Process/Memory Abstraction in R5O5

A Project Report Submitted

for the Open Ended Lab/Project

(Course No.: ID 3801)

by

Ahmed Zaheer Dadarkar (111701002)

Rakesh Kumar (111701024)

and

Muhammed Yaseen (111701032)

Sirpa Sahul (111701028)

under the guidance of

Dr. Sandeep Chandran

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

CERTIFICATE

This is to certify that the work contained in this report entitled ***"Process / Memory Abstraction in R5O5"*** *is a bonafide work of* ***Ahmed Zaheer Dadarkar (Roll No. 111701002), Rakesh Kumar(Roll No. 111701024), Sirpa Sahul (Roll No. 111701028)****,* ***Muhammed Yaseen (Roll No. 111701032)*** *carried out under my supervision for the course Open Ended Lab/Project (ID 3801).*

Dr. Sandeep Chandran

Assistant Professor

Discipline of Computer Science and Engineering

Indian Institute of Technology Palakkad

Acknowledgements

We are extremely thankful and pay our gratitude to our guide Dr. Sandeep Chandran, for his constant support, guidance, patience and encouragement throughout the project and we wish to seek his support in the future as well. We are grateful for having a mentor like him.

We extend our gratitude to Dr. Sovan Lal Das and the other members of the OELP Evaluation committee, Indian Institute of Technology Palakkad for giving us this opportunity.

We would like to extend our gratitude to the RISC-V International contributing members for an extensive specifications document on RISC-V ISA[1], which has come very handy in developing the R5O5 OS model. We also express our gratitude to Charles Crowley, author of the book *Operating Systems: A Design-Oriented Approach*[2], and the developers of *xv6 - a teaching operating system developed in the summer of 2006 for MIT's operating systems course*[3], which had paved the way for building the foundations of R5O5.

[1] https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf
[2] https://www.cs.unm.edu/~crowley/osbook/begin.html
[3] https://pdos.csail.mit.edu/6.828/2019/xv6.html

Contents

Chapter 1

Introduction

An OS is responsible for giving a simple abstraction of the underlying hardware to the application programmer. Hence it is also responsible to manage the underlying resources on behalf of the programmer. Therefore, an OS developer should understand the functioning of the underlying hardware, and design management policies suitably. In a typical curriculum of an undergraduate program in Computer Science and Engineering, a student is exposed to the functioning of underlying hardware in a Computer Organization course, and the design of management policies and abstractions is introduced in the Operating Systems course.

A very good way to introduce design principles of OS is to make students build an OS from scratch. This will ensure that students understand all the steps involved in building an OS, and the implications of various management policies. However, this approach is time consuming, and cannot be finished in a 14-week course (and its associated lab).

Another approach is to give an existing OS to students and ask them to modify some management policies. Although this approach is practical for a semester long course, it does not introduce students to steps involved in abstracting away the details of the underlying hardware.

In this project, we did a balance between these two approaches of teaching OS by developing a teaching operating system - RISC-V OS (R5O5). R5O5 closely follows the concepts introduced in the textbook *Operating Systems: A design-oriented approach by Charles Crowley* that is prescribed in the syllabus of the Operating Systems course at IIT Palakkad (CS3010). This textbook has sample code snippets for each concept, but these snippets are not complete. Therefore, a student would still find it challenging to build an OS from scratch, but has reference material handy in case of any difficulty. R5O5 will also make the students able to see the OS they develop run on a RISC-V machine using the RISC-V simulator *QEMU*.

R5O5 is implemented using the driver supports from *xv6*. The assembly code instructions are written in RISC-V wherever it is required and the rest of the OS is an implementation of the OS design discussed in the course textbook by Charles Crowley.

1.1 Organization of the Report

- **Chapter 1, Introduction** - Gives an introduction to the project work, explains the motivation behind it and gives a brief description of its implementation and application.
- **Chapter 2, Review of Past Works -** Describes the previous works which are included or utilized in the development of R5O5.
- **Chapter 3, Hardware -** Describes the tools and technologies used to set up a hardware on top of which the R5O5 is being run.
- **Chapter 4, Building the base of R5O5 using xv6 -** Explains how the device drivers for the console and the disk are abstracted in R5O5.
- **Chapter 5, Memory Abstractions in R5O5 -** Describes the implementation of the OS modules that are responsible for abstracting the disk and the memory of a RISC-V machine.
- **Chapter 6, Process Abstractions in R5O5 -** Describes the implementation of the OS modules that are responsible for abstracting the processor of a RISC-V machine into R5O5 processes.
- **Chapter 7, Source Code -** Explains the layout of the code, the code guidelines that are followed in the project and the debugging support set up for the project.
- **Chapter 8, Conclusion and Future Scopes -** Gives a summary of the development work and discusses the scope of improvements possible in R5O5.

Chapter 2

Review of Past Works

We have used the following works extensively in the development of R5O5. An extensive reading and a walkthrough of the codebase and the documentation of each of these works were done as part of this project.

- **Simple Operating System (SOS):**
    - SOS is a model of a simple operating system introduced by Charles Crowley is his Operating Systems Textbook[2]. The textbook discusses the design of the SOS and provides the code snippets of the different parts of the OS.
- **xv6:**
    - xv6 is a simple, Unix-like teaching operating system developed by Massachusetts Institute of Technology(MIT). We have studied the codebase and the design of xv6 to carry out low level driver and disk abstractions in R5O5.

Chapter 3

Hardware

1. QEMU

QEMU is a generic and open source machine emulator and virtualizer. We have used it as a machine emulator to emulate the RISC-V Architecture. Running RISC-V on qemu requires a disk image which would be used to emulate the disk and the kernel binary which would be the operating system that would run on this RISC-V emulated machine. We have taken the additional support (low level drivers) required to run an OS from the xv6 operating system developed by MIT.

2. RISC-V

Our OS is built to run on 64-bit RISC V which supports Sv39 virtual addressing. This architecture defines 64-bit registers and instructions to operate on these registers. There are three privilege levels - Machine, Supervisor and User. But, our OS is built to utilize only Machine and User mode, the former for running the kernel, while the latter for running user programs. Sv39 means that the least 39 bits of the 64 bit virtual address are used, while the top 25 bits remain unused. Also, Sv39 supports a 3-level paging hardware which provides a larger number of pages and also efficient use of memory. Paging is completely disabled in Machine mode, but can be enabled or disabled in Supervisor and User modes.

Chapter 4

Building the base using xv6

Our operating system is modeled after SOS, but since SOS is too abstract that it does not talk about the implementation of low level driver code, we were required to set up these drivers. We thought that we could use the implementation of the driver code which was used by Xv6, hence we imported the UART and Disk driver source code from Xv6.

1. UART

We have imported the UART driver source code which is being used to send characters to the screen of the hardware, and receive characters typed using the keyboard. We have also imported a layer of abstraction over the UART drivers in the form of console operations from Xv6. These operations allow us to read from the console and write to the console, thereby completely abstracting the IO to these two operations (i.e. any IO operation to the console can be written using these two operations).

2. Disk Driver

We have also imported the Disk driver source code which first initializes the disk by writing into the memory mapped registers of the disk. Once initialized, it provides operations to read a block from the disk and write a block to the disk. These operations use the interface of buffer-based reading and writing of blocks (which is explained in the disk section of chapter 5).

Chapter 5

Memory Abstraction

The memory abstraction component of R5O5 consists of buffered reading and writing of disk blocks by the disk driver and the higher levels of abstraction over this buffered reading and writing such as reading and writing a number of bytes from the disk to a memory location and vice versa, where the memory location can be a physical memory location or a virtual memory location. We also have functions for allocating and freeing memory pages - the memory allocator, and functions which would be used for setting up paging in User mode. Finally, we have implemented the program loader which would load the program from disk into memory by appropriately parsing the ELF file of the program present in the disk.

1. Disk Abstraction

Read / Write operations to a disk block is costly. So, operating systems maintain a buffer cache to store some of the recently used disk blocks in the memory. Read / Write operation in a memory is less costlier. We have implemented a LRU Buffer Cache for faster Read / Write operations on the disk.
The file buffer.c maintains the Buffer Cache List and provides several functions needed for the disk abstraction. Each buffer can store one disk block. The file contains these helper functions for reading, writing, releasing, etc. the buffers.

- **Buffer Initialization:** binit() initializes the Buffer Cache by creating a doubly linked list of the buffers.
- **Reading a disk block into the buffer :** bread(uint bockno) reads the given block number into a buffer and returns the address of the buffer. It uses the disk driver function diskRW() to read the block (if the block is not cached).
- **Writing a buffer back to the disk:** bwrite(Buffer *b) uses the driver function diskRW() to write back the buffer into the corresponding disk block.
- **Releasing a buffer:** brelse(Buffer *b) releases the buffer b. It decrements the reference count of the buffer. If the count becomes zero, then the buffer is released and attached to the head of the Buffer list.

Our disk abstraction module (buffer.c) provides these two major functionalities:

1. **Reading from a disk location to a memory location (kernel / user)**
    1.1. **kernel space:** readBytes(uint64 diskBlockNum, uint64 offset, uint64 nBytes, uchar *memoryLocation) reads the given number of bytes from the disk block and the offset into the given memory location. It uses the above helper

functions like bget(), bread(), etc. for reading. The memory location is physical i.e., no address translation is required.

1.2. **user space:** readBytesVirtual(uint64 diskBlockNum, uint64 offset, uint64 nBytes, uchar *memoryLocation, PageTable *pagetable) does the same thing as above but as the address is virtual, the address is translated and then the bytes are copied. The address translation is done using the given pagetable and the function virtualToPhysical() (paging.c).

2. **Writing from a memory location (kernel / user) to a disk location**

2.1. **kernel space:** writeBytes(uint64 diskBlockNum, uint64 offset, uint64 nBytes, uchar *memoryLocation) writes the given number of bytes into the disk block and the offset from the given memory location. It uses the above helper functions like bread(), bwrite(), etc. for writing. The memory location is physical i.e., no address translation is required.

2.2. **user space:** writeBytesVirtual(uint64 diskBlockNum, uint64 offset, uint64 nBytes, uchar *memoryLocation, PageTable *pagetable) does the same thing as above but after the address translation.

2. Memory Allocator

The memory allocator of R5O5 first initializes a stack of free memory pages. Now, if a page is requested, it provides the topmost page on the free page stack if available, else raises an error. If a page is to be freed after usage, it is provided back to the memory allocator, which would place the page back at the top of the stack. The memory allocator source code is present in the file memoryAllocator.c

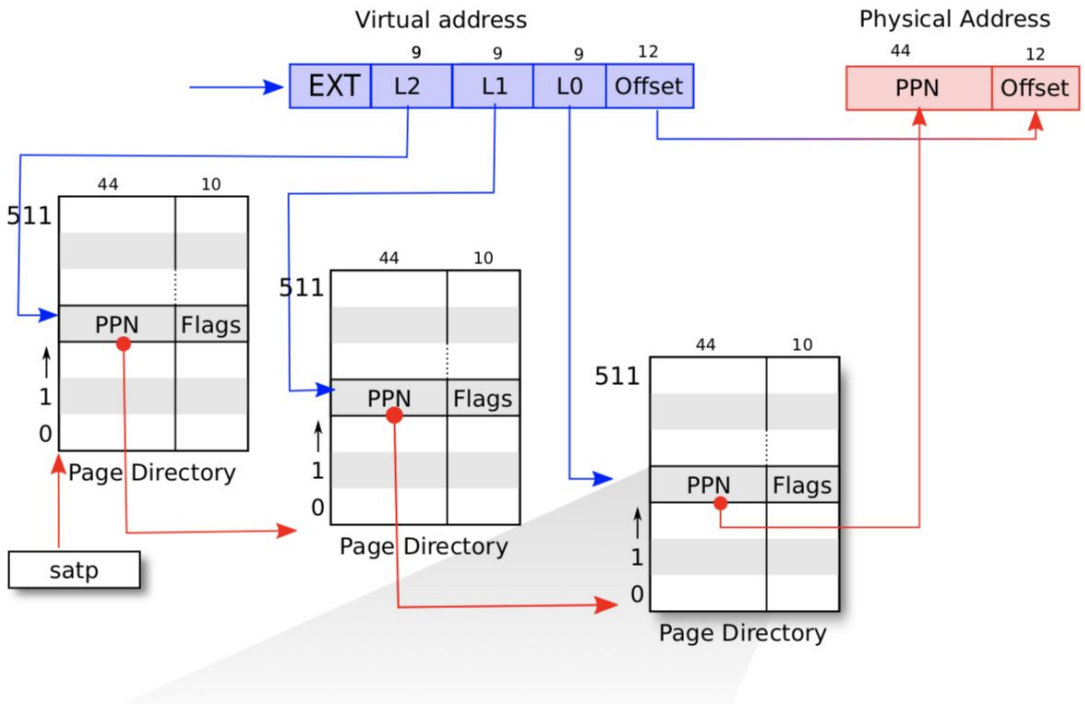
3. Paging Functions

The file paging.c provides many helper functions which would be used by other modules of the OS, for example the process module. We have helper functions for retrieving the offsets, physical page numbers, virtual page numbers, and more from the provided physical and virtual addresses.

It was mentioned earlier that the paging system of RISC V supports 3-level page tables, where there is a single root page directory at level 2 (the actual page table), then for every valid entry in the root page directory, we have an intermediate page directory at level 1. Now, for every intermediate page directory, and for every valid entry in that page directory, we have a final page directory at level 0 which contains the page table entries which map to physical pages used by the user process to which the page table belongs to. Hence, page directories at level 2 and level 1 contain page table entries which map to other page

directories, while page directories at level 0 map to physical pages used by the user process to which the page table belongs to.

The three level page table structure is shown in the diagram below (from Xv6 book):



- **EXT**: Unused (or extra) 25 bits
- **L2**: 9 bits at the right of EXT which would be used to index into the root page directory and find the page number of the intermediate page directory.
- **L1**: 9 bits at the right of L2 which would be used to index into the intermediate page directory and find the page number of the final page directory.
- **L0**: 9 bits at the right of L1 which would be used to index into the final page directory and find the physical page number corresponding to virtual address.
- **offset**: 12 bits at the right of L0 which are the offset into the virtual page (determined by L2, L1 and L0 as mentioned above), and hence into the physical page to which the virtual page corresponds to.

Our paging module implements the following helper function definitions: -

- **allocatePageDirectory:** Allocate a page directory, and initialize all entries to be invalid
- **allocatePageTable:** Allocate an empty page table.
- **getFlags/setFlags:** Get/set flags of a page table entry
- **getPageNum/setPageNum:** Get/set page number of a page table entry

- **isVirtualPageAllocated:** Checks if the given virtual page number is mapped to a physical page number.
- **mapVirtualPage:** Maps the given virtual page to the given physical page.
- **allocateVirtualPage:** Allocate a physical page and map it to the given virtual page if not already mapped, else raise an error.
- **virtualToPhysical:** Output the physical address corresponding to the given virtual address if it exists, else raise an error
- **deallocatePageTable:** Go over all the valid final, intermediate and root page directories and deallocate the pages.

4. Create Process - Program Loader

The creation of a process can be said to have two parts - loading the program from the disk into the memory which belongs to the memory abstraction component, and then updating the process structures which belongs to the process abstraction component. Here, we only describe the program loading part of the creation of the process.

Program loading, performed by the program loader in `createProcess.c` can itself be divided into two parts - (1) parsing the ELF file to get the program segments and then (2) loading the segments from disk to memory.

(1) ELF Parser / Reader

The ELF parser function is present in elfReader.c, it parses the program binary (which is an ELF file) present in the disk and returns the entry point into the program and a linked list in which each node contains information about a single program segment described in the ELF file.

(2) Loading Program Segments

The function **programLoader** in createProcess.c first reads the ELF file in the disk to get information about program segments, then loads the program segments one by one from the disk into the memory by using helper functions provided by the buffer.c.

Chapter 6

Process Abstraction

1. Process structures

We have used some structures for creating processes out of programs. The structures are related to process abstraction.

- **SaveArea:** Structure which will store the information related to registers and the next instruction from where it has to execute when it goes from Ready to Running state. And when context switching happens, all the information about the process will be stored in the save area: -
    - **reg**: stores the register content
    - **epc**: stores the exception program counter value
    - **satp**: stores the satp value corresponding to this process.

- **Process Descriptor:** Structure contains the following fields.
    - **SaveArea *sa*** which will contain structure for save area
    - **slotAllocated**: boolean indicating whether slot allocated or not
    - **timeLeft**: time left for the process to run in one go.
    - **state**: state of the process

2. Create Process - Setting up Process Structure Information

Create Process first finds a free slot in the process descriptor table, and then calls the program loader to the program from the disk into the memory as described above in the memory abstraction component. Then it initializes the entries of the process descriptor table corresponding to the process just created.

3. Context switching

In Operating Systems, context switch is the process of storing the state of a process so that it can be restored later and resume its execution. Operating systems usually have two types of context switches : kernel-kernel switch and user-kernel switch. R5O5 also supports these two context switches:

1. **Kernel Context (**kernelvec.S**):** In the kernel mode, mtvec register (stores the interrupt vector table's address) points to the kernelvec (a function defined in the kernelvec.S). When an interrupt comes in the kernel mode, the control gets transferred to the kernelvec function. The function stores all the general purpose registers, epc and

mstatus registers into the kernel save area. Then it calls the kernelInterruptHandler() for processing the interrupt and once the interrupt is processed, the kernelvec function restores back the registers and the kernel continues its execution from where it left.

2. **User Context (uservec.S):** The file uservec.S has the definition of these two functions:
   2.1. **uservec:** It is responsible for the user-kernel transition. In the user mode, mtvec register points to the uservec function. When an interrupt comes in the user mode, the control gets transferred to the uservec function. The function stores all the general purpose registers, epc and satp registers into the user save area of the corresponding entry of the process in the process descriptor table. It calls the userInterrruptHandler() for processing the interrupt.
   2.2. **userret:** It is responsible for the kernel-user transition. It restores back the general purpose registers and returns the control back to the user mode (control registers like satp, mepc, etc are already written by the runProcess() function in the process.c file).

4. Interrupt handler

Whenever an interrupt occurs, the interrupt handler will be called and it will be responsible for handling the interrupts that occur in the kernel mode/user mode depending upon which interrupt handler was called. As of now, only device interrupts can occur in kernel mode (hence we would have to wait in the kernel until a device interrupt occurs for a particular device operation - for example disk read). And timer interrupts would be handled by the user interrupt handler and kernel interrupts would be handled by kernel Interrupt Handler.

1. User Interrupt Handler

So when an interrupt occurs in user mode then the control comes first to uservec and kernel will store the information of the process in it's save area and it will point out the interrupt vector handler which is responsible for executing the interrupt and then after the interrupt is executed then it will change the state of the process if required and then control backs to the uservec then it will restore the process state from the save area and it will execute the process.

2. Kernel Interrupt Handler

The kernel interrupt handler first gets the cause of the interrupt, then checks whether it is an external (device) interrupt or not. If it is, then it goes on to handler the device interrupt. The only allowed device interrupts are Disk interrupts and UART interrupts.

Chapter 7

Source Code

1. Project Repository

The complete source code for the project is available here. We have used the features offered by the version control very extensively to maintain a clean codebase. We have automated the commands used for making the OS to run using makefiles wherever it was possible.

```
make qemu: boots up R5O5
make qemu-gdb: boots up R5O5 in debug mode
```

2. Coding standards

The source code is written stricting to a strict set of coding guidelines. The *master* branch contains the latest version of R5O5. Features get added to the *master* branch only after peer reviews of the collaborators.

The split-up of the repository is as follows:

- **kernel**: All the codes written for the running of OS are stored here.
- **logs**: The log information of R5O5 is stored here for analysis of errors or unexpected interruptions.
- **mkfs**: The abstraction of disk for R5O5
- **information**: Will contain the metadata of the programs loaded into the disk, which is not currently implemented.

The following coding guidelines are strictly enforced throughout the codebase:

- All the global variables to be stored in kernel/globalData.c
- All the declarations are to be done in kernel/declarations.h
- All the function declarations are to be done in kernel/function.h

- Every function should be properly commented indicating the purpose of the function, additional comments within the function should be added wherever it is necessary.
- One tab should be equal to four spaces.
- No line of code should exceed more than 80 characters.

3. Debugging using gdb

We use the gdb debugging tool to try to debug errors which cannot be simply inferred by using print statements or going through the source code. We follow the execution path of instructions while setting breakpoints, watchpoints and extracting register or variables values (by using the symbol table from the kernel binary). The gdb setup for qemu has been taken from Xv6 (present in the Makefile). The file .gdbinit.tmpl-riscv is used by make qemu-gdb for setting up the gdb for the RISC-V architecture.

Chapter 8

Conclusion and Future Scopes

1. Conclusion

The work done in this project was towards developing an operating system which can be used for educational purposes. It can be used in operating systems labs and for teaching operating systems. We tried to build an operating system with minimal features. The low-level drivers and the various abstractions set up in the R5O5 can help the students to directly get started with the development of OS. At present, R5O5 is all set for further enhancements with a solid low level interface and a program loader. The memory abstractions, paging and the disk abstractions are implemented and tested thoroughly. We have also set up interrupt handlers and the context switching part.

2. Future scopes

The present model of R5O5 can be improved further with the implementation of basic system calls and by introducing advanced OS concepts like semaphores, deadlock prevention mechanisms and threads. These additional features can be added to the codebase with minimal efforts now as all the building blocks required for development are set right now, and ideally can be completed within a period of one month work, other conditions remaining the same. Also, an extensive documentation can be helpful for the students to get familiarised with the existing codebase.