CSE 536: Advanced Operating Systems

Assignment 0: Getting Started (ungraded)

1 Brief Description of the xv6 OS and RISC-V

xv6 is an education-focused operating system (OS) designed by folks from MIT, namely Franz Kaashoek, Robert Morris, and Russ Cox (alongside contributions from many other individuals). Like Linux and MacOS, xv6 follows the structure of UNIX-based operating systems, and it is a loose reimplementation of Dennis Ritchie and Ken Thompson's UNIX version 6 (v6). UNIX-based OSs are commonplace today, and each of them typically consists of 3 main components: the kernel, the shell, and the applications/file system. Kernel manages the system operations (e.g., memory management, resource allocation, etc.) and the Shell is used to interact with the kernel.

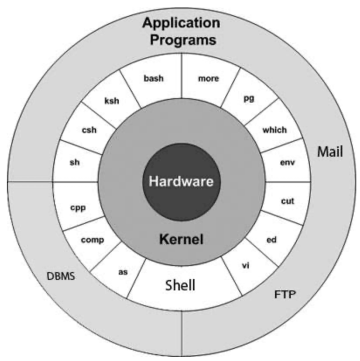

Figure 1: UNIX system architecture (source).

Suggested reading

The UNIX Time-Sharing System, Dennis M. Ritchie and Ken L.Thompson.
xv6: A Simple, Unix-like Teaching Operating System, Russ Cox, Franz Koshoek, and Robbert Morris.

xv6 is designed as a *monolithic* kernel, meaning all functionality of the OS is merged into one large program (please check notes from the 2nd lecture for more details). While modern monolithic kernels are terribly large (e.g., Linux contains more than 31 million code lines), the same cannot be said for xv6, since it is a fairly simple and self-contained kernel (approximately 6 thousand code lines).

Question.1

What are the advantages and disadvantages of monolithic kernels?

In this class, the implementation of xv6 we are concerned with is written in C and designed for the RISC-V architecture. This architecture was introduced at UC Berkeley in the early 2000s. RISC stands for Reduced Instruction Set Computing which at a high-level means that the architecture does not provide CPU instructions that perform complex tasks (e.g., run a cryptographic algorithm like AES). A famous example of RISC architecture is ARM, which is widely used in mobile phones (e.g., Qualcomm Snapdragon) and in modern Apple machines. In contrast to RISC, there is a CISC or Complex Instruction Set Computing architecture, which is designed to allow the CPU to perform complex tasks. A famous example of CISC architecture is the widely-used x86 architecture co-designed by Intel and AMD.

Tidbit

You might ask the following question: *"why are we using xv6 instead of a well-known OS?"*
For better or worse, your instructor's belief is that understanding core OS concepts is too complex on modern OSs like Linux (and heavens forbid Windows or MacOS). The goal of this course is to help you understand core concepts, instead of having you understand the vagaries of modern OSs.

2 Running xv6 on ASU Apporto Instances (or Personal Machines)

The xv6 operating system is designed for the RISC-V architecture (as mentioned in the previous paragraph). Hence, generally to run the xv6 OS, we require (1) a CPU capable of running RISC-V instructions and (2) a compiler toolchain capable of producing RISC-V assembly code. Requirement (2) is straightforward (we can easily download and set up a RISC-V compiler), but requirement (1) is very tricky since none of our CPUs support RISC-V instructions natively (more than likely you are running an x86 or ARM CPU). Hence, to fulfill the requirement (1), we will emulate a RISC-V CPU using a system emulator called QEMU. It emulates the machine's CPU processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. This is perfect for our use case to learn RISC-V architecture and implement OS functionality.

The following explains how to set up xv6 on your personal machines and Apporto VMs.

Important note

Apporto VM instances have QEMU and the RISC-V GNU toolchain pre-installed. Hence, you can skip to step-5 if you are using Apporto.

2.1 Setting up xv6 on your Linux, WSL, or Mac machines

Please follow the steps below to install on a Linux/WSL machine. For a Mac machine, please follow steps in the repository's README.

1. Clone the CSE 536 GitHub repository and checkout the getting-started branch.
   ```
   git clone https://github.com/ASTERISC-Teaching/cse536-release.git
   git checkout getting-started
   ```
2. Download and install the RISC-V GNU compiler toolchain.
   ```
   cd install/linux-wsl && ./linux-toolchain.sh
   ```
3. Download and install the RISC-V capable QEMU.
   ```
   cd install/linux-wsl && ./linux-qemu.sh
   ```
4. Export the installed components to PATH so that you can access them from shell.
   ```
   cd install/linux-wsl && source .add-linux-paths
   ```

   Tidbit

   Step 4 only exports the toolchains and QEMU to PATH for the currently running shell. If you exit this shell, you will have to repeat step 4. You can avoid this repetition (Google it ;)).

5. Clone and run xv6.
   ```
   git clone https://github.com/mit-pdos/xv6-riscv.git
   cd xv6-riscv
   make qemu
   ```

6. You should see a screen like the following, which basically opens the xv6 shell.

Figure 2: Terminal view - make qemu

The `make qemu` command creates a QEMU virtual machine (VM) and runs the xv6 kernel inside the VM. There are different parameters provided, which you can inspect by looking under the line (make qemu) in the Makefile. More details in the next section.

Once booted, you're in the xv6 kernel which shows you a shell terminal (look at '$ ...'). Play around and try commands like ls, cat, etc. They should work similarly to a normal shell. To find out all the commands supported by the shell, run 'ls'.

Tidbit

To exit xv6 and go back to your host terminal shell, execute Ctrl+A then X.

Please use the following link as additional reference. If you run into issues, reach out to the TAs for help.

2.2 Opening Apporto on Canvas

Apporto is a virtual cloud environment with many different OS instances (e.g., Linux, Windows, MacOS) provided by ASU to students through canvas.

1. Go to your ASU canvas website and select the canvas for CSE 536.
2. On your canvas, there should be an "Apporto Virtual Lab" on the left tab.
3. Apporto should open an Ubuntu-based operating system environment for you.

Important note

It is crucial to periodically backup your code. For instance, GitHub provides unlimited private repositories. Do NOT use a public repository for your assignment code!

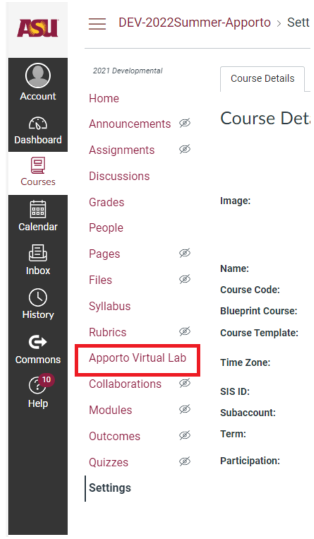

Figure 3: Finding Apporto on Canvas.

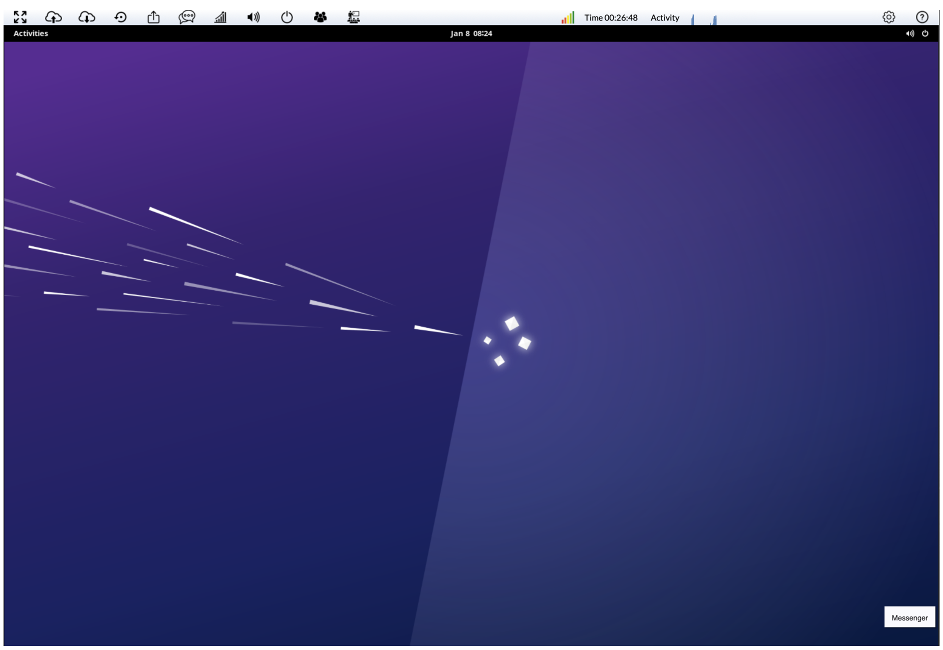

Figure 4: An Apporto instance.

2.3 Running xv6 on Apporto

In the Apporto cloud instances provided to you, QEMU and RISC-V GNU toolchains should be pre-installed and configured to run xv6 beforehand.

Tidbit

To test out if QEMU and RISC-V toolchains are correctly installed on the Apporto instance, please run the following commands: `qemu-system-riscv` and `riscv64-unknown-elf-gcc`.

1. Open a terminal within the Ubuntu instance.
2. Clone the xv6 RISCV OS repository from GitHub.
   `git clone https://github.com/mit-pdos/xv6-riscv.git`
3. Navigate into the cloned directory.
   `cd xv6-riscv`
4. Run the xv6 OS.
   `make qemu`

3 Debugging xv6 using the GNU debugger (GDB)

A debugger is your friend when dealing with systems-level programming (aka designing an OS), and GDB is your best friend. To run xv6 in debugging mode with GDB, please follow the steps below:

1. Instead of running `make qemu` on step 4 in subsection 2.3, you can run gdb for xv6 using `make qemu-gdb`. This will produce an output similar to the one below.

   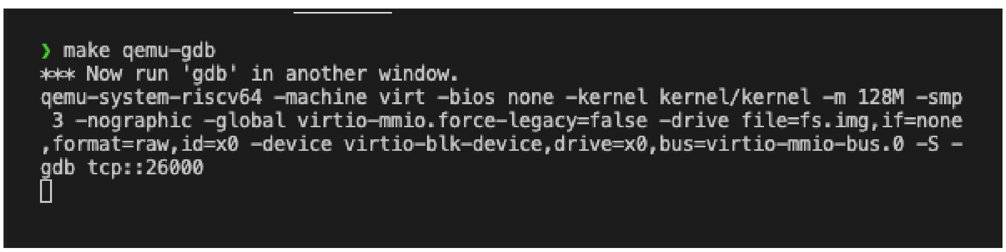

   Figure 5: Terminal view - make qemu-gdb

   Essentially, the system starts in debug mode and waits for a GDB instance to be attached.
2. On a separate terminal window, navigate to the xv6-riscv folder.
3. Then, run `riscv64-unknown-elf-gdb` to attach a GDB instance to the QEMU VM.

Tidbit

It is important to run GDB in the xv6-riscv folder because we have set up various configurations for the debugger in the .gdbinit script in the folder. If you're interested, check out that script.

Now you can debug execution by using the following commands on the gdb terminal:

- `continue` (or `c`) to resume execution until breakpoint.
- `b func` to set a breakpoint at a certain function (func).
- `b filename:line` to set a breakpoint at a certain line of a file.
- `si` to single-step the execution by running one instruction and stopping.
- `s` to single-step the execution by running one C code line and stopping.

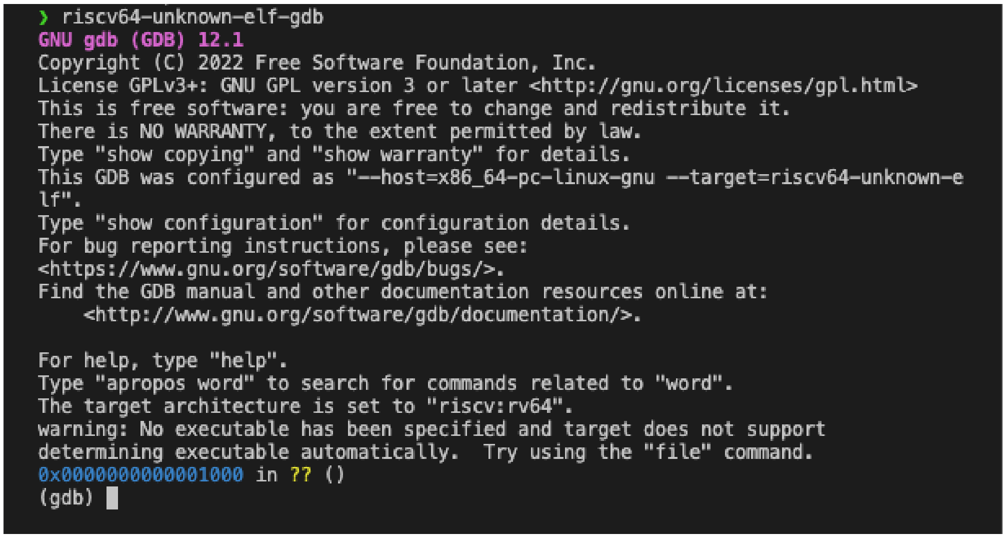

Figure 6: Terminal view - GDB debugger

- `info registers` will show you the value of various CPU registers. You can run it at any breakpoint.
- `bt` shows you a backtrace of the stack, which includes the nested functions you have executed. For example, if you executed A() and inside A() you executed B(), it will show you B → A.

Suggested reading

RISC-V GDB Tutorial, SHAKTI Development Team

4 Miscellaneous

4.1 Understanding the xv6 Code Directory Structure

The xv6 OS from the GitHub repository has the following 3 directories in the root directory.

1. `kernel/` contains assembly and C code files that belong to the kernel. Check: (1) start.S which contains first assembly code that runs when xv6 boots, and (2) main.c which contains first C code that runs when xv6 boots.
2. `user/` contains code files related to user programs (e.g., ls, cat, etc.).
3. `mkfs/` contains code files for building an xv6 file system and image.

Suggested reading

RISC-V Assembly Language, Stephen Marz

5 Acknowledgements

Special thanks to Divyam Mehta, Ayushman Bilas Thakur, and Mukul Mahadik for their help in designing this handout.