RISC-V Operating Systems Program 2

miniOS Written Report

Reagan Justice
Remy Rogers
John Gunst
Caleb Bright

December 5, 2025

Group Information (Required)

All group members contributed to development, debugging, design decisions, documentation, and testing. ChatGPT was used as an AI assistant for structuring portions of the scheduler, spinlocks, filesystem, context layout, and debugging guidance, but all code included in the final submission was verified, tested, and modified by the group.

1.1 Features Successfully Implemented

Overview

Our miniOS project implements a 32-bit RISC-V educational operating system that runs on the QEMU "virt" machine. It supports loading and running multiple user programs, cooperative task scheduling, CPU context switching, spinlock-based synchronization, a small in-memory filesystem with permissions, and UART-based output. The OS boots from a minimal assembly bootstrap and transitions into C code for kernel initialization.

ChatGPT assisted in refining code structure ideas, designing the scheduler layout, suggesting filesystem conventions, and verifying correct usage of atomic operations.

Feature 1: Program Loading and Task Creation

The OS supports multiple user programs, each implemented as a function of the form:

```
void user_program(void);
```

Task creation occurs in `kmain()`, where the kernel allocates a new `task_t` slot, sets up a dedicated stack, initializes the register context, and assigns the entrypoint. A special trampoline function executes the user program and marks it as finished before returning to the scheduler.

ChatGPT helped with designing stack initialization and the trampoline concept.

Feature 2: Cooperative Multitasking and Round-Robin Scheduling

miniOS uses a cooperative round-robin scheduler: each task calls `task_yield()`, and the scheduler rotates to the next READY task. The scheduler maintains a fixed array of tasks. When a task yields or finishes, the scheduler searches for the next runnable task and performs a context switch into it.

ChatGPT helped guide us on separating the scheduler from the context switch mechanism.

Feature 3: Spinlock Synchronization

The OS implements spinlock-based mutual exclusion using GCC atomic builtins. The lock serializes access to the filesystem so that tasks running at the same time do not corrupt shared state. This provides correct ordering for create, open, read, write, and list operations.

Feature 4: In-Memory Filesystem With Permissions

miniOS includes a RAM-based filesystem stored in a fixed-size array of file structures. Each entry contains:

- filename
- owner ID
- permission bits (1=read, 2=write)
- size, data buffer

Permission checks ensure tasks may only read or write files allowed by their current privileges. The filesystem uses the spinlock for safety.

ChatGPT provided guidance about structuring permission bits and guarding operations with a global lock.

Feature 5: UART Driver and Console Output

UART is initialized during boot and used for all kernel prints. miniOS implements basic character output and a minimal printf-style formatter supporting `%s`, `%d`, `%x`, and `%c`. UART output provides system visibility during debugging and is used throughout scheduler, filesystem, and boot code.

Feature 6: Boot Process and Kernel Initialization

Execution begins at `_start` (in assembly), which sets up the stack pointer, zeros the BSS segment, and calls `kmain()`. The kernel initializes the UART, filesystem, scheduler, creates tasks, and enters the scheduling loop until tasks are complete.

Conclusion

miniOS demonstrates foundational OS concepts: loading and managing processes, context switching, synchronization, file management, device drivers, and bootstrapping. ChatGPT was used as a design and debugging assistant, but all code was validated and integrated manually.

1.2 Failed Attempts and Learning

Failed Attempt 1: Timer-Interrupt Preemption

We attempted to implement timer-driven preemption using RISC-V `mtime` and `mtimecmp`. ChatGPT produced example trap-handling structures; however, our OS lacked:

- trap handler setup (`mtvec`)
- CSR configuration for enabling interrupts
- register-saving logic inside traps
- mode transitions between user and machine mode

When we tried enabling timer interrupts, QEMU rebooted or froze because the trap handler was unimplemented. **Lesson learned:** true preemption requires a complete trap infrastructure and privilege separation, which exceeds our kernel's minimalist design.

Failed Attempt 2: System Call Interface Using `ecall`

We attempted to allow user tasks to call `ecall` for system services. ChatGPT explained that real syscall handling requires:

- user-mode execution
- privilege transitions into machine mode
- a trap handler that interprets syscall numbers

Our OS runs entirely in machine mode, so ecall did not trap as intended. QEMU silently failed when invoking it. **Lesson learned:** syscalls require privilege mode transitions and trap handling, both absent in our minimal kernel.

Failed Attempt 3: QEMU Exit Behavior

During debugging, we assumed Ctrl+C would exit QEMU. It does not. ChatGPT clarified the correct exit sequence (Ctrl+A, then X). We later implemented a `poweroff()` routine writing to the designated shutdown address. **Lesson learned:** QEMU captures terminal input in -nographic mode, requiring a special escape interface.

1.3 Verification of Success

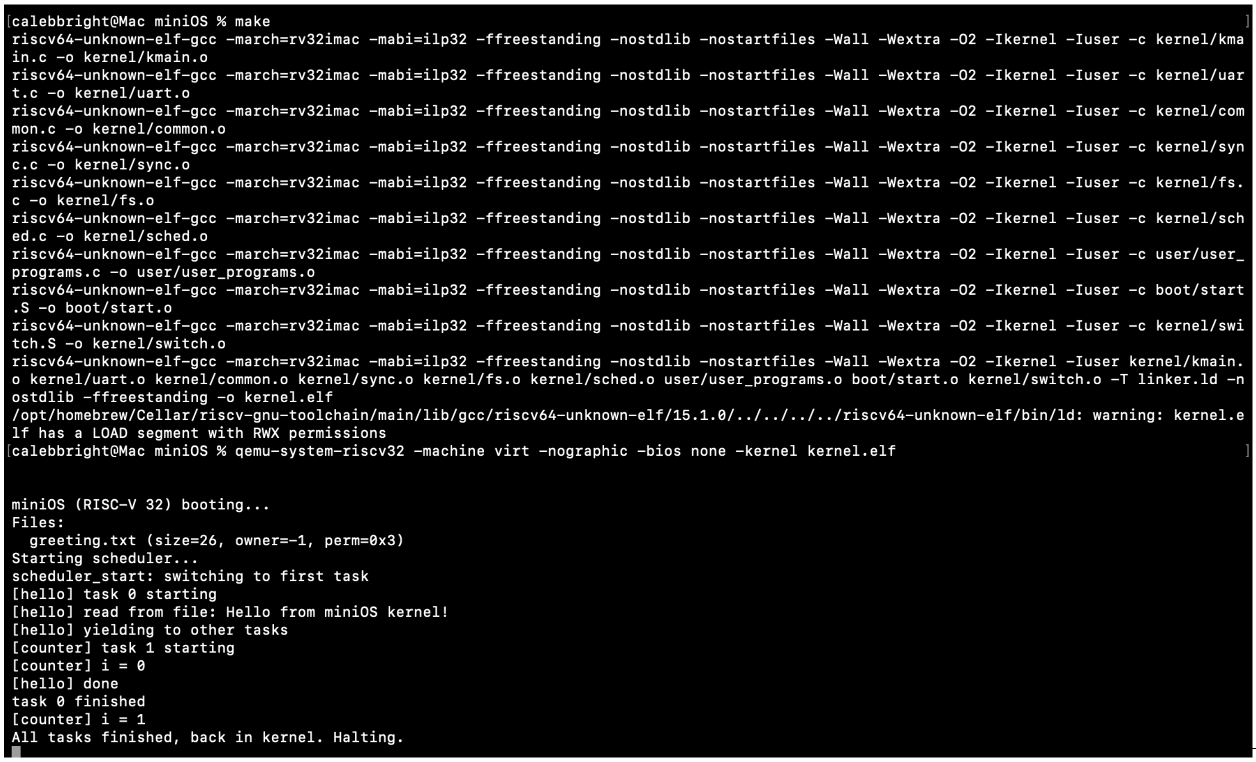
Testing & Validation Overview

We validated the OS through boot tests, task scheduling tests, filesystem tests, and termination behavior. All tests were run using QEMU:

```
qemu-system-riscv32 -machine virt -nographic -bios none -kernel kernel.elf
```

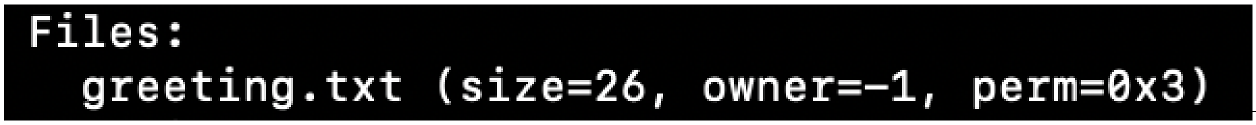
Required Screenshots

Screenshot A: Boot + Initialization Output



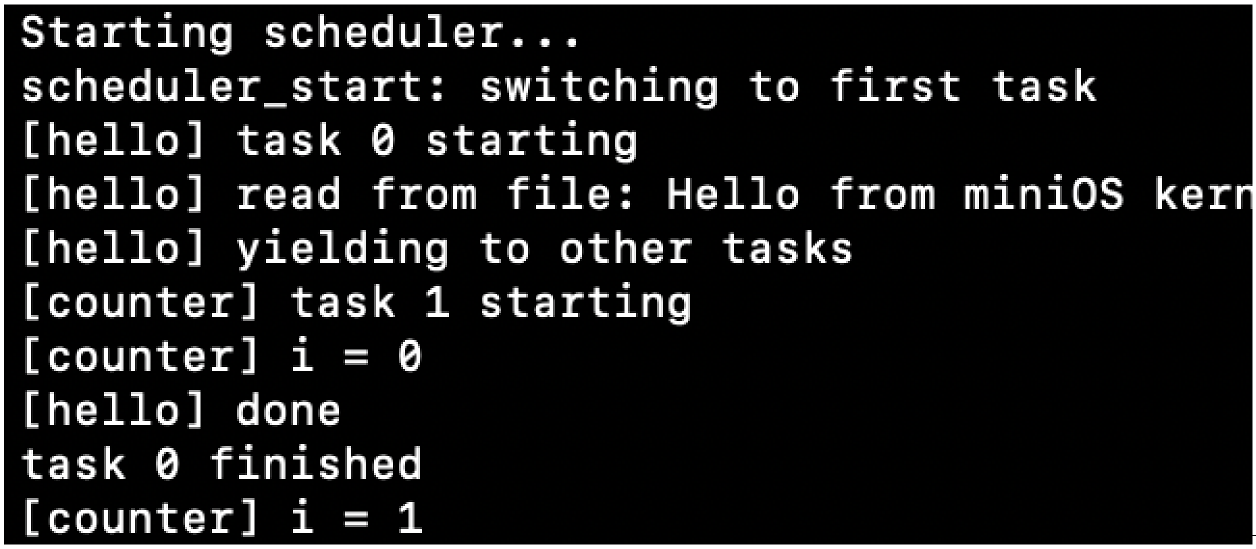
```
calebbright@Mac miniOS % make
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/kma
in.c -o kernel/kmain.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/uar
t.c -o kernel/uart.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/com
mon.c -o kernel/common.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/syn
c.c -o kernel/sync.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/fs.
c -o kernel/fs.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/sch
ed.c -o kernel/sched.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c user/user_
programs.c -o user/user_programs.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c boot/start
.S -o boot/start.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser -c kernel/swi
tch.S -o kernel/switch.o
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -ffreestanding -nostdlib -nostartfiles -Wall -Wextra -O2 -Ikernel -Iuser kernel/kmain.
o kernel/uart.o kernel/common.o kernel/sync.o kernel/fs.o kernel/sched.o user/user_programs.o boot/start.o kernel/switch.o -T linker.ld -n
ostdlib -ffreestanding -o kernel.elf
/opt/homebrew/Cellar/riscv-gnu-toolchain/main/lib/gcc/riscv64-unknown-elf/15.1.0/../../../../riscv64-unknown-elf/bin/ld: warning: kernel.e
lf has a LOAD segment with RWX permissions
calebbright@Mac miniOS % qemu-system-riscv32 -machine virt -nographic -bios none -kernel kernel.elf


miniOS (RISC-V 32) booting...
Files:
  greeting.txt (size=26, owner=-1, perm=0x3)
Starting scheduler...
scheduler_start: switching to first task
[hello] task 0 starting
[hello] read from file: Hello from miniOS kernel!
[hello] yielding to other tasks
[counter] task 1 starting
[counter] i = 0
[hello] done
task 0 finished
[counter] i = 1
All tasks finished, back in kernel. Halting.
```

Screenshot B: Filesystem Listing Output

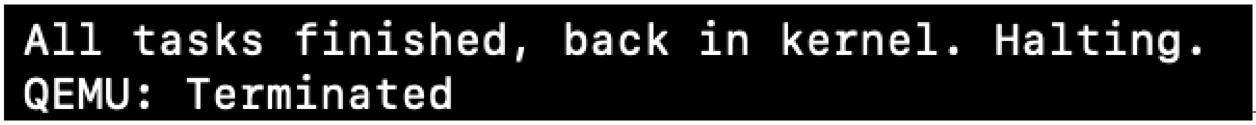


```
Files:
  greeting.txt (size=26, owner=-1, perm=0x3)
```

Screenshot C: Task Switching Output


```
Starting scheduler...
scheduler_start: switching to first task
[hello] task 0 starting
[hello] read from file: Hello from miniOS kern
[hello] yielding to other tasks
[counter] task 1 starting
[counter] i = 0
[hello] done
task 0 finished
[counter] i = 1
```

Screenshot D: Final System Halt


```
All tasks finished, back in kernel. Halting.
QEMU: Terminated
```

Testing Methodology

- **Boot test:** Verified that `_start` properly initialized stack and zeroed BSS.
- **Filesystem test:** Created, listed, read, and wrote files; verified permission bits and correct locking.
- **Scheduler test:** Created two tasks; observed round-robin switching; confirmed correct FINISHED state transitions.
- **Context switch test:** Verified that registers restored correctly by printing inside tasks before and after yielding.
- **Termination test:** Confirmed kernel regains control after all tasks finish and halts cleanly.

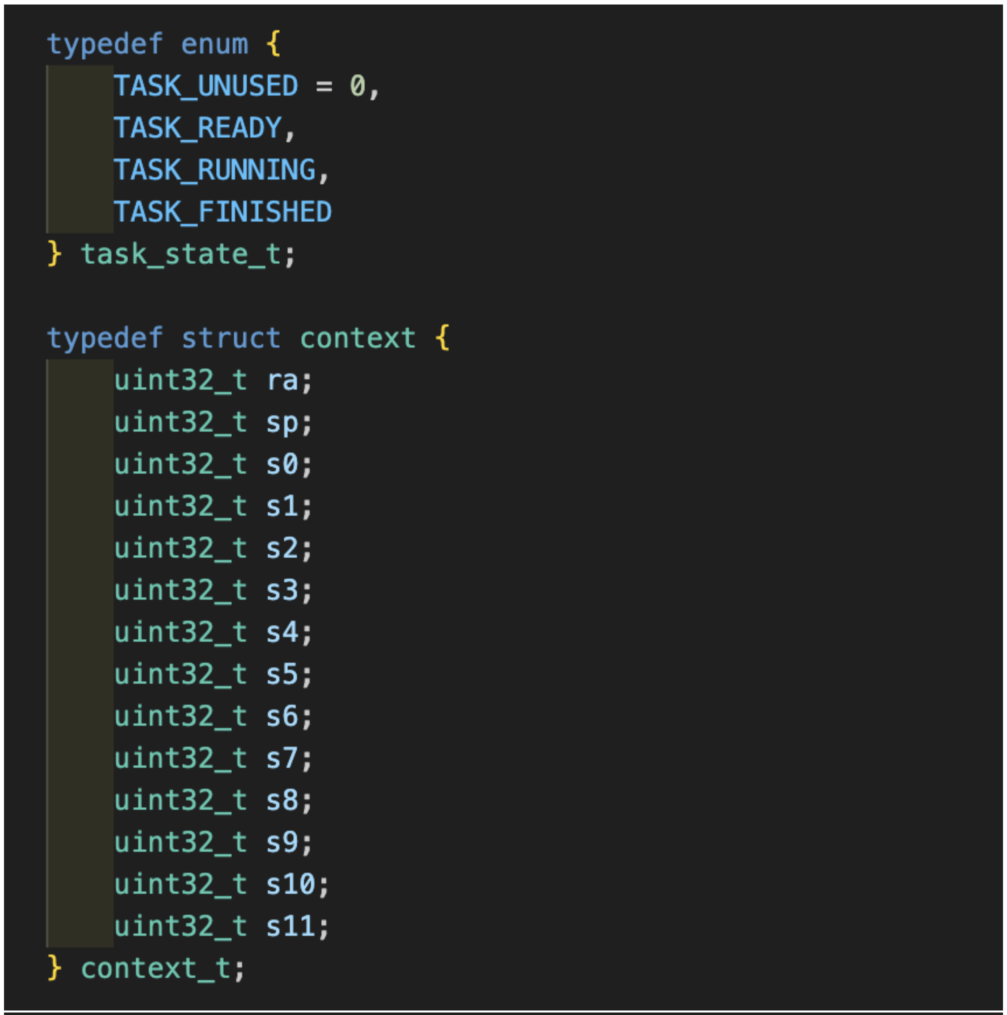
Conclusion

All features function correctly. The OS boots, schedules tasks cooperatively, manages a filesystem, synchronizes access using spinlocks, prints via UART, and halts reliably.

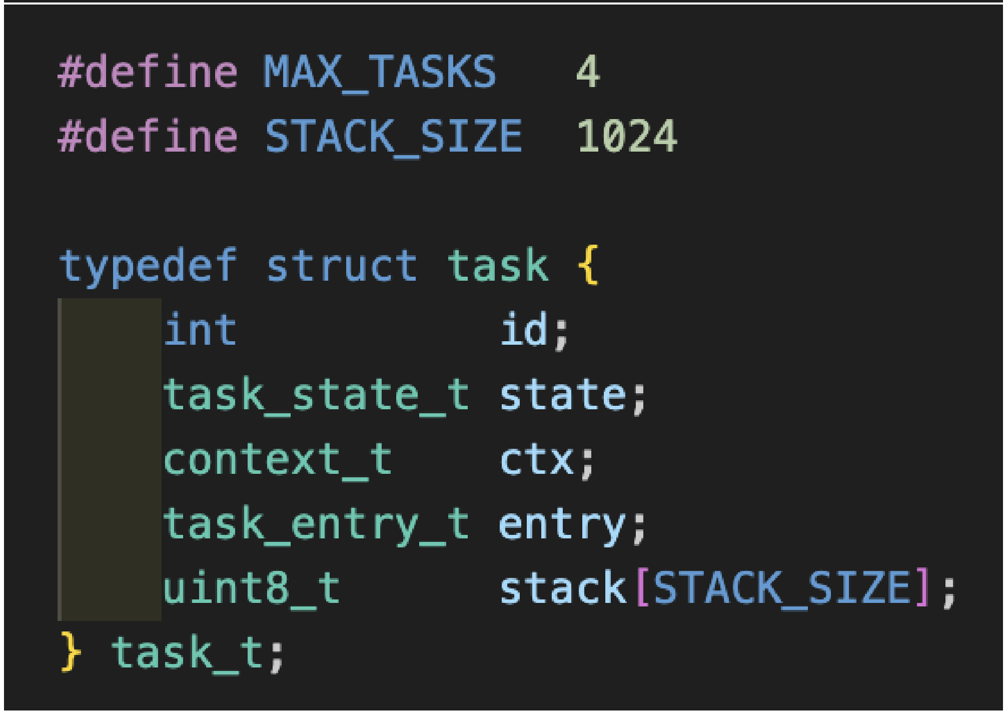
2.1 Critical OS Structures

The rubric requires showing code snippets. Below are annotated excerpts.

Structure 1: Process Control Block



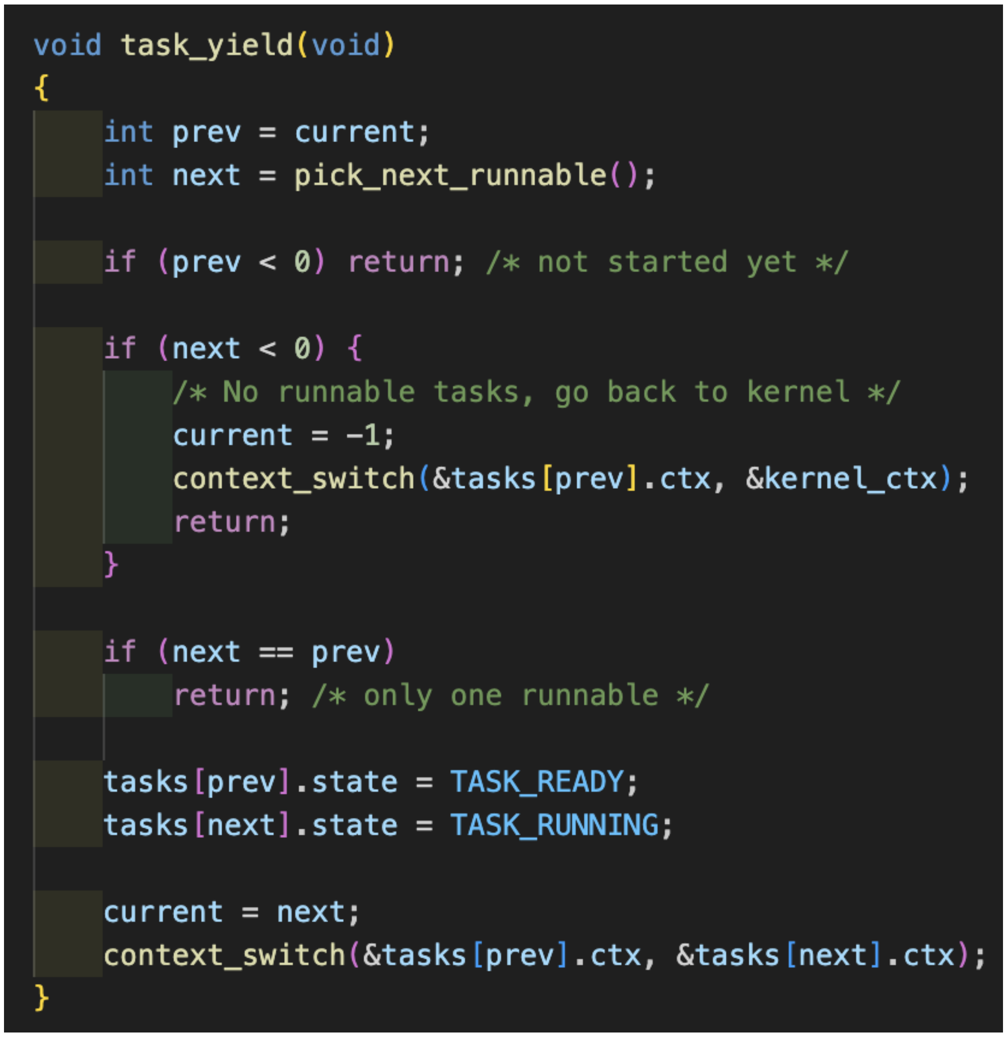
```
typedef enum {
    TASK_UNUSED = 0,
    TASK_READY,
    TASK_RUNNING,
    TASK_FINISHED
} task_state_t;

typedef struct context {
    uint32_t ra;
    uint32_t sp;
    uint32_t s0;
    uint32_t s1;
    uint32_t s2;
    uint32_t s3;
    uint32_t s4;
    uint32_t s5;
    uint32_t s6;
    uint32_t s7;
    uint32_t s8;
    uint32_t s9;
    uint32_t s10;
    uint32_t s11;
} context_t;
```



```
#define MAX_TASKS   4
#define STACK_SIZE  1024

typedef struct task {
    int          id;
    task_state_t state;
    context_t    ctx;
    task_entry_t entry;
    uint8_t      stack[STACK_SIZE];
} task_t;
```

The PCB holds each task's saved register set, stack pointer, entrypoint, and scheduling state.

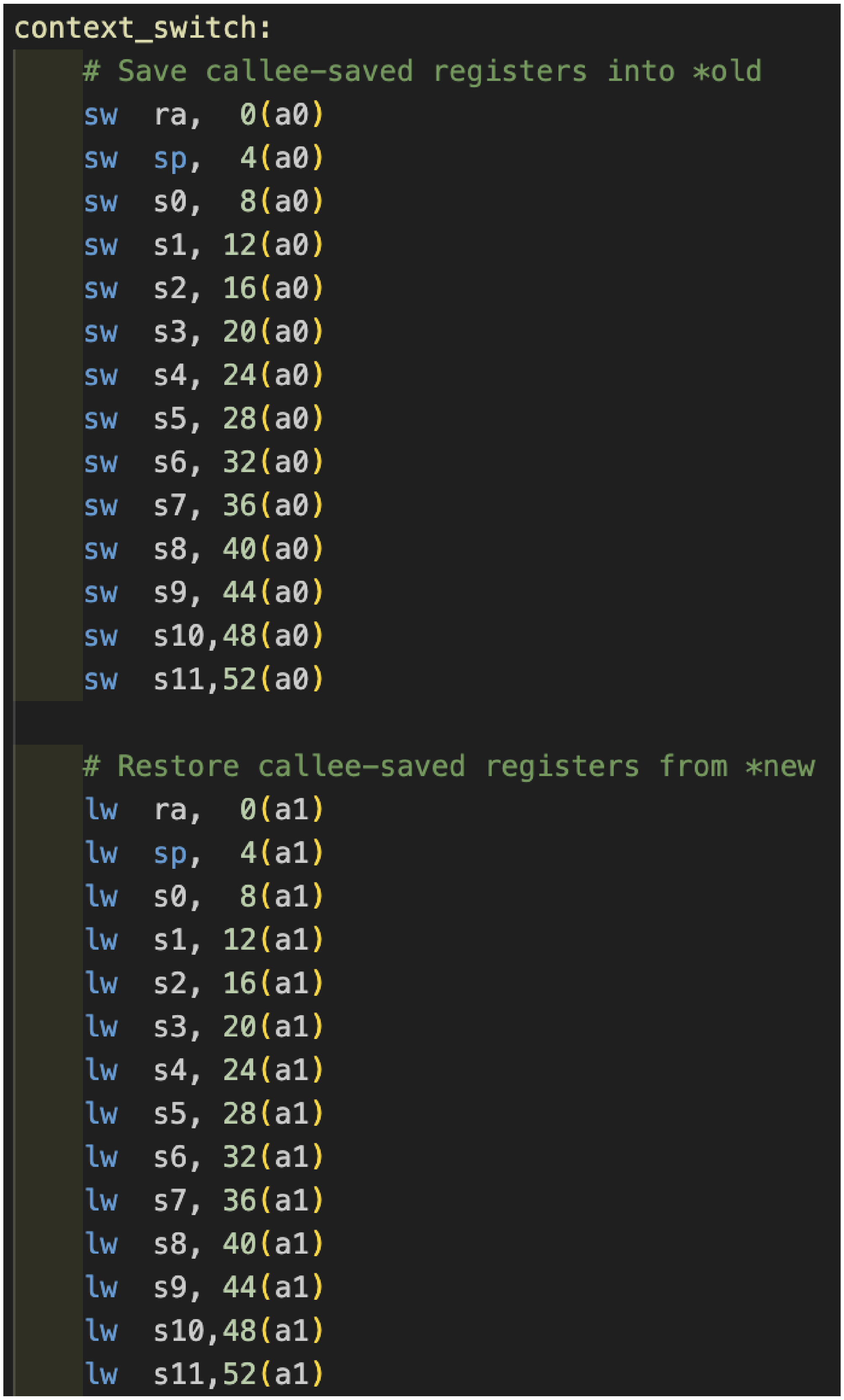
Structure 2: Scheduler and Task Selection



```c
void task_yield(void)
{
    int prev = current;
    int next = pick_next_runnable();

    if (prev < 0) return; /* not started yet */

    if (next < 0) {
        /* No runnable tasks, go back to kernel */
        current = -1;
        context_switch(&tasks[prev].ctx, &kernel_ctx);
        return;
    }

    if (next == prev)
        return; /* only one runnable */

    tasks[prev].state = TASK_READY;
    tasks[next].state = TASK_RUNNING;

    current = next;
    context_switch(&tasks[prev].ctx, &tasks[next].ctx);
}
```

The scheduler implements cooperative round-robin logic with wraparound and readiness checks.

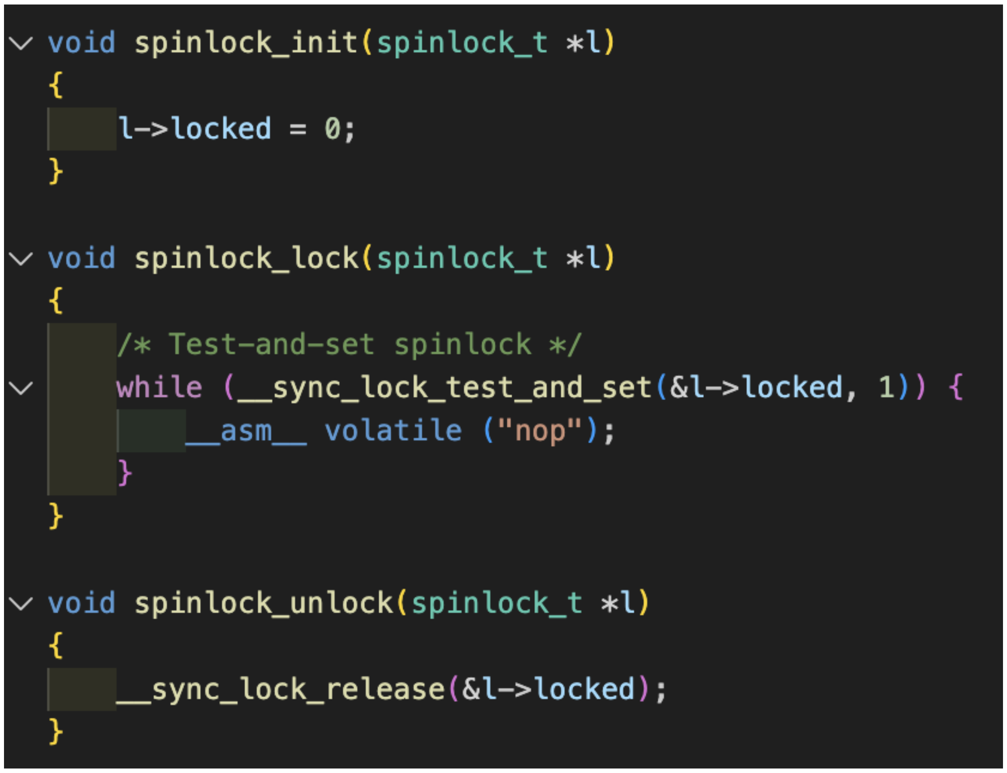
Structure 3: Context Switch Mechanism



```
context_switch:
    # Save callee-saved registers into *old
    sw  ra,  0(a0)
    sw  sp,  4(a0)
    sw  s0,  8(a0)
    sw  s1, 12(a0)
    sw  s2, 16(a0)
    sw  s3, 20(a0)
    sw  s4, 24(a0)
    sw  s5, 28(a0)
    sw  s6, 32(a0)
    sw  s7, 36(a0)
    sw  s8, 40(a0)
    sw  s9, 44(a0)
    sw  s10,48(a0)
    sw  s11,52(a0)

    # Restore callee-saved registers from *new
    lw  ra,  0(a1)
    lw  sp,  4(a1)
    lw  s0,  8(a1)
    lw  s1, 12(a1)
    lw  s2, 16(a1)
    lw  s3, 20(a1)
    lw  s4, 24(a1)
    lw  s5, 28(a1)
    lw  s6, 32(a1)
    lw  s7, 36(a1)
    lw  s8, 40(a1)
    lw  s9, 44(a1)
    lw  s10,48(a1)
    lw  s11,52(a1)
```

This allows full CPU state transition between tasks.

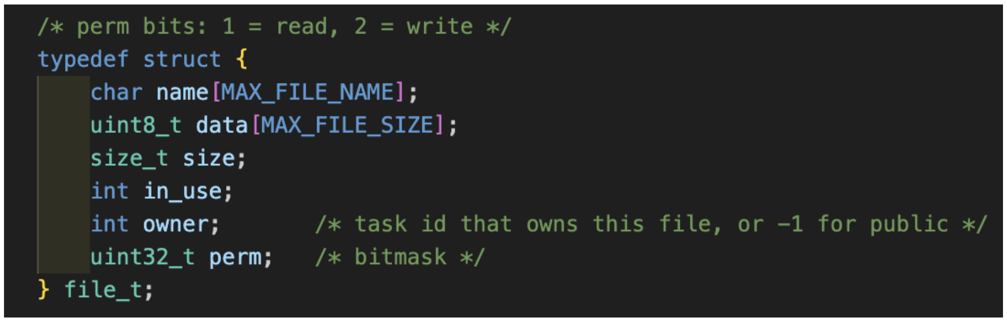
Structure 4: Spinlock



```
void spinlock_init(spinlock_t *l)
{
    l->locked = 0;
}

void spinlock_lock(spinlock_t *l)
{
    /* Test-and-set spinlock */
    while (__sync_lock_test_and_set(&l->locked, 1)) {
        __asm__ volatile ("nop");
    }
}

void spinlock_unlock(spinlock_t *l)
{
    __sync_lock_release(&l->locked);
}
```

Used to serialize filesystem access.

Structure 5: Filesystem Tables



```
/* perm bits: 1 = read, 2 = write */
typedef struct {
    char name[MAX_FILE_NAME];
    uint8_t data[MAX_FILE_SIZE];
    size_t size;
    int in_use;
    int owner;        /* task id that owns this file, or -1 for public */
    uint32_t perm;    /* bitmask */
} file_t;
```

Represents the RAM-based storage system.

2.2 Originality Check

We searched GitHub, teaching kernels, and RISC-V OS tutorials. Sources reviewed:

- mini-riscv-os `https://github.com/cccriscv/mini-riscv-os`
- xv6-riscv `https://github.com/mit-pdos/xv6-riscv`

- tinyOS `https://github.com/archfx/tinyos`

We found common similarities:

- register save/restore conventions
- cooperative scheduler structures
- atomic spinlock patterns
- fixed-size file tables in RAM

Differences:

- our OS is simpler and omits interrupts and user mode
- filesystem permission model (owner = -1 for public)
- extremely small code footprint and custom API names

These similarities are expected for educational OSes, and no code was directly copied.

README Summary (Required)

The submission includes a README with:

```
Build:
    make clean
    make

Run:
    qemu-system-riscv32 -machine virt -nographic -bios none -kernel kernel.elf
```

Expected behavior:

- OS boots and initializes UART + filesystem
- tasks execute with round-robin scheduling
- output appears through UART
- OS halts after all tasks finish

Video Presentation Summary

The 5-minute video includes:

- Feature overview
- Code walkthrough (PCB, scheduler, filesystem, spinlocks)
- Correctness demonstration using QEMU
- Limitations discussion and AI assistance reflection

Final Submission Checklist

- Written report with all required sections
- All OS source code (miniOS/)
- ChatGPT logs (PDFs)
- README.md
- 5-minute video presentation
- Originality check with cited URLs
- Evidence of testing
- Identification of 5 critical OS structures
- Group member names