BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2025
https://excel.fit.vutbr.cz

MatteOS

Matěj Bucek*

Abstract
This work focuses on the design and implementation of a simple operating system for the RISC-V architecture in C++. The system utilizes OpenSBI and U-Boot for bootstrapping and provides fundamental functionality, including memory management and protection, process scheduling, device drivers, and partial support for the FAT32 file system. The result of this work is a functional operating system kernel that enables multitasking and basic hardware interaction. The main contribution of this work is the creation of an open platform for RISC-V experimentation and low-level software development.

*xbucek17@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The growing popularity of the RISC-V architecture in recent years has sparked significant interest across both industry and academia. Its openness, extensibility, and potential to rival established architectures like x86 and ARM make it a compelling platform for research and education. Despite this momentum, accessible resources for learning low-level system programming on RISC-V remain limited. There is a strong need for educational tools that provide hands-on experience while remaining simple enough to be understood and extended.

Existing operating systems with RISC-V support, such as Linux, are often too complex and difficult to grasp for developers new to this field. While many OS development tutorials exist, they are typically either x86-focused, overly simplistic, or tailored to a specific hardware configuration with no real device detection. This creates a steep barrier for those who wish to explore RISC-V or dive into operating system development.

A few educational projects and tutorials for RISC-V exist, such as the OS blog by Stephen Marz [1], which serves as a solid starting point. It introduces core RISC-V concepts, memory management, interrupt handling, processes, and even a block device driver. Although an excellent resource, it has limitations: it does not use a bootloader, runs entirely in the so-called machine mode (M-mode), lacks dynamic device detection, and relies heavily on hard-coded addresses.

This work presents a minimal yet functional operating system kernel for the RISC-V architecture, implemented in C++. It aims to strike a balance between simplicity and realism, offering a manageable codebase while supporting essential OS features. The system includes memory management and protection, process scheduling, basic device drivers, partial FAT32 file system support, and more (see Figure 1.). It leverages industry-relevant technologies and standards such as OpenSBI, U-Boot, Flattened Device Tree (FDT), VirtIO, PLIC, UART (NS16550A), thereby increasing its practical relevance and educational value.

The primary contribution of this work is the development of an open, educational operating system for RISC-V that enables students and developers to explore kernel development in a real-world context. It offers hands-on experience with key subsystems and hardware interfaces, bridging the gap between oversimplified tutorials and overly complex production systems. By focusing on clarity and modularity, the system serves as a practical platform for experimentation, teaching, and further development in the field of low-level RISC-V software.

2. Technologies

The core operating system is written in C++, complemented by RISC-V Assembly for low-level components. The use of an object-oriented design aligns well with the modular structure required in an operating system.

QEMU serves as the virtualization platform. It was the first emulator to support RISC-V and integrates well with the other components of the system. OpenSBI acts as a RISC-V standard supervisor interface, providing a hardware abstraction layer that simplifies and accelerates OS development.

U-Boot, recommended for use with OpenSBI in the FW_DYNAMIC configuration, works seamlessly within this architecture. The Flattened Device Tree (FDT) provides a standard method of hardware description, allowing the system to automatically detect and configure attached devices—an approach commonly used in embedded environments.

Finally, VirtIO is employed for efficient device virtualization, offering a standardized interface for emulated hardware components.

3. Architecture

MatteOS is a monolithic kernel that runs a shell as a userspace process. Internally, the kernel is organized into several distinct subsystems, as illustrated in Figure 1. The memory management subsystem is responsible for paging and kernel memory block allocation. System management maintains essential system information and includes a built-in timer. The device and driver subsystem implements support for several hardware components, including drivers for the SiFive PLIC, NS16550A UART, and VirtIO block devices. For device discovery, MatteOS relies on the DeviceTree protocol and uses the SBI (Supervisor Binary Interface) for platform-specific interactions. Process management in MatteOS is thread-based—each process consists of one or more threads, which are scheduled independently by the kernel. The Virtual File System (VFS) abstracts access to file systems and provides a consistent interface for implementing new ones. At present, support is available for FAT32 and a custom in-memory file system, RamFS. Finally, the interrupt management subsystem handles the configuration and abstraction of interrupt controllers. It defines how interrupts are processed and enables context switching, which allows the kernel to transition between userspace and kernelspace execution.

4. Achievements

The entire boot process was successfully configured and implemented in MatteOS, utilizing U-Boot and OpenSBI on the QEMU platform. All subsystems shown in Figure 1. were at least partially implemented. The Memory Manager handles physical memory allocation and address-space virtualization using three-level page tables, and provides a dedicated kernel memory allocator.

The Device Tree (FDT) is used for automatic hardware detection and configuration. Devices and drivers are managed through Device and Driver Managers, with support for block devices, console devices, and the interrupt controller. Drivers were implemented for VirtIO block devices, the NS16550A serial console, and the PLIC interrupt controller.

The Virtual File System (VFS) supports two backends: a partially implemented FAT32 driver and RamFS, a memory-based file system for virtual files, directories, and device interaction.

Support for processes, context switching, and system calls was also implemented. The Process Manager handles creation and memory mapping of processes, while the Scheduler uses the Round-Robin algorithm. Timekeeping and timed interrupts are provided by the Timer subsystem. A simple shell interface demonstrates user interaction with the system.

The Figure 4. illustrates the interaction with MatteOS after a successful boot, showcasing a simple command-line interface. Two user processes are executed: the first is a background process that prints "`Hello from dummy process!`", attempts to open the file `hello.txt` from the FAT32 file system, and reports the success of this operation. It then continues to run in the background, periodically confirming its activity.

The second process runs in the foreground and provides a simple shell interface. It supports commands such as `excel`, prints "`Hi, Excel@FIT!`", and also `stats`, which displays a summary of basic system statistics.

5. Future goals

Future development of MatteOS aims to include an ELF loader and a system library accessible from userspace to support more complex applications. The FAT32 file system will be fully implemented, and VirtIO support extended to additional device types. Plans also include creating a custom IDL for code generation and implementing the necessary drivers to enable booting on the MPi-MQ1PL platform.

Acknowledgements

I would like to thank my supervisor prof. Ing. Tomáš Vojnar, Ph.D. for his valuable help and advice.

References

[1] Stephen Marz. The adventures of os: Making a risc-v operating system using rust. online, 2019. https://osblog.stephenmarz.com/.