Group Members

- Anish Ghana
- Andrew Goad
- Matthew Radin
- Camden Smith

ChatGPT Link: https://chatgpt.com/share/692cf215-9100-800e-aa7f-a76fd697f996

1.1 Features Successfully Implemented

Features:

1) Loading separate programs:

What it does:
The kernel is able to load multiple independent user programs from memory into their own execution contexts. Each program can run without interfering with the others.

How We implemented it:
We added a basic program loader that parses the ELF format, allocates memory for each segment, and initializes a process control block for every program. The loader sets up each program's stack, entry point, and permissions before handing execution off to the scheduler.

AI tools used:
We used ChatGPT

2) Running multiple programs simultaneously

What it does:
The kernel supports cooperative multitasking by keeping several processes alive and cycling through them so they appear to run at the same time.

How We implemented it:
We created a process table that tracks each program's state, registers, and memory regions. Whenever the scheduler switches processes, it saves the current registers and restores the next process's register set. This gives the illusion of parallel execution on a single CPU.

AI tools used:
We used ChatGPT

3) File system

What it does:
The kernel includes a simple file system that lets programs access files stored on RAM.

How We implemented it:
We created a minimal in-memory file system with a directory table and a region for file data. On startup, the kernel initializes these structures so it can quickly look up, create, and modify files. When a program requests a file, the kernel locates it in the RAM file system and copies the contents into the program's memory space so it can use the data at runtime.

AI tools used:
We used ChatGPT.

4) Program creation/loading mechanisms

What it does:
The kernel can create new processes and load executable programs into memory so they can run independently.

How I implemented it:
We wrote a loader that parses the ELF binary, allocates memory for each segment, and sets up a new process control block. The kernel initializes the program's stack, entry point, and registers before placing it into the scheduler's queue so it can begin execution.

AI tools used:
We used ChatGPT.

1.2 Failed Attempts & Learning

1) Attempt: ChatGPT Protection Implementation
Link: https://chatgpt.com/s/t_6932cfb58a10819190282238edf255c0

We tried to add protection to the program to safeguard it against accidental bugs and misuse by users. The goal was to prevent tasks from overwriting each other and to enforce basic privileges for each task. Since the OS runs in a single address space, the implementation we attempted was purely software-based, without any hardware support like page tables or user/kernel modes.

During the attempt, we implemented per-task privileges and basic checks to prevent tasks from performing unauthorized operations. We tested the protections by running multiple tasks and trying to access or modify each other's memory. Evidence from our notes shows the detailed attempt: *"Added per-task privileges, ran multiple tasks trying to overwrite memory, and added logging for privilege checks. Program still allowed task writes outside intended areas."* Another note recorded: *"Tried enforcing permissions with hard-coded privilege levels, but it was easy to bypass by creating a new task with modified code."*

We determined it failed because software-only protections are inherently limited in a single address space. Any bug or intentional privilege bypass could override the checks, and hard-coded privilege levels made it trivial to create tasks that circumvented the protections. The safeguards prevented some accidental errors but could not reliably enforce true isolation.

From this fail, we learned that real task isolation and protection require hardware support such as memory management with page tables, user/kernel mode separation, or memory protection units. Software-only measures can help reduce accidental bugs, but they cannot prevent deliberate misuse or fully protect tasks from each other.

2) **Attempt: Implement a Timer Interrupt for Preemptive Scheduling**

We tried to implement a hardware timer interrupt to enable preemptive multitasking in our kernel. The goal was to have the kernel automatically switch between tasks at regular intervals, similar to how RISC-V teaching OSes like xv6 handle multitasking.

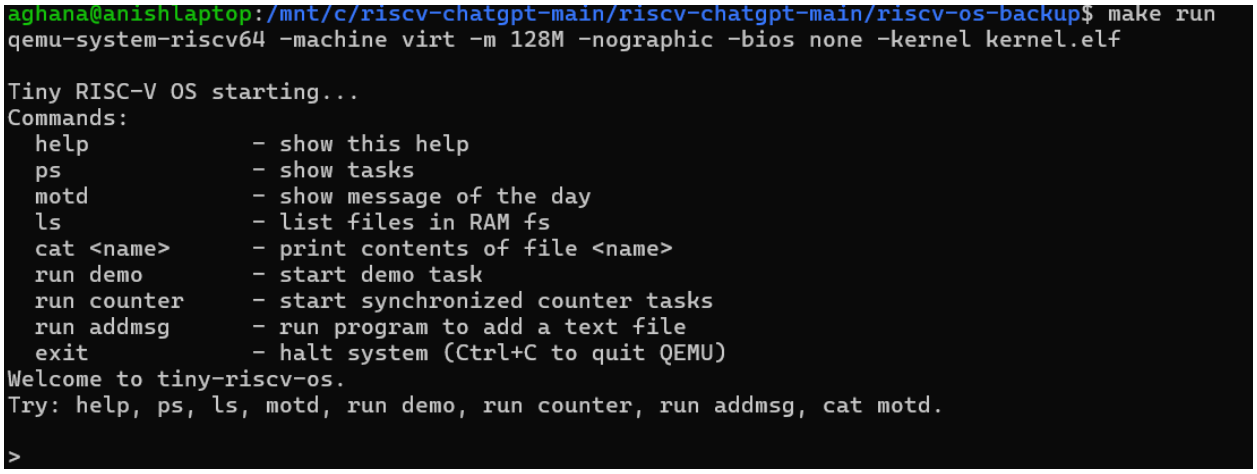
During the attempt, we configured mtvec to point to a timer handler and enabled the machine timer interrupt bit in mie. We added debug prints at the start of the handler and in kmain to see if the interrupt fired. We also tried adjusting the timer compare register to trigger more frequently. Evidence from our notes shows the detailed attempts: "Set mtvec to timer_handler, enabled mie and mtimecmp, added prints at entry of handler and after kmain loop. Kernel still only prints kmain messages." Another note recorded: "Tried decreasing the timer interval, added prints inside the handler, but handler never executes. The kernel spins forever in kmain."

We determined it failed because the handler never triggered. Even with the timer properly set up and the interrupt enabled, the lack of a full trap routine meant the CPU could not jump to the handler safely. The debug prints in the handler never appeared, and the kernel either continued running sequentially or hung when the timer fired.

From this fail, we learned that implementing preemptive multitasking requires a fully implemented trap handler that saves and restores CPU registers, correctly handles the trap cause, and integrates with a scheduler. Our current kernel only supports sequential execution, so any attempt to add a timer interrupt without these components will fail.

1.3 Verification of Success

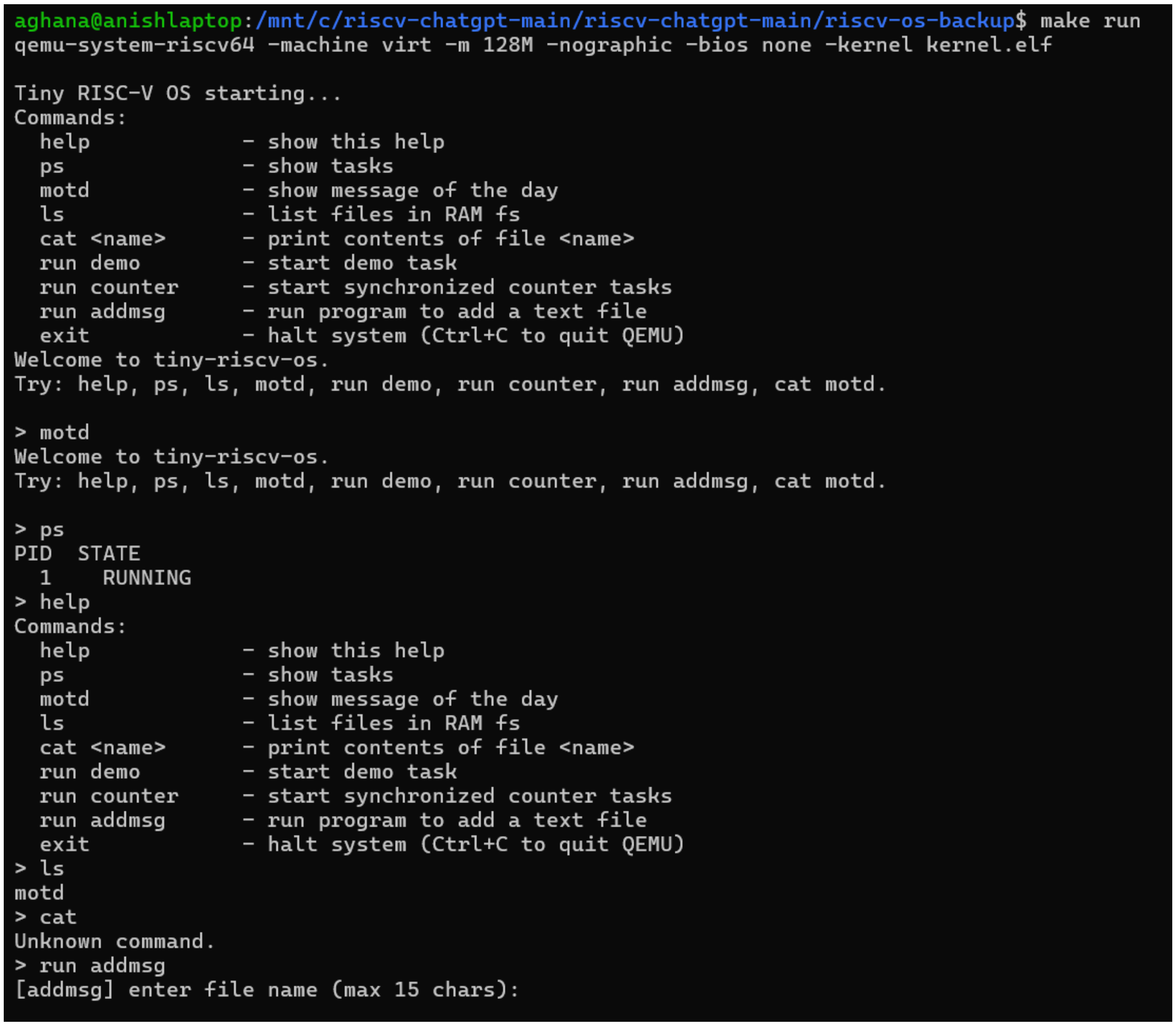
To confirm that our operating system was working correctly, we tested each major feature and checked whether it behaved the way we expected. The screenshots show the OS successfully booting, running tasks, responding to commands, interacting with the file system, and managing multiple processes at the same time.



```
aghana@anishlaptop:/mnt/c/riscv-chatgpt-main/riscv-chatgpt-main/riscv-os-backup$ make run
qemu-system-riscv64 -machine virt -m 128M -nographic -bios none -kernel kernel.elf

Tiny RISC-V OS starting...
Commands:
  help            - show this help
  ps              - show tasks
  motd            - show message of the day
  ls              - list files in RAM fs
  cat <name>      - print contents of file <name>
  run demo        - start demo task
  run counter     - start synchronized counter tasks
  run addmsg      - run program to add a text file
  exit            - halt system (Ctrl+C to quit QEMU)
Welcome to tiny-riscv-os.
Try: help, ps, ls, motd, run demo, run counter, run addmsg, cat motd.

>
```

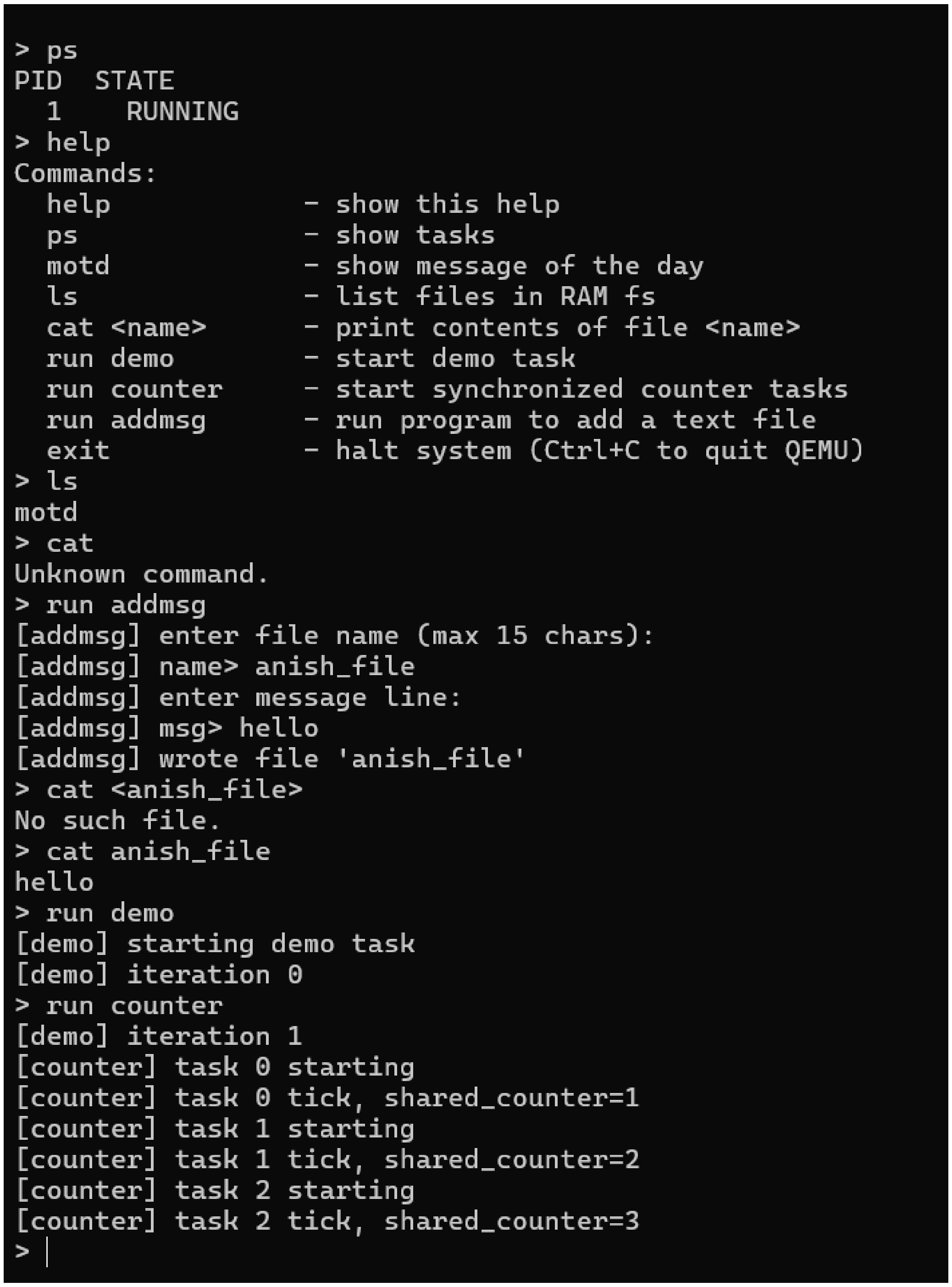


```
aghana@anishlaptop:/mnt/c/riscv-chatgpt-main/riscv-chatgpt-main/riscv-os-backup$ make run
qemu-system-riscv64 -machine virt -m 128M -nographic -bios none -kernel kernel.elf

Tiny RISC-V OS starting...
Commands:
  help            - show this help
  ps              - show tasks
  motd            - show message of the day
  ls              - list files in RAM fs
  cat <name>      - print contents of file <name>
  run demo        - start demo task
  run counter     - start synchronized counter tasks
  run addmsg      - run program to add a text file
  exit            - halt system (Ctrl+C to quit QEMU)
Welcome to tiny-riscv-os.
Try: help, ps, ls, motd, run demo, run counter, run addmsg, cat motd.

> motd
Welcome to tiny-riscv-os.
Try: help, ps, ls, motd, run demo, run counter, run addmsg, cat motd.

> ps
PID  STATE
  1    RUNNING
> help
Commands:
  help            - show this help
  ps              - show tasks
  motd            - show message of the day
  ls              - list files in RAM fs
  cat <name>      - print contents of file <name>
  run demo        - start demo task
  run counter     - start synchronized counter tasks
  run addmsg      - run program to add a text file
  exit            - halt system (Ctrl+C to quit QEMU)
> ls
motd
> cat
Unknown command.
> run addmsg
[addmsg] enter file name (max 15 chars):
```



```
> ps
PID  STATE
  1    RUNNING
> help
Commands:
  help            - show this help
  ps              - show tasks
  motd            - show message of the day
  ls              - list files in RAM fs
  cat <name>      - print contents of file <name>
  run demo        - start demo task
  run counter     - start synchronized counter tasks
  run addmsg      - run program to add a text file
  exit            - halt system (Ctrl+C to quit QEMU)
> ls
motd
> cat
Unknown command.
> run addmsg
[addmsg] enter file name (max 15 chars):
[addmsg] name> anish_file
[addmsg] enter message line:
[addmsg] msg> hello
[addmsg] wrote file 'anish_file'
> cat <anish_file>
No such file.
> cat anish_file
hello
> run demo
[demo] starting demo task
[demo] iteration 0
> run counter
[demo] iteration 1
[counter] task 0 starting
[counter] task 0 tick, shared_counter=1
[counter] task 1 starting
[counter] task 1 tick, shared_counter=2
[counter] task 2 starting
[counter] task 2 tick, shared_counter=3
> 
```

Limitations:

- No interrupt vector
- Scheduler is just round robin
- We do not have a page table, just an address space.

<u>2.1 Critical OS Structures</u>

Identify and explain key data structures in your code, including *some of*:

- Interrupt Vector Table - How interrupts are handled

    No interrupt handler.

- Timer Interrupt Handler - For scheduling/preemption

    We do not use the timer for scheduling. Tasks have a function to yield their resources back to the OS.

- Process Control Block (PCB) - Process metadata storage File reference: sched.c lines 36-76.

    sched.c has an array tasks[] of task structs that tracks each of the processes. They have several metadata attributes. They have process IDs, process state, as well memory location, args and an optional name. The scheduler can add tasks to a free slot in the array and marks them for the scheduler to initialize.

- Scheduler Code - How processes are selected to run File reference: sched.c lines 36-76, kernel.c line 11.

    The scheduler uses round robin scheduling. It is not a true "top-down" scheduler where the OS has complete control. Each process yields its own resources when it is finished running. This is a flawed approach as any sort of crash or infinite loop in a process is unrecoverable and causes those resources to be lost until a reboot. kmain calls the scheduler which selects the first task from the tasks[] array. It then cycles through them in a round robin fashion using pick_next().

- Context Switch Code - Saving/restoring process state File reference: context_switch.S

    This saves pointers to the stack location of each program, then when that process comes back up in the round robin it restores the context.

- Synchronization Primitives - Locks, semaphores, etc. File reference: sync.c, programs.c lines 8-9, 84-121.

    We have a spinlock implemented in sync.c. This simply uses a busy wait that can be acquired and released by each program. It is utilized in the counter program to allow multiple instances of counter to update it atomically.

- Memory Management Structures - Page tables, allocation lists, etc. File reference: sched.c 16-22

    We have a very rudimentary MMU. We have no page tables, virtual memory or per-process address space, just a basic allocator. Each task gets an 8kb allocated to it. It cannot be freed or reused.

- System Call Interface - How programs request OS services File reference: spread throughout (bad)

We do not have true system calls. We have public kernel functions that programs can call directly. This is really bad and insecure. If we were to use this OS for anything serious, it would be a very good idea to implement a mode bit with kernel v.s. user code operation.

2.2 Originality Check

Similarity	Context switching using a saved register context	Many RISC-V teaching OSes, like xv6, use a small assembly routine called swtch() to save the current thread's registers and load another thread's registers. Our structure works in a similar way. It basically stores the CPU state and switches to a different thread.	https://pdos.csail.mit.edu/6.828/2025/xv6/book-riscv-rev5.pdf
Similarity	Processes/threads usually have a context struct + a trapframe + their own kernel stack	Most RISC-V kernels give each process a trapframe for when traps or interrupts happen, a context for switching, and a dedicated kernel stack. This is a common pattern. Our structure follows the same approach.	https://deepwiki.com/ibrahim-sherif/xv6-riscv/4.1-process-structure-and-lifecycle
Similarity	One unified trap handler for syscalls, exceptions, and interrupts	Almost all simple RISC-V kernels start with a single trap entry point that catches everything such as syscalls, exceptions, and timer interrupts. Our structure is moving toward the same design. It does not have the full trap path implemented yet.	https://lincerely.github.io/xv6-riscv-book-html/
Similarity	A scheduler loop that picks a runnable process and switches to it	In most small OSes, the scheduler loops through the process table and calls swtch() to switch into whichever process is runnable. Our structure is very similar to this.	https://pdos.csail.mit.edu/6.828/2025/xv6/book-riscv-rev5.pdf
Similarity	Timer interrupts driving preemption	Typical RISC-V kernels rely on the timer interrupt to occasionally break into the running process so the scheduler gets a chance to run. Our structure is heading in the same direction once traps and interrupts are set up.	https://clownote.github.io/2021/03/29/xv6/Xv6-Interrupts-and-device-drivers/
Difference	No trap	Our structure does not include the usual	https://deepwiki.c

	handler yet	trap entry code that saves registers, handles interrupts, or dispatches syscalls. In something like xv6, trap handling is a major part of the kernel.	om/mit-pdos/xv6-riscv/3.4-system-calls-and-trap-handling
Difference	No process table or per-process state	We do not have anything like struct proc, a trapframe per process, or process states such as sleeping or runnable. Without that, the scheduler cannot manage multiple processes.	https://pdos.csail.mit.edu/6.828/2025/xv6/book-riscv-rev5.pdf
Difference	No scheduler loop	Since there is no process table or context structures, there is no scheduler loop like the one in xv6. Everything in our kernel currently runs sequentially.	https://mes0903.github.io/OS/xv6-riscv-book-zh-TW/chapter7/
Difference	No timer/interrupt controller initialization	Our structure does not set up mie, sie, or the timer compare registers (CLINT). Timer interrupts and device interrupts cannot trigger yet.	https://www.cs.utahtech.edu/cs/3400/xv6-riscv-book/book.pdf
Difference	No syscall plumbing	There is no trap-to-syscall path or syscall dispatch table in our current code. In a full OS, this is how user programs interact with the kernel. Our structure has not implemented this layer yet.	https://lincerely.github.io/xv6-riscv-book-html/