CSE321: Operating Systems Laboratory

Reading Materials on Files and Components of xv6-riscv

Learning Objectives

- ➔ Understand the overall structure of the xv6-riscv operating system.
- ➔ Identify the major directories, files, and components of xv6-riscv.
- ➔ Explain, at a high level, how xv6 boots, creates processes, handles system calls, and manages memory.
- ➔ Relate user programs and system calls to the corresponding kernel files and layers (processes, traps, file system, devices).

1. What is xv6-riscv?

xv6 is a small, Unix-like teaching operating system that re-implements the ideas of Sixth Edition Unix (V6) on a modern architecture. The xv6-riscv version targets the 64-bit RISC-V architecture and is widely used in operating systems courses. Its codebase is intentionally small and readable, so that students can study how an operating system works internally.

At a high level, the system consists of a privileged kernel and many unprivileged user processes:

Further reading: xv6 RISC-V book, Chapter 1 (Overview and kernel/user split)

```
    user space
+--------------------+         +--------------------+
|    shell process   |         |    cat process     |
|  (sh user prog)    |         |  (user program)    |
+----------^---------+         +----------^---------+
           | system calls                 |
           +---------------+--------------+
                           |
   kernel space            v
+-----------------------------------------+
|               Kernel                    |
|  (processes, memory, files,             |
|   devices, scheduler, etc.)             |
+-----------------------------------------+
```

Diagram 1: User Space vs Kernel Space (Conceptual View)

User programs live in the user space and cannot directly access hardware. Whenever they need services such as file I/O, process creation, or memory management, they issue system calls, which transfer control into the kernel (kernel space). The kernel then performs the requested action and returns the result to the user program.

2. Top-Level Directory Layout of xv6-riscv

When you open the xv6-riscv source tree, you will see several important files and directories at the top level:

- ➔ **kernel/** Contains the operating system kernel source files (C and assembly).
- ➔ **user/** Contains user programs (such as sh, ls, cat) and a tiny user-space library.
- ➔ **mkfs:** A tool that builds the initial xv6 file system image (fs.img) from a set of files.
- ➔ **Makefile:** It describes how to build xv6 and its user programs, as well as how to run xv6 under QEMU.
- ➔ **Other files** include Helper scripts, configuration files, and documentation (e.g., README).

Further reading: xv6 RISC-V book, Sections 2.1–2.4 (Code organization and file layout)

```
xv6-riscv/
├── Makefile            (build rules)
├── mkfs                (builds fs.img)
├── kernel/             (OS kernel, runs in supervisor mode)
│   ├── entry.S
│   ├── start.c
│   ├── main.c
│   ├── proc.c, exec.c, swtch.S
│   ├── trap.c, syscall.c, ...
│   ├── vm.c, kalloc.c
│   ├── fs.c, file.c, log.c, bio.c, ...
│   └── drivers: uart.c, console.c, virtio_disk.c, plic.c, ...
└── user/               (user programs + tiny libc)
    ├── init.c, sh.c
    ├── ls.c, cat.c, echo.c, ...
    ├── user.h, ulib.c, usys.S, printf.c
    └── tests: usertests.c, etc.
```

Diagram 2: Simplified Directory Tree

This tree provides a map of where to look when you want to modify a specific part of the system. For example, process-related code is mostly located in the kernel/proc directory.c, while shell code lives in user/sh.c.

3. Kernel Files by Responsibility

The xv6 source is organised by responsibility. Each kernel file focuses on a specific subsystem, such as booting, processes, traps, memory, devices, the file systems, or locking. The table below summarises the main groups and representative files.

Group	Key Files (kernel/)	Short Responsibility
Boot	entry.S, start.c, main.c	First code: switch to kernel, initialise subsystems.
Processes	proc.c, exec.c, swtch.S, sysproc.c	Process table, scheduler, fork/exec/exit, process system calls.
Traps	kernelvec.S, trampoline.S, trap.c, syscall.c	Trap entry/exit, system call dispatch.
Memory	vm.c, kalloc.c	Virtual memory, page tables, physical page allocation.
Devices	console.c, uart.c, plic.c, virtio_disk.c, printf.c	Console, UART, interrupt controller, disk driver, kernel printing.
File system	bio.c, fs.c, log.c, file.c, sysfile.c, pipe.c	File system structures, buffer cache, logging, pipes.
Misc	spinlock.c, sleeplock.c, string.c	Locks, basic utility routines.

In the rest of this lecture note, we walk through the main groups, explain what each file does, and show how the pieces interact using simple diagrams.

4. Booting xv6-riscv: The Boot Path

When xv6 starts, the CPU begins executing a small boot loader (provided by QEMU and not part of xv6 itself). The boot loader eventually jumps into the xv6 kernel at the entry point defined in kernel/entry.S. From there, a short sequence of files brings the system into a usable state and launches the first user process.

Further reading: [xv6 RISC-V book, Section 2.6 (Starting xv6 and the first process)]()

```
Hardware / QEMU
   |
   |  (boot ROM + boot loader, outside xv6)
   v
[1] entry.S        (kernel/entry.S)
    - sets up stack
    - jumps to start()
   |
   v
[2] start.c        (kernel/start.c)
    - runs in machine mode
    - configures timer, interrupts
    - switches to supervisor mode
    - calls main()
   |
   v
[3] main.c         (kernel/main.c)
    - initialises memory, process table, file system,
devices
    - calls userinit()  -> start user program 'init'
    - enters scheduler loop
   |
   v
[4] init (user/init.c)
    - creates console, starts shell 'sh'
   |
   v
[5] sh (user/sh.c)
    - reads commands, runs other programs
```

Diagram 3: Boot Sequence (From Hardware to Shell)

The main responsibilities of each file are:

- ➔ **entry.S:** First kernel instructions after the boot loader. Sets up a stack and jumps to C code in start.c.
- ➔ **start.c:** Runs in RISC-V machine mode, configures hardware (timer, interrupts), switches to supervisor mode, and calls main().
- ➔ **main.c:** Initialises core subsystems (memory allocator, process table, file system, device drivers), starts the first user process (init), and then enters the scheduler.

5. Processes and the Process Address Space

Processes are the basic units of execution in xv6. The kernel keeps track of all processes, schedules them on CPUs, and provides each process with its own virtual address space.

- ➔ **proc.c:** Manages the process table, process states, and the scheduler. Functions such as allocproc(), scheduler(), fork(), exit(), and wait() live here.
- ➔ **exec.c:** Implements the exec() system call, which replaces the current process image with a new program loaded from an ELF executable.
- ➔ **swtch.S:** Assembly code that performs a context switch by saving and restoring CPU registers when switching between processes.
- ➔ **sysproc.c:** Implements process-related system calls (sys_fork, sys_exit, sys_wait, sys_kill, sys_sleep, sys_uptime, and so on).

Further reading: xv6 RISC-V book, Chapters 2–3 (Processes and address spaces)

```
high addresses (MAXVA)
   +----------------------------+
   | trampoline + trapframe     |  (for trap entry/return)
   +----------------------------+
   |            heap            |  (grows up with malloc/sbrk)
   |          (dynamic)         |
   +----------------------------+
   |         user stack         |  (grows down)
   +----------------------------+
   |      user text + data      |  (code + global variables)
   +----------------------------+
 low addresses (0)
```

Diagram 4: Simplified Process Address Space

The file vm.c is responsible for creating and manipulating the page tables that implement this layout. The file kalloc.c provides a simple page allocator that hands out 4096-byte physical pages to vm.c and other parts of the kernel.

6. Traps and System Calls: Entering and Leaving the Kernel

A trap is a controlled transfer of execution from user space into the kernel. Traps can be caused by system calls (explicit requests from user code), exceptions (such as page faults), or hardware interrupts (such as timer or device interrupts). xv6 handles traps using a combination of assembly and C files.

- ➔ **trampoline.S:** Assembly code at a fixed virtual address used when traps occur from user space. It saves user registers into a trapframe and switches to a kernel stack.
- ➔ **kernelvec.S:** Trap entry code used when the CPU is already in the kernel.

➔ **trap.c:** C-level trap handlers (usertrap(), kerneltrap()) that classify traps, call the appropriate system call or device handler, and manage returning to user space.

➔ **syscall.c:** Dispatches system calls based on the system call number, fetches arguments from the trapframe, calls sys_* functions, and stores the return value back in the trapframe.

Further reading: [xv6 RISC-V book, Chapter 4 (Traps and system calls)]

```
User process (e.g., sh)
----------------------------------
 in user space:
   write(fd, buf, n);      // user C call
     |
     v
   usys.S stubs            // put SYS_write in a7, do ecall
----------------------------------
 Trap to kernel (ECALL)
----------------------------------
   trampoline.S            // save registers to trapframe
     |
     v
   usertrap() (trap.c)     // sees "system call" cause
     |
     v
   syscall() (syscall.c)   // switch on system call number
     |
     v
   sys_write() (sysfile.c) // calls lower file system layers
----------------------------------
 Return to user
----------------------------------
   syscall() puts return value in trapframe
   usertrapret() + trampoline.S restore registers
   back to user code, write() returns
```

Diagram 5: System Call Path (User → Kernel → User)

This diagram also shows where you would add a new system call. You would assign it a number, create a stub in user/usys.S, add an entry in syscall.c, and implement the kernel-side function in an appropriate sys*.c file.

7. Memory Management Components

xv6 provides each process with its own virtual address space and manages physical memory using a simple page allocator. Two central files are involved in memory management:

➔ **kalloc.c:** Implements a very simple physical page allocator backed by a free list of pages. Functions kalloc() and kfree() allocate and free 4096-byte pages.

- ➔ **vm.c:** Manages page tables and virtual to physical mappings. It creates new page tables for the kernel and for processes, maps segments (text, data, stack), and provides helpers such as copyin, copyout, and copyinstr.

Further reading: xv6 RISC-V book, Chapter 3 (Virtual memory system)

```
              Process code in proc.c / exec.c
                             |
                     needs memory
                             |
                             v
+-----------------+        +-----------------------+
|  vm.c           |<------>| Hardware page tables  |
|  (virtual mem)  |        |  (RISC-V Sv39)        |
+-----------------+        +-----------------------+
       |
       v
+-----------------+
|  kalloc.c       |
| (physical page  |
|    allocator)   |
+-----------------+
       |
       v
Physical RAM
```

Diagram 6: Memory Management Stack

When a new process is created, or exec() loads a new program, vm.c sets up the virtual address space and requests physical pages from kalloc.c as needed.

8. Device Drivers and Interrupt Handling

xv6 interacts with hardware devices such as the serial port (UART), the virtual disk, and the interrupt controller through simple device drivers.

- ➔ uart.c: Low-level driver for the UART serial port, used to implement the console I/O device.
- ➔ console.c: Higher-level console device. Provides line editing and connects the UART to file descriptors so that read and write on standard input/output work.
- ➔ plic.c: Driver for the RISC-V platform-level interrupt controller (PLIC), which routes external interrupts (such as UART and disk) to CPUs.
- ➔ virtio_disk.c: Block device driver for the virtual disk used by QEMU, implementing read and write of disk blocks.
- ➔ printf.c: Implements a minimal printf() used by the kernel, which outputs characters using the UART driver.

Further reading: xv6 RISC-V book, Chapters 5–6 (Device drivers and interrupts)

```
kernel code (printf in printf.c)
            |
            v
console.c  (consolewrite)
            |
            v
uart.c     (uartputc_sync)
            |
            v
UART hardware  ->  QEMU terminal window
```

Diagram 7: Console Output Path

When you see output in the QEMU window (for example, kernel messages or shell prompts), it is traveling along this path from printf() down to the UART device.

9. File System Layers and Files

xv6 implements a simple Unix-like file system. The implementation is layered so that each part has a clear role. The main files involved are:

- **fs.c:** Implements on-disk file system structures such as the superblock, inodes, and directories, and provides operations to read and write file contents.
- **bio.c:** Implements a buffer cache that caches disk blocks in memory, reducing the number of physical disk accesses.
- **log.c:** Provides a simple write-ahead logging mechanism to keep the file system consistent in the presence of crashes.
- **file.c:** Manages in-memory struct file objects and per-process file descriptors.
- **sysfile.c:** Implements file-related system calls (open, read, write, close, link, unlink, mkdir, chdir, fstat, and so on).
- **pipe.c:** Implements pipes, which are in-memory buffers used for inter-process communication.

Further reading: xv6 RISC-V book, Chapters 8–10 (File system structure and logging)

```
User space:
  read(fd, buf, n)
        |
        v
Kernel (sysfile.c):
  sys_read() -> fileread()
        |
        v
File descriptor layer (file.c):
  struct file -> inode or pipe or device
        |
        v
Inode / directory layer (fs.c):
  readi() finds which disk blocks hold file data
        |
        v
Buffer cache layer (bio.c):
  bread() / bwrite() get blocks from cache or disk
        |
        v
Log + block allocator (log.c + fs.c)
        |
        v
Disk driver (virtio_disk.c)
        |
        v
Virtual disk (fs.img on host)
```

Diagram 8: File System Stack for read()

This layered design means that higher layers (such as sysfile.c and file.c) do not need to know the details of how blocks are fetched from disk; they simply use the abstractions provided by the file system and buffer cache.

10. Concurrency and Locking Components

Because xv6 can run on multiple CPUs, the kernel must protect shared data structures from concurrent access. Two types of locks are provided:

➔ **spinlock.c:** Implements spin locks, which cause a CPU to busy-wait (spin) until the lock becomes free. These are used for short, critical sections where sleeping would be too expensive.

➔ **sleeplock.c:** Implements sleep locks, which put a thread to sleep while it waits for a lock, making them suitable for longer operations such as file system actions.

➔ **string.c:** Implements basic string and memory functions (memcpy, memmove, memset, strcmp, strlen, and so on) to avoid depending on an external C library.

Further reading: [xv6 RISC-V book, Chapter 6 (Locks and concurrency)]()

```
Multiple CPUs
   |
   v
 code in fs.c / bio.c / proc.c
   |
   +-- acquire(&lock) ----+
   |                      |   critical section
   |                      v
   |                   shared data
   +-- release(&lock) ----+
```

Diagram 9: Spinlock Usage (Conceptual)

Any code that accesses a shared data structure (such as the process table, file table, or buffer cache) must hold the appropriate lock to prevent race conditions.

11. User-Space Files and Programs (user/ Directory)

The user/directory contains programs that run in user space as well as a tiny support library that provides system call wrappers and basic functions.

- **user.h:** Contains function prototypes and type definitions for user programs, including system call declarations.
- **ulib.c:** Implements basic C library functions (string handling, memory operations, simple wrappers) for user programs.
- **printf.c:** User-space printf implementation.
- **usys.S:** Assembly stubs that issue ecall instructions to enter the kernel for system calls.
- **init.c:** The first user program started by the kernel. It sets up the console and then starts the shell (sh).
- **sh.c:** The shell program. It reads commands from the user, uses fork() and exec() to run other programs, and waits for them to finish.
- **Utilities** such as ls.c, cat.c, echo.c, mkdir.c, rm.c, kill.c, sleep.c, and others provide basic Unix-style commands.
- **Test programs**, such as usertests.c, exercise the kernel and help you check the correctness of your changes.

Further reading: [xv6 RISC-V book, Chapters 1–2 and 4 (User programs and system call interface)]()

```
user program (e.g., ls.c)
    |
    v
user.h + ulib.c  (write(), read(), open(), ...)
    |
    v
usys.S stubs     (assembly ECALL wrapper)
    |
    v
kernel (syscall.c + sys*.c)
```

Diagram 10: User Program Calling a System Call

12. Bringing It All Together: From Build to Shell Prompt

Let us summarise the life cycle of xv6 from building the system to running a shell command:

I. On the host machine, the Makefile compiles the kernel (files in kernel/), compiles user programs (in user/), and uses mkfs to build the initial file system image (fs.img).

II. When you run xv6 in QEMU, the boot loader loads the xv6 kernel and jumps to entry.S, which leads to start.c and main.c to launch the first user process init.

III. The init program creates the console device and starts the shell sh. The shell reads commands from the user and uses fork() and exec() to create and run new processes.

IV. Whenever a user program calls a system call (such as read, write, open, fork, exec), the CPU executes an ecall instruction, which traps into the kernel via trampoline.S, trap.c, and syscall.c.

V. Inside the kernel, various subsystems cooperate: proc.c and swtch.S manage processes and context switches; vm.c and kalloc.c manage memory; the file system layers in fs.c, bio.c, log.c, file.c, and sysfile.c manage files and directories; device drivers in uart.c, console.c, plic.c, and virtio_disk.c interact with hardware; and locks in spinlock.c and sleeplock.c protect shared data.

VI. The kernel eventually returns control to user space, and the user process continues execution as if it had simply called a C function.

By understanding how these files and components fit together, you gain a roadmap for navigating the xv6 source. This roadmap is essential for later labs and assignments where you will modify the scheduler, add system calls, change the file system, or extend the memory manager.

13. Suggested Reading and Practice Activities

To reinforce this lecture, students are encouraged to:

- ➔ Read the introductory chapters of the xv6 book, focusing on the overview of the kernel and the description of each file.
- ➔ Open the xv6 source tree and locate the files mentioned in this note. Skim through entry.S, start.c, main.c, proc.c, trap.c, syscall.c, vm.c, and fs.c to match the code to the diagrams.
- ➔ As a small lab exercise, add a simple new system call (for example, hello()) that prints a message from the kernel when called by a user program. Identify which files you must change and how they are connected using the diagrams in this lecture.

Over time, this high-level mental model will make it much easier to understand and modify specific parts of xv6 during your operating systems course.

References and Further Reading

- ➔ Overview of xv6-riscv design, file layout, and code examples: xv6: a simple, Unix-like teaching operating system (RISC-V edition)
- ➔ Official xv6-riscv source code repository (all files mentioned in this note): mit-pdos/xv6-riscv on GitHub
- ➔ Course materials, labs, and assignments using xv6-riscv: MIT 6.S081 / 6.1810 Operating Systems Engineering
- ➔ Background on RISC-V privilege levels, page tables, and traps: RISC-V Instruction Set Architecture Specifications
- ➔ Emulator used to run xv6-riscv: QEMU official website

- ➔ **Mapping from this note to xv6 book chapters:**
  - ◆ **Sections 1–3** (overview, directory layout, kernel files) ↔ **Chapters 1–2**.
  - ◆ **Sections 4–6** (boot path, processes, traps/system calls) ↔ **Chapters 2–4**.
  - ◆ **Sections 7–9** (memory, file system, device drivers) ↔ **Chapters 3, 5–6, 8–10**.
  - ◆ **Sections 10–11** (locking, user programs) ↔ **Chapters 6–7** and **the user programs appendix**.