# Lab0 - Environment Setup

## Lab0: XV6 and UNIX Utilities

This lab will familiarize you with xv6 and development environment.

### Boot xv6

You can do these labs on your own computer (**recommended**) or on Sledge. If you use your own computer, have a look at the **lab tools page** for setup tips, including how to set up the environment in Windows (WSL2) and macOS

If you use Sledge, please add the following statement to your `.bash_profile` to access the tools.

```
# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin
PATH=/usr/local/pkgs/qemu-5.10/bin:/usr/local/pkgs/riscv-gnu-toolchain/bin:$PATH

export PATH
```

Fetch the xv6 source for the lab and check out the `util` branch:

```
$ git clone https://github.com/mit-pdos/xv6-riscv-fall19.git xv6-riscv
Cloning into 'xv6-riscv'...
...
$ cd xv6-riscv
$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

This repository differs slightly from the book's xv6-riscv; it mostly adds some files. If you are curious look at the git log:

```
$ git log
```

The files you will need for this and subsequent lab assignments are distributed using the Git version control system. Above you switched to a branch (**git checkout util**) containing a version of xv6 tailored to this lab. 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. Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can *commit* your changes by running:

```
$ git commit -am 'my solution for util lab exercise 1'
Created commit 60d2135: my solution for util lab exercise 1
 1 files changed, 1 insertions(+), 0 deletions(-)
$
```

You can keep track of your changes by using the **git diff** command. Running **git diff** will display the changes to your code since your last commit, and **git diff origin/util** will display the changes relative to the initial xv6-riscv code. Here, `origin/util` is the name of the git branch with the initial code you downloaded for the class.

Build and run xv6:

```
$ make qemu
riscv64-unknown-elf-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o ker
...
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_zombie user/zombie.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_zombie > user/zombie.asm
riscv64-unknown-elf-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/zombie.sym
mkfs/mkfs fs.img README  user/xargstest.sh user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
balloc: first 591 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```

If you type `ls` at the prompt, you should see output similar to the following:

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2227
xargstest.sh   2 3 93
cat            2 4 32864
echo           2 5 31720
forktest       2 6 15856
grep           2 7 36240
init           2 8 32216
kill           2 9 31680
ln             2 10 31504
ls             2 11 34808
mkdir          2 12 31736
rm             2 13 31720
sh             2 14 54168
stressfs       2 15 32608
usertests      2 16 178800
grind          2 17 47528
wc             2 18 33816
zombie         2 19 31080
console        3 20 0
```

These are the files that `mkfs` includes in the initial file system; most are programs you can run. You just ran one of them: `ls`.

xv6 has no `ps` command, but, if you type **Ctrl-p**, the kernel will print information about each process. If you try it now, you'll see two lines: one for `init`, and one for `sh`.

To quit qemu type: **Ctrl-a x** (press **Ctrl** and **a** at the same time, followed by **x**).

### Debugging tips

Here are some tips for debugging:

- 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. Have a look at this example code and make sure you understand why it produces the results it does.

  A few common pointer idioms are particularly 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 this CS-oriented overview of Git.

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

- You may find that your print statements produce a lot of 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`.

- Print statements are often a sufficiently powerful debugging tool, but sometimes being able to single step through some assembly code or inspect variables on the stack is helpful. To use gdb with xv6, run `make qemu-gdb` in one window, run `gdb-multiarch` (or `riscv64-linux-gnu-gdb` or `riscv64-unknown-elf-gdb`) in another window (if you are using Sledge, make sure that the two

windows are on the same Sledge machine), set a break point, 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 see what assembly code the compiler generates for the xv6 kernel or find out what instruction is at a particular kernel address, see the file `kernel/kernel.asm`, which the Makefile produces when it compiles the kernel. (The Makefile also produces `.asm` for all user programs.)
- If the kernel causes an unexpected fault (e.g. uses an invalid memory address), it will print an error message that includes the program counter ("sepc") at the point where it crashed; you can search `kernel.asm` to find the function containing that program counter, or you can run `addr2line -e kernel/kernel` *`pc-value`* (run `man addr2line` for details). If you want a 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, perhaps due to a deadlock, 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 Ctl-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.