6.828 lab tools guide

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.

Debugging tips:	Kernel User environments
Reference:	JOS makefile JOS obj/ GDB QEMU

Debugging tips

Kernel

GDB is your friend. Use the `qemu-gdb` target (or its `qemu-gdb-nox` variant) 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 GNUmakefile 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`
: Build everything and start QEMU with the VGA console in a new window and the serial console in your terminal. To exit, either close the VGA window or press `Ctrl-c` or `Ctrl-a x` in your terminal.

`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-gdb`
: Like `make qemu`, but rather than passively accepting GDB connections at any time, this pauses at the first machine instruction and waits for a GDB connection.

`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 6.828, 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`.

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 00008000 -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.

`info pg`

(Lab 2+) Display the current page table structure. The output is similar to `info mem`, but distinguishes page directory entries and page table entries and gives 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.

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.