CSE 536: Advanced Operating Systems Portfolio

Venkata Swaraj Kotapati
vkotapat@asu.edu
Arizona State University, Tempe, AZ

***Abstract*—Operating systems serve as a vital link between user applications and hardware, fulfilling diverse roles such as facilitating user interaction, ensuring uninterrupted application usage, optimizing system speed, and maintaining overall stability. This course comprehensively covers the multifaceted functions of operating systems, offering hands-on experience with the xv6 operating system through purposeful assignments.**

I. Introduction

This project portfolio encompasses a series of advanced operating system enhancements, exploring diverse aspects of operating system design and functionality. Four distinct assignments have been undertaken to address specific challenges and introduce novel features to the xv6 operating system.

Collectively, these four assignments showcase a progressive journey of enhancing xv6, covering bootloader development, memory management, multitasking, and virtualization. The portfolio demonstrates a comprehensive understanding of operating system internals, addressing critical challenges to create a more robust and feature-rich xv6 operating system.

II. Solution

Assignment 1 : The initial assignment focuses on understanding the boot process and custom bootloader development in the xv6 operating system. Through a step-by-step approach, the bootloader is crafted, incorporating essential elements such as linker scripts, stack setup, dynamic kernel loading, and RISC-V Physical Memory Protection (PMP). The assignment delves into the intricacies of boot ROM execution, ensuring a robust foundation for subsequent enhancements. Following are the task breakdowns for this assignment.

Inspecting QEMU Boot ROM Execution: This phase involved leveraging GDB for a comprehensive analysis of the Boot ROM execution. By executing *./run gdb* to run xv6 in debug mode and launching *riscv64-unknown-elf-gdb* in a separate terminal window, we aimed to identify the loading address, meticulously scrutinize execution steps, and determine the final jump address after execution.

Writing a Bootloader Linker Script and Booting into Assembly: The subsequent step focused on defining a robust linker descriptor in bootloader/bootloader.ld. This descriptor specified the starting address and meticulously organized bootloader sections such as .text, .data, .rodata, and .bss. Additionally, we set up the entry function and confirmed correct execution by verifying the bootloader's entry function using GDB.

Setting Up a Stack for C Code: To facilitate seamless execution of C code, we configured a stack. The stack address was loaded into the stack pointer register to ensure the proper execution of C functions after *_entry*.

Loading User-Selected OS Kernels: Functionality was implemented to dynamically load various OS kernels. This involved determining the kernel load address using ELF headers, copying the kernel binary to the specified address (*kernload-start*), executing the OS kernel, and passing essential system information.

Setting Up the RISC-V PMP Feature: The focus shifted to configuring RISC-V Physical Memory Protection (PMP) to enable memory access at S-mode. TOR and NAPOT configurations were implemented to effectively isolate memory regions and govern access. The TOR configuration isolated the upper 11 MBs, while the NAPOT configuration isolated regions 118-120 MB and 122-126 MB.

Enabling Secure Boot: Secure boot functionality was integrated to ensure the integrity of the loaded kernel. In the event of tampering detection, the system initiated the loading of a recovery kernel. The validation process included SHA-256 hash checks for the integrity of the loaded kernel.

Assignment 2 : The second assignment introduces on-demand paging and copy-on-write mechanisms to xv6, elevating memory management capabilities. Key tasks include enabling on-demand binary loading, designing a page fault handler, extending on-demand loading to heap memory, implementing page swapping to disk for heap memory, and introducing the Working Set Algorithm (WSA). These enhancements optimize memory usage and facilitate efficient page management. Following are the task breakdowns for this assignment.

Enable On-Demand Binary Loading: Revamped process creation in xv6 by introducing on-demand loading of a program binary's contents. Identified on-demand processes, skipped program section loading, and updated print statements. Tested by running make qemu and comparing outputs.

Design a Page Fault Handler : Implemented a page fault handler to load program binary contents on-demand. Redirected page fault exceptions, found faulting page address, loaded program binary page from disk, and updated print statements. Tested by running echo Hello World on shell.

Enable On-Demand Heap Memory: Extended on-demand loading to heap memory. Prevented heap page allocation, tracked heap pages, handled heap page faults, and updated print statements.

Implement Page Swapping to Disk for Heap : Implemented page swapping for heap memory using FIFO algorithm. Tracked heap page load time, total resident heap pages, evicted victim page to free disk region, and retrieved swapped page from disk.

Implement the Working Set Algorithm : Replaced the FIFO page swap algorithm with the Working Set Algorithm (WSA). Tested the algorithm robustly and uploaded test cases.

Implement Copy-On-Write (CoW) during Fork : Enhanced fork() with copy-on-write (CoW) optimization. Implemented uvmcopy_cow (), called it in fork (), tracked shared pages in a CoW group, and enabled CoW in the page fault handler.

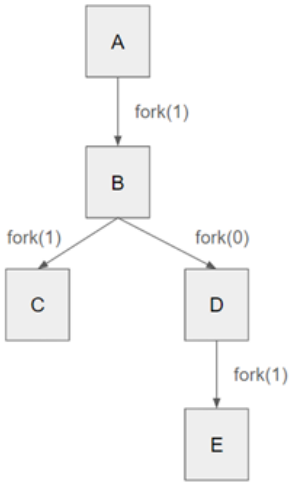

Fig. 1. Process A, B & C belong to the same CoW group, whereas process D & E belong to another group.

Assignment 3 : The third assignment delves into user-level thread management, adding a layer of sophistication to process execution within xv6. By implementing user-level threads, the assignment introduces a user-level threading library (ULTLib) responsible for thread creation, switching, yielding, and destruction. Various scheduling algorithms, including round-robin, first-come-first-serve, and priority scheduling, are incorporated, enhancing the multitasking capabilities of the operating system. Following are the task breakdowns for this assignment.

User-Level Threading Library (ULTLib)Library Initialization: Upon initialization, a data structure (similar to proc in the kernel) is created to track each user-level thread. Assigned the first kernel-provided thread as the user-level scheduler thread within this structure. Implemented this initialization in the ulthread_init() function.

Thread Creation : To create a thread, the starting function address, initial arguments, stack location, thread priority, and a context save location is obtained from the user process using ulthread_create(). Created and maintained the context save location in ULTLib. Updated the thread as RUNNABLE for scheduling.

Thread Switch: The user-level scheduler thread is invoked to schedule threads after creation and each time a user-level thread yields its CPU. Implemented an assembly function to save the current thread's registers and load registers from the next scheduled thread's context. Initialized the registers for correct function argument passing.

Thread Yield and Destroy : Implemented ulthread_yield() for a thread to give up its execution, asking the scheduler to schedule a different thread. At the thread's end, ulthread_destroy() is called to signal completion. Worked on state changes to YIELD or FREE and perform necessary housekeeping tasks before switching back to the scheduler.

Thread Scheduling Decisions : The user-level scheduler thread determines which thread to schedule based on round-robin, first-come-first-serve, or priority scheduling algorithms. A scheduler is developed that is aware of these algorithms, making decisions based on the specified policy during library initialization.

Testing : Conducted testing for FCFS, Priority, and Round Robin scheduling algorithms. Rigorous testing, including corner situations, and designed additional test cases for thorough validation.

Assignment 4 : The final assignment explores trap and emulate virtualization, offering a unique approach to designing a virtual machine (VM) within the xv6 framework. The VM runs as a user-mode process, while privileged instructions trap to a virtual machine monitor (VMM). Tasks include initializing and maintaining the VM's privileged state, redirecting traps, decoding and emulating trapped privileged instructions, and emulating physical memory protection (PMP). This assignment extends the capabilities of xv6 into the realm of virtualization. Following are the task breakdowns for this assignment.

Privileged Virtual Machine State Initialization : To facilitate privileged state tracking, an initialization process creates and maintains a data structure. This structure includes registers like machine trap handling, setup trap, information state, physical memory protection, satp, and execution mode. All registers are initialized, with special attention to the mvendorid register, which is set to the hexadecimal code "cse536." Additionally, different VM execution modes are

defined here.

Track Virtual Machine Execution and Redirect Traps : VM processes are identified by their names starting with "vm-." The VMM tracks these processes and redirects user traps raised for privileged instructions to the trap_and_emulate() function, as defined in trap-and-emulate.c. Notably, this redirection includes handling traps for the ecall instruction.

Decode Trapped Privileged Instruction : Upon trapping privileged instructions, the VMM decodes them to enable subsequent emulation. The decoding process involves extracting the opcode of the instruction (op), identifying source registers (src1 and src2), and determining the destination register (dst). A crucial part of this decoding is the mapping between register codes and the actual source and destination registers.

Emulate Decoded Instructions : Implemented ulthread_yield() for a thread to give up its execution, asking the scheduler to schedule a different thread. At the thread's end, ulthread_destroy() is called to signal completion. Worked on state changes to YIELD or FREE and perform necessary housekeeping tasks before switching back to the scheduler. Following the decoding of privileged instructions (csrr, csrw, sret, mret, ecall), the VMM proceeds to emulate their operation in software. Specific descriptions for each instruction are provided:

- **csrr:** Moves the value from a privileged register to an unprivileged register.
- **csrw:** Moves the value from an unprivileged register to a privileged register.
- **sret:** Transfers control from S-mode to U-mode entry point.
- **mret:** Transfers control from M-mode to S-mode entry point.
- **ecall:** Transfers control from U-mode to S-mode or M-mode, also transferring control from S-mode to M-mode. Additionally, the VM is shut down if 0x0 is written to mvendorid.

Emulating Physical Memory Protection (PMP) : To emulate PMP without paging, the VMM performs three subtasks. Firstly, it creates a copy of the VM process' page tables for U/S-mode execution. Secondly, on the newly-created page tables (referred to as PMP tables), the VMM unmaps regions of memory that are inaccessible based on PMP registers. Thirdly, when the VM transitions into U/S-mode (e.g., using sret), the VMM switches the VM process' page tables to the PMP tables. The assumption is made that a 4MB memory region from 0x80000000 to 0x80400000 is available for PMP.

III. Results

- Successfully enhanced xv6's boot process with a refined bootloader. Implemented a robust linker script, ensured proper stack setup for C code execution, dynamically loaded user-selected OS kernels, configured RISC-V PMP, and integrated secure boot functionality.
- Significantly improved memory management in xv6. Introduced on-demand binary loading, a dedicated page fault handler, on-demand heap memory loading, and disk-based page swapping. The Working Set Algorithm (WSA) optimizes memory usage for improved adaptability.
- Introduced a user-level threading library (ULTLib) to enhance multitasking. Successfully implemented thread creation, switching, yielding, and destruction. Integrated round-robin, first-come-first-serve, and priority scheduling algorithms for efficient process execution.
- Extended xv6 with trap and emulate virtualization. Initialized and maintained VM privileged state, redirected traps for user-level execution, decoded and emulated trapped privileged instructions, and emulated Physical Memory Protection (PMP). Resulted in a virtualization framework for xv6, allowing VM execution as user-mode processes.

Collectively, these assignments demonstrate tangible improvements in bootloader functionality, memory management, multitasking capabilities through user-level threads, and the introduction of virtualization within the xv6 framework. The results showcase an enhanced understanding and practical application of advanced operating system concepts.

IV. Contribution

All assignments within this project were completed independently, representing solo efforts. I handled the design, coding, and testing phases, demonstrating a comprehensive understanding of advanced operating system principles.

V. Skills and Knowledge Acquired

Before embarking on this course, my knowledge of operating systems was limited, with only foundational concepts from my bachelor's studies. This course, designed to explore advanced operating system principles, served as a comprehensive guide, filling the gaps in my understanding.

Through hands-on assignments and theoretical exploration, I gained an in-depth insight into the internal workings of operating systems. What initially seemed like disparate concepts gradually coalesced into a unified understanding of the intricate interplay i.e an operating system. The course not only equipped me with theoretical knowledge but also provided practical experience through assignments that covered various aspects of operating systems.

As a result of this immersive learning experience, I now possess the skills to navigate and comprehend open-source Linux code, paving the way for potential contributions to the broader community. Moreover, I am confident in my ability to pursue an entry-level role in operating system design, armed with the practical understanding gained from completing the

assignments.

Most importantly, this project has demystified the internal mechanisms of my system. I no longer view my computer as a black box; instead, I comprehend how the different components collaborate harmoniously as part of the overarching operating system. This newfound knowledge instills confidence not only in my academic pursuits but also in my ability to navigate and contribute to the broader realm of operating systems.

REFERENCES

[1] Xv6-public: Xv6 OS.

[2] A. Belay, "6.828: Using Virtual Memory," MIT.edu. https://pdos.csail.mit.edu/6.1810/2022/lec/l-usingvm.pdf.

[3] F. Embeddev, "RISC-V instruction set manual, volume I: RISC-V user-level ISA," Five EmbedDev. https://five-embeddev.com/riscv-isa-manual/latest/rv32.html.

[4] R. C. F. K. Morris, "xv6: a simple, Unix-like teaching operating system," MIT.edu. https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf.