Computer Science 432
Spring 2022
Lab 2: Getting to Know `xv6`*, due next Monday before lab.*

Objective. To get an introduction to `xv6` on `RISC-V`.

Discussion. This week we will start using `xv6`, the student-friendly unix operating system developed at MIT. We will download and build `xv6`, and then write some user-space programs that interact with this very simple kernel. You should not feel that we've made any compromises by using `xv6`, it's simply a matter of distilling the O/S down to the "simplest complex" version of an operating system. You will be thankful down the road.[1]

It is assumed that you have read Chapter 1 of the MIT text. If you have not, *you should not begin this lab until you have finished the reading.*

Preparation. We have already installed the tools necessary for you to download and install the `xv6` operating system on lab machines. I encourage you to use the Unix and Ward labs. If you wish to install everything on your own machine, you can follow the directions found in the `Resources` section of the class website.

Assuming you're using our lab equipment, or a machine that has been appropriately prepared, you can now download sources for `xv6`, typically as part of a lab assignment. This week we will clone the O/S as part of the `lab2` repo:

```
$ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab2.git
```

This will install a basic version of `xv6` in the directory `lab2`. To build and run the O/S, you should:

```
$ cd lab2
$ make qemu
```

The makefile will make sure that the O/S is compiled using the RISC-V cross-compiler.[2]

When we write code for labs involving `xv6`, the make system will compile our code for us, and copy the resulting executable into a file system for `xv6` to access after the O/S boots.

Once the system is made, the `qemu`-based `RISC-V` emulator (pronounced: "kee-moo"), fully simulates the hardware as it boots and runs the operating system.

When you're in `xv6` be aware that, at this point, the operating system has only the most basic functionality. For example, there is a `cat` program, but no `more` or `less`. Even the shell is pretty limited: there are no environment variables (e.g. `PS1` or `PATH`). In return, you are able to see (and, soon, understand) the source code for these user-space utilities in the `user` directory. Similarly, the kernel support for the operating system has been simplified.

To leave `qemu` you must press CONTROL-A, then X.

Assignment. The work this week involves familiarizing yourself with `xv6` and the workflow necessary to add new user-level utilities.

1. As we described, above, clone the `lab2` repository:

   ```
   $ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab2.git
   $ cd lab2
   ```

2. Test-build the operating system:

   ```
   $ make qemu
   ... lots of compiling ...
   xv6 kernel is booting

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

[1] Be aware that I've captured the version of `xv6` that corresponds to our version of the textbook. Our labs, as well, are written to make use of a specific version of the `xv6` source code. While you can find these resources on the web at MIT, please be aware that those versions will necessarily deviate from our particular focus in this course.

[2] A cross-compiler is simply a compiler that runs on one type of architecture, but generates executables for another. For example, if you are on a Windows machine but developing applications for a phone, you're using a cross-compiler that runs on an Intel box, but generates executables that run on ARM devices. Here, we are simply trying to have the compiler generate code for a RISC-V architecture.

Notice that the prompt for `xv6` is `$`. Look around and see what utilities are available for you to use.

There is no `ps` command in `xv6`, but if you type CONTROL-P `xv6`

will print information about each process.

Exit the operating system by typing CONTROL-A, then X.

3. In the user-space directory, `user`, implement a simple version of the `sleep(1)` program. Your program should accept a single integer argument that indicates how long, in *ticks*, the program should pause before exiting. A "tick" is the length of time between interrupts generated by the timer chip. Your program should print an error message if the user fails to specify an argument.

   Here are some things to consider, as you implement `sleep`:

   - You may find it useful to to look at other user-space programs to get a sense about how to obtain command-line arguments in `xv6`. Also, notice how include files are typically specified.
   - There is a rudimentary library of routines provided for you in `ulib.c`. You can access these by including `user/user.h` in your code. This library contains, for example, a simple implementation of `atoi(3)`.
   - Note that `user/user.h` also declares the system calls that are available. For example, there is a `sleep(2)` entry point whose argument is the number of ticks to pause. This kernel call is implemented in `sys_sleep` in `kernel/sysproc.c` (because it's related to process scheduling). The actual system call is to the kernel in the `RISC-V` assembly `user/usys.S`: it places a sleep-specific constant in `a7`, and then performs an `ecall`. (This file is constructed as a side-effect of the building process. If you don't see that file, type `make user/usys.S` in top level directory of the lab.)
   - In `xv6`, you need call `exit` at the end of your `main` program. This, as you know, terminates the process and returns all resources to the operating system. (In more advanced implementations, the `main` method is called by a `start` routine that performs this task for us.)
   - When you are finished with implementing `sleep.c`, make sure you add it to the `UPROGS` list in the `Makefile` in the top folder. This forces a compile of your code and a transfer of the executable to the `xv6` file system.

4. Write a program, `find`, that takes, as arguments a directory and the name of a possible file. It searches the directory's tree and prints relative paths to files with that name. Your program should work similarly to this:

```
$ make qemu
...
init: starting sh
$ mkdir home
$ echo purple cows >home/waldo
$ mkdir away
$ echo mammoths >away/waldo
$ find . waldo
./home/waldo
./away/waldo
$
```

   Here are some considerations:

   - You will find it useful to look at `user/ls.c` to see how directories are read. Observe the use of `read` and `fstat`.
   - The file `kernel/param.h` defines `MAXPATH`, the maximum length of a file name path.
   - Use recursion to drive the search process into subdirectories.
   - Every directory has entries for "`.`" and "`..`". Do not recursively search in these directories.
   - Feel free to (from within `xv6`) make subdirectories and add files to test your `find` command. These file structures will persist until you perform a `make clean` in the top level directory of the lab.
   - You will be manipulating strings. Look at `user/user.h` for library functions that will help you. For example, you'll find `strcmp` and `strcat` allow us to compare and concatenate strings.
   - Update the `UPROGS` variable in the `Makefile` to incorporate your `find` utility in the `xv6` file system.

5. Write a simple version of the unix `xargs` program. This program accepts, as arguments, a unix command. It reads, as input, individual space-delimited words. For each word read, `w`, it executes the unix command with `w` as the final argument to the command.

   Here is how we might imagine it working:

   ```
   $ make qemu
   ...
   init: starting sh
   $ echo alice bob carol >friends
   $ xargs echo hello <friends
   hello alice
   hello bob
   hello carol
   $
   ```

   - Here, you will want to use the `fork`, `exec`, and `wait` commands. You should make sure that each command is finished before the next one is started.
   - Note that the file `kernel/param.h` defines `MAXARG`, the maximum number of arguments that can be passed to `exec`.
   - To read words from the input, you will need to write a routine that uses `read` to read and return one character at a time from standard input. Be aware than when `read` hits the end-of-file it returns a status of 0. Let's agree that a printable "word" character is a character with ASCII value greater than `' '` and less-than or equal to `'~'`.
   - When you're finished with your `xargs` implementation, add it to `UPROGS` in the top-level `Makefile` and test it out in `xv6`.

   Notice that this is also possible:

   ```
   $ make qemu
   ...
   init: starting sh
   $ mkdir home
   $ echo purple cows >home/waldo
   $ mkdir away
   $ echo mammoths >away/waldo
   $ find . waldo | xargs wc
   1 2 12 ./home/waldo
   1 1 9 ./away/waldo
   $
   ```

6. Make sure you add `sleep.c`, `find.c`, and `xargs.c` in the `lab2` repository, and that you commit changes to your `Makefile`. Push your commits to the server for discussion and evaluation. (After a `make clean`, a `git status` should report no changes that need to be committed.)

Thought Questions. Please think about the following questions before we meet in small groups.

1. As we add utilities, like `sleep` to the `user` directory, they are compiled to executables that start with an underscore. Why?
2. You may observe that the length of a "tick" does not seem to correspond to any particular length of time. Why is the operating system interested in getting interrupted once each tick? Given that, how does not O/S designer pick a length of time for the timer interrupt?
3. How does calling `sleep` impact the scheduling of other processes on the system?
4. There is no `PATH` environment variable on `xv6`. How does the shell (`user/sh.c`) find the commands it executes?

⋆