A Basic Microkernel for the RISC-V Instruction Set Architecture

Benjamin William Mezger
University of Vale do Itajaí, Brazil
ben@edu.univali.br

Fabricio Bortoluzzi
University of Vale do Itajaí, Brazil
Noroff University College, Norway
fb@univali.br

Cesar Albenes Zeferino
University of Vale do Itajaí, Brazil
zeferino@univali.br

Paulo Roberto Oliveira Valim
University of Vale do Itajaí, Brazil
pvalim@univali.br

Douglas Rossi Melo
University of Vale do Itajaí, Brazil
drm@univali.br

ABSTRACT

Computer processors provide an abstract model known as the instruction set architecture, which serves as an interface between the available hardware and the software. Application developers need to communicate with these types of hardware, and having to learn each computer specification is difficult and time-consuming. Operating systems provide an abstraction towards the available computer hardware and user software. They manage computer resources to enable application programmers to communicate with the available hardware. This work introduces an academic-oriented operating system for the RISC-V architecture, a de facto instruction set architecture standard, and compares the solution with other small operating systems using the same architecture. As the main contribution, this work provides an extensible operating system to introduce students to operating system development.

KEYWORDS

Operating Systems, Computer Architecture, RISC-V.

1 INTRODUCTION

One of the essential software of a computer system is the Operating System (OS), whose job is to provide application programmers an abstract set of resource interfaces of the hardware without worrying about the hardware complexity it needs to make use of [1]. Application programmers occasionally need to store data into a disk, read data from memory, connect and communicate with a remote host without worrying about how to access and use the functionality provided by the hardware, so it depends on the OS to manage these resources for them.

The OS acts as an intermediary between the user and the computer hardware. The OS provides an abstraction by enabling users to concentrate on doing their necessary tasks or software developers to engineer applications with ease [1].

With the demand for small, portable, and straightforward OS for deploying on embedded computers, there has been a broad category of operating systems developed for each purpose. In contrast, computer architecture research has tried to simplify computer components for easier integration and better performance [2].

The RISC-V is an open Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA) which enables internal modification and design. Berkeley University developed it to reduce the cost of software by permitting more reuse with an open ISA. The RISC-V implementation can be easily extended by designing custom extensions when additional functionality is desired [3].

The study of OS and computer architecture has been highly dependent on current technology. However, these technologies are either proprietary, hardening internal review, or too complex for academic classes of operating systems and computer architecture [4]. RISC-V breaks such barriers enabling students and lecturers to enrich their class material further.

A variety of kernels have been ported to RISC-V, including the Linux kernel [5], Amazon's FreeRTOS [6], Redox OS [7], and the Zephyr Project [8]. However, these solutions address the application in embedded systems and do not focus on beginners in the study of operating systems.

This work provides the project and implementation of a basic microkernel for academic purposes, by enabling students and lecturers to dive into computer system research and ease their understanding of computer architecture and operating systems development. This works aims to be used as a laboratory course material for both Operating System and Computer Architecture courses by providing a simple UNIX-based RISC-V kernel.

The remainder of this paper is structured as follows. Section 2 presents the background and a review of related work. Section 3 describes how the proposed microkernel was structured and implemented, and Section 4 describes how to adopt this work in OS classes. Section 5 discusses the obtained results, and Section 6 presents conclusions and future work.

2 BACKGROUND

2.1 Operating Systems

Computer software can be classified into two categories: system programs, responsible for managing hardware resources, and application programs, which execute the actual work that the user requests [9]. The OS's job is to provide application programmers a clean abstract set of resource interfaces of the hardware, leaving the hardware complexity to the OS, and enabling the programmer to focus on the application they are developing [1].

Operating Systems has three main objectives:

(1) They aim to offer *convenience*, making computer hardware more usable.
(2) They must enable computer system resources to be used efficiently, for example, making sure the memory is efficiently allocated.
(3) They must have the *ability to involve*, meaning that the OS should be engineered in a way that is easy to extend, develop, test and introduce new features [2].

The kernel is an essential part of the OS, and therefore, it has special rights comparing to other parts of the system [10]. It is loaded into memory on boot, initializes any required services, and waits for an event. An essential principle in kernel design is the separation of policies and mechanisms, determining what will be done and how it will be done, respectively. This separation is important, as policies are likely to change over time [11].

As systems tend to grow over time, they must be engineered carefully to work properly and be easily modified. The kernel can be engineered as a monolithic kernel or as a microkernel.

Monolithic kernels (Figure 1) runs all services such as processes, memory management, and interrupt handling in the kernel space, in a layered approach. The bottom layer is the hardware, and the upper layer is the user interface [11]. Typically, a monolithic kernel is implemented as a single process, with all its components sharing the same address space [2].

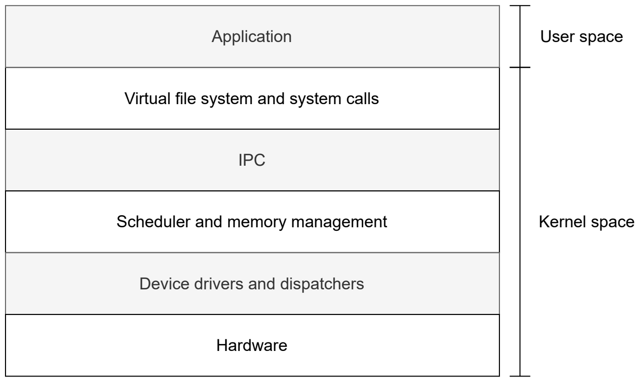
Application
Virtual file system and system calls
IPC
Scheduler and memory management
Device drivers and dispatchers
Hardware
User space
Kernel space

Figure 1: A common monolithic design

Microkernels (Figure 2) removes all nonessential components from the kernel and implements them as system and user-level programs [11]. These services are processes which are known as a server. They encapsulate memory management, process managing, Interprocess Communication (IPC), and so on [10]. The microkernel's main functionality is to provide a communication mechanism between the client program and the servers. If the client requires to read a file, it must interact with the file server through communicating indirectly by exchanging messages with the kernel [11].

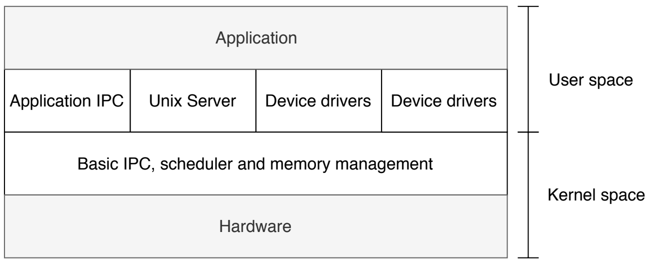

Figure 2: A common microkernel design

2.2 Computer Architecture

Although there are different distinctions made between computer architecture and organization, the former refers to what systems and application programmers see, which are the attributes that have a direct impact on the execution of a program, where the latter refers to the operational unit and its interconnections that make the architectural specifications. The ISA, the numbers of bits used to represent data types, the Input and Output (I/O) structure, and approaches for memory addressing are all organizational issues that need to be structured [2].

The computer organization creates a hierarchy of hardware attribute details transparent to the programmer, such as the interface between the computer and peripherals, the memory technology used, the type of processor and control signals [2]. Figure 3 presents a simple computer organization hierarchy.

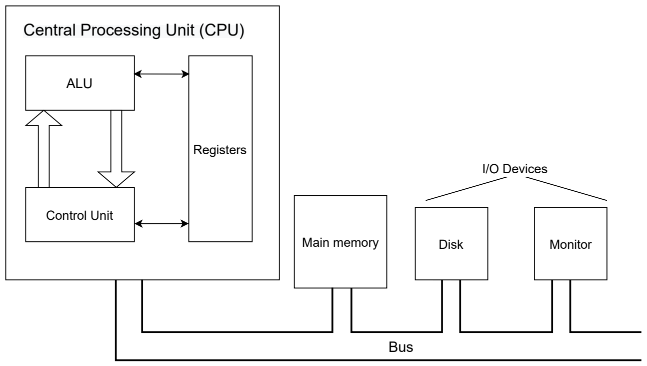

Figure 3: An overview of a simple computer organization

Computer architectures can be classified in Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC). RISC architecture provides a large number of general-purpose registers, the use of a compiler to optimize register use, a simple and limited instruction set, and optimizes the instruction pipeline [12]. In a CISC ISA, the compiler needs to do very little work to translate high-level language into assembly level. However, though it generates smaller code sizes, it needs higher cycles per second [13].

2.3 RISC-V

RISC-V is an open RISC ISA offering both 32- and 64-bit support. The RISC-V's ISA is modular, having a base architecture and several available extensions, enabling multiple variants to coexist. The base architecture is the RV32I, which will not be altered to enable programmers to rely on a stable architecture. The modules are standard extensions on which the hardware designers can choose whether to include them or not. The modular architecture enables small-scale applications with little energy consumption to be deployed with the required extensions to run [3].

RISC-V supports multiple software stacks, from executing a single application running on an Application Execution Environment (AEE) to multiple programmed OSs running on a single hypervisor. RISC-V provides three modes of execution: Machine-mode (M-mode), User-mode (U-mode), and Supervisor-mode (S-mode), having M-mode the only mandatory mode. Execution modes enable programmers to protect different software stack components and guide programmers on building secure systems. An operation not permitted by the privilege level will cause RISC-V to raise an exception, trapping into the underlying execution environment. Code executing in M-mode is commonly trusted, as it has low-level access to the machine implementation. The U-mode and S-mode are indented for conventional applications and OS, respectively [14].

2.4 Related Work

There are numerous OS ports for the RISC-V, from lower to higher complexity. The OSKernel is a small kernel written in Rust with basic scheduling support [15]. The RISC-V Proxy Kernel and Boot Loader project aim to provide a lightweight application execution environment that hosts statically-linked RISC-V Executable and Linkable Format (ELF) binaries [16]. The Core-OS-RISCV is an xv6-like OS for the RISC-V with Virtual Memory support, traps, and interrupts and process scheduling [17].

FreeRTOS has ported RISC-V RV32I and RV64I machine support, announced by Amazon in 2019 [6]. The Linux Kernel supports multiple architectures and has ported RISC-V in 2017 [18]. Redox [7] is an OS written in Rust with support to multiple architectures, including RISC-V. Finally, Zephyr, a real-time OS, has ported RISC-V to its kernel [8].

With the growing complexity of OS and computer architecture, many students rely on implementing a small subset of the OS or complex proprietary ISA. There are few basic academic-oriented resources for studying the RISC-V's software support due to its novelty.

Our work aims to provide an academic-oriented kernel for the RISC-V ISA, enabling lecturers and researchers to apply the proposed microkernel in OS and computer architecture classes and students to build their OS.

3 MICROKERNEL DEVELOPMENT

Figure 4 gives a brief overview of how the proposed microkernel is structured. From top to bottom, user-mode runs as the last layer of the abstraction, in which user programs and device drivers run. The second layer is where the Interprocess Communication (IPC), memory management, and the Central Processing Unit (CPU) scheduler executes. The third layer represents hardware and software communication in an abstract manner. The underlying layer shows SiFive's System-on-Chip (SoC) processor architecture doing the communication both ways with the kernel.

The kernel layer contains the bare minimum core functionality for the system to work. It is platform-dependent on the RISC-V, but it provides extensibility for different architectures. The primary function of the last layer is to provide a set of privileged kernel calls to drivers and servers, including writing to I/O ports and copying data between addresses and spaces.

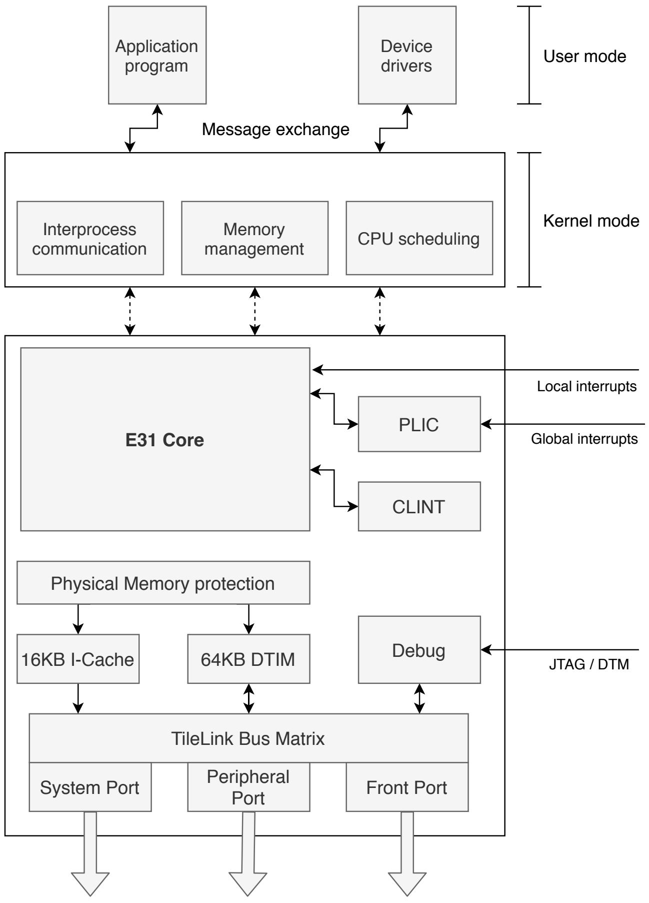

Figure 4: The microkernel and the HiFive1 board overview

3.1 Materials

This work uses the C language and RISC-V assembly language for the implementation of the microkernel. The latter is used to initialize the required CPU components and configure the processor to run in kernel-mode, user-mode, handling traps, exceptions, and saving and restoring context.

Qemu's RISC-V machine platform [19] is used to run the generated kernel binary and remote debugging with GNU GDB debugger. SiFive provides their custom Software Development Kit (SDK) for developing on their custom development board. However, to enable interoperability against different RISC-V boards and machines, the kernel is compiled without the SDK, which results in a smaller binary size. The kernel is also compiled without GCC's standard library and implements `crt0.S`, required by the compiler, enabling custom execution startup routines linked into a C program, for performing any initialization work required by the kernel before calling the program's `main` function.

3.2 Processes

A process may be represented as an entity that consists of some elements. Two essential elements of a process are program code and a set of data associated with the code. While the process is executing, it can be uniquely characterized by some elements, for example: (i) the current state, such as running if the process is currently being executed; (ii) the Program Counter (PC) pointing to the next instruction to be executed; (iii) memory pointers of the program code and data associated; and so on [20]. This work provides two distinct C functions to simulate a process. Therefore, it requires recompilation of the kernel when updating one of the process functions if one needs an update.

3.3 Trap handling

Traps are ways of the CPU notifying the kernel when an error occurs, or action is required. RISC-V uses a function pointer to a physical address in the kernel. When a trap occurs, RISC-V switches to M-mode and jumps to the function address.

When the kernel is booting, it calls `_setup_mtrap` for setting `mtvec` to the address of `_trap_entry`, finally, it enables RISC-V's Direct Mode Access (DMA) and M-mode interrupts. The state diagram of Figure 5 shows the `_setup_mtrap`. When a trap happens, RISC-V will jump to the address of `_trap_entry` (Diagram 6). `_trap_entry` will save the current state of the registers by allocating space in the stack pointer, saving the current stack pointer to the first function argument, and jump and link to `trap_handler`. The C function expects to receive a `trapframe_t` with the previous stack information. The `trapframe_t` has the previous register's state, the trap code, `epc`, `cause` and any required RISC-V Control Status Register (CSR) for handling traps.

The trap handler will catch the error and jump to the function related to the error code, treat it, return to `trap_handler`, and finally, restore the context and return. The errors are statically mapped into an array of function pointers. This approach enables easy jump to trap handlers by using the map as a jump table.

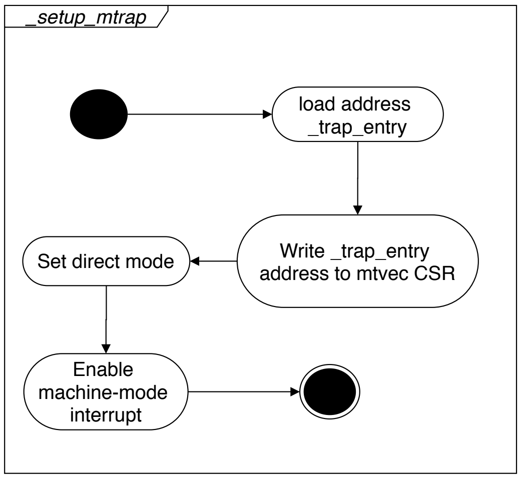
_setup_mtrap
load address _trap_entry
Write _trap_entry address to mtvec CSR
Set direct mode
Enable machine-mode interrupt

Figure 5: State diagram of the trap setup

3.4 Execution modes

The system memory is structured into segments, where each segment stores the context of a mode (U-mode and S-mode) with the appropriate read, write and execute flags. The S-mode code

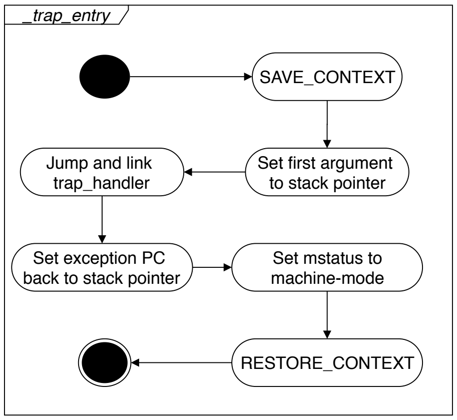
_trap_entry
SAVE_CONTEXT
Set first argument to stack pointer
Jump and link trap_handler
Set exception PC back to stack pointer
Set mstatus to machine-mode
RESTORE_CONTEXT

Figure 6: State diagram of the trap entry

has a predefined protected memory location, any other mode that tries to access the S-mode memory address space will receive an exception. When the kernel has been set up, it allocates a memory segment for the U-mode before switching to it. The U-mode address space has read, write, and execute permissions within its segment.

When the CPU enters S-mode, it initializes the CPU timer and software interrupt handlers required to be treated if an interrupt occurs. Figure 7 shows all context the supervisor keeps track of when running. The S-mode controls U-mode related tasks, some of the available hardware and software interrupt. The supervisor keeps the U-mode context in a global C-language `struct`, keeping a pointer to all of these components.

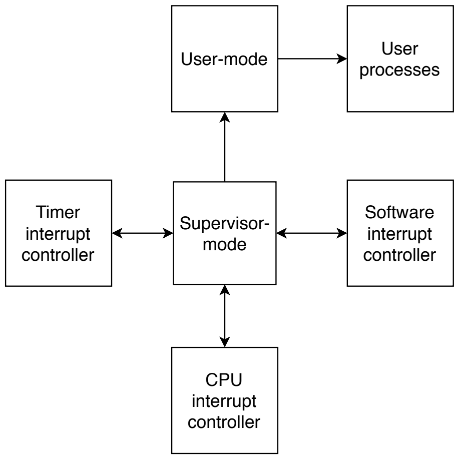

Figure 7: The supervisor hierarchy overview

The U-mode is responsible for creating a map of all the required processes to run. This mode runs indefinitely unless any error happened or an interrupt is triggered. While U-mode is being executed by the CPU, the timer will trigger an interrupt when the S-mode is supposed to run again. This scenario is illustrated in Figure 8.

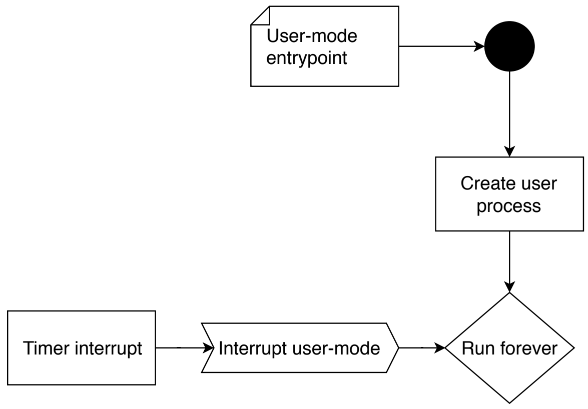

Figure 8: Activity diagram of the user-mode

3.5 Extending the kernel

The kernel is designed to facilitate development, support multiple architectures, and facilitate new components to be implemented in the kernel. Figure 9 introduces how the code-base is structure.

The driver directory is shown in Figure 9 (a). It holds all available drivers supported by the kernel, enabling each to be compiled separately by updating the `make.config` file. It also specifies which drivers should be compiled built into the kernel.

The kernel supports multiple architectures by creating the code in the `arch` directory, as seen in Figure 9 (b). Although this work provides only RISC-V support, researchers can easily add new architectures by using different architectures as class material.

The supervisor, scheduler, and any kernel-related code are located in the `kernel` directory, as shown in Figure 9 (c). Although `main.c` initializes the kernel, the `init.c` handles the architecture type during compilation by checking the specified C-language macro passed as a compiler argument.

Public and private Application Programming Interface (API) headers are located in the `include` directory in Figure 9 (d), enabling APIs to be easily accessed, modified, and propagated against the kernel upon compilation.

The kernel requires a shared library with general utilities, such as printing and string manipulation, enabling all packages to use the available library code, like `printk` for outputting kernel information through Universal Asynchronous Receiver-Transmitter (UART), as seen in Figure 9 (e).

The kernel compilation is managed with a global `Makefile`, which recursively reads the `make.config` for each package, enabling packages to be individually compiled with their respective flags and options.

4 ACADEMIC ADOPTION

4.1 The kernel as a course material

This work can be applied in OS classes in three main topics: (i) processes and threads, (ii) memory management, and (iii) I/O operation. Due to the lack of file-system support and a robust memory management system, some topics may have to be implemented into the kernel by the student during or before the course.

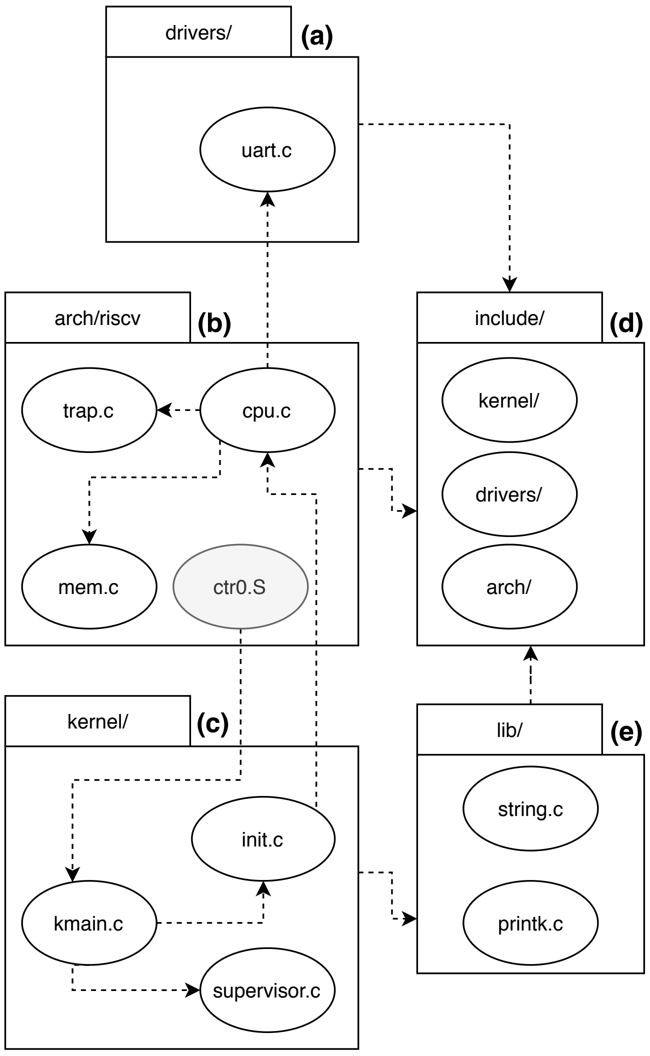

Figure 9: The microkernel package organization.

This work provides static user-programs that enable students to attach processes to the scheduler in user-mode. This approach improves the student's understanding of processes and scheduling and enables the implementation of complex process management solutions.

Although this work provides segmented memory, students can either implement a better memory-model or understand how memory is handled within kernel-mode and user-mode. The student may either run and modify the code during development or analyze memory with a debugger or Qemu-like system.

The I/O subsystem enables an understanding of how the system communicates with the external environment. The kernel currently supports only UART-based communication, which could help students understand how the I/O system works and port new I/O components into the kernel.

```
src / supervisor . c :32:  [supervisor_mode_entrypoint] 1 Entered supervisor mode with kcontext 0x80001418
src /cpu.c :130:           Number of CPU harts is 1
src /cpu.c :141:           Address of cpu is 0x80000c5c
src / interrupt . c :194:  [ init_interrupt ] 1 Registering interrupt handler
src / interrupt . c :150:  [ _init_cpu_intrp ] 1 Address of CPU 0x80000c5c interrupt controller is 0x80000058
src / interrupt . c :182:  [ _init_software_intrp ] 1 Software interrupt 0x80000c54 enabled for cpu Ox80000c5c
src / exceptions . c :54:  [ _init_cpu_intrp ] 1 Failed to register CPU 0x80000c5c EXCP handler 0x200103be for code 12
src / exceptions . c :54:  [ _init_cpu_intrp ] 1 Failed to register CPU 0x80000c5c EXCP handler 0x200103be for code 13
src / exceptions . c :54:  [ _init_cpu_intrp ] 1 Failed to register CPU 0x80000c5c EXCP handler 0x200103be for code 14
src / exceptions . c :54:  [ _init_cpu_intrp ] 1 Failed to register CPU 0x80000c5c EXCP handler 0x200103be for code 15
src /pmp.c :14:            [init_pmp] 1 Address of PMP is 0x80000d80
src /pmp.c :24:            [init_pmp] 1 Address of PMP config is Oxf
src / supervisor . c :70:  [supervisor_mode_entrypoint] 1 Created user SP 1 0x80001108
src / interrupt . c :107:  [ _init_timer_intrp ] 1 Address of Timer interrupt controller is 0x80000c54
src / interrupt . c :122:  [ _init_timer_intrp ] 1 Timer interrupt EXCP registered for 0x80000c54 id 7
src / interrupt . c :132:  [ _init_timer_intrp ] 1 Timer interrupt 0x80000c54 enabled for cpu Ox80000c5c
src / supervisor . c :74:  [supervisor_mode_entrypoint] 1 Droping to user−mode
```

Figure 10: Boot log

The package architecture shown in Figure 9 enables a student to choose a component to study: (a) relates to device driver code and communication ports; (b) holds all RISC-V related code, trap handling, and memory management; (c) contains kernel-related logic, such as supervisor and initialization; (d) holds all interfaces and C headers; and (e) keeps the shared library, such printing, and string support. This structure facilitates finding a way around the kernel.

4.2 Constructionist Approach

Relevant principles for the organization of complex learning processes are defined in [21]. Based on those principles, it should be possible to stimulate the interaction between students and the OS by:

- interaction spirals, where the student can continuously build on what they have already learned.
- lectures covering key topics, leaving students with the responsibility to play an active role in filling the gaps left by theory with practice.

As a result of this constructivist approach, students should feel empowered for autonomous interaction with his learning object, the OS. Lecturers, on the other hand, should place themselves as mediators capable of answering questions in a reflexive fashion, causing students to think instead of reacting to ready-made answers.

Our constructionist goals when creating the OS target the acquisition of progressive success, to be achieved along the time, based on these assumptions:

- Students are not familiar with operating systems when they start having classes. Experimentation to be held on the OS must facilitate taking the right decisions and following the right direction.
- The environment where students practice is adequate, and it should not add unnecessary complexity for the interaction with the OS when testing for a hypothesis.
- Hacking the OS should be as close as possible to more complex systems, given the proportions.
- Group interaction, although optional, should be made possible and bring richer experiences.

5 RESULTS

5.1 Kernel execution

The supervisor first initializes the required CPU utilities before initializing all the interrupt handlers and exception trap handlers. By default, interrupts are disabled when executing the supervisor, preventing any interrupt from happening while configuring any components or executing any other task. This initialization is illustrated in Figure 10, outputting the supervisor logs.

The kernel then creates the U-mode memory segment and locks its memory segment before switching to U-mode. The lock is required to avoid any other mode to read, write, or execute from the kernel segment. Finally, the supervisor starts the timer and switches to U-mode, enabling user-related processes to run.

The supervisor logs information about the important actions as they happen, such as initializing or catching an interrupt. All interrupts must be initialized before dropping to U-mode, but they may be enabled or disabled at run time. In U-mode, the call to C's `libc` function `printf` raises a system call exception, allowing the supervisor to catch it, treat it, and return the result to the caller.

When the kernel raises a timer interrupt, it switches back to the supervisor, calls the interrupt handler, disables the timer within its function scope, and runs the user-space process. Any interrupt generated by the U-mode is triggered and returned to the caller.

5.2 Cost and performance

The kernel is compiled against GCC's version 8.3.0 with no optimization flags enabled, targeting the Sifive's HiFive1 Rev-B development board, among with FreeRTOS and Zephyr for size comparison.

The binary size is computed by running GNU's `stat`, which displays information related to a file. The resulting binary is 123KB, 4.3 times smaller comparing to FreeRTOS basic example application, which resulted in a binary of 560KB, and 2.7 times smaller than Zephyr (367KB) with the default kernel configuration. This result is due to the higher complexity of the compared kernels.

Given the HiFive1 board runs a SiFive E31 core, and can operate up to 320MHz, we can divide the cycle count from CSR `cycle` register by the operation frequency, resulting in the execution time. Table 1 presents the cycle count of the supported modes.

Context	Cycles	Time (ms)
Boot process	48,872	9.16
Supervisor-mode initialization	290,927	54.54
User-mode initialization + process	39,379	7.38

Table 1: Kernel cycle count per mode

The total kernel execution time, from boot up to the execution of a `hello world` process in U-mode, took 37,300,080 cycles. Comparing with FreeRTOS executing the same task, the cycle count was 36,295,935 cycles. When comparing with Zephyr, it took a total of 36,562,675 cycles, being very close to FreeRTOS. The complexity and code optimization of both Zephyr and FreeRTOS are much greater in comparison to this work, therefore they are expected to have fewer cycles. However, our proposal presented a similar cycle count.

6 CONCLUSIONS

The developed kernel has shown a cycle count similar to FreeRTOS and Zephyr, and a smaller binary size due to its simplicity and lack of compiler and code optimization. The comparison against other kernels helps lecturers to decide whether to apply this kernel in laboratory classes or the others.

The simplicity of the kernel enables a newcomer to center the attention on studying how an OS can be developed and integrated with the available hardware. The provided kernel organization enables porting new architecture and features when required. RISC-V helps academics to entirely study the design and specification of the architecture, due to its open ISA design and rich software support.

This work presents a small and extensible kernel by providing a segmented memory management, process scheduling, supervisor and user-mode support, and a structure for porting new architecture. This work is not intended to run on a production environment, as it lacks security measures, scalability and has little driver support. The missing features can all be improved in future work by students and lecturers during classes as a learning opportunity or as a research topic by extending or improving this work.

ACKNOWLEDGMENTS

This work was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) [grants number 315287/2018-7 and 436982/2018-8],

REFERENCES

[1] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014. ISBN 9780133591620.
[2] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall Press, Upper Saddle River, NJ, USA, 7th edition, 2011. ISBN 9780132309981.
[3] David Patterson and Andrew Waterman. *The RISC-V Reader: an open architecture Atlas.* Strawberry Canyon, Berkeley, California, 2017.
[4] M. D. Hill, D. Christie, D. Patterson, J. J. Yi, D. Chiou, and R. Sendag. Proprietary versus open instruction sets. *IEEE Micro*, 36(4):58–68, 2016.
[5] Damien Le Moal Atish Patra. Linux kernel on risc-v: Where do we stand?, 07 2018. Western Digital.
[6] Jeff Barr. New – risc-v support in the freertos kernel, February 2019. URL https://aws.amazon.com/blogs/aws/new-risc-v-support-for-freertos-kernel/.
[7] Jeremy Soller. Redox-os, 04 2015. URL https://www.redox-os.org.
[8] Linux Foundation. Zephyr, 05 2019. URL https://www.zephyrproject.org/zephyr-rtos-featured-in-risc-v-getting-started-guide/.
[9] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. ISBN 0131429388.
[10] Benjamin Roch. Monolithic kernel vs. microkernel. *TU Wien*, 1, 2004.
[11] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied operating system concepts.* John Wiley & Sons, Inc., New York [u.a.], 2001.
[12] William Stallings. *Computer Organization and Architecture: Designing for Performance.* Prentice Hall Press, Upper Saddle River, NJ, USA, 8th edition, 2009. ISBN 9780136073734.
[13] Andrew S. Tanenbaum. *Structured Computer Organization (5th Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. ISBN 0131485210.
[14] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. The risc-v instruction set manual volume ii: Privileged architecture ratified version 1.11. Technical Report v.20190608, EECS Department, University of California, Berkeley, Jun 2019. URL https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf.
[15] Stephen Marz. Osblog, 10 2019. URL https://github.com/sgmarz/osblog. Github Repository, Commit 76715da537e8d07e305d1c76faa925b58579291b.
[16] RISC-V. Risc-v proxy kernel, 06 2019. URL https://github.com/riscv/riscv-pk. Github Repository, Commit 46cd5082c5d324bb843be35d8130aa9d44068d7d.
[17] Alex Chi. core-os-riscv, January 2020. URL https://github.com/skyzh/core-os-riscv.
[18] Linux Torvalds. Risc-v port for linux 4.15 v9, 11 2017. URL https://lkml.org/lkml/2017/11/15/640.
[19] Michael Clark. Risc-v qemu part 2: The risc-v qemu port is upstream, April 2018. URL hhttps://www.sifive.com/blog/risc-v-qemu-part-2-the-risc-v-qemu-port-is-upstream.
[20] William Stallings. *Operating Systems: Internals and Design Principles, 9/e.* Pearson IT Certification, Indianapolis, Indiana, USA, 9th edition, 2018. ISBN 9352866711.
[21] Jerome Bruner. *Toward a Theory of Instruction.* Harvard University Press, Harvard, 1966. ISBN 0674897005, 9780674897007, 0674897013, 9780674897014.