Lab guidance

Hardness of assignments

Each assignment indicates how difficult it is:

- Easy: less than an hour. These exercise are typically often warm-up exercises for subsequent exercises.
- Moderate: 1-2 hours.
- Hard: More than 2 hours. Often these exercises don't require much code, but the code is tricky to get right.

These times are rough estimates of our expectations. For some of the optional assignments we don't have a solution and the hardness is a wild guess. If you find yourself spending more time on an assignment than we expect, please reach out on piazza or come to office hours.

The exercises in general require not many lines of code (tens to a few hundred lines), but the code is conceptually complicated and often details matter a lot. So, make sure you do the assigned reading for the labs, read the relevant files through, consult the documentation (the RISC-V manuals etc. are on the reference page) before you write any code. Only when you have a firm grasp of the assignment and solution, then start coding. When you start coding, implement your solution in small steps (the assignments often suggest how to break the problem down in smaller steps) and test whether each steps works before proceeding to the next one.

> Warning: don't start a lab the night before a lab is due; it much more time efficient to do the labs in several sessions spread over multiple days. The manifestation of a bug in operating system kernel can be bewildering and may require much thought and careful debugging to understand and fix.

Debugging tips

Here are some tips for debugging your solutions:

- Make sure you understand C and pointers. The book "The C programming language (second edition)" by Kernighan and Ritchie is a succinct description of C. Some useful pointer exercises are here. Unless you are already thoroughly versed in C, do not skip or skim the pointer exercises above. If you do not really understand pointers in C, you will suffer untold pain and misery in the labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

  A few pointer common idioms are in particular worth remembering:

  - If `int *p = (int*)100`, then `(int)p + 1` and `(int)(p + 1)` are different numbers: the first is `101` but the second is `104`. When adding an integer to a pointer, as in the second case, the

integer is implicitly multiplied by the size of the object the pointer points to.
  - `p[i]` is defined to be the same as `*(p+i)`, referring to the i'th object in the memory pointed to by p. The above rule for addition helps this definition work when the objects are larger than one byte.
  - `&p[i]` is the same as `(p+i)`, yielding the address of the i'th object in the memory pointed to by p.

  Although most C programs never need to cast between pointers and integers, operating systems frequently do. Whenever you see an addition involving a memory address, ask yourself whether it is an integer addition or pointer addition and make sure the value being added is appropriately multiplied or not.

- If you have an exercise partially working, checkpoint your progress by committing your code. If you break something later, you can then roll back to your checkpoint and go forward in smaller steps. To learn more about Git, take a look at the Git user's manual, or, you may find this CS-oriented overview of Git useful.

- If you fail a test, make sure you understand why your code fails the test. Insert print statements until you understand what is going on.

- You may find that your print statements may produce much output that you would like to search through; one way to do that is to run `make qemu` inside of `script` (run **`man script`** on your machine), which logs all console output to a file, which you can then search. Don't forget to exit `script`.

- In many cases, print statements will be sufficient, but sometimes being able to single step through some assembly code or inspecting the variables on the stack is helpful. To use gdb with xv6, run make **`make qemu-gdb`** in one window, run **`gdb-multiarch`** (or **`riscv64-linux-gnu-gdb`**) in another window (if you are using Athena, make sure that the two windows are on the same Athena machine), set a break point, followed by followed by 'c' (continue), and xv6 will run until it hits the breakpoint. See Using the GNU Debugger for helpful GDB tips. (If you start gdb and see a warning of the form 'warning: File ".../.gdbinit" auto-loading has been declined', edit ~/.gdbinit to add "add-auto-load-safe-path...", as suggested by the warning.)

  If you want to learn more about how to run GDB and the common issues that can arise when using GDB, check out this page.

- If you want to see what the assembly is that the compiler generates for the kernel or to find out what the instruction is at a particular kernel address, see the file `kernel.asm`, which the Makefile produces when it compiles the kernel. (The Makefile also produces `.asm` for all user programs.)

- If the kernel panics, it will print an error message listing the value of the program counter when it crashed; you can search `kernel.asm` to find out in which function the program counter was when it crashed, or you can run **`addr2line -e kernel/kernel`** ***`pc-value`*** (run **`man addr2line`** for details). If you want to get backtrace, restart using gdb: run 'make qemu-gdb' in one window, run gdb (or riscv64-linux-gnu-gdb) in another window, set breakpoint in panic ('b panic'), followed by followed by 'c' (continue). When the kernel hits the break point, type 'bt' to get a backtrace.

- If your kernel hangs (e.g., due to a deadlock) or cannot execute further (e.g., due to a page fault when executing a kernel instruction), you can use gdb to find out where it is hanging. Run run 'make qemu-gdb' in one window, run gdb (riscv64-linux-gnu-gdb) in another window, followed by followed by 'c' (continue). When the kernel appears to hang hit Ctrl-C in the qemu-gdb window and type 'bt' to get a backtrace.

- `qemu` has a "monitor" that lets you query the state of the emulated machine. You can get at it by typing control-a c (the "c" is for console). A particularly useful monitor command is `info mem` to print the page table. You may need to use the `cpu` command to select which core `info mem` looks at, or you could start qemu with `make CPUS=1 qemu` to cause there to be just one core.

It is well worth the time learning the above-mentioned tools.