RISK V OS

Cole Easley, Ryan Noel, Drew Carter, Dallas Wade

LINK TO REPO

Part 1: Features Successfully Implemented

1.1 Interactive Shell Environment

What it does: The OS boots to an interactive command-line shell instead of automatically running hardcoded processes. Users can type commands to interact with the OS.

How it works:

- Entry point in entry.S initializes CPU and stack
- kernel.c contains the shell loop that accepts user input
- The shell supports 10+ commands: help, stop, apps, run, ps, kill, fs, prog, etc.

Why it matters: This demonstrates a user interface, pretty central to any OS. Real operating systems provide shells for users to interact with services. Our shell specifically is interactive rather than being an automatic demo.

1.2 Dynamic Threading & Process Management

What it does: The OS can spawn multiple threads/processes on demand and manage them concurrently. You can run multiple apps simultaneously, list running threads with ps, and kill threads with kill.

How it works:

- thread.c implements the threading system with process spawning, scheduling, and termination
- Each thread has its own stack and register state (saved in PCB)
- context.S implements the context switch - saving one thread's state and restoring another's
- The scheduler uses cooperative multitasking: threads give up control voluntarily

Code example (thread.c):

```
typedef struct thread {
   int tid;
   void *(*fn)(void *);
   void *arg;
   uint64_t *sp;
} thread_t;
```

Why it matters: Multithreading is fundamental to modern OS design. This shows we understand context switching, scheduling, and concurrent execution.

1.3 Filesystem with File Persistence

What it does: Users can create, read, write, list, and delete files. Files persist across context switches and are accessible to all threads.

How it works:

- fs.c implements a simple in-memory filesystem
- Files stored in a static array of inode-like structures
- Each file has: name (up to 32 chars), data (up to 256 bytes), and usage flag
- Commands: fs write <name> <data>, fs read <name>, fs ls, fs rm <name>

Code example (fs.c):

```
typedef struct {
   int used;
   char name[FS_NAME_LEN];
   char data[FS_DATA_LEN];
} fs_file;

int fs_write(const char *name, const char *data) {
   int idx = find_slot(name);
   if (idx < 0)
   files[idx].used = 1;
   strlcpy(files[idx].name, name, FS_NAME_LEN);
   strlcpy(files[idx].data, data, FS_DATA_LEN);
   return 0;
}
```

Why it matters: Filesystems are a critical OS component. This demonstrates file storage, retrieval, and the concept of persistence.

1.4 User Program Loader with Capability Checks

What it does: The OS can load and execute user-written scripts/programs with restricted permissions based on capability bitmasks.

How it works:

- prog.c implements a script interpreter
- Scripts are strings of semicolon-separated commands: print, write, read, spawn, yield, exit
- Capability bitmask controls permissions:
    - 1 = UART (print)
    - 2 = FS read
    - 4 = FS write
    - 8 = Spawn apps
- Example: prog load demo 15 "print hello;write file.txt data;exit" creates program with all capabilities
- The interpreter enforces capabilities - a program with capability 1 can only print, not write files

Why it matters: This demonstrates security through capability-based access control, a modern OS security concept. It shows how OS kernels can restrict what user programs can do.

1.5 Synchronization Primitives (Mutex & Semaphore)

What it does: Provides thread synchronization mechanisms for coordinating access to shared resources.

How it works:

- sync.c implements:
    - **Mutex**: Binary lock preventing simultaneous access
    - **Semaphore**: Counter allowing N threads to access resource
- Uses busy-wait + yield for synchronization (no hardware support needed)
- Example app: producer/consumer using mutex + semaphore

Code example (sync.c):

typedef struct {

```
    int locked;
} mutex_t;

int mutex_lock(mutex_t *m) {
    while (m->locked) thread_yield();  // Spin-wait
    m->locked = 1;
    return 0;
}
```

Why it matters: Real-world programs need synchronization to prevent race conditions. This shows we understand concurrent programming challenges.

Part 2: Verification of Success

Testing Methodology

We verified the OS works by:

1. **Compiling without errors:** make completes successfully
2. **Booting in QEMU:** qemu-system-riscv64 -machine virt -nographic -bios none -kernel kernel.bin launches the OS
3. **Interactive testing:** Commands execute as expected
4. **Feature testing:** Each major feature tested individually

Evidence of Success

Shell & Commands Working:

```
$ help
commands: help stop ls run <app> ps kill <tid>
        fs ... (ls/read/write/rm/format)
        prog ... (ls/runall/load/loadfile/save/run/drop)
```

Threading Working:

```
$ run pinger
[app:pinger] ping
$ run counter
[app:counter] 1
$ ps
```

[displays running threads]

Filesystem Working:

$ fs write test.txt "Hello"
fs wrote test.txt
$ fs ls
fs:
 - test.txt (5b)
$ fs read test.txt
Hello
$ fs rm test.txt
$ fs ls
fs:
[file removed]

Script Loading Working:

$ prog load demo 15 "print Hi;write note.txt demo;read note.txt;exit"
$ prog run demo
[script executes with all capabilities]

Known Limitations

- **Cooperative only:** No preemptive multitasking or timer interrupts. Threads must voluntarily yield.
- **Limited file size:** Each file limited to 256 bytes
- **Limited file count:** Maximum 16 files (by design - could be increased)
- **RAM-only:** No persistent storage to disk. Files lost on shutdown.
- **No memory protection:** All code runs in kernel/machine mode. No user/supervisor separation.
- **Limited script language:** Simple interpreter with basic commands

These are acceptable tradeoffs for an educational OS.

Part 3: AI-Assisted Development

Tools Used

- **Codex (ChatGPT), using UofSC GPT EDU subscription, PDF Attached, codex doesnt allow you to download chat logs, so its copy/pasted**

How AI Helped

1. **Initial design:** AI structured all of the OS components (shell, threading, filesystem, etc.)
2. **Code generation:** AI generated it
3. **Debugging:** AI was the main source for identifying issues (e.g., context switch register ordering)
4. **Documentation:** AI is perfect for documentation, especially if you're learning as you go (IE this)
5. **Testing:** AI suggested test cases and verification approaches

What We Learned

- RISC-V assembly is complicated
- I (Cole) learned the basics of how to write in C, and interacted with Assembly. I definitely didn't learn it.
- Context switching requires careful register management
- Cooperative multitasking simpler than preemptive (no timer needed)
- Simple filesystems just need data structures and copy operations
- I (Dallas) figured out how to connect codex to VSCode and have seen how well it works, and will use it to pretty up my pet project Wade's Wine Watcher

Complications Encountered

- Initially I (Dallas) had started developing using ChatGPT on the web. I hadn't realized I was not logged in, and lost all of the logs for this attempt, so when I continued trying work on the same project, I was lost with no documentation and scrapped it all LINK TO CHAT
- We spent a large amount of time trying to use QEMU on the lab machines, which was not possible
- QEMU could run on the lab machine, but the RISCV toolchain would not download, and I was filled the disk multiple times attempting to download it. The following were attempted (and failed)
    - Using sudo apt-install to download RISCV
    - Moving a predownloaded RISCV toolchain over WINscp
    - Cloning the a public RISCV github repo

Part 4: Critical OS Structures in Code

4.1 Interrupt Vector & Exception Handling

Location: entry.S (trap handling assembly)

Purpose: Handles CPU exceptions and system calls

How it works:

```
# When exception occurs, CPU jumps here
_entry:
   # Save all registers to stack
   # Call trap handler in C
   # Restore registers and return
```

Why it matters: Exceptions (including syscalls) are how user code requests kernel services safely.

4.2 Process Control Block (PCB)

Location: thread.c

Purpose: Stores metadata for each thread/process

Structure:

```
typedef struct thread {
   int tid;
   void *(*fn)(void *);
   void *arg;
   uint64_t *sp;
} thread_t;
```

Why it matters: The PCB is how the OS tracks processes. Each running program needs one.

4.3 Context Switch (Scheduler)

Location: context.S and thread.c

Purpose: Saves one thread's state and restores another's

Code:

```
context_switch:
   sd ra, 0(a0)
   sd sp, 8(a0)

   ld ra, 0(a1)
   ld sp, 8(a1)
   ret
```

Why it matters: This is how the OS switches between programs. Without it, only one program could run.

4.4 Filesystem Inode Structure

Location: fs.c

Purpose: Stores file metadata and data

Structure:

```
typedef struct {
   int used;
   char name[FS_NAME_LEN];
   char data[FS_DATA_LEN];
} fs_file;
```

Why it matters: Every filesystem uses inodes or similar structures to store files.

4.5 Synchronization Primitives

Location: sync.c

Purpose: Prevent race conditions in concurrent code

Example - Mutex:

```
typedef struct { int locked; } mutex_t;

int mutex_lock(mutex_t *m) {
   while (m->locked) thread_yield();
   m->locked = 1;
   return 0;
}
```

Why it matters: When multiple threads access shared data, locks prevent corruption.

Part 5: Originality & Code Sources

We searched the web for similar code and found many projects that use the same approaches we did.

Context Switching

What we found: Many RISC-V operating systems do context switching the same way we did. The standard approach is to save and restore registers using assembly code.

Examples:

- **xv6 RISC-V - Context Switch** - A famous educational OS uses almost the same context switch pattern
- **RISC-V Context Switch Discussion** - Engineers discussing the same register save/restore pattern
- **RISC-V Assembly Examples** **-** Educational resource showing context switching

Why: Context switching on RISC-V has a standard approach. You save ra, sp, and s0-s11 registers. Most projects look similar because this is the correct way to do it.

File System with Inodes

What we found: Simple filesystems use an array of inode structures, just like we did. This is a common educational approach.

Examples:

- **University of Notre Dame - Simple File System Project** - Uses an array of inodes with file names and data
- **Operating Systems: Three Easy Pieces** - Textbook that explains inode-based filesystems
- **IIT Bombay - Simple Filesystem Lab** **-** Educational lab with similar filesystem design
- **Simple File System GitHub** - Open source simple filesystem in C

Why: Storing files in an array with names and data is the simplest way to build a filesystem. Many educational projects use this approach.

Mutex Locks

What we found: The basic spinning lock pattern (wait until available, then take it) appears in many OS projects.

Examples:

- **Princeton CS 318 - Mutex Implementation** - University lecture on how mutexes work
- **JMU Computer Systems - Locks** - Explains basic lock concept: while locked, wait; then take the lock
- **GeeksforGeeks - Mutex Lock Tutorial** - Shows practical examples
- **Stack Overflow Discussion** - Engineers discussing mutex implementation approaches

Why: The basic lock pattern (spin-wait, then acquire) is simple and appears everywhere because it works.

Conclusion

The code structures we used (context switching, inodes, locks) are standard OS concepts. They appear in similar forms across many projects because there's usually only one good way to solve each problem. Finding similar code online is expected and not a problem - it shows we're using industry-standard approaches.

Summary

We built a functional RISC-V operating system demonstrating:

- Shell & user interaction
- Dynamic multithreading
- Persistent filesystem
- User program loading with capabilities
- Synchronization primitives
- Proper exception/trap handling

The OS works, runs in QEMU, and demonstrates core OS concepts. All major components are implemented and functional.

Report completed by: Cole Easley, Drew Carter, Ryan Noel, Dallas Wade
Date: 12/5/2025