* address *paddr*. If *N* is omitted, it defaults to 1. This is the physical memory analogue of GDB's `x` command.

### info registers

Display a full dump of the machine's internal register state. In particular, this includes the machine's *hidden* segment state for the segment selectors and the local, global, and interrupt descriptor tables, plus the task register. This hidden state is the information the virtual CPU read from the GDT/LDT when the segment selector was loaded. Here's the CS when running in the JOS kernel in lab 1 and the meaning of each field:

```
CS =0008 10000000 ffffffff 10cf9a00 DPL=0 CS32 [-R-]
```

`CS =0008`

The visible part of the code selector. We're using segment 0x8. This also tells us we're referring to the global descriptor table (0x8&4=0), and our CPL (current privilege level) is 0x8&3=0.

`10000000`

The base of this segment. Linear address = logical address + 0x10000000.

`ffffffff`

The limit of this segment. Linear addresses above `0xffffffff` will result in segment violation exceptions.

`10cf9a00`

The raw flags of this segment, which QEMU helpfully decodes for us in the next few fields.

`DPL=0`

The privilege level of this segment. Only code running with privilege level 0 can load this segment.

`CS32`

This is a 32-bit code segment. Other values include `DS` for data segments (not to be confused with the DS register), and `LDT` for local descriptor tables.

`[-R-]`

This segment is read-only.

## info mem

**(Lab 2+)** Display mapped virtual memory and permissions. For example,

```
ef7c0000-ef800000 00040000 urw
efbf8000-efc00000 00080000 -rw
```

tells us that the `0x00040000` bytes of memory from `0xef7c0000` to `0xef800000` are mapped read/write and user-accessible, while the memory from `0xefbf8000` to `0xefc00000` is mapped read/write, but only kernel-accessible.

QEMU also takes some useful command line arguments, which can be passed into the JOS makefile using the `QEMUEXTRA` variable.

## make QEMUEXTRA='-d int' ...

Log all interrupts, along with a full register dump, to `qemu.log`. You can ignore the first two log entries, “SMM: enter” and “SMM: after RMS”, as these are generated before entering the boot loader. After this, log entries look like

```
4: v=30 e=0000 i=1 cpl=3 IP=001b:00800e2e pc=00800e2e SP=0023:eebfdf28 EAX=00000005
EAX=00000005 EBX=00001002 ECX=00200000 EDX=00000000
ESI=00000805 EDI=00200000 EBP=eebfdf60 ESP=eebfdf28
...
```

The first line describes the interrupt. The `4:` is just a log record counter. `v` gives the vector number in hex. `e` gives the error code. `i=1` indicates that this was produced by an `int` instruction (versus a hardware interrupt). The rest of the line should be self-explanatory. See `info registers` for a description of the register dump that follows.

Note: If you're running a pre-0.15 version of QEMU, the log will be written to `/tmp` instead of the current directory.

---

Questions or comments regarding CS 444/544? Send e-mail to the TAs and/or the instructor.