Experiences with the ChCore Experimental Operating System Kernel

Haibo Chen
Shanghai Jiao Tong University
Shanghai, China
haibochen@sjtu.edu.cn

Yubin Xia
Shanghai Jiao Tong University
Shanghai, China
xiayubin@sjtu.edu.cn

Jinyu Gu
Shanghai Jiao Tong University
Shanghai, China
gujinyu@sjtu.edu.cn

Abstract

ChCore is an experimental operating system kernel built from the ground up for both education and research. Compared to other similar kernels, ChCore is unique in three aspects: 1) ARM-centric in light of a recent shift of computing from x86 to ARM; 2) microkernel-based given a resurgent interest in microkernel from industry; 3) bimodal for both education and research such that students can easily switch from learning to researching. This paper will describe our five-year experiences in teaching and experimenting operating systems with ChCore.

CCS Concepts

• **Software and its engineering → Operating systems**; • **Social and professional topics → Computational science and engineering education**.

Keywords

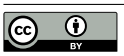
Operating system labs, microkernel, computer science education

ACM Reference Format:
Haibo Chen, Yubin Xia, and Jinyu Gu. 2026. Experiences with the ChCore Experimental Operating System Kernel. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026), February 18–21, 2026, St. Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770762.3772610

1 Introduction

Operating system (OS) is still a core course in computer science education, which helps students understand OS from both internal and external perspectives. Students can benefit from the OS course in at least three aspects: 1) construct efficient high-level software by understanding its interaction with OS; 2) learn to design and optimize an OS by writing one; 3) learn core principles of computer science (e.g., 13 of 41 *computer science principles* originated from OS [14]).

We believe gaining all the three benefits requires a bottom-up approach by providing students with opportunities to implement a concrete OS kernel, in addition to the principles and designs covered by lectures. Such an approach, while challenging for students to a certain level, helps students gain a deep understanding of how the computer system works by "getting their hands dirty", and ultimately be more competitive.



SIGCSE TS 2026, St. Louis, MO, USA

ACM ISBN 979-8-4007-2256-1/26/02
https://doi.org/10.1145/3770762.3772610

This paper describes the ChCore experimental OS kernel. ChCore follows a "divide-and-conquer" approach by dividing the kernel into a set of small programming labs with incremental difficulties, including booting a machine, memory management, user processes, multiprocessing, file systems and shell, and device I/O. This set of labs forms a small yet runnable OS kernel either in QEMU [11] or commodity boards (e.g., Raspberry Pi 3/4). ChCore provides code of the framework and only leaves the core code for students to fill, according to the offered code, comments and documentation. Automated test cases and scripts are provided to ease the process of testing and debugging.

We are not the first to provide OS kernels to help teach OS courses. There has been a long line of research and practices, including Pintos [18], Nachos [12], PortOS [10], GeekOS [17], JOS [5] and xv6 [13]. ChCore distinguishes itself from prior projects in mainly the following aspects: 1) Warmup labs on interesting applications like large language models (LLM) inference and GUI games, enabling students to gain concrete understanding of OS mechanism design goals; 2) ARM-centric while many others are x86-centric; 3) microkernel-based while many others are monolithic kernels; 4) connecting teaching to research such that students may further do research using ChCore for advanced studies.

We have used ChCore for teaching in our university for five years and gained some experiences and encouraging feedbacks during the process. Meanwhile, students have been using ChCore to conduct further research for finishing their graduation projects and got papers published in top OS conferences. The lab materials are available at https://github.com/SJTU-IPADS/OS-Course-Lab. The rest of this paper will introduce the design guidelines, each programming lab, and our experiences in detail.

2 Design Guidelines

Commodity kernels like Linux are complex and have tens of millions of lines of code, which is hard for a student to learn step by step. Briefly, the design of ChCore follows the following design guidelines:

Warmup labs. We provide warm-up labs on two interesting applications (LLM inference and GUI game), which allow students to first examine OS API usage from an application perspective, helping them understand the necessity of modules included in the OS and the interfaces (system calls) it provides.

Incremental difficulties. The kernel is divided into a set of programming labs with incremental complexity. Further, the framework code provided is decreasing from the first to the last programming labs. In this way, a student can finish the first programming lab in an almost effortless way and can thus build their confidence at the very beginning. The latter labs are becoming more and more

challenging such that students can enjoy the process of eventually challenging their own capabilities.

Microkernel-based. According to a survey [8], most of the OSes for undergraduate students are based on monolithic kernel architectures. However, there has been a recent resurgent interest in industry in building microkernel-based OSes, including Google's Fuchsia OS [4]. Besides, commodity OSes like Windows, macOS and iOS are heavily impacted by microkernel's design. For example, XNU, the kernel of macOS and iOS, essentially comprises the Mach microkernel [16] and the BSD execution environments. We envision that there is a gap for students to learn how a microkernel-based OS can be built. Using microkernel for teaching may help students be better prepared to work in industry and do advanced research.

ARM-centric. Currently, many OS programming labs are mainly x86-centric, given the prevalence of x86-based processors used in desktops and servers. Some OS courses are trying to be platform-neutral. We believe it is important to give students a chance to understand the low-level architectures such that they can readily understand and optimize performance and correctness issues during research and work. In contrast to being x86-centric, ChCore is designed to be ARM-centric (though it also supports x86) given the dominant occupation of ARM on smart phones and embedded devices. Moreover, Apple has recently announced Mac computers based on ARM M1 chips [2]. We currently do not choose RISC-V as done in MIT 6.828 course [1] because we think RISC-V is still under active development and thus premature for teaching undergraduate students in our university.

Bridging the gap between teaching and research. As active researchers in the OS community, we feel that there is still a gap between the undergraduate courses for OS and for students who want to do research after the course. Hence, we design ChCore with the goal of being not only a teaching OS, but also a base OS for advanced research. In this way, students interested in pursuing advanced OS designs can seamlessly continue their research with the same OS base and contribute to the research prototype.

3 ChCore Labs Overview

ChCore is composed of a set of labs and each lab may have some additional challenges, which are optional and will have bonuses. There are two types of test set: one is a local test set, released to students; the other is a remote test set, which runs on a server provided by the lecturers. The two test sets are different, and the final score is combined with the scores of both test sets.

3.1 Warmup Labs

Students are required to implement two applications, which are basically about invoking OS APIs. One application is a chatbot based on llama.cpp [7] that requires concurrent serving of network users similar to ChatGPT, through which students learn to use multithreading and networking APIs. Another application is a Pokemon game based on LVGL [6] that requires keyboard event handling, save functionality, and window movement/scaling capabilities, through which students understand inter-process communication between GUI frontend and backend, the Wayland protocol, and file system interface usage. Students should get a concrete understanding of the purpose of OS modules and APIs in the following labs.

3.2 Lab0: ARM Assembly

This lab has two parts. The first part is for the students to get familiar with ARM assembly language and the Qemu emulator. The second part is to read and comprehend an assembly program.

The students are not required to write any code in this lab. Instead, they are instructed to first read the given code with the help of "ARM ISA reference guide" [9]. The students also need to use GDB on an x86-64 (virtual) machine (by Qemu) to debug the AArch64 code, using *gdb-multiarch* rather than the common *gdb*. The students will learn how to set breakpoints and use commands like "where" and "readelf", which will be used in future labs.

After that, every student needs to read and run the provided assembly code of some program. The program, which was inspired by the bomb lab from CSAPP [3], requires the student to enter the correct password. Because only assembly code is accessible, students must understand the most common ARM assembly instructions and then infer the password by reading the assembly code.

Test cases: The passwords assigned to different students are different. Each student must upload their own results and gets passed if the password is correct.

Learning goals: Learning or developing the OS requires a basic understanding of the assembly for the target architecture. In this lab, the students will learn ARM assembly code and know how to debug ARM binary with *gdb* and Qemu.

3.3 Lab1: Booting a Machine

This lab contains two parts. The first part is to learn about the power-on bootstrap procedure and examine the boot loader for ChCore. The second part delves into the kernel initialization phase.

In the first part, the students will read one assembly code file, named *boot/start.S* which contains the very first instructions after booting, and be able to explain each step in the procedure. Then the students need to read *boot/init_c.c* to understand the process of loading kernel as an ELF binary.

The second part explains the process of formatted printing in the kernel. It first explains how C language uses the stack on AArch64, and asks the students to write a kernel monitor function to print the backtrace of the stack. The backtrace contains a list of saved Program Counter (PC) values from the nested *bl* instructions (similar to *call* in x86) that led to the current point of execution. The students are encouraged to consider more cases, e.g., more than one argument, for which they will get bonus.

Test cases: The grading script gives a grade by comparing the standard output of this lab with the correct output. Since the optimization level of compilers determines the results, we fix the optimization level in critical functions. Students can study the assembling of the lab code to give the information on detailed calling conventions.

Learning goals: Lab1 has two learning objectives. First, the students will understand the details of system booting process after power on, including the format of ELF binary and steps of kernel loading. Second, by printing the backtrace of function invocation, they will also know the layout of memory, which will also be very helpful for future debugging.

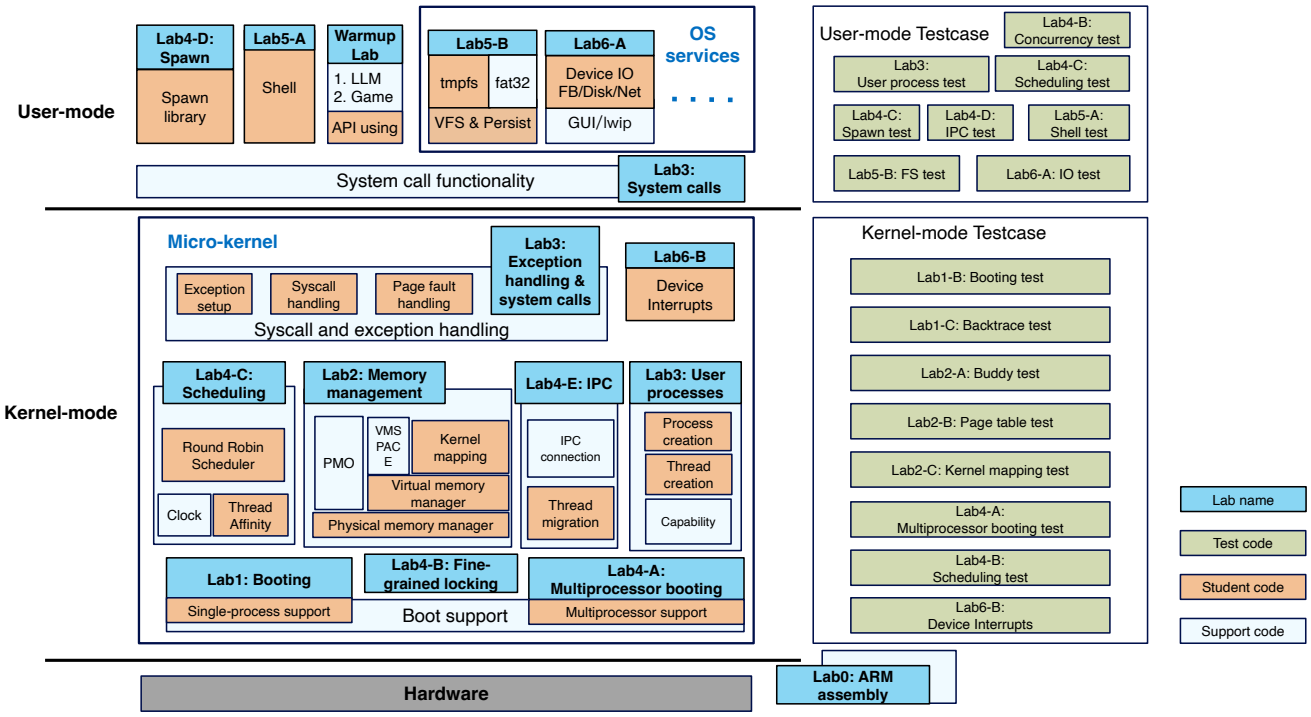


Figure 1: Components of ChCore, support code for assignments and test cases.

3.4 Lab2: Memory Management

This lab asks the students to write a memory manager, which has two components: a physical memory allocator that allocates and frees physical memory in 4KB granularity, and a virtual memory controller that maps and unmaps memory pages through MMU.

The physical memory allocator needs to keep track of which parts of physical memory are free and which are in use. In ChCore, it implements a simple buddy system to organize the physical pages. ChCore uses *block* for memory management. Each memory block has an order n. The size of a block of order n is proportional to 2^n. Power-of-two block sizes make address computation simple. The students need to understand how blocks are merged and split.

For the virtual memory component, we first give the introduction of two ARM TTBR (Translation Table Base Register), TTBR0_EL1 and TTBR1_EL1, and how ChCore uses these two registers for address translation; then give a detailed description on the format of ARM's page table entry, as well as three memory descriptors, including table descriptor, block descriptor and page descriptor. The students are asked to write a set of routines to manage page tables, including inserting and removing virtual-to-physical mappings and allocating page table pages when needed.

Based on the two components, the students need to finish kernel address space management. There are two challenges in this lab. Challenge-1 is to map the kernel space in the page granularity (by default, only the user process can map virtual address in page granularity). Challenge-2 is to support huge page for ChCore, which requires to set the page attribution bit carefully.

Test cases: The tests of Lab2 are divided into two parts. One is the unit test for the buddy system and page table management, and the other is the kernel runtime test for kernel mapping. We use *minunit*, a unit test tool, to help the students verify their code rapidly. In the buddy system test, we test the page allocating and freeing functions with corner cases to consider. In the page table test, we test the page table mapping function with a given virtual address, physical address and permission. After the students finishing part A or part B, they can run the test files to get the unit report. As for kernel mapping, we already embed the runtime test during the kernel booting process.

Learning goals: There are three learning objectives: first, the students will learn the classic buddy memory allocation algorithm for physical memory management. Second, they will be familiar with both the hardware support for virtual memory and the corresponding software component. Third, by reading the initialization code of virtual memory during the boot process, the students will understand how different addressing modes are transferred smoothly, which is a critical step for system initialization.

3.5 Lab3: User Processes

In this lab, the students are required to implement necessary kernel functions to run a user-mode process, including setting up data structures to keep track of user processes, creating a single-threaded user process, loading an image into the process and making it running. It is also required to implement several system calls as well as handle some exceptions the user process causes.

In ChCore, all kernel resources exposed to applications are managed by the *capability mechanism*. Applications can access a kernel object through an integer identifier, called a *cap*. To make an analogy with file descriptor in Linux, a file descriptor is a cap, and the

file itself is a kernel object. The mapping relationship between caps and objects is shared by all threads within the same process. The students are not required to implement any logic of the capability mechanism, but should be familiar with the interface to allocate cap and object, and know how to query an object with a cap.

Since there is no file system yet, ChCore combines all executable ELF images of user applications to a single file and embeds it into the kernel. The students are asked to prepare kernel stack, user stack, user heap, and set up memory region to load the code and data from the ELF image.

The first system call executed in user mode is "svc #0". The students are asked to read ARM's manual to know the concepts of exception and interrupt in ARM (which are slightly different from Intel's) as well as the process of handling them, and then set up the exception vector table. The current lab focuses on handling synchronous exceptions and leaves asynchronous exceptions to later labs.

Following system calls are required to implement in this lab:

- **sys_putc**: to put a character onto the screen, which is used by *printf*.
- **sys_exit**: to exit a thread.
- **sys_create_pmo**: to create a *pmo* (physical memory object).
- **sys_map_pmo**: to map a *pmo*; will be used to test page fault (with the previous one).
- **sys_handle_brk** (*bonus*): to create or expand the user heap.

Meanwhile, the page fault exception handling should also be implemented. Our script will use the two *pmo*-related system calls to trigger page fault for testing.

Test cases: Lab3 is accompanied by ten tests focusing on Part B and C, since Part A cannot be tested individually. Most of the tests are user programs leveraging kernel capacities on exception handling and system calls. Tests for part B require properly loading user programs and handling exceptions happened in user mode, which is done by triggering invalid instruction exceptions in user programs. Tests for part C depend on correctly implementing all the required system calls as well as the page fault handler. Some extra tests are designed for corner cases like unaligned memory addresses or multi-header ELF files. All the tests only produce deterministic output. If any problem happens, the grading script will point out the difference between the expected output and your actual output for error correction.

Learning goals: There are three learning objectives in this lab. First, the students will learn the capability mechanism, which is a critical component for microkernel. Second, they will learn the lifecycle of a user-level program, from ELF loading to process exiting. Third, by implementing several system calls, the students will know the interaction between user program and the kernel. They will get more familiar with the ARM hardware through this lab.

3.6 Lab4: Multiprocessing

This lab enables the support of running multiple processes on multiple CPU cores and communication between different processes. It contains five parts. In Part A, the students are asked to boot more than one CPU core while using one big lock for concurrency control. In Part B, the students are required to replace the big lock with fine-grained locks to enable more concurrency. In Part C, the students need to implement a basic round-robin scheduler to schedule multiple threads among all CPU cores, and use hardware timer interrupts to support preemption. Part D asks to implement a user-level function named `spawn()`, which creates a new process to run a program. Part E focuses on inter-process communication (IPC) support which allows the interaction between different processes.

The students first need to modify the boot process to initialize multiple CPU cores. When booting up, the Raspberry-Pi-3 board (chosen as the lab platform since its popularity) starts all the CPU cores up simultaneously, but only one, the primary core, is able to make progress to initialize itself and other cores are blocked and waiting for the primary core to active them one by one. In order to avoid race condition, the students are asked to use a big kernel lock to ensure only one CPU core can run in the kernel mode. The interrupt is disabled during kernel execution.

The above big kernel lock is a spin lock. We then ask students to implement both the ticket lock and the read-write lock by referring to the lock specification and the provided spin lock implementation. We also point out some potential locations in the kernel (all the necessary locations are included) where locks may be required. Students must carefully determine whether a lock is required and whether a dead lock is possible, replacing the big kernel lock with fine-grained locks.

Each CPU core has its own run-queue, which tracks all the ready-to-schedule threads on that core. One thread can only be in one core's queue. If there is no runnable thread, a CPU core will run its own idle thread, which does nothing but waiting for an interrupt. The scheduling algorithm is round-robin. Once there is a timer interrupt, the kernel will handle the interrupt, update the current thread's scheduling info, and preempt it if it runs out of its scheduling budget. In order to support scheduling threads across different CPU cores, we also ask the students to implement processor affinity, which enables a thread to bind to a specific CPU core. Now a thread can run on any CPU cores instead of only the core on which it is created.

Next is to allow a user-mode program to execute a specific file. In Linux, a typical way is to first call `fork()` and then `exec()`. In ChCore, we ask the students to use `spawn()` instead, which combines both functionalities. Once an application invokes `spawn()` with an ELF file, ChCore will create a new process (child), map each segment in the binary ELF to the process, setup a running environment for the child process including user stack initialization and argument passing, and create its main (first) thread. The parent process also needs to transfer *PMOs* and *caps* to the child process.

The IPC mechanism of ChCore is similar to the "migrating thread model" [15]. ChCore's IPC interfaces are not the one used by traditional message passing (`send()/recv()`). Instead, it is like a client/server model, where the IPC request receiver is a server, and the sender is a client. Before starting to serve, a server needs to register its callback function first. ChCore will bind a client to the server by creating a connection between the two. The client can send a request through the connection, and its thread will be migrated to the server's process to handle the request, and migrated back to the client with the return value after handling. The students can also implement the traditional interface, `send()` and `recv()`, as a challenge.

Test cases: Lab4 contains both kernel-mode and user-mode tests. For each processor, the first time it attempts to exit the kernel mode, it checks whether the multiprocessor booting (Part A) works correctly. The end-to-end functionality tests of the fine-grained locking (Part B) and the scheduler (Part C) are in user-mode. We also require the students to submit a document describing their locking mechanisms, such as what variables to protect and how to avoid deadlock. Meanwhile, to prevent the students from spending a large amount of time debugging their scheduler in the user mode, we also provide kernel-mode functionality tests, robustness tests, and stress tests for the scheduler. For process spawn (Part D) and IPC (Part E), we prepare most of the code. Thus, the coding tasks are not heavy, and we only provide user-mode test functionality tests. All the tests' outputs are deterministic, and a grading script is used to compare the expected and actual outputs.

Learning goals: This lab has five learning objectives. First, the students will know the differences between single core and multiple cores from the perspective of OS. Second, they will get familiar with the basic lock implementations and carefully use fine-grained locks in the kernel. Third, they need to learn how a multi-core scheduler works. Then, by comparing `fork()` with `spawn()`, the students are supposed to be able to tell the pros and cons of both designs and how to choose one for different scenarios. Last but not least, the students will learn how to implement one of the most fundamental mechanisms of microkernel, IPC, and know the differences of various design choices.

3.7 Lab5: File System and Shell

This lab contains two parts. The first part asks the students to implement *tmpfs* which is an inode-based file system and runs as a user-level file system server. The inode-based file system does not require many disk operations; instead, it is in-memory. The students are required to implement basic file system facilities, including `open()`, `read()`, `write()`, scanning directory, pathname resolving, file persistency, etc. The file system server exposes functions on top of ChCore's IPC mechanism. Besides, students also need to implement a simplified virtual file system (VFS) layer. The second part is to implement a shell based on the file system.

Test cases: Tests for Lab5 are all user-mode tests. The file system tests consist of a series of operations on files, such as *read*, *write*, and *scanning*. The *VFS* layer needs to support both the *tmpfs* made by students themselves and the provided *FAT32*. The tests for the shell simulate a user typing shell commands, and the grading script will check the standard output of the shell.

Learning goals: There are two learning objectives in this lab. First, the students will learn the internal components of a file system. In the lecture, we will also introduce file systems that are not based on inode, like FAT, GFS, etc., and also file systems designed not for hard disks, like flash, non-volatile memory (NVM), etc. The students will be asked to compare these file systems with the classical inode-based file system. Second, the students will know how a shell work, from command typing to execution.

3.8 Lab6: Device I/O

The last lab asks the students to port three drivers, a storage driver, a framebuffer driver, and a network driver on the Raspberry Pi 3/4 board. Furthermore, it asks the students to connect the drivers to other OS modules, such as file system (FAT32), GUI (LVGL), and network stack (lwip).

Test cases: Tests for Lab6 validates whether the warmup applications (the LLM inference server and the GUI game) can correctly access the required devices.

Learning goals: The main learning objective of this lab is how a device driver looks like and how the OS interacts with a device through its driver. With the concrete code samples and the lab practice, we find that students can get deeper understanding of these concepts than before.

4 Experiences

4.1 Experience with Teaching

Asking questions is an effective way to get feedback to know how well the students can understand and master the contents of the course. We divide the degree of mastery to three levels, and ask different questions to see which degree the students have achieved.

The first level is being able to explain the reasons for a given phenomenon. For example, given two ways to copy a directory containing Linux kernel source code from a local disk to a remote-mounted NFS disk: one way is to copy the directory directly with the **cp** command; the other way is using **tar** to generate an archive file containing all the source code, copying it to the destination disk, and extracting the files. In practice, the second way is far more efficient than the first way. The students are required to answer why by explaining where the time goes. At this level, the students need to be familiar with how different components of OS work, e.g., file systems, network, IPC, etc., and should also be able to connect phenomenons with implementations.

The second level is being able to tell the pros and cons of a given design or system. For example, OS usually uses *block* as the abstraction of the disk. The block size can be defined by software. The students are required to tell how different configurations of block size will affect the execution, and analyze which scenarios are more suitable for a given configuration. We can also give two or more designs targeting the same goal, and ask the students to make a comparison of the two. At this level, the students need to know the metrics of the operating system, including latency, throughput, utilization, security, compatibility, etc., and should be able to make choices on different design options.

The third level is being able to propose a new design to solve a given problem. For example, most of the existing file systems are designed for hard disks. When non-volatile memory (NVM) is getting prevalent, it cannot be fully utilized due to the limitations of file systems. How to design a new file system to take full advantage of the characteristics of NVM could be a challenging problem. We ask the students to consider the differences between hard disk and NVM and why the design of file system does not fit the NVM. The students will further be asked to design one component or more of a file system for NVM, like the crash consistency mechanism or organization of directory entries.

4.2 Experience with Research

ChCore is also planned to be used as the a research OS kernel to investigate innovative ways to reshape modern OS architectures.

Each year, there are about 12-18 students in our university who have done the ChCore labs are using ChCore to experiment new OS mechanisms. On average, there are two papers published in top system conferences, including SOSP, USENIX security, USENIX ATC, etc. One student group even won the best paper award in SOSP, the most prestigious conference in operating systems.

One specific example is using ChCore for experiment new IPC designs with modern hardware. IPC is known as "Achilles heel" of a microkernel operating system, whose performance significantly impacts the overall system performance. Although there has been a long line of research to improve IPC designs, IPC performance has yet to be optimized. ChCore has been used recently by students to exploit how modern hardware features for fine-grained isolation to devise a new IPC design with both low overhead and strong isolation. Specifically, they have exploited how the memory protection keys (MPK) from recent Intel processors can be used to support strong isolation and fast cross address space calling. This has led to an order of magnitude improvement of IPC performance, resulting in a publication in a top conference in computer systems.

4.3 Feedback from Students

ChCore does assist students in better understanding some important OS mechanisms. Our school's new OS course (with ChCore) has been taught to two classes of juniors. Based on the experiences described in Section 4.1, we developed a final exam to assess the students' mastery. In the first year, the median final exam score was 82, rising to 85 in the second year (full score is 100). As the labs get more mature, the score is about 90 in the latest year's exam. The exam difficulty and grading rules remained (almost) the same. In our school's evaluation, more than 80% of students said that doing Labs was very helpful in mastering the mechanisms and principles of OS, which also helps them have a deeper understanding of the exam questions.

In addition to the students' supervisory feelings, we can tell from the graduate selection interviews that these two classes have a better understanding of OS mechanisms as a result of completing the ChCore labs. An interview question about the OS virtual memory is a typical evidence. We ask students what will happen if an application accesses memory address 0 and why. Most students, regardless of whether they are taking our new OS course, are aware that it will trigger a page fault exception because memory address 0 is an invalid address. However, when we further ask how the OS determines the legality of memory addresses, students in previous years have rarely answered correctly, despite the fact that we also ask them to complete the JOS labs [5]. In the recent two years, more than 60% of students who completed the ChCore labs can provide a precise answer because they read the related code in ChCore's lab framework when completing the labs. ChCore makes the conceptual introduction more concrete and simple for students to grasp.

We used JOS labs for about ten years before designing ChCore labs. JOS is widely used in OS education and inspires our ChCore labs. ChCore labs, on the other hand, have two advantages. First, it has more in common with a modern OS. Despite the fact that ChCore has a small code base, it simplifies most policies while retaining the core mechanisms. For example, it supports on-demand paging (mechanism) for virtual memory management while just allocating one physical memory page for the faulting virtual memory. ChCore presents the basic designs and implementations for such mechanisms so that students can become acquainted with the OS mechanisms that share the same high-level concept even across modern OSes such as Linux. Yet, JOS, as a representative of existing OS labs, eliminates some basic mechanisms for modern OSes, which also explains why students failed to answer the above interview question after finishing JOS labs. Second, ChCore is ARM-centric and microkernel-based, which can broaden students' horizons on advanced OS topics, which makes ChCore become a suitable complementary to existing courses. On one side, ChCore introduces microkernels which has gained increasing attention from industry, particularly in mobile platforms and vehicles, and guides students to compare different OS architectures. On the other side, ChCore familiarizes students with ARMv8 architecture which dominates mobile and IoT devices and is marching towards the server and desktop fields in the future.

Most students can catch up and enjoy the labs. We conducted surveys and established an online forum to collect students' feedback including the labs' difficulty. The following are some of the most notable outcomes. Over 80% of students can complete each lab (Lab0-Lab4) on their own in a timely manner (*Result-1*), which is also confirmed by their lab grade. Because ChCore labs, slides, and videos from our OS course are all freely available on the Internet, students from other universities can also write ChCore labs and participate in the online forum. In the forum discussions and individual blogs, they said that the lab content is very enriching and the challenging exercises are quite interesting. Overall, most students enjoy the labs because they can learn a lot through reading and writing the code (*Result-2*). In the first year, 15% students said they struggled with Lab1 and Lab2 because they were unfamiliar with ARM assembly. In the second year, we add Lab0 for warming up, which mitigates the problem (*Result-3*). Besides, some students may be unable to complete some labs on time. We will give them runnable object files (binary code) so that they can move on to the next lab without having completed the previous ones.

5 Conclusions

In this paper, we described our experiences in using ChCore, an experimental operating system kernel for operating system education. In contrast to prior systems, ChCore embodied several design guidelines, including being ARM-centric and microkernel-based, as well as aiming to bridge teaching and research. We also described our experiences in using ChCore for both teaching and research. In the future, we plan to evolve ChCore with more modern features and make it more customizable to design programming lab sets for different levels of students.

Acknowledgments

Thanks to individuals including teaching assistants and students who participated in our operating system courses. Thanks to the anonymous reviewers for their valuable comments. Thanks to professor Binyu Zang for the guidance and support.

References

[1] 2020. MIT 6.828: Operating System Engineering. https://pdos.csail.mit.edu/6.828/2020/.
[2] 2020. WWDC 2020: Apple to announce move to ARM-powered Macs. https://www.computerworld.com/article/3562068/wwdc-2020-apple-to-announce-move-to-arm-powered-macs.html.
[3] 2022. Computer Systems: A Programmer's Perspective. https://csapp.cs.cmu.edu/.
[4] 2022. Google's Fuchsia Operating System. https://fuchsia.dev/.
[5] 2022. MIT JOS. https://pdos.csail.mit.edu/6.828/2021/overview.html.
[6] 2025. Light and Versatile Graphics Library. https://lvgl.io/.
[7] 2025. llama.cpp. https://github.com/ggml-org/llama.cpp.
[8] Charles L Anderson and Minh Nguyen. 2005. A survey of contemporary instructional operating systems for use in undergraduate courses. *Journal of Computing Sciences in Colleges* 21, 1 (2005), 183–190.
[9] ARM. 2018. *ARM Instruction Set: Reference Guide*.
[10] Benjamin Atkin and Emin Gün Sirer. 2002. PortOS: an educational operating system for the Post-PC environment. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*. 116–120.
[11] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 41–46. http://www.usenix.org/events/usenix05/tech/freenix/bellard.html
[12] Wayne A Christopher, Steven J Procter, and Thomas E Anderson. 1993. The Nachos instructional operating system. In *USENIX Winter*. 481–488.
[13] Russ Cox, M Frans Kaashoek, and Robert Morris. 2011. Xv6, a simple Unix-like teaching operating system. http://pdos. csail. mit. edu/6.828/2012/xv6. html.
[14] Peter J Denning. 2003. Great principles of computing. *Commun. ACM* 46, 11 (2003), 15–20.
[15] Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to A Migrating Thread Model.. In *USENIX Winter*. 97–114.
[16] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. 1992. Microkernel operating system architecture and Mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. 11–30.
[17] David Hovemeyer. 2001. GeekOS: An instructional operating system for real hardware. *GeekOS.(nd). Retrieved March* 16 (2001).
[18] Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The pintos instructional operating system kernel. In *Proceedings of the 40th ACM technical symposium on Computer science education*. 453–457.