Operating Systems Concepts

Introduction to xv6

CS 4375, Fall 2025

Instructor: MD Armanuzzaman (***Arman***)

marmanuzzaman@utep.edu

September 8, 2025

Summery

- Processes
    - Process representation in OS
    - Process creation
    - Process termination
    - Context switching
    - Process queues
    - Process scheduling
    - Interprocess communication

Agenda

- Introduction to xv6
  - What is xv6?
  - RISC-V
  - Hardware-software stack of xv6
  - xv6 system calls
  - Code-base overview

xv6

- *xv6* is MIT's re-implementation of UNIX v6
    - Written in ANSI C
    - Runs on RISC-V and x86
        - We will use the RISC-V version with the QEMU simulator
    - Smaller than v6
    - Preserve basic structure (processes, files, pipes. etc.)
    - Runs on multicores
    - Got paging support in 2011

Ken Thompson &
Dennis Ritchie, 1975

xv6

- To understand it, you'll need to read its source code
    - It's **not** that hard
    - Source code:
        - https://github.com/Tomal-kuet/xv6-riscv-labs (for this course assignments)
        - Forked from https://github.com/mit-pdos/xv6-riscv; to avoid unexpected updates during this course
    - Book/commentary
        - xv6: a simple, Unix-like teaching operating system
        - https://pdos.csail.mit.edu/6.1810/2024/xv6/book-riscv-rev4.pdf

Why xv6?

- **Why study an old OS instead of Linux, Solaris, or Windows?**

Why xv6?

- **Why study an old OS instead of Linux, Solaris, or Windows?**
- Big enough
  - To illustrate basic OS design & implementation
- Small enough
  - To be (relatively) easily understandable
- Similar enough
  - To modern OSes
  - Once you've explored xv6, you will find your way inside kernels such as Linux

Why RISC-V?

- RISC-V: open standard instruction set architecture (ISA) based on RISC principles
  - High quality, loyalty free, license free
  - Multiple proprietary and open-source core implementations
  - Supported by growing software ecosystem
  - Appropriate for all levels of computing system, from **microcontrollers** to **supercomputers**
- Fun to use toolchains for the new architecture

RISC-V®

RISC-V: The Free and Open RISC Instruction Set Architecture

Apple shows interest in RISC-V chips, a competitor to iPhones' Arm tech

RISC-V chip technology could be used for tasks like AI and computer vision.

Intel Will Offer SiFive RISC-V CPUs on 7nm, Plans Own Dev Platform

By Joel Hruska on June 24, 2021 at 8:36 am | Comments

xv6 Structure

- Monolithic kernel
  - Provides services to running programs
- Processes uses system calls to access system services
- When a process call a system call
  - Execution will enter the kernel space
  - Perform the service
  - Return to the user space

Implementation

- RISC-V
    - 32 bit
- Multicore
- C and Assembly
    - 6,000 lines of code
- QEMU emulation

xv6 hardware and software stack

<table>
<tr><td colspan="2">User programs</td></tr>
<tr><td colspan="2">Xv6 kernel</td></tr>
<tr><td>QEMU (emulating RISC-V hardware)</td><td>Other user programs</td></tr>
<tr><td colspan="2">Linux (e.g., Ubuntu 22.04)</td></tr>
<tr><td colspan="2">VMM (e.g., VMWare)</td></tr>
<tr><td colspan="2">Native OS (e.g., MacOS, Windows)</td></tr>
<tr><td colspan="2">Hardware</td></tr>
</table>

Features of xv6

- Processes
- Virtual address spaces
- Files, Directories
- Pipes
- Multitasking
    - Time slicing
- 21 system calls

What is missing?

- All the complexity of real OS
  - Millions of lines of code
- User IDs, Login
- File protection: no such bit
- Mountable file systems
- Paging to disk
- Sockers, support for networks
- Interprocess Communication
- Device driver: Only two
- User code/apps: limited

xv6 System Calls

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n zero bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

xv6 kernel source files

File	Description
bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	exec() system call.
file.c	File descriptor support.
fs.c	File system.
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
spinlock.c	Locks that don't yield the CPU.
start.c	Early machine-mode boot code.
string.c	C string and byte-array library.
swtch.S	Thread switching.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
virtio_disk.c	Disk device driver.
vm.c	Manage page tables and address spaces.

xv6 Setup

- Mac OS, Linux: install directly on top of your OS
- Windows: Subsystem for Linux (WSL 2) with Ubuntu 20.04
- Toolchain
  - You need a RISC-V toolchain and QEMU for RISC-V

Install dependencies:

```
$ sudo apt-get install git build-essential gdb-multiarch qemu-
  system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

xv6 Setup

- Mac OS, Linux: install directly on top of your OS
- Windows: Subsystem for Linux (WSL 2) with Ubuntu 20.04
- Toolchain
  - You need a RISC-V toolchain and QEMU for RISC-V

Install dependencies:

```
$ sudo apt-get install git build-essential gdb-multiarch qemu-
  system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

xv6 Setup

- Repository
  - https://github.com/Tomal-kuet/xv6-riscv-labs
- Mirror on your own github

Mirroring a repository

1. Open Terminal.
2. Create a bare clone of the repository.

```
git clone --bare https://github.com/EXAMPLE-USER/OLD-REPOSITORY.git
```

3. Mirror-push to the new repository.

```
cd OLD-REPOSITORY
git push --mirror https://github.com/EXAMPLE-USER/NEW-REPOSITORY.git
```

4. Remove the temporary local repository you created earlier.

```
cd ..
rm -rf OLD-REPOSITORY
```

xv6 Setup

```
$ cd xv6-riscv-labs
$ make qemu
```

ls:

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2226
cat            2 3 23960
echo           2 4 22784
forktest       2 5 13144
grep           2 6 27320
init           2 7 23888
kill           2 8 22744
ln             2 9 22696
ls             2 10 26192
mkdir          2 11 22848
rm             2 12 22832
sh             2 13 41720
stressfs       2 14 23848
```

User Programs

Announcement

- Introduction to xv6
  - Overview of the code base
  - Homework 1 will be released
    - Due in two weeks (hard deadline)
  - **Get the installation done this week**
    - We will do a in class exercise next week