UNIVERSITY OF NAIROBI

EMBEDDED OPERATING SYSTEM FOR IOT DEVICES

By James Kiarie

Registration Number: P15/ 1667/2019

Supervised by Dr. Wanjiku Ng’ang’a

August 2023

This project proposal has been submitted in partial fulfillment of the requirements of the Bachelor of Science in Computer Science as offered by the University of Nairobi.

Declaration

This project is my original work and to the best of my knowledge, this work has not been submitted for any other award in any University.

Student: **Wagura James Kiarie** Registration Number: **P15/1667/2019**

Signature: ____________________ Date: __________________________

This project has been submitted as a partial fulfillment of the requirements for a Bachelor of Science in Computer Science of the University of Nairobi with my approval and the University Supervisor

Supervisor: **Dr. Wanjiku Ng'ang'a**

Signature: ____________________ Date: __________________________

Acknowledgements

Dr. Wanjiku, for kind guidance.

Phillip Opperman, writer of “Writing an OS is Rust” blog. The blog single-handedly made OS development stop looking like a craft reserved for wizards.

Stephen Marz, writer of the “The Adventures of OS” blog. Took the OS development tutorials beyond just writing “hello world”.

Lin Clark, for simplifying complex webassembly jargon using mortal language and simple cartoon illustrations.

The writers of “Wasmachine: Bring IOT up to speed with Web Assembly OS” research paper. This was the bedrock of this project.

Kevin Hoffman, the author of the book : *Programming WebAssembly with Rust .* The book captures the use of wasm off-browser in an elaborate manner.

All the Rust, wasm and riscv teams for creating awesome and well-thought-out tech. Tech that will hopefully stand the test of time.

Table of Contents

Table of Figures.

CHAPTER 1: INTRODUCTION

1.1 Background

It is a dream of many a computer scientist to control matter. To make all matter all around us programmable.

One step towards this direction is through embedded programming. Embedded programming on its own might sound simple, all you have to do is to read the data sheet associated with the specific piece of hardware, abstract that data sheet using data structures and finally manipulate the exposed registers using MMIO programming. Deployment looks simple too, sealing off the programming jack pin sounds enough.

With the progress of IoT, things have changed. Things have become complex. Both embedded hardware and software are more complicated. For example, reading the hardware data sheet is not enough, you have to understand the ISA associated with that hardware, the corresponding hardwired security implementations and all additional compatible circuit extensions. Embedded software now deals with network connectivity, bringing in a whole set of cybersecurity modules and cloud integration intricacies.

The general IoT architecture is depicted below [2]:

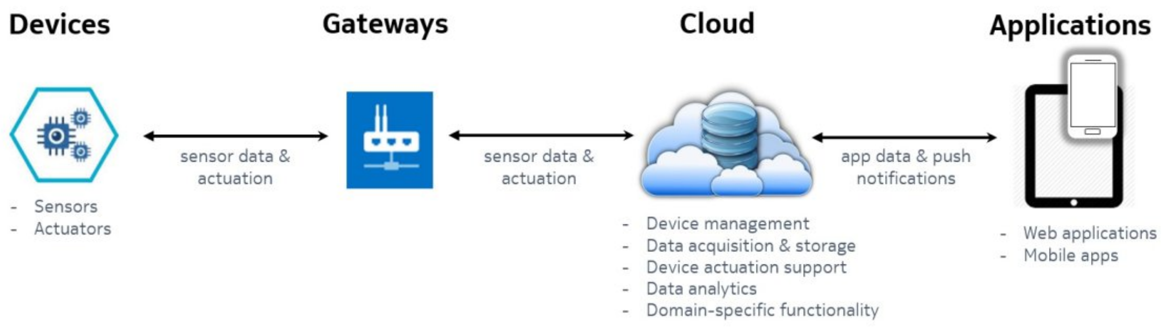


Fig. 1. Common generic end-to-end (E2E) IoT architecture.

Figure 1: The General IoT Architecture

As a result of the above IOT architecture, building an average IOT system requires the development team to use a lot of unassociated technologies and techniques. For example; Assembly language for the hardware, C/Rust for embedded programming, distributed programming for building the immediate network infrastructure, Docker and Kurbenetes for deploying and orchestrating micro-services over the cloud, web languages to build a website that acts as a remote interface to the embedded devices. Moreover, the development team might also be required to build an accompanying mobile application that acts as an interface to the embedded devices. A language like Kotlin or Swift might be used to create that mobile application.

Here is an illustration of the different technologies that may get used in tandem with each other [2]:

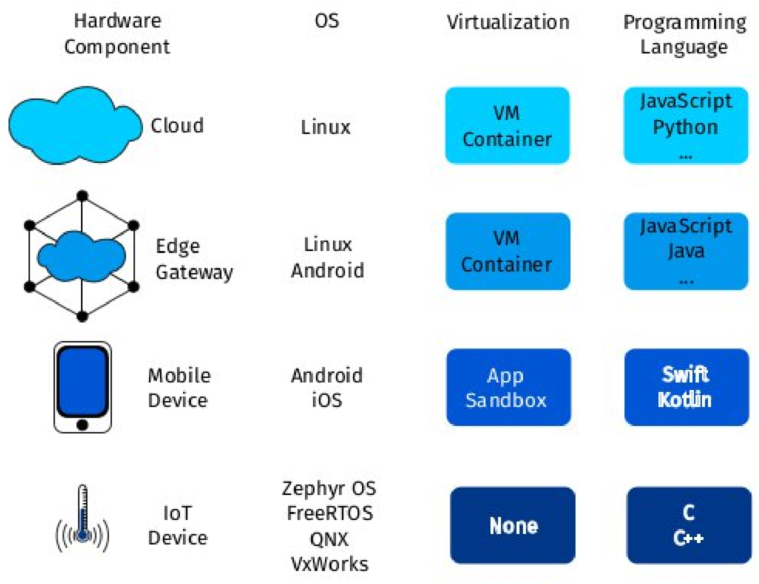


Figure 2. Illustrated Example of Current Platform Diversity in the Context of IoT Systems.

Figure 2 Illustration of the Platform Diversity in a generic IoT architecture.

This complexity can be traced to three causes; One is, heterogeneity of the devices used in terms of hardware and runtime implementations. Two, inconsistency of the communication protocols used between the heterogeneous devices and finally, the necessity to use specialized tools and programming languages.

Below are the solutions as proposed in the paper by Mikkonen [2].

To solve the problem of heterogeneity of devices, Mikkonen proposes a universal API that describes the abstraction and interaction with known generic hardware devices. For example, the API should specify how to abstract and interact with generic cameras or heat sensors. This solution does not force people to use the same hardware, but it makes sure that all known devices are accessed in a determinate way.

To solve the inconsistency of communication protocols used between heterogeneous devices, Mikkonen proposes a standard set of communication protocols to be specified for each known device-device interaction. The paper was in support for the use of existing web protocols for simplicity and easy adoption.

The above two problems have been partially solved by initiatives such as Web of Things Standards (WoT) and Open Communication Foundation (OCF).

The third problem; having too many tools and programming languages involved in development, causes two kinds of complexity. One, It makes it hard for a small team to properly master and keep up with the tools involved. Two, It makes it hard to migrate code from one node to another within this distributed network of IOT devices. Code migration portability is important because of reasons like code re-usability or code maintainability. However, the major reason why code migration portability is important is because current IOT favors edge computing (fog computing). In edge computing, it is expected that at some point, the edge device may run the algorithms that are usually ran in the cloud (for example a data processing algorithm).

If the data processing algorithm found in the cloud node is written in python, does that mean that the developer has to re-write that algorithm in C/Rust just for it to run on the edge device?

This code portability problem can be solved through two methods: One, finding a programming language or tool that gets explicitly and solely used throughout the project OR finding a way to package the programs written in different languages into portable packages.

Here is an image showing an example implementation of the two solutions mentioned above; **using only one** programming language for all the development modules **versus** using many different languages for the development of the same modules but limiting the modules to be packaged using **one** consistent packaging system:

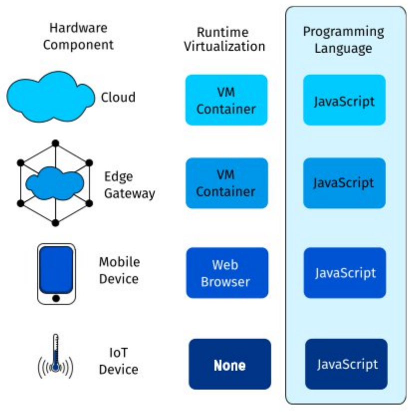


Figure 3: An Illustration of how one language can be used throughout the whole architecture.

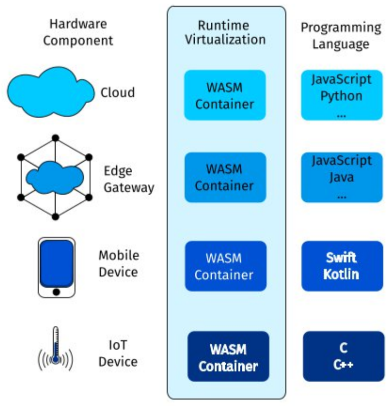


Figure 4: An Illustration of Multiple Languages being used to create modules that run on-top of a constant Runtime

Solution one is unfavorable, each language or tool has its own special purpose. It is not okay to use a single language to solve every problem, a good carpenter does not use a hammer to fix everything. For example, using JavaScript to program micro-controllers using tools like Espruino is feasible, however this decision might cause inefficiencies. JavaScript is an interpreted language, the JIT compiler continuously runs as the program runs, this might guzzle too much energy. Moreover, if you derail off the JIT's happy path, the performance of the program might become unpredictable. JavaScript in this case is a feasible solution, but an expensive one.

A more suitable use of tools would be: to use JavaScript to make the User-facing apps and use C/C++/Rust to write code for the embedded devices. Python and Julia are preferred for writing the data processing algorithms.

Solution two is not a perfect solution, but it is considerably favorable; developers use the languages they are comfortable in, languages that were purposefully made for those jobs; but all the programs get compiled to web assembly binary. In a sense, all programs have been written in one language, targeting one machine and predetermined runtime.

This solves the code migration problem by making the resultant programs portable. Each program can run on any machine, as long as that machine can run a webassembly runtime. The portability is achieved by packaging the programs as containers or setting up compatible virtual machines on top of the incompatible execution environments. However, solution two does not reduce the number of technologies being used simultaneously. Developers still use a combination of many languages and tools, and that is an understandable trade-off.

It is hard to build a one-size-fits all tool or language; at least solution 2 acknowledges this pseudo-fact, unlike solution 1 which forces you to use only one language throughout the development.

Solution two is majorly implemented through the use of container technologies like Docker. There have been propositions and early implementations of using web assembly containers instead of mainstream solutions like Docker. This is because webassembly containers are comparably smaller in size and the corresponding wasm runtimes also have a small memory occupancy.

But if we look at the comparison objectively, Docker has the same properties as wasm functionality-wise. The true strength of wasm lies in the implementation of a universal host system API; WASI (Web Assembly System Interface) and its default capability-based security.

With WASI, containers can be packaged for any host system, be it an operating system or an Application like Word or VScode. As long as the API of that host system is loosely defined, you can interface a wasm module with it. WASI is just like POSIX, but it is meant for more than just operating systems.

It is better to choose tools whose design and purpose will last for a long time. WASI might last for a long time.

1.2 Problem Definition

1.2.1 Human Problem:

Developers are enticed to package their Software into wasm modules so as to reap its benefits; portability, compactness and security. Wasm containers have the potential to be the de-facto standard for deploying IoT apps. However, developers find it hard to deploy and run wasm containers on embedded devices at near native speed.

The wasm ecosystem is immature **but** quickly growing. Most of the available wasm tools are browser-focused, only a few projects focus on building wasm tools for the IOT space.

One tool that would be beneficial to have is an embedded operating system that safely executes wasm containers on top of CPUs at near-native speed.

At the moment, developers who need such a tool have resorted to two solutions;

On one hand, developers embed the wasm runtimes in their bare-metal program executables. Let us call this method Method_A. For example, the developer might create a rust binary that has used a wasm-runtime as an embedded crate. This method is not favorable because it means that the developer will not be able to run multiple programs on the same riscv CPU. To run a new program, the developer has to adjust the code for the executable and re-compile it. Having an OS enables one to run multiple isolated processes on top of a shared hardware, without having to recompile the kernel.

Furthermore, Method_A is unsuitable for modern IOT development because it makes remote code modification impossible. It forces recompilation of the entire executable with each update.

On the other hand, developers resort to running the wasm runtime on top of another operating system like Tock or embedded Linux. Let us call this method Method_B. This method allows swapping of programs and remote code modification. You get to enjoy the capability security system of wasm and use the extensible WASI interface. However this method is unfavorable because the layers of abstraction sacrifice out the execution speed of the wasm programs. Under Method_B, the wasm programs run on-top of a wasm runtime that runs on-top of an OS. Those are too many layers of abstraction.

1.2.2 Technical Problem

Running wasm containers on top of a runtime that runs ontop of an OS is slow because the code execution experiences two kinds of latencies ; System-call-normalization latency and context-switching latency.

The System-call-normalization latency is caused by the normalization process. The normalization process is the mechanism of converting a WASI function call to have the same function signature as the host’s kernel system calls. Normalization happens when there is a difference in the function prototypes of the WASI API syscalls and the Native syscalls.

System-call-normalization happens during the compilation of wasm files to the corresponding native binary files. When an AOT compiler is used, the normalization latency does not affect the execution speed of the binary file at runtime. If a JIT compiler or an interpreter is used, the System-call-normalization latency **will** affect the execution speed of the binary file at runtime.

Context switching latency occurs when the wasm executable calls for a system call function via WASI API. This is because the CPU stops executing the wasm process and instead switches to the kernel process, just to execute a system call function. The switching between the two processes is costly but necessary, but this is made worse by the fact that the CPU has to additionally switch CPU modes from user-mode to machine-mode or supervisor-mode.

1.3 Objectives

For the project to be complete, the following research and implementation objectives had to be completed.

Research Objectives

i. Understand the different kernel designs and implementations.
ii. Understand the implementation aspects of wasm containers and wasm runtimes.
iii. Understand the IOT development infrastructure and figure out the problems faced in IOT development.
iv. Understand the Docker technology and its downsides in comparison to wasm.

System Objectives

a. Design a kernel that has basic functionalities needed in the embedded space. The design should outline how the modules interact with each other and how the Webassembly runtime should get ported as the program-loader and program-executioner.
b. Build a kernel that supports user input and output.
c. Build a kernel that has a virtual page management system.
d. Build a kernel that supports a file system.
e. Extend the minimal kernel to incorporate a Webassembly runtime as its program-loader and executioner.
f. Modify the extended kernel to export its system calls using WASI function prototypes.

1.4 Project Justification

Web Assembly programs execute slowly in embedded devices when compared to native programs. Native programs have good performance but with poor portability and security control. This project aims to implement an OS that makes web assembly programs run at near-native speed in embedded devices while still taking advantage of the capability-based security system of wasm. You get the performance of native execution and the portability and capability-based security of wasm.

The resultant operating system will be a proof of concept that wasm-based operating systems will be the default operating systems in devices used in the IOT infrastructure. The OS acts as a proof of concept of the wasmachine proposed in the paper [4] with the aim achieving an isomorphic IOT architecture [2].

1.5 Scope

The resultant kernel will run on top of a Qemu-emulated Riscv CPU. The kernel will not be able to run on other ISAs. The Kernel will not run atop physical ISAs without tweaking the code.

The resultant kernel will not have standard features such as networking capabilities or sound and graphics. The kernel will only be limited to the functionalities specified in the System Analysis Chapter.

CHAPTER 2: LITERATURE REVIEW

The literature review discusses the technologies used within the IOT infrastructure and how they fit into the project. The conclusion section of the literature review outlines the summary of the literature review and how it affected the design and implementation of the kernel.

2.1 The IOT development structure.

IoT development requires a range of software technologies, from mobile development to cloud and analytics. The common end-to-end architecture for IoT systems consists of edge devices, gateways, cloud, and applications as depicted below.

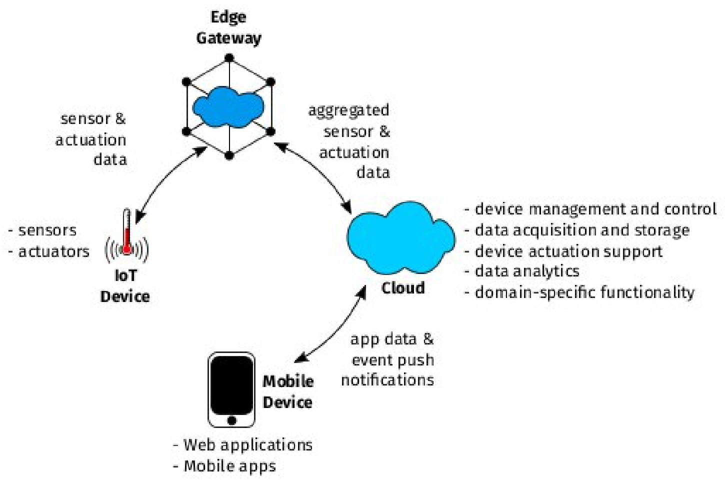

Edge Gateway
sensor & actuation data
aggregated sensor & actuation data
- sensors
- actuators
IoT Device
- device management and control
- data acquisition and storage
- device actuation support
- data analytics
- domain-specific functionality
Cloud
app data & event push notifications
Mobile Device
- Web applications
- Mobile apps

Figure 5: An Illustration of the different components in an IoT architecture.

IoT systems encompass embedded development, meaning that at the edge, the developer has to deal with a resource constrained environment. Embedded devices typically have low processing power, low memory and limited energy supply.

The Edge Gateway connects the edge devices to a local network or the internet and consequently a cloud platform. Gateways are intermediary devices located closer to the devices/things, often at the edge of the network. They collect, pre-process, and filter data from connected devices, reducing latency and network traffic by performing local data processing and analysis. Gateway devices may also provide additional functions like protocol translation and security features like monitoring.

Cloud services form the backend infrastructure that stores, processes, and analyzes the data collected from IoT devices. They enable centralized management, real-time monitoring, and remote control of connected edge devices.

The Web and Mobile Apps provide a way to interact with both the cloud and edge devices, they provide a form of user-interface to the backend. IoT app development is dominated by mobile and web development frameworks more than desktop frameworks like Tauri or Electron.

Under this architecture, the backend services are packaged as independent micro-services; this helps in making the backend software more modular. The micro-services get wrapped as containers. A popular container choice is Docker. To manage the containers, developers employ container orchestration software such as Kurbenetes.

The bigger dream of IOT development is to build a programmable world, ubiquitous technology. To achieve this vision, fog computing and edge computing gradually emerged. These two technologies have no clear definition, they both propose that computing tasks should be transferable across many nodes. The main point being that the cloud should not be the only place where core processes are executed. This proposition is achievable because the processing power, storage and energy usage found in embedded environment has improved; we now have advanced CPU chips and high-volume storage components in the embedded chips.

The challenge that arises is that the programming languages and tools used to write and execute the data processing applications found in the cloud might be not be supported by the runtime software found in the edge devices. As a result the software portability required by the fog computing proposal becomes hard to implement. As mentioned in the report introduction chapter, the solution to this is to:

a) defining common combinations of device-to-device communication standards
b) Using a common API standards
c) Using an isomorphic wrapper (container) to wrap the software modules.[2]

2.2 Web Assembly

Webassembly is an intermediate language for a virtual stack CPU. This intermediate language can be represented in assembly form or binary form. In other words, it is a language made for a virtual ISA (Instruction set architecture). And it so happens that the architecture is stack-based.

To better understand how wasm functions, one needs to understand the concept of abstraction. Abstraction is an important concept in Computer Science, if a problem has too many details or it has multiple implementation alternatives, just represent it in simple wrappers and let the dirty implementations stay under that wrapper . For example, in the past, there were so many Operating systems cropping up. Every kernel developer came up with unique ways of implementing system calls. To simplify things, developers agreed to create a common System call abstraction, POSIX. Under this abstraction, they defined standard system call function signatures. From there on, developers had to use those agreed upon signatures but underneath those prototypes, they had the freedom to implement the functions in their own way.

Webassembly employs the same abstraction technique. It abstracts an execution environment. Core webassembly abstracts a CPU. Combining Wasm and WASI abstracts the CPU and the runtime running ontop of that CPU. That runtime could be an OS or another application — it could be anything that has an exposed API.

Being that core webassembly abstracts CPUs, the wasm bytecode can be compiled to suit specific CPU ISAs. This makes wasm bytecode to be Architecture-independent.

Using the WASI API, web assembly abstracts any underlying system, be it an operating system or an ethereum system. As long as you use the WASI prototypes in your wasm bytecode, that code can be compiled for that specific implementation of the target system. This makes wasm platform-independent.

All high level programming languages need to get compiled to the CPU target architecture. Using the LLVM compiler, a developer can compile any high level language to target the wasm virtual machine. This will in turn generate a .wat or .wasm file. A .wat file stands for (webAssembly Text). This .wat file contains human readable bytecode. The .wasm file contains machine code meant for the wasm virtual CPU.

The webassembly file is known to be compact and memory-efficient. This is greatly attributed to the fact that the bytecode targets a stack-based system instead of a register-based system thereby reducing the number of arguments required for each assembly instruction.

2.3 WebAssembly Runtimes

A runtime may mean a lot of things. In the context of this section, the word runtime means a piece of software that provides and manages resources needed by another running process. For example, if a person plays a video game on a windows operating system, the windows operating system can be referred to as the runtime for the video game.

A web-assembly runtime is a piece of software that provides and manages resources needed a wasm program. The web-assembly runtime interfaces with the underlying host system. For example, the run-time might interface with a kernel or a bare metal hardware interface.

A wasm runtime acts as the intermediary between the host system and the wasm modules. It performs the following functions :

Module loading and validation: The runtime locates the location of the wasm module, reads it and stores the content in a read-only buffer. It then validates the WebAssembly module, ensuring that it adheres to the WebAssembly specifications and security requirements. It verifies the module's structure, type signatures, and validates its bytecode.

Module compilation : For the webassembly modules to get executed by the underlying CPU, it needs to be compiled to native machine code. The Wasm Runtime contains a compiler specifically for this job. Note that the compiler might be of any kind depending on the runtime’s specific implementation. For example, the compiler might be an Ahead-Of-Time compiler, a Just-In-Time compiler or even a combination of both.

Import and export handling: WebAssembly modules can import functions and data from the host environment, such as JavaScript. The runtime facilitates the interaction between the WebAssembly module and the host environment by handling imports and exports. It resolves the dependencies and connects the module's imports to the corresponding functions and data in the host environment.

Memory management: The runtime handles the memory management for the WebAssembly module.By interacting with the underlying operating system, It initiates commands that allocate and manage linear memory, which is a contiguous block of memory accessible to the module. The runtime provides functions for allocating, resizing, and manipulating this memory.

Execution: By interacting with the underlying OS, the runtime executes the WebAssembly module's bytecode instructions. It provides an execution environment where the module can run, interpreting or compiling the bytecode to machine code for efficient execution. The runtime manages the execution stack and handles control flow, including function calls, loops, and branches.

Capability-Based Security : By default, wasm modules are sandboxed, they cannot invoke any system calls or host functions. To do so, they reference the WASI functions in their bytecode OR they explicitly import host functions. Each function call made by the wasm module gets inspected and validated by the wasm runtime. The Runtime checks if the function affects only the accessible resources, For example, if the wasm module is given access to file “x” only, the runtime will flag an error when the module tries to access any other file apart from file “x”.

Garbage collection: Some Wasm runtimes incorporate garbage collection mechanisms to automatically reclaim memory that is no longer in use by the WebAssembly module. Garbage collection helps manage memory resources efficiently and prevents memory leaks.

Interoperability: Wasm runtimes often provide interfaces or APIs that allow the WebAssembly module to interact with the host environment. This enables communication between the WebAssembly module and the surrounding application or system. For example, it may provide functions to access the file system, network, or other platform-specific capabilities.

Performance optimization: The runtime may employ various techniques to optimize the execution of WebAssembly modules. This can include just-in-time (JIT) compilation, where the bytecode is dynamically compiled to machine code for improved performance. Runtimes can also utilize ahead-of-time (AOT) compilation to generate optimized machine code before execution.

Some of the runtime examples include : wasmtime, wasmer and wasmi. There are tens of runtimes listed on github [5].

2.4 Security Capabilities of Wasm and Wasm Runtimes

WebAssembly (Wasm) and Wasm runtimes provide several security capabilities to ensure the safe execution of code. Here are some of the key security features:

Sandbox Execution: Wasm code runs within a sandboxed environment, which isolates it from the host system and other code. This sandboxing prevents malicious code from accessing sensitive resources or interfering with the underlying system. The mechanism for implementig this sandbox execution is entirely up to the wasm runtime.

Memory Safety: Wasm enforces memory safety by using a linear memory model with bounds checking. This prevents buffer overflows and other memory-related vulnerabilities that can lead to security exploits. Wasm runtimes ensure that memory accesses stay within the defined bounds, preventing unauthorized access to data. A wasm module cannot directly access memory that is not withing the allocated linear memory.

Validation and Verification: Before execution, Wasm modules are validated and verified by the runtime. This process ensures that the module adheres to the Wasm specifications, has correct type signatures, and is free from structural errors. Invalid or malicious modules are rejected, preventing potential security risks.

Sandboxed APIs: Wasm runtimes provide a set of sandboxed APIs that allow controlled interaction with the host environment. These APIs provide limited access to system resources, such as file I/O, networking, or graphics, while preventing direct access to sensitive operations or resources. This helps mitigate potential security threats by enforcing access restrictions. This sandboxed APIs ensure that a wasm module only gets to use certain functions only. For example, you can refuse to provide the networking API to a calculator app. This forms the basis of capability-based security.

Controlled Imports and Exports: Wasm modules can import and export functions, but the runtime controls these interactions. Import functions are explicitly linked to trusted and safe host functions, preventing unauthorized access or execution of arbitrary code. Exported functions are also subject to runtime enforcement, ensuring that they are used appropriately and securely by the host environment.

2.5 Porting of webAssembly Runtimes to No-std environments

Majority of the wasm runtimes have been built to run on the browser. But there have been many upcoming runtimes that have been built to run off-browser. For example wasmtime, wamR, Wasmi and Wagi. A good comparison of the different runtimes has been done by Appcypher[5]

Some of them have been implemented as standalone command-line programs while some have been implemented as language-embeddable libraries.

Wasmtime was a compelling option being that it supports AOT compilation. Wasm3 is embeddable in a no-std environment but it is written in C. It has an Interpreter instead of an AOT compiler. Having an AOT compiler is crucial. An interpreter or a JIT compiler introduces unpredictability in performance and resource inefficiency; they waste energy, time and memory.

Wasmi proved to be the best candidate. It was the only embeddable runtime that easily ran in a no-std environment. It can easily be imported as a rust-library crate. Porting wasmtime proved to be too hard a hill-climb. Wasmtime is tightly coupled to the operating systems that it runs on. The downside of using wasmi is that wasmi uses an interpreter instead of an AOT compiler.

2.6 WASI and POSIX

WebAssembly System Interface (WASI) is a system interface that has been developed to enable WebAssembly (Wasm) applications to run securely and efficiently across different platforms and operating systems. WebAssembly itself is a binary instruction format designed for safe and efficient execution on web browsers and other environments. WASI extends the capabilities of WebAssembly beyond the web, allowing it to be used in a wide range of contexts, including server-side applications, edge computing, and more. [6][7][8].

For example, wasi-nn is an extension-module to WASI that provides neural network inference capabilities. It enables WebAssembly applications to perform machine learning inference using standardized APIs, making it easier to deploy and execute machine learning models.

One of the primary goals of WASI is to provide a consistent runtime environment for WebAssembly applications across different platforms, architectures, and operating systems. This means that developers can compile their code once and have confidence that it will run consistently across various execution environments.

POSIX on the other hand is a System Interface description purposefully built to abstract Operating system functionalities. POSIX defines a standard set of interfaces, functions, and behaviors that operating systems should provide if they want to be POSIX-compliant. The goal is to ensure application portability across different Unix-like operating systems.

Advantages of POSIX in IoT:

1. Compatibility: Many existing IoT devices and systems might already have POSIX support, making it easier to develop and port applications to these devices.
2. Familiarity: Developers who are experienced with POSIX programming might find it more comfortable and familiar to work with this standard, as opposed to learning and adopting new technology like WASI.
3. Legacy Systems: IoT deployments often involve older devices that lack the resources or compatibility for new technologies. POSIX support could cater to such legacy systems.

Advantages of WASI in IoT:

1. Security Isolation: One of the key benefits of WASI is its strong security isolation. It restricts applications from accessing the underlying system in ways that could pose security risks. This is crucial for IoT devices, where security vulnerabilities can have serious consequences.
2. Portability: While POSIX provides compatibility across various Unix-like systems, it might not cover the entire IoT landscape, which includes diverse devices with different operating systems or even bare-metal environments. WASI containers offer a more consistent runtime across a wider range of systems.

   If an IoT device uses a specialized real-time operating system (RTOS) that does not fully adhere to POSIX standards. Porting a POSIX-based application to this device might require significant modifications due to differences in APIs and behaviors.

   While POSIX provides portability within the scope of Unix-like systems, WASI extends this portability to a wider range of environments.

   In summary, WASI is a super-set of POSIX. WASI covers more domains, it goes beyond abstracting an operating system. WASI uses capability-based security by default while POSIX leans towards user-based security.

2.7 Kernel Designs

The kernel designs vary, there are no definite designs. For example, one might say that kernel A is a micro-kernel because the kernel services run as user processes, while another person might say that the same kernel is a monolith because its drivers are implemented as part of the core kernel. Naming designs is subjective.

Tock

The Tock Operating System is an open-source, event-driven operating system designed primarily for low-power, embedded systems and Internet of Things (IoT) devices. It is specifically engineered to address the challenges posed by constrained hardware resources, real-time requirements, and security considerations. [9][10]

Here are some key design aspects of the Tock Operating System:

Event-Driven Architecture: Tock uses an event-driven architecture, where applications and kernel components respond to asynchronous events, such as sensor readings or timer expirations. This approach allows for efficient use of resources, as components are activated only when necessary

Multithreading and Isolation: Tock supports multithreading, allowing multiple tasks to run concurrently. Each task has its own stack and memory space, which helps prevent one task from interfering with another. This isolation improves system reliability and security.

Small Kernel Size: Tock's kernel is designed to be compact, which is important for resource-constrained devices. This small kernel size minimizes memory overhead and leaves more resources available for applications.

Applications are loadable, one need not compile the application together with the kernel. Tock separates the Apps from the kernel using the Memory Protection Unit(MPU); to this regard, Tock assumes that the host system has an MPU. But if the underlying host architecture does not have an MPU, one can configure the kernel to use a virtual MPU.

Tock architectural Design

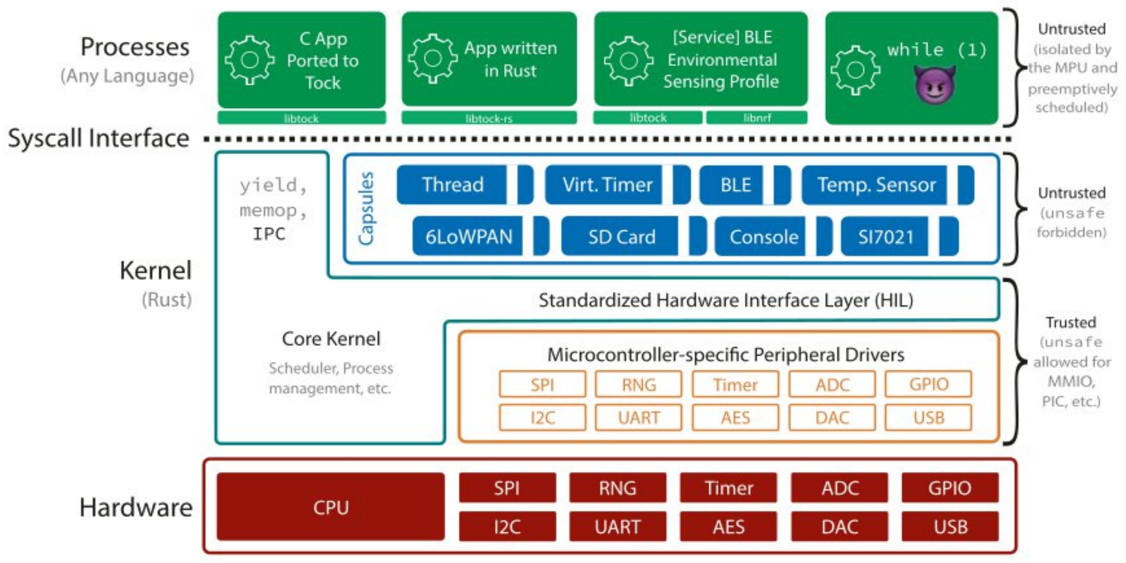


Figure 3-1. *The Tock stack (source: https://github.com/tock/tock)*

Figure 6: The Tock Kernel Architecture

In Tock, the drivers (both low-level and high-level drivers) are not part of the Kernel. The Apps and Services are also not part of the kernel. The Apps are programs that users can directly interact with.

The services are programs that continuously run in the background and are meant to be summoned by other software. They are not typically used by users. They can be things like a server.

The low-level drivers are pieces of software that come pre-witten inside the hardware. They can directly interact with the hardware. These drivers expose a Hardware Interface Layer to the Kernel.

These low level drivers are board-specific and device-specific. To abstract the low-level drivers, the kernel takes the low level and specific Interfaces and exposes them to the Higher level drivers using a kernel_defined_standard Interface.

The capsules (ie. High_level_Agnostic_drivers), provide APIs that can be imported to be part of the SYSCALL interface.

The Tock kernel is considered memory safe because the compiler does all the memory checks at compile time. The compiler does not check memory safety within unsafe rust blocks. So the tock kernel developers were compelled to reduce the number of unsafe blocks in any kernel Rust code.

The problem becomes that all drivers will end up having unsafe code. This is because registers are accessed using raw pointers. Tock solves this problem by defining a register interface below the low_level_drivers.

This register interface provides consistent safe functions for accessing memory. These safe functions are unsafe functions that have been wrapped with safe code. This ensures that all drivers do not get to have the necessity to declare any unsafe block. They only have to call the memory_interface safe functions. Tock does not allow any unsafe code in the Capsules.

Functions of the Tock Kernel :

1. Schedule the Apps and services. Who gets to use the CPU?

2. Memory Management : Allocate/deallocate/track memory needed by the Apps and Services

3. Provides mechanisms for Inter-process communication

4. Providing an interface for the Apps and Services

5. Providing an interface to drivers (High_level_drivers AND low_level_drivers)

Tock System Call implementation

Tock uses both synchronous and asynchronous system calls ; synchronous calls for simple tasks that take negligible time and asynchronous calls for operations that might take some time. For Example getting the number of GPIO pins (General Purpose I/O pins) is done synchronously while reading and writing to a peripheral device is done asynchronously.

How Tock Handles The Problem of Dynamic Memory Allocation

Dynamic memory allocation, often done using functions like malloc() and free(), allows programs to request and release memory from the system's heap during runtime. While dynamic memory allocation is powerful, it can introduce several challenges and complexities.

For example, it may cause fragmentation of the physical memory. Paging does not solve this issue, it just eases it. Fragmentation can lead to performance latency when it’s time to defragment the fragmented physical memory.

Moreover, dynamic heap allocation may make performance to be unpredictable. In real-time systems or systems with strict timing constraints (common in embedded systems), dynamic memory allocation can introduce unpredictable delays due to the variable time it takes to allocate and de-allocate memory.

The Tock Embedded Operating System, designed for resource-constrained embedded systems, takes a different approach to memory management to address these challenges:

- **No Dynamic Allocation by Default**: Tock avoids dynamic memory allocation by default. It uses a "stack-only" memory model for applications, which means that memory for data structures, buffers, and variables is allocated statically at compile time. This eliminates runtime memory allocation and the associated problems.
- **Predictability:** By using a fixed memory model, Tock ensures predictable and deterministic behavior, making it suitable for real-time and safety-critical applications.

2.8 Docker versus Wasm

Docker and Webassembly (Wasm) containers are two distinct technologies that share the common purpose of packaging and deploying applications, yet each is tailored to address specific needs and use cases. Docker containers have gained immense popularity for their ability to encapsulate applications, along with their dependencies and runtime environment, in a portable package. Operating at the OS level, Docker offers process-level isolation through virtualization, enabling applications to run within separate environments. While Docker containers offer good isolation, security concerns can arise from vulnerabilities in the host OS or the Docker runtime.

On the other hand, Webassembly (Wasm) containers introduce a new paradigm of lightweight and secure application deployment. These containers are designed for exceptional portability, able to execute applications across diverse environments such as web browsers, edge devices, and cloud servers. The secret to Wasm's portability lies in its standardized runtime environment, achieved through the Webassembly System Interface (WASI). Unlike Docker, Wasm containers prioritize strong security through robust isolation. Wasm modules operate in a sandboxed environment, preventing direct access to host resources and enhancing overall security.

Resource efficiency is another domain where Docker and Wasm containers differentiate themselves. Docker containers carry the overhead of a full OS user space, often resulting in larger container sizes and higher resource consumption. Conversely, Wasm containers embrace a minimalist approach. By packaging only the essential code and dependencies, they ensure lightweight deployment and optimize resource utilization. The encapsulation of Docker's OS user space within its containers offers versatility across programming languages, whereas Wasm's design accommodates multiple languages through compiler toolchains, allowing developers to choose their preferred language without excessive overhead.

However, the advantages that wasm containers have over docker containers get quickly overshadowed by the fact that Docker has a more established ecosystem. There are comparably more tools that seamlessly integrate with Docker. For example, **Kurbenetes**.

In the realm of Internet of Things (IoT), Kubernetes has emerged as a prominent platform for container orchestration, streamlining the deployment of containerized applications across diverse IoT devices. Orchestrating wasm containers using Kurbenetes is currently being developed.

This is where projects like Krustlet come into play. Krustlet, an initiative within the Kubernetes ecosystem, introduces the prospect of seamlessly integrating Wasm workloads into Kubernetes clusters. By leveraging the capabilities of Krustlet, developers can efficiently deploy Wasm modules alongside traditional containerized applications. This innovation holds the promise of extending Kubernetes' reach into the embedded space, enabling a unified orchestration platform that caters to the specific demands of IoT devices while capitalizing on the benefits of Wasm containers' security and efficiency.

2.9 Conclusions from the Literature Review

A monolithic kernel architecture was chosen over the micro-kernel architecture. The decision to implement the project using a monolithic architecture was driven by several factors that aligned with the project's goals and requirements. Firstly, the project's scope and complexity were relatively modest, with a limited set of functionalities that needed to interact closely and frequently. A monolithic architecture allowed for seamless communication between different components, facilitating rapid development and debugging. Additionally, the project’s short timeline favoured a simplistic and straightfoward design; a monolithic approach.

Considering that the applications run in isolated wasm environments within the wasm runtime, it would be an overkill to further isolate them to the user address space (micro kernel architecture). Furthermore, the concept of the project is to run user processes in kernel mode.

Wasm containers were chosen over Docker containers. Wasm containers seem more promising than Docker containers in terms of resource efficiency and portability. The wasm containers are smaller and the wasm runtimes themselves have small memory footprints.

WASI-compliance was chosen over POSIX-compliance. WASI-compliance was found to be more favourable based on three reasons. Firstly, sticking to WASI over POSIX will reduce the latency caused by the function-normalization done by the wasm-runtime. Additionally, using WASI makes the kernel to be extensible to domains that were not initially meant for operating systems, this is because POSIX only covers OS-related standards while WASI covers what POSIX covers plus more. WASI is a superset of POSIX. Lastly, WASI's capability-based security model provided better isolation between applications, reducing the potential attack surface. Implementing capability-based regulation in POSIX is not as straightforward as in a WASI execution environment.

Using both synchronous and asynchronous system calls for different use-cases was found favorable. Incorporating both synchronous and asynchronous system calls into the project's architecture was a deliberate strategy aimed at optimizing task execution. By leveraging synchronous calls for lightweight and rapid operations, the project ensured minimal overhead and swift processing of simple tasks that demanded negligible time. On the other hand, the inclusion of asynchronous calls was a strategic move to address more time-intensive operations. These asynchronous calls allowed the system to efficiently handle tasks that might require extended processing time without causing bottlenecks or delays in other parts of the application. This duality in system calls not only optimized resource utilization but also provided a balanced approach to task execution, enhancing the overall responsiveness and performance of the project’s architecture. Just like the Tock system, I/O system calls and other heavy operations will be done asynchronously while quick operations will get done synchronously.

The emergence of the Kruslet project made the idea of the orchestration of wasm containers become feasible. Kurbenetes support for wasm containers further encourages the notion that maybe in the future, wasm containers will become the norm.

The problem of dynamic memory allocation may have an impact in real-life critical embedded systems but considering that this is a learning project, a system that implements the Tock heap allocator design would be an overkill and hard to implement. The Heap estimations would also be hard to gauge.

Wasmi runtime was chosen over other wasm runtimes. The project will use the Wasmi runtime because it supports no-std capabilities. Wasmi is easy to port to a bare metal environment. Other runtimes like wasmtime are more developed than wasmi, but porting them to a no-std environment is an uphill task.

CHAPTER 3: METHODOLOGY

This section describes the process used in actualizing the project

Being that the project was an uncharted territory for the implementor, there was no clear plan at the very beginning. The plan only became clearer as the research and half-implementations were undertaken.

The process was divided into six phases:

a) Knowledge and skill building
b) Research
c) System Analysis
d) System Design
e) Implementation
f) Support Phase (Testing and Documentation)

3.1 Knowledge and Skill building

Under this phase, the developer was required to get familiar with the business domain and the technologies involved. The business domain being **the provision of development tools** for the IOT development.

Knowledge and skill building boiled down to reading the necessary research papers associated with the project, familiarizing with the technologies involved, implementing any modules within reach and repeating the loop.

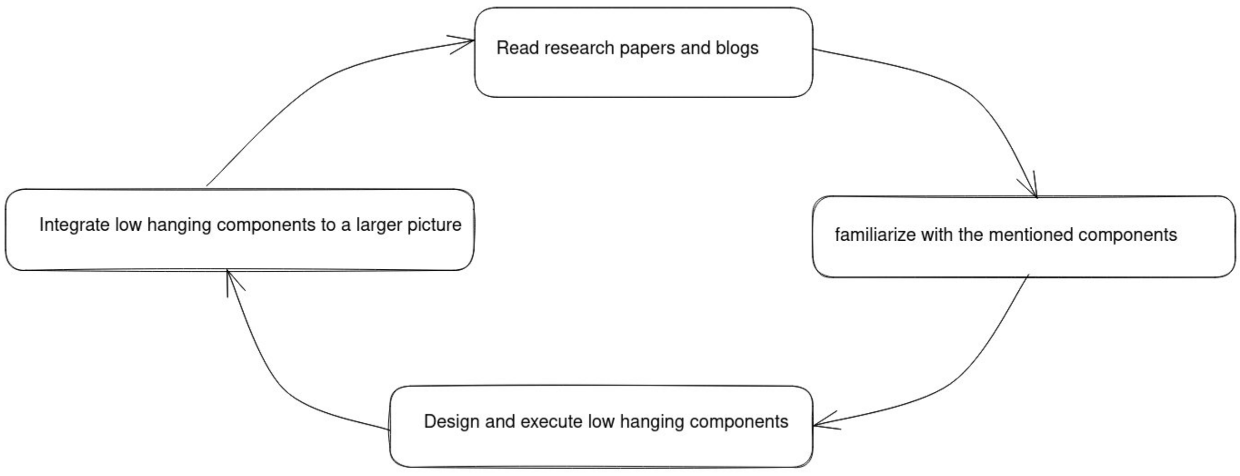

Read research papers and blogs
familiarize with the mentioned components
Design and execute low hanging components
Integrate low hanging components to a larger picture

Figure 7: Methodology used in Knowledge and Skill Building.

3.2 Research Methodology

The research methodology involved looking for scholarly documents concerning the involved subjects and understanding the applicable components. This mainly touched on the web assembly and IOT infrastructure topics. The key papers have been referenced in the appendix section of the report.

As for the Rust and Riscv knowledge, the official documentation and online tutorials were sufficient. The books used have also been attached in the Appendix.

The Kernel design and implementation knowledge was extracted from three blog tutorials and referenced text books in the appendix.

ChatGPT was also used as a guide during research, It helped save days of googling or reading textbooks and datasheets with hope of finding a specific line of information. Sometimes ChatGPT was inaccurate but reading official documentations about datasheets or softwares clarified issues quickly.

To help retain the knowledge gained, the Feynman technique was applied, the implementer wrote a tutorial [11] as they researched and implemented the project.

It is worth noting that the research methodology adopted herein did not involve the administration of questionnaires or the conduct of interviews. The focus of this study rested upon a comprehensive examination of existing resources, drawing from established academic works, authoritative documentation, and the synthesis of contemporary technologies.

3.3 System Analysis

System analysis involved understanding the business requirements and determining which features and functionalities were needed to satisfy those business requirements in a feasible manner. Feasible in terms of time, knowledge, money and human energy.

The first step was analyzing which functional requirements were required by a generic Embedded Operating system. This was followed by arranging the identified requirements by their priority order. The priority level of each requirement was based on whether the requirement was needed to show a proof of concept of the project. For example, implementing the standard input and output took more priority than implementing a File system. This is because the proof that wasm functions worked needed to be displayed, storing the wasm programs in a File system could be important but it does not directly affect the concept of running wasm programs at kernel-level.

Technical feasibility, time feasibility and labor-feasibility was determined continuously during research and implementation.

3.4 System Design

System design was done continuously as the research and implementation happened. Designing each kernel involved the following steps: drawing component diagrams, narrowing down to the data-flow diagrams and finally outlining the pseudo-code and expected API notations.

With time, the designed components were integrated with each other accordingly. The design process mixed up both a top-down approach and a bottom-up approach.

3.5 System Implementation

Implementation took a bottom-up approach. The kernel modules were implemented starting from the modules that were least-independent on other modules to the modules that were most-dependent on multiple modules.

3.6 Support Phase

To support the project and make it maintainable, continuous documentation and testing was necessary.

Documentation took two forms : Report Documentation and Tutorial Documentation. Tutorial documentation outlines how to recreate the project, it describes the actual implementation steps taken. Report Documentation was done last. It summarizes the the whole process.

Testing was done in two categories : Unit Tests and Integration tests. The unit tests focused on testing the functionalities of isolated modules while the integration tests focused on how the different modules interacted with each other.

CHAPTER 4: SYSTEM ANALYSIS

The main business requirement was to deliver a minimal embedded kernel that exported WASI system calls. Below are the derived functional requirements.

4.1 Functional Requirements

Under this segment, the minimal kernel stands for a kernel that has not integrated the wasm runtime in its code. The term extended kernel describes the result of integrating the minimal kernel with the wasm runtime.

1. The minimal kernel should provide a terminal user interface as its standard output and the keyboard as the standard input.
2. The minimal kernel should have a page grained memory allocation system
3. The minimal kernel should have a byte-grained memory allocation for both kernel and user space heaps
4. The minimal kernel should have a memory management unit that runs in the Riscv Supervisor mode
5. The minimal kernel should be able to handle both external and internal interrupts with the help of the CLINT and PLIC
6. The minimal kernel should support user processes
7. The minimal kernel should export both system calls falling under FileSystem access, memory allocation and process handling.
8. The minimal kernel should provide persistent storage by providing a filesystem that runs atop a hard-disk.
9. The extended kernel should export its system calls using the WASI syscall prototypes
10. The extended kernel should be able to load wasm containers and execute them as distinct processes.
11. The extended kernel should provide a simple capability based interface for inspecting the capabilities exposed to a user process.

4.2 Feasibility Study

As mentioned in the Methodology Chapter, feasibility analysis was done continuously in tandem with research and implementation.

4.2.1 Technical Feasibility

The technical feats required to satisfy the functional requirements were divided into two; the feat to build a minimal kernel and the feat to build an extended kernel. Being that the implementor had no prior experience in kernel development, it was hard to port an existing kernel like xv6. A more practical approach was to build a minimal kernel from scratch in order to understand how to extend kernels using WASI.

During research, it was realized that WASI was an entire branch of knowledge on its own. Kernel development knowledge was already a sizable branch too. It was thus evaluated that implementing a kernel that exported WASI calls from scratch was unfeasible. With the set project deadlines, it was better to concentrate on building a minimal kernel only.

4.2.2 Time Feasibility

The project was estimated to take around 8 months in order to be complete. It was determined to be time-feasible. The project was divided into two phases : Building the minimal kernel and Extending the minimal kernel with WASI calls. It was estimated that building the minimal kernel would have taken six months and extending it would have taken one month. The eighth month was to be dedicated to testing, documentation and optimizing the implemented modules

However, this time analysis was poorly done. Implementing the minimal kernel ended up consuming the entire 8 months.

CHAPTER 5: SYSTEM DESIGN

5.1 Introduction

System design involved sketching out the possible high-level implementations. It ranged from drawing block diagrams, dataflow diagrams and pseudo-code flowcharts.

The tools used were mermaid.js, excalidraw and mdbook.

As mentioned in the Methodology section, system design was done using a mixture of top-down and bottom-up approaches. It was done continuously and in tandem with research and implementation phases.

The problem that arose from the chosen method was that the modules grew more and more dependent on each other as the project progressed. This forced the designer to keep on redrawing the designs with the new changes.

For example, the UART design might seem independent at first, but it you reach the section of trap handling, you realize that the PLIC and the UART are quite intertwined. Their memories are also getting handled by the MMU, a component that you only realize is needed later on.

The above problem introduced the problem of re-drawing components. Today the designer may take an hour designing how the UART works. Two weeks later the designer trashes the previous image because it is incompatible with the PLIC. To solve this problem, the designer resorted to using mermaid.js, a tool that uses code to draw designs. Excalidraw made it easier to modify the images.

5.2 Solution Chosen

The solution to the human problem in chapter 1 is to build an embedded operating system that runs wasm containers at near-native speed.

The solution to the mentioned technical problem is to eliminate or reduce the intensity of the two latencies responsible for impeding the execution speed of wasm containers. The methods of reducing the two latencies has been discussed below.

Handling the System-call-normalization latency

The system-call-normalization latency can be completely eliminated by making the kernel export its native system calls using the prototypes used in the WASI API [3], this means that the webassembly runtime will not need to normalize functions. Eliminating the need to normalize functions completely rids the latency caused by the normalization process. No more time is spent on normalization.

Handling the Context-switching Latency

It is possible to completely eliminate the context switching expense by making the wasm modules part of the kernel code. Such that the CPU will only execute one process throughout its up-time. But this solution requires the developer to compile the wasm modules together with the kernel code. This means that whenever the wasm app gets updated, the developer has to recompile the entire kernel. This is highly unmaintainable. This solution is suitable when performance is more important than maintainability. It is especially unsuitable in the IoT space where it is ideal to update device programs remotely (via the network).

Completely eliminating context switching comes at the cost of maintainability, for this reason, it becomes more viable to focus on reducing the cost of context switching instead of completely eliminating it.

Context switching from a user-level process to a kernel-level process is more expensive than context switching from a kernel-level process to another kernel-level process. This is because switching from user-level process to a kernel-level process requires more overhead: The CPU has to switch modes, re-map the Memory Management page tables and switch the execution stack. On the other hand, switching from one kernel-level process to another kernel-level does not require the CPU to switch modes and re-map the page tables. Switching from one kernel-level process to another kernel-level process is cheaper.

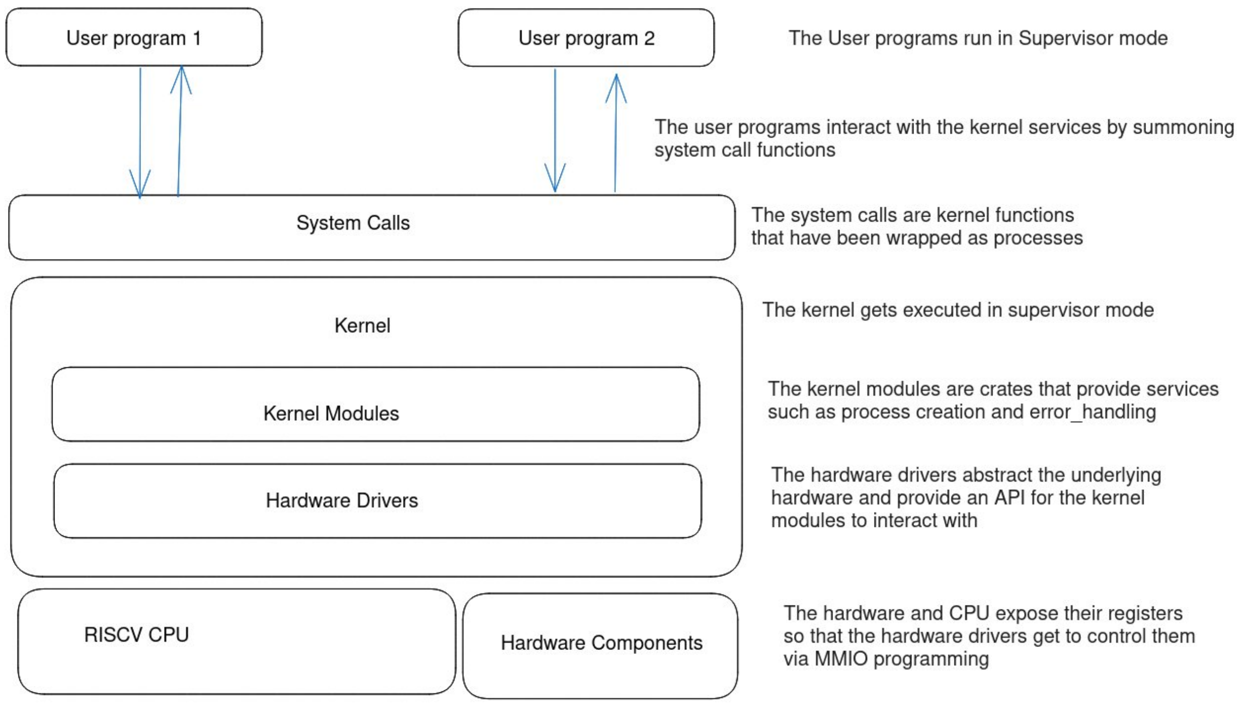
Chosen Solution

Based on the above discussion on the different methods of solving the latencies, the project went with building a kernel that:

a) Has the wasm runtime embedded in it, as part of the kernel code
b) Uses an Ahead-of-time compiler to compile the wasm modules
c) Exports its system calls using WASI function prototypes
d) Runs the loaded wasm containers in kernel mode.

5.3 The Mini-Kernel

The mini-kernel is the kernel without the WASI extension. Below is the overview of the components that interact with the Kernel.



User program 1
User program 2
The User programs run in Supervisor mode
The user programs interact with the kernel services by summoning system call functions
System Calls
The system calls are kernel functions that have been wrapped as processes
Kernel
The kernel gets executed in supervisor mode
Kernel Modules
The kernel modules are crates that provide services such as process creation and error_handling
Hardware Drivers
The hardware drivers abstract the underlying hardware and provide an API for the kernel modules to interact with
RISCV CPU
Hardware Components
The hardware and CPU expose their registers so that the hardware drivers get to control them via MMIO programming

Figure 9: High-level architecture of the Kernel, without the embedded Wasm Runtime

The mini-kernel runs simple user programs in supervisor mode.

The user programs get stored in a secondary memory, preferably a hard-disk. The user programs then get loaded onto the Random Access memory together with the kernel image. Storing the code in the RAM ensures that the CPU gets direct access to the code. It makes instruction and data fetches to be less expensive in comparison to fetching them from the hard-disk.

In the context of this project, a notable architectural choice involves both the Kernel code and user programs operating within supervisor mode. Traditionally, executing user programs in supervisor mode is perceived as a security risk, given the potential for these programs to gain access to privileged CPU instructions and registers, compromising system integrity. However, the project addresses this concern by implementing a controlled sandbox environment for executing user programs within supervisor mode. This sandbox enforces limitations on the available instruction set, effectively mitigating potential hazards associated with uncontrolled access to privileged operations.

One key aspect bolstering the security of the system lies in the exclusive utilization of WebAssembly (Wasm) containers. These containers are confined to utilizing only the host functions that have been explicitly exposed by the host's glue code. This encapsulation ensures that the interaction between the wasm programs and the host environment is tightly controlled and well-defined. By adhering to a predetermined set of secure functions and resources, the project guarantees that wasm programs access only components vetted as safe and sanctioned.

Fine-Grained Access Control

Further enhancing security, the wasm code's interactions with the kernel are strictly regulated. The wasm code can solely invoke functions that have been made accessible through the kernel's system call API. This restricted access layer serves as an initial line of defense against unauthorized system interactions. Moreover, an additional layer of security granularity can be imposed, wherein specific wasm modules are limited to interacting with only a subset of the open API. This feature enhances the system's ability to customize access permissions and minimizes potential vulnerabilities.

Dynamic Monitoring and Inspection

To bolster the oversight of wasm modules' behavior, the wasm runtime plays a crucial role in dynamically assessing the system calls invoked by loaded wasm modules. This real-time scanning acts as an inspector, enabling detection of any deviation from the specified subset of system calls. Such monitoring reinforces the project's security framework by promptly identifying instances where wasm modules attempt to use system calls outside their predefined and secure subset, thereby preventing unauthorized or potentially malicious activities.

In summary, by executing both the Kernel code and user programs within a controlled supervisor mode sandbox, and by restricting wasm programs to a well-defined set of host functions and system calls, this project establishes a robust security foundation. The dynamic monitoring and inspection capabilities offered by the wasm runtime further fortify the system's security posture, ensuring that the execution environment remains impervious to unauthorized access and potential threats.

From Figure 6, it can be seen that the kernel comprises of kernel modules and device drivers. The drivers abstract the underlying hardware and provide an ALI for the kernel modules to interact with.

5.3.1 Kernel Modules

The kernel modules include the following:

5.3.1.1: The Boot-loader glue code

A typical bootloader does the following operations:

1. Finds the memory address of the Kernel program by searching through the external memory devices that have been plugged into the Motherboard.
2. Loads the Kernel image onto the RAM. Note that it only loads the required sections, it might not load the entire image.
3. Prepare the values found in the CPU registers to suit the execution of the kernel. For example making the Stack pointer to point to the stack of the kernel.
4. Transferring control to the kernel. This is done by making the program counter point to the entry point of the kernel

The Bootloader in this project is much simpler. It does not have to look for the kernel image in a secondary memory like a hard-disk, instead, the Kernel and the bootloader are one program.

In the case of this project, the boot program is the one that is responsible for:

1. Setting up the environment for the kernel to run in machine mode
2. Transferring the control to kinit() (kernel running in machine mode)
3. Designating a specific place where the CPU will jump to if a trap occurred when kernel is in either machine mode or supervisor mode.
4. Initiating an environment for the kernel to run in supervisor mode.

The boot code is the one that contains the _start function. The _start function is the entry point for the whole kernel image. So it is the first place the CPU points to in the RAM after the Qemu firmware has loaded the entire kernel image onto the RAM.

Here is a Bird's view of the general boot process:

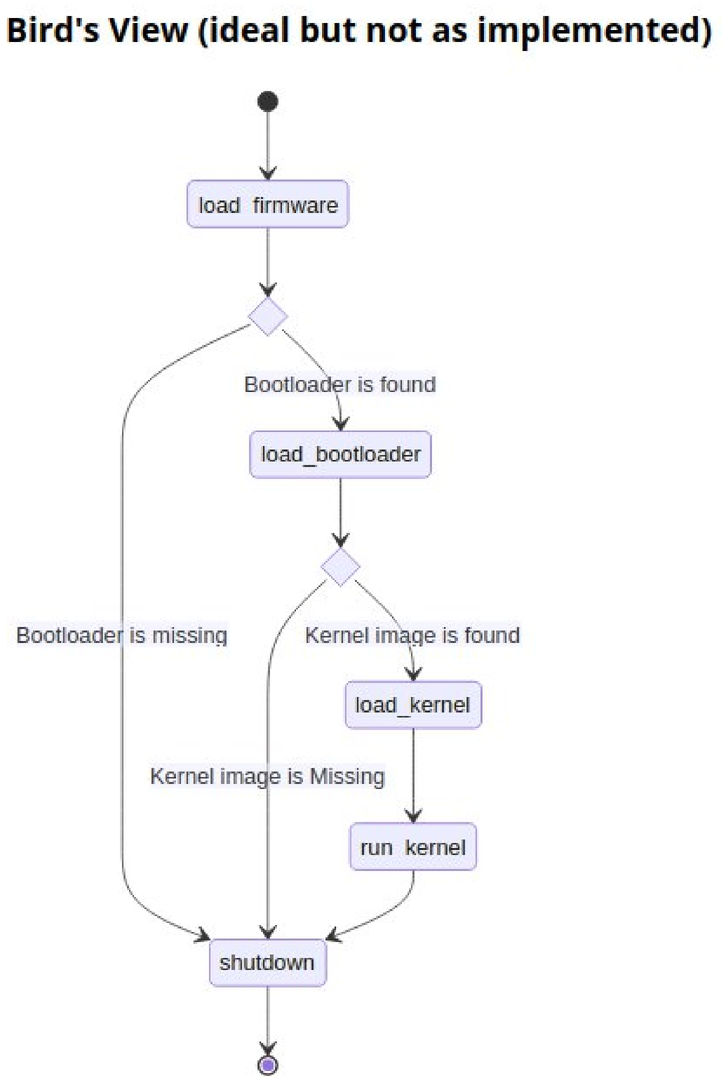
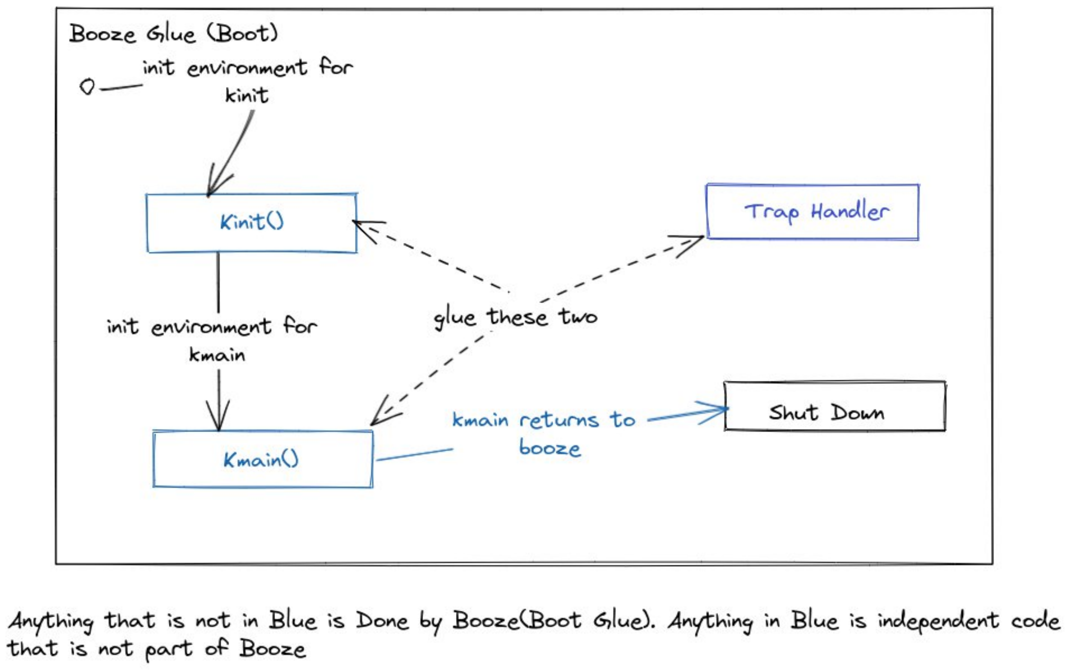


Figure 10: An Illustration of the generic Boot process.

The motherboard's firmware looks for the bootloader code in all the attached secondary memory devices. It checks whether a hard-disk is flagged as 'Bootable'. If no boot loading code is found, the machine throws an exception and shuts down. If a bootloader is found, it gets loaded onto the RAM and its execution gets started.

The bootloader prepares the CPU's context in preparation to accommodate the execution of the oncoming kernel Image. The bootloader then looks for the kernel image and loads it onto the RAM. The CPU then begins executing the kernel image starting from its entry point.

Instead of the boot code being a program that gets called once(as seen above), it becomes the glue code that gets called whenever kernel goes out of scope. It encompasses the entire kernel. It acts as glue code as seen below:



Anything that is not in Blue is Done by Booze(Boot Glue). Anything in Blue is independent code that is not part of Booze

Figure 11: The Bootloader implemented as a glue that links the different parts of the kernel image

Below is a clearer demonstration how the boot code has been implemented:

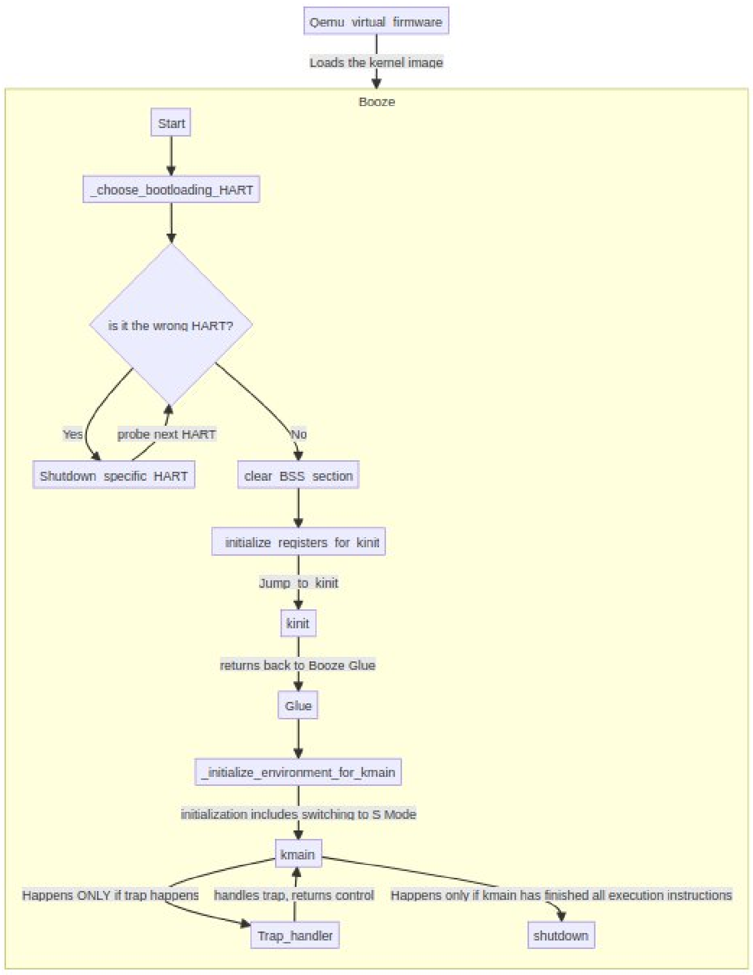

Qemu virtual firmware
Loads the kernel image
Booze
Start
_choose_bootloading_HART
is it the wrong HART?
Yes
probe next HART
No
Shutdown specific HART
clear BSS section
initialize registers for kinit
Jump to kinit
kinit
returns back to Booze Glue
Glue
_initialize_environment_for_kmain
initialization includes switching to S Mode
kmain
Happens ONLY if trap happens
handles trap, returns control
Happens only if kmain has finished all execution instructions
Trap_handler
shutdown

Figure 12: The Program Flow of the Bootloader

5.3.1.2 : The Memory Initializer

The memory initializer is responsible for demarcating all memory sections. It forms abstractions that get used by the Page Manager and the Byte Manager. The Memory initializer is responsible for demarcating the RAM into Descriptors and Pages. It is also responsible for segregating the different memory meant for different purposes. For example, it demarcates the memory section used by the kernel code and the memory sections used by the user applications

5.3.1.3: The Page Manager

The page manager allocates and deallocates pages from the heap as requested by user applications and the kernel.

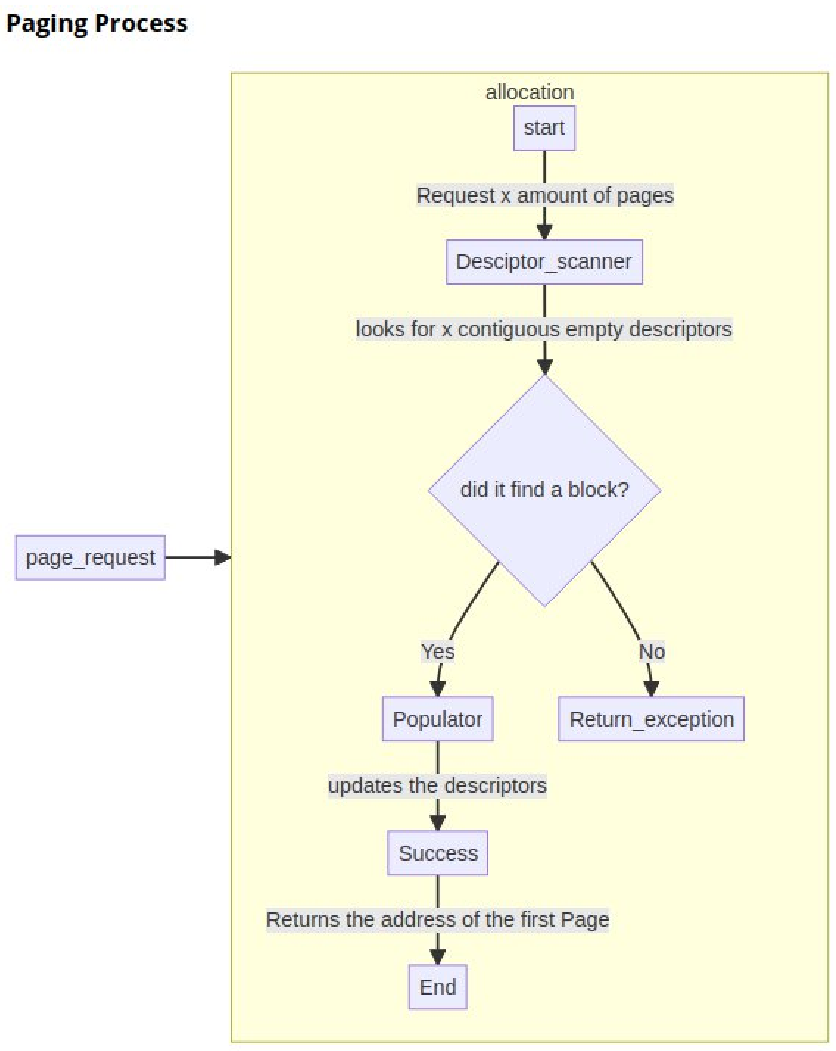


Figure 13: An Illustration of the Page allocation process

When the page allocator gets a page request from a user application, it scans through the set of page descriptors while looking for a sufficient contiguous space. If it finds space, it returns the first physical address of the contiguous page.

Deallocation also works in the same manner. The Page de-allocator scans through the set of populated descriptors and frees the targeted descriptors only. To enforce data cleaning, the page de-allocator zeroes out the data in the actual pages that were associated with the descriptors that were freed. This prevents any data-leaks.

5.3.1.4: The Byte Manager

The byte manager manages how bytes are allocated and deallocated bytes within a page. The byte allocation system allocates space per-byte, much like how malloc is implemented in C++. To keep track of which bytes have been allocated within a single page, the byte allocator uses a linked list as demonstrated below :

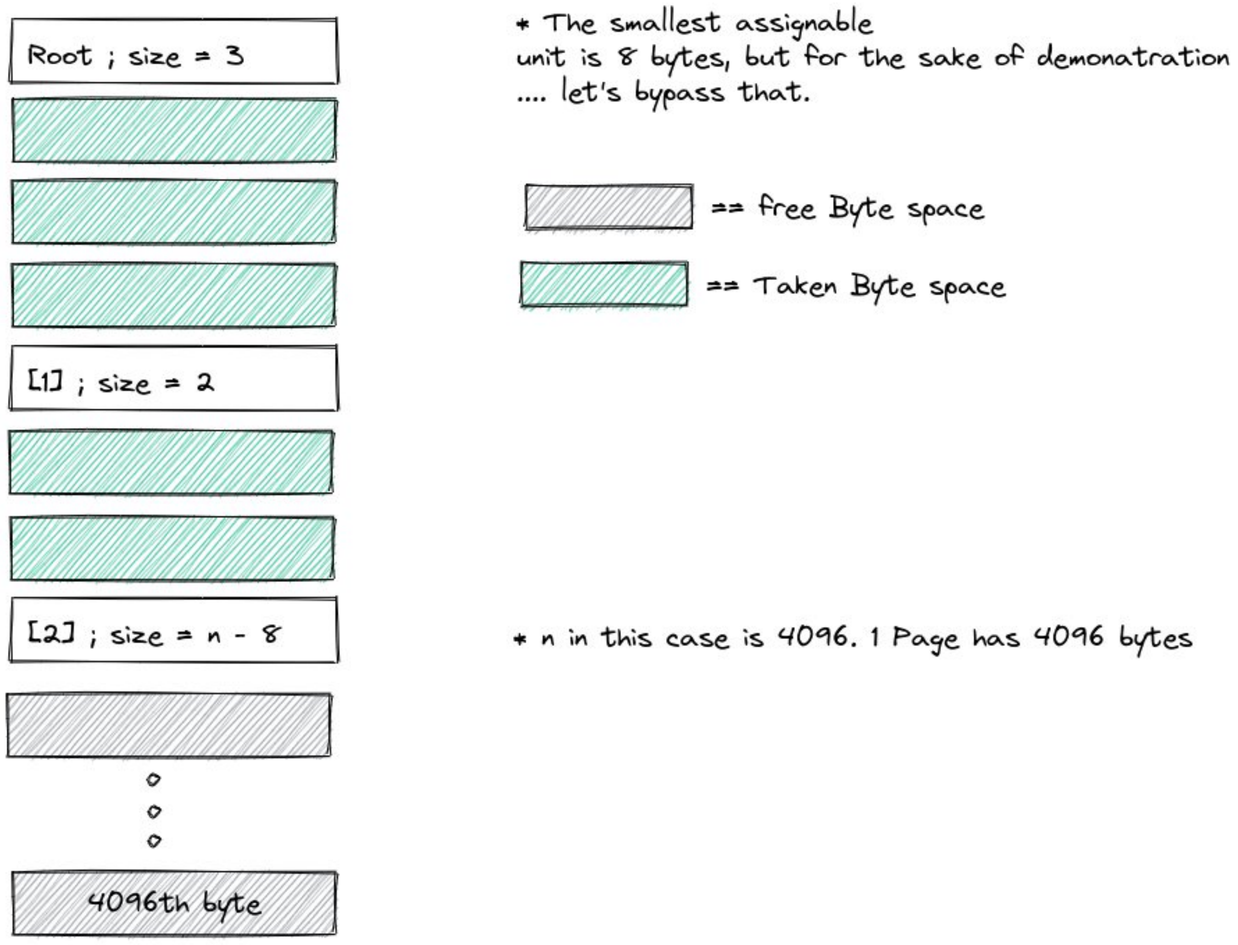

Root ; size = 3
[1] ; size = 2
[2] ; size = n - 8
4096th byte
* The smallest assignable unit is 8 bytes, but for the sake of demonatration .... let's bypass that.
== free Byte space
== Taken Byte space
* n in this case is 4096. 1 Page has 4096 bytes

Figure 14: Demarcation of a Page in order to keep track of allocated and un-allocated bytes

Using a linked list to keep track of allocated and unallocated memory pages offers several advantages.

Firstly, linked lists provide flexibility in accommodating varying memory segment sizes. Unlike arrays, which have fixed sizes, linked lists can dynamically allocate memory for each entry, allowing the allocator to efficiently manage both small and large memory requests without excessive waste or fragmentation.

Secondly, linked lists enable efficient memory allocation and deallocation operations. When a memory request is made, the allocator can traverse the linked list to find a suitable memory segment. Once a segment is allocated, it can be efficiently marked as used by updating the linked list pointers. Similarly, when memory is deallocated, the allocator can quickly update the linked list to mark the segment as available, without needing to shift or rearrange other memory segments.

Thirdly, linked lists facilitate coalescing of adjacent free memory segments. As memory is allocated and deallocated, free memory segments may become contiguous. With a linked list, it's relatively straightforward to identify neighboring free segments and merge them into a larger, continuous block of free memory. This helps in reducing memory fragmentation and optimizing memory utilization.

Lastly, linked lists provide a straightforward mechanism for traversing and managing memory segments. The allocator can maintain separate linked lists for allocated and free memory, allowing efficient iteration through both lists for various purposes such as memory allocation, deallocation, coalescing, or memory compaction.

In summary, using a linked list to manage allocated and unallocated memory pages in a byte allocator offers the benefits of flexibility, efficiency, and simplification of memory management operations, contributing to a more effective memory allocation strategy.

5.3.1.5: The Memory Management Unit

The instructions in the Elf files of programs typically reference virtual addresses. The CPU cannot execute an instruction that has a virtual address. For this reason, everytime the CPU encounters a virtual address in an instruction, the CPU uses the MMU circuitry to translate the virtual address into a physical address.

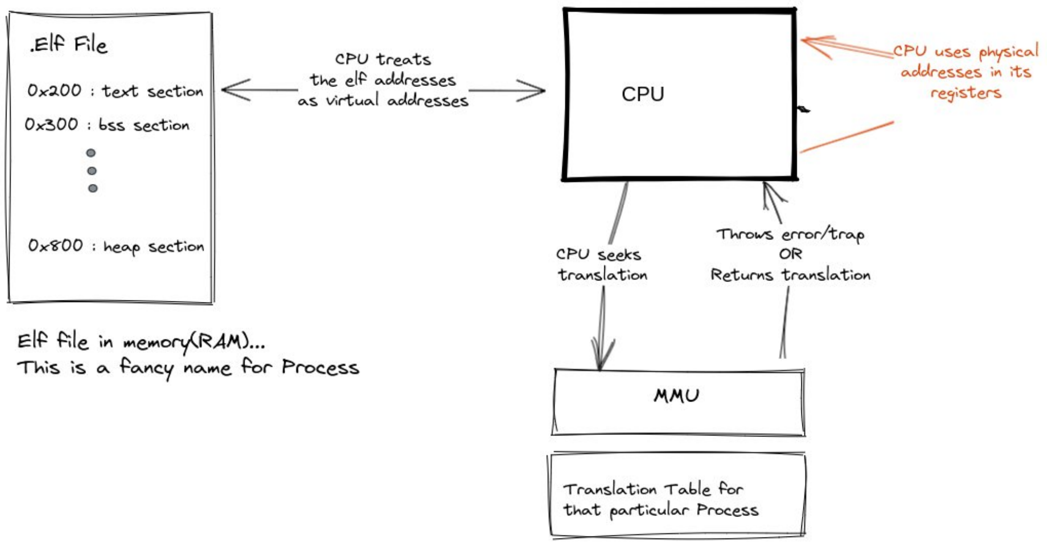

.Elf File
0x200 : text section
0x300 : bss section
0x800 : heap section
Elf file in memory(RAM)...
This is a fancy name for Process
CPU treats the elf addresses as virtual addresses
CPU
CPU uses physical addresses in its registers
CPU seeks translation
Throws error/trap OR Returns translation
MMU
Translation Table for that particular Process

Figure 15: Illustration of how the MMU fits into the architecture.

5.3.1.6: Identity Mapper

The Identity Mapper updates the Kernel MMU translation table while the kernel in machine mode. This enables the kernel to access all memory sections while being in supervisor mode.

5.3.1.7: The Block Driver

The block driver is responsible for managing and controlling the data exchange between the computer's operating system and block storage devices, such as hard drives or solid-state drives. It translates high-level read and write commands issued by the operating system into low-level instructions that the storage device can understand, ensuring efficient storage and retrieval of data in fixed-size blocks.

5.3.1.8: The standard Input and Output

The Standard input is the keyboard while the standard output is the console. The kernel interacts with these two components via the UART device and the UART driver.

5.4 The Hardware Components

The system's hardware components encompass a comprehensive set of essential elements, as outlined below:

1. Hard Disk: The primary storage medium for persistent data storage.
2. RAM (Random Access Memory): The volatile memory utilized for program execution and data storage.
3. CPU (4 cores): The central processing unit comprising four cores for computational tasks.
4. MMIO Memory Region: Memory-Mapped I/O region for interacting with hardware peripherals.
5. UART Converter: The hardware module responsible for converting parallel data signals into serial data signals and vice versa. It facilitates serial communication between the motherboard and external devices.
6. Core Local Interruptor (CLINT): A component handling local interrupts for cores.
7. PLIC (Platform Level Interrupt Controller): The controller managing interrupts for various platform components.
8. Console: The output display interface for user interactions.
9. Keyboard: The input interface for user commands and interactions.

Hardware Driver Functions

The tasks executed by hardware drivers predominantly fall within two distinct categories: Firstly, drivers may manipulate hardware behavior by modifying the status of pertinent registers. Alternatively, a driver might abstract the underlying hardware, providing a more user-friendly interface. It's worth noting that not every hardware component necessitates a dedicated driver. This is due to Rust's capability to directly access raw memory addresses of component registers without configuring communication protocols. For instance, the Core Local Interruptor (CLINT) operates efficiently without requiring a driver, exemplifying this direct access approach.

The UART Driver

The UART device serves as a hardware intermediary, bridging the gap between parallel and serial connections by converting signals from serial to parallel and vice versa. Positioned between the motherboard and peripheral devices employing serial connections like mice, keyboards, and console outputs, the UART device facilitates seamless communication between these components, enabling the efficient exchange of data.

In the project, the UART serial input connection has been connected to the Keyboard while its output connection has been connected to the Qemu console output. The UART links the motherboard and the external devices as shown in the image below

Overview

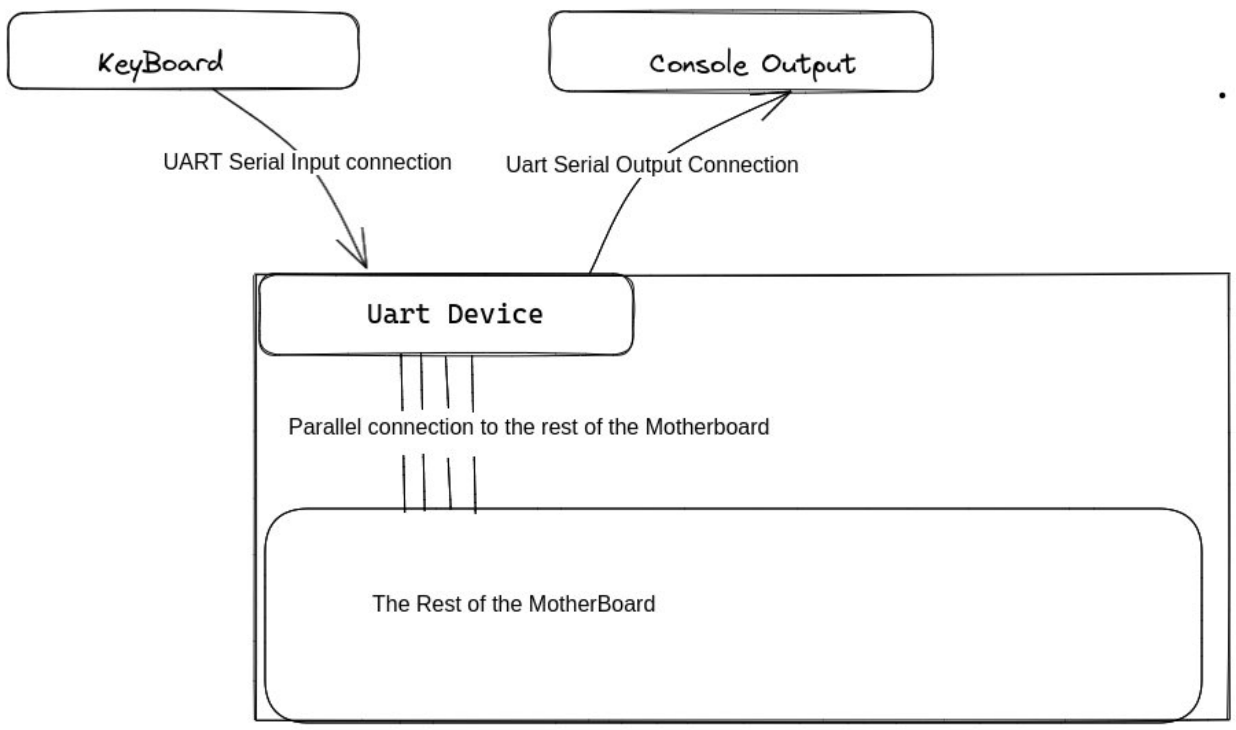

KeyBoard
Console Output
UART Serial Input connection
Uart Serial Output Connection
Uart Device
Parallel connection to the rest of the Motherboard
The Rest of the MotherBoard

Figure 16: Interaction of the UART with other components

The Keyboard is treated as the standard input device while the console is treated as the standard output device.

When a key is pressed on the keyboard, bytes are sent to the UART buffer. Once the UART buffer receives the input bytes, it sends an interrupt signal to the CPU to inform it that the UART has new data that needs to be read. The Interrupt signal is first received by the PLIC. The PLIC passes the interrupt to the CPU after it has filtered and prioritizes the UART interrupt in relation to other possible interrupts.

The CPU falls into a trap and it summons the trap handler. The trap handler takes care of the trap by reading the input bytes from the UART buffer and storing them into the standard input buffer.

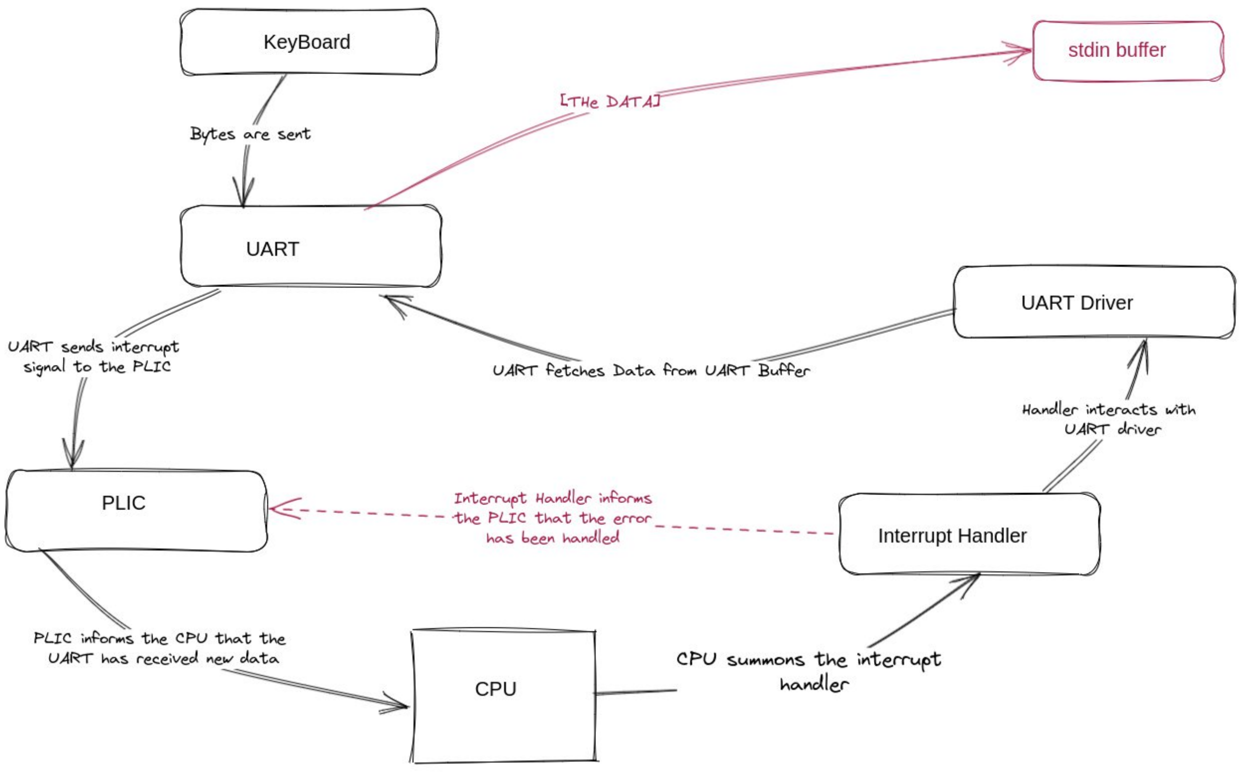

KeyBoard
stdin buffer
[THe DATA]
Bytes are sent
UART
UART Driver
UART sends interrupt signal to the PLIC
UART fetches Data from UART Buffer
Handler interacts with UART driver
PLIC
Interrupt Handler informs the PLIC that the error has been handled
Interrupt Handler
PLIC informs the CPU that the UART has received new data
CPU
CPU summons the interrupt handler

Figure 17: An illustration of the interrupt-driven interaction between the keyboard and the UART

Memory Sections

The memory sections being handled include : the RAM, the dedicated I/O section and the Hard-disk

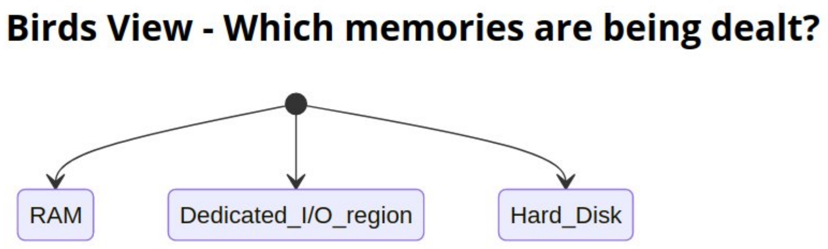


Figure 18: An illustration of the memory sections that have been abstracted in the project.

The RAM and the dedicated I/O sections get accessed directly by the CPU. At the start of the machine, the kernel image gets loaded onto the RAM, and it occupies the entire RAM as depicted below

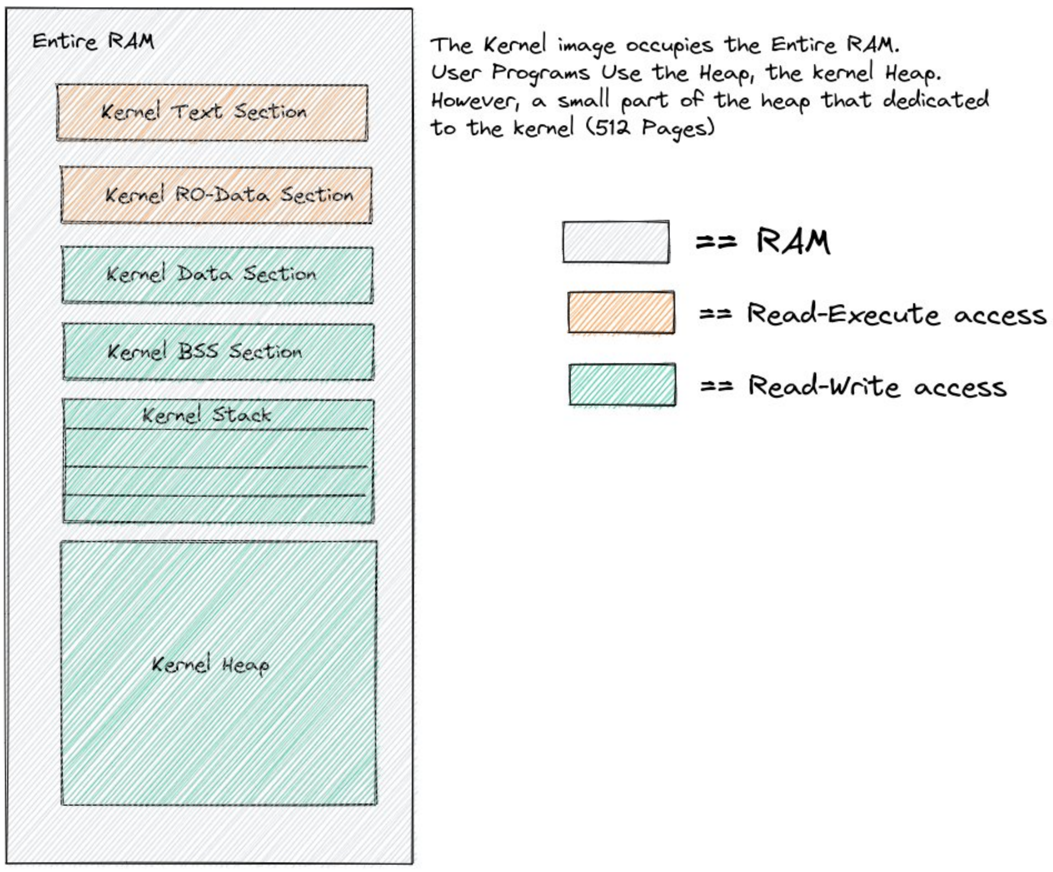


Figure 19: Layout of the entire RAM after the kernel Image gets loaded.

The kernel image heap occupies the largest part of the RAM. This space gets dynamically allocated to both the kernel and all loaded user programs. To ease the performance latency and time latency introduced by memory segmentation, paging was chosen to subdivide the heap.

Each page occupies 4096 bytes and deallocation and allocation gets done by a Page Manager. To clearly segregate kernel pages from User pages, the Heap gets demarcated into descriptors, dedicated kernel pages and user pages. This has been illustrated below.

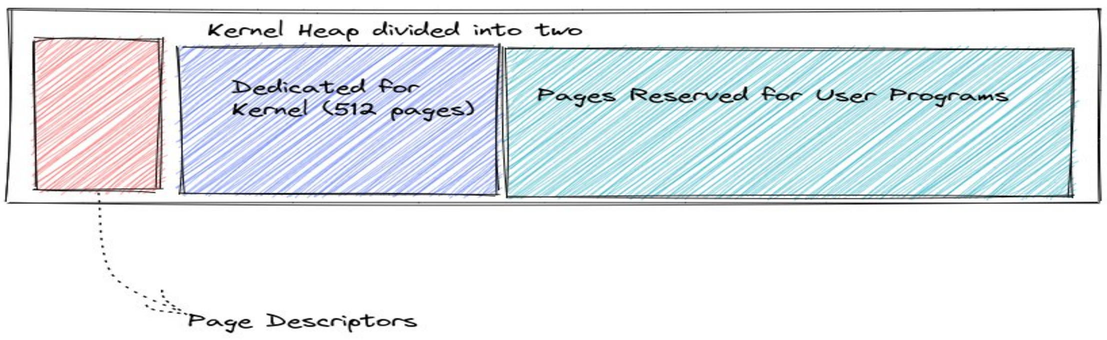

Kernel Heap divided into two
Dedicated for
Kernel (512 pages)
Pages Reserved for User Programs
Page Descriptors

Figure 20: An illustration of how the heap gets divided into Descriptors and dedicated Pages

CHAPTER 6: SYSTEM IMPLEMENTATION

This section discusses how different components were implemented. The code repository can be found at: https://github.com/kiarie404/Hobo-OS

6.1: System setup

The kernel was written for Qemu-virt riscv machine. This compelled the developer to set up a cross-compiling toolchain. The Riscv-GNU toolchain [12] was initially used but the Rust Default toolchain proved more user friendly in the end. The Riscv GNU had installation difficulties at first, this was followed by configuration subtle tweaks and bugs. Moreover, it required the use of Makefiles which were very cumbersome.

The Rust toolchain however has an easier time installing and configuring. There is no need to write Makefiles, cargo automatically takes care of dependency and compilation dependencies. All the chain tools (linker, compiler, Assembler, debugger, binary file inspectors) are installed by default when you install the language compiler. In one command.

The Rust toolchain also supports RISCV assembly, you can write asssembly code within rust code. You can link assembly files and the toolchain will take care of combining them into object files.

The rust toolchain allows one to download compiler backends called targets. Like for the case of this project, the target installed was `riscv64gc-unknown-none-elf`.

The process of linking the different object files and determining the layout of the final ELF file was defined via a linker script. Configuration on the dependency management and compilation was done via cargo.

The toolchain setup was followed by making a binary file that could run on bare metal, without standard Library support. This involved setting up the right nightly compiler features and defining the error handling personality of the executable.

6.2 The Bootloader

The boot program was responsible for :

1. Setting up the environment for the kernel to run in machine mode
2. Transferring the control to kinit() (kernel running in machine mode)
3. Designating a specific place where the CPU will jump to if a trap occurred when kernel is in either machine mode or supervisor mode.
4. Initiating an environment for the kernel to run in supervisor mode.

6.3 The UART Driver

The UART driver controls and interacts with the physical UART device.

In the case of Qemu, the console output and the keyboard input use the same UART device. This is because the transmit_out channel is connected to the console output while the receive channel is connected to the keyboard.

The UART emulated in Qemu is the NS16550A UART chipset. It is accessed and controlled using MMIO programming. The Base address of the UARTs begins at 0x1000_0000 and each UART device is given an offset of 0x100 (256 bytes)

The UART has 8 physical registers that can be interpreted as 12 logical registers, this is because some of the physical registers can be used differently under different contexts. For example The Buffer Register can be used as an input register when the UART is idle, but when the UART is not idle, the same register will be treated as an output register.

In the implementation, there was an option of using the USB communication protocol. The USB also does the conversion of parallel signals to serial signals. The USB has higher transfer speeds than the UART connection. The USB can do 20 Gbps while the UART does around 1 Mbps!
However, the project uses the UART because of two reasons :

Its simplicity in configuration. The UART is easier to configure when compared to the USB. This is a learning project, it is expensive to delve into the complex nature of USBs.

It uses less power than the USB. The project is targeting embedded systems; the less power the machine consumes, the better.

There is a trade-off the speed of the USB for the simplicity and power-efficiency of the UART.

Implementation of the UART

Implementing the UART driver involved initializing communication configurations and making the UART reads and writes to be interrupt driven via the PLIC.

6.4 Byte Management

The byte management module was implemented only for the kernel Heap. User applications solely rely on the page allocator. The Byte manager used a linked list instead of an array of descriptors in order to ensure that memory was used efficiently as discussed in the System design chapter.

6.5 The Memory Management Unit

Programs' Elf files commonly contain instructions that point to virtual addresses. However, the CPU cannot directly execute instructions with virtual addresses. Consequently, whenever the

CPU encounters a virtual address within an instruction, it employs the MMU circuitry to convert the virtual address into a corresponding physical address.

The MMU in the project has been programmed to enforce access rights to certain physical memory addresses, such that a translation will fail if an access right is being violated. If the translation fails, a page fault is thrown by the MMU and the interrupt handler handles it. In this case, the MMU acts as a memory protector, ensuring translations only happen when all access rights are adhered to.

In Machine Mode, RISCV provides a mechanism of protecting memory called Physical Memory Protection(PMP). But PMP does not scale well. Disadvantages of the PMP mechanism only allow the creation of 16 sections. However, with a virtual paging system memory system you can get more granularity. PMP does not cater for a system that runs multiple complex applications where each application may need its own set of pages with different access levels.

Each Process gets assigned its own translation tables, therefore, each process gets its own address space

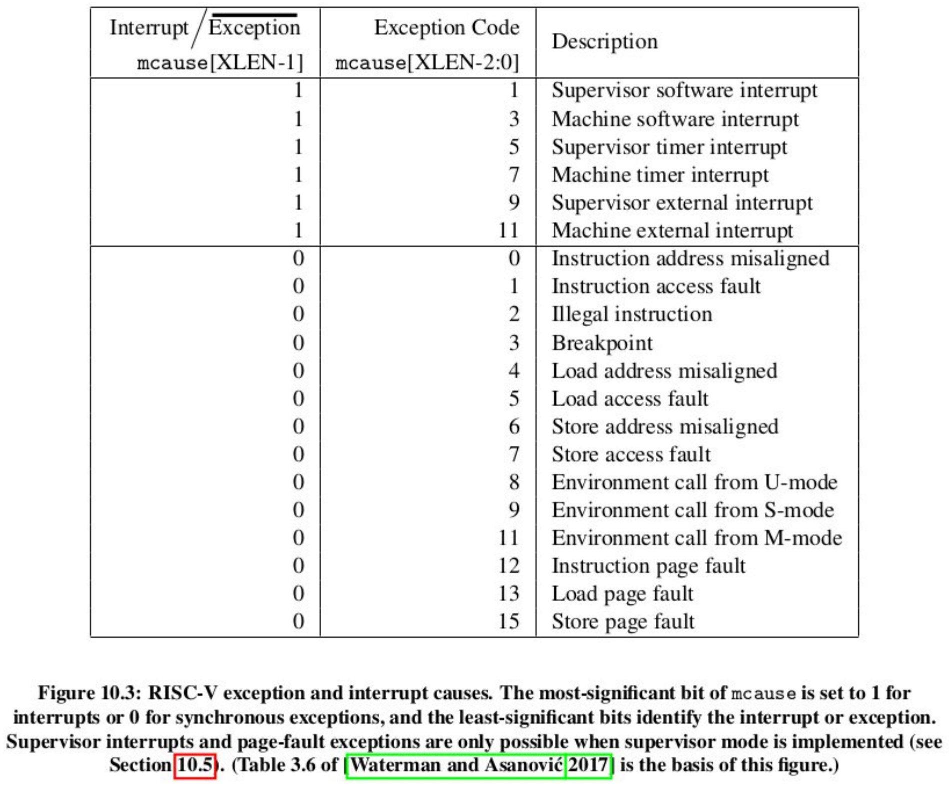
6.6.1 Interrupt and Exception Handling

Privileged modes give you access to some registers that can be used to configure how the CPU deals with Interrupts and exceptions.

An exception occurs when the disturbance comes from the code that is currently getting executed. For example, if the current instruction tried to write to a read_only memory location, an exception will occur.

An interrupt occurs when the disturbance does not come from the code executing in the subject HART. This disturbance might come from another HART or the PLIC; something that is not the code running in the subject HART

Riscv acknowledges the following Exceptions and Interrupts. The project stuck to only these interrupts.



Interrupt / $\overline{\text{Exception}}$ mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

Figure 10.3: RISC-V exception and interrupt causes. The most-significant bit of `mcause` is set to 1 for interrupts or 0 for synchronous exceptions, and the least-significant bits identify the interrupt or exception. Supervisor interrupts and page-fault exceptions are only possible when supervisor mode is implemented (see Section 10.5). (Table 3.6 of [Waterman and Asanović 2017] is the basis of this figure.)

Figure 21: Exceptions and Interrupts that have been considered for the project.

All exceptions have been handled in a similar manner, save for the three environment call exceptions.

If an Environment call happens from U-mode or S-mode, the trap handler switches to machine-mode and executes the required privileged code.

If an Environment call happens in M-mode, the kernel falls into an artificial state in which it cannot recover from, it then shuts down.

All other exceptions are treated as unrecoverable and result in the display of an elaborate error message followed by a kernel shutdown.

The Timer interrupt has not been fully handled. It can be detected but it cannot be used in the scheduling process yet. For this reason it gets disabled at start-up.

6.6.2 Handling External interrupts

The project relies on the PLIC (Platform level Interrupt Controller)circuitry to manage all external interrupts. The PLIC contains many interrupt pins that are connected to all external sources.

The PLIC interfaces with the CPU via a physical Interrupt pin. The interrupt pin is enabled, disabled and configured using the MIE register (the machine interrupt Enable register)

The meie bit is found within the MIE register. It enables the acceptance of external interrupts.

The function of the PLIC is to receive all external interrupts, choose which interrupts have to get filtered out and sort out the remaining interrupts by their order of priority. From there, it provides a way for the CPU to pick the pending interrupts one by one.

Below is a diagram that shows how the PLIC fits into the architecture

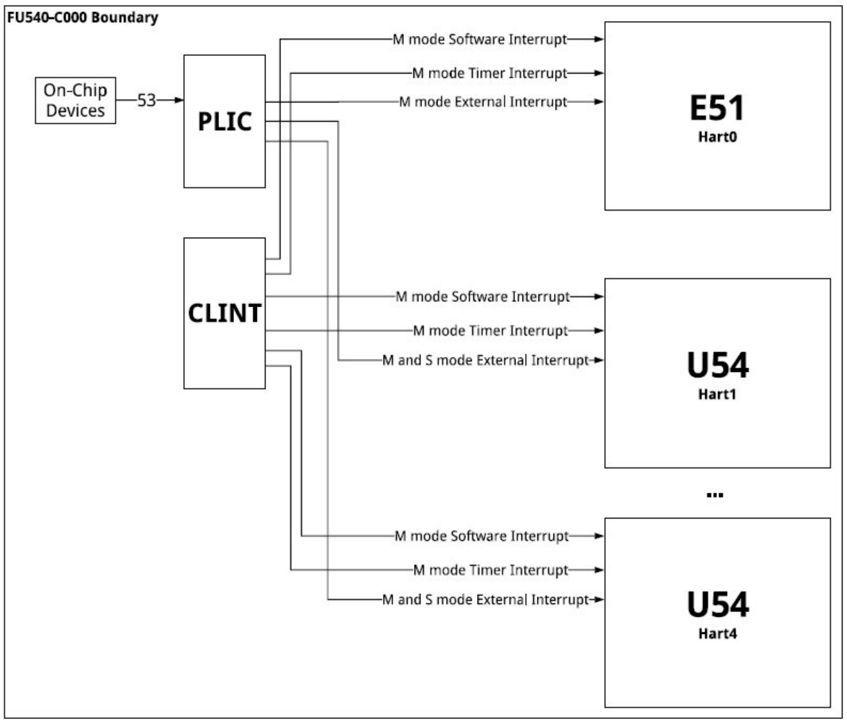

FU540-C000 Boundary
On-Chip Devices
53
PLIC
CLINT
M mode Software Interrupt
M mode Timer Interrupt
M mode External Interrupt
E51
Hart0
M mode Software Interrupt
M mode Timer Interrupt
M and S mode External Interrupt
U54
Hart1
...
M mode Software Interrupt
M mode Timer Interrupt
M and S mode External Interrupt
U54
Hart4

Figure 3: FU540-C000 Interrupt Architecture Block Diagram.

Figure 22: An illustration of how the PLIC, CLINT and CPU interact.

6.7 The Identity Mapper

The identity Mapper maps the whole RAM and IO memory to the translation table of the kernel. Identity mapping ensures that the kernel code still has access to all memory sections despite being in supervisor mode. Mapping in this case is special because it does not matter whether the mapped pages have been allocated or not.

6.8 The Block Driver

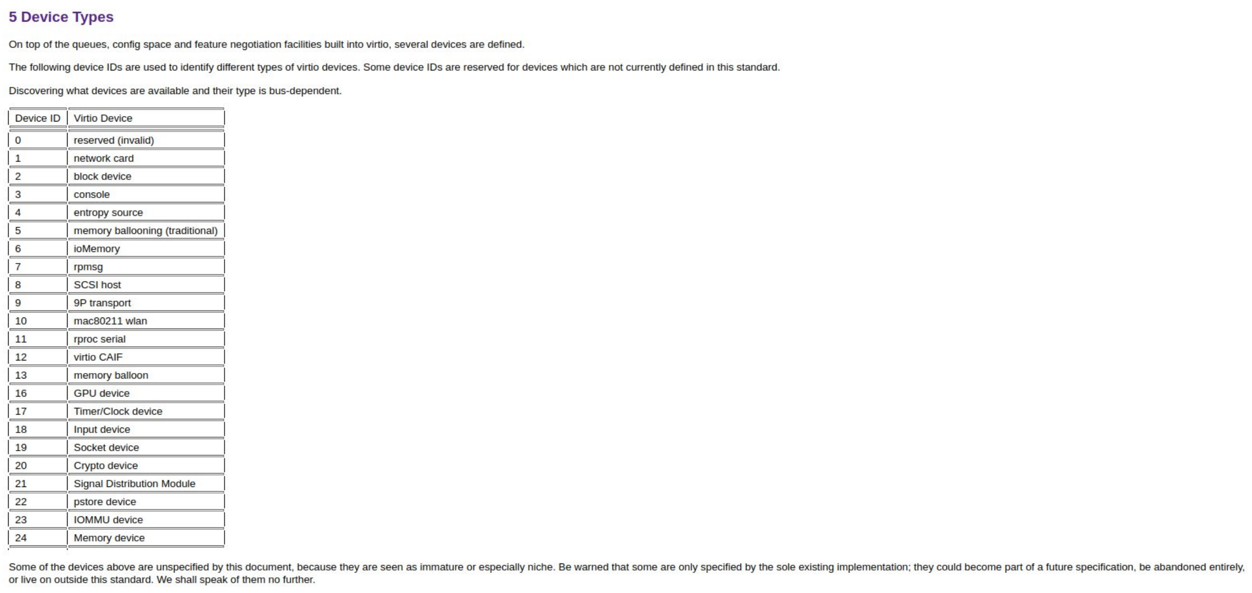
The block driver interfaces with the hard-disk. The project used a virtual Hard-disk that used the VirtIO protocol. The virtual hard-disk was initially a file that had been turned to a loop device using the Losetup command line tool. The Hard-disk has a maximum size of 32 Mibs.

VirtIO protocol is a communication protocol. It defines how a virtual machine communicates with a hypervisor. It also defines the API of a virtual device.

Qemu exposes virtual devices in the MMIO dedicated memory section. It puts the virtio devices (backwards) from 0x1000_1000 to 0x1000_8000. For example if we decide to attach

only one virtio device, we will attach it at 0x1000_8000 instead of 0x1000_1000.
 Qemu supports 8 Virtio buses. Each having 4096 byte register space.

Below is a list of the device IDs as used in Qemu.



5 Device Types

On top of the queues, config space and feature negotiation facilities built into virtio, several devices are defined.

The following device IDs are used to identify different types of virtio devices. Some device IDs are reserved for devices which are not currently defined in this standard.

Discovering what devices are available and their type is bus-dependent.

Device ID	Virtio Device
0	reserved (invalid)
1	network card
2	block device
3	console
4	entropy source
5	memory ballooning (traditional)
6	ioMemory
7	rpmsg
8	SCSI host
9	9P transport
10	mac80211 wlan
11	rproc serial
12	virtio CAIF
13	memory balloon
16	GPU device
17	Timer/Clock device
18	Input device
19	Socket device
20	Crypto device
21	Signal Distribution Module
22	pstore device
23	IOMMU device
24	Memory device

Some of the devices above are unspecified by this document, because they are seen as immature or especially niche. Be warned that some are only specified by the sole existing implementation; they could become part of a future specification, be abandoned entirely, or live on outside this standard. We shall speak of them no further.

Figure 23: Device IDs as used in Qemu.

The block driver probes the bus memory section, negotiates features and sets the default configurations between the block driver and the hard-disk. It also co-ordinates the communication between the driver and the block by managing the descriptors in the VirtiQueues.

Implementing the block driver proved complex and time consuming. So the project ported the block driver that was implemented by Stephen Marz[13]

6.9 Porting the wasm Runtime to a no-std environment

Porting Wasmi runtime to a no-std environment was successful. The process involved reading the wasmi crate documentation and finding the version that worked seamlessly with the project. The latest version 0.30.0 did not seamlessly run in the no-std space, the “core” feature had been removed ; compiling it gave out compilation errors.

Version 0.4.0 seamlessly supported no-std, however, it was compiled by an early version of the rust compiler that did no fully support inline assembly. Considering that the project depended on inline assembly, wasmi version 0.4.0 was not chosen.

Version 0.9.0 found a balance. With a little tweaking, it ran on a no-std environment. It was also compiled with a compiler that supports inline assembly.

6.10 Exporting system calls as WASI calls

This project objective proved to be too ambitious to be implemented within the project designated time. The implementor exported the system calls as they were, they did not use

WASI prototypes. This was because the WASI prototypes required functionalities that were not yet implemented.

6.11 Integrating the Minix 3 file system

Making the virtual hard-disk to implement the minix 3 file system was successful, however, integrating it and implementing an interface for it proved difficult. Attempts to port it from another project yielded bugs and unexpected behavior. The File system is unimplemented.

6.12 Unit Testing

Not all modules have unit tests. Testing in a no-std environment was challenging but with a little compiler tweaking, the implementor managed to run unit tests without the need of employing third party libraries.

The Rust Compiler built in test framework was not used for the unit tests. This was because the framework was dependent on a library called "test". The "test" library is dependent on the standard library. Since the project was based on a no-std environment, the default test-framework used by the rust compiler became unusable. Using a nightly feature of the compiler, a custom no-std framework was defined.

6.13 Integration Testing

Six inclusive integration tests were performed.

The standard input and output were tested if their executions were poll-based and they passed. However the interrupt driven approach occasionally produced unexpected results. For example, if the stdout and stdin get invoked simultaneously, the output generated or the input received may have missing or extra data. This is because the two processes share the same UART without a mutex or a co-ordinator to make sure invocations act in a deterministic way.

The Memory Management Unit integrated well with the Rust Global allocator. As a result, it was possible to allocate bytes, vectors, strings and other data structures that are dependent on heap allocations.

The Memory management Unit performed successful virtual address translations and mappings while the CPU was in Supervisor-mode. The MMU rejected the translation and mapping of invalid physical and virtual addresses. The MMU also rejected the mapping of valid addresses using an invalid access map.

The **Identity Mapper integrated well with the MMU,** the kernel pages that were mapped into the kernel translation table were accessible without throwing unexpected page faults.

The ported block driver was able to integrate with the virtual Hard-disk. An attempt to write to the hard-disk was successful. An attempt to read from the hard-disk was also successful.

The Page allocator integrated well with the MMU and memory initializer. The page allocator was able to allocate the pages that were demarcated by the memory initializer. The Page allocator worked well in both Machine mode and supervisor mode, this proved that the MMU mapping and translation in the background was seamless.

6.14 Tools used

1. The Rust Language – The rust language was the main language used in the project.Among many reasons, it was chosen because of its implementation of memory safety, its support for inline assembly, its package management and its elaborate compiler messages.
2. The Riscv Language
3. Cargo Binutils – Cargo Binutils helped in analyzing object files.
4. Mdbook – Mdbook assisted in documenting the project using the MarkDown Language
5. Mermaid.js – MermaidJS wa used in drawing diagrams using text and code. It made diagram modification easier
6. Excalidraw – Excalidraw was used to draw illustrations from within the text editor.
7. Losetup – Losetup was used to convert regular data files to oop devices.
8. Riscv-GNU toolchain – the Riscv-GNU toolchain was initially used to cross-compile code to target the bare-metal riscv platform.
9. Gnu Debugger – The GNU debugger was used to debug the program. It proved especially useful in situations where print statements could not work. For example, identifying the values of the CPU registers during runtime.
10. Wasmi – wasmi was useful in loading and executing wasm modules
11. Wabt(Web Assembly Binary Toolkit)
12. Qemu – Qemu acted as the default riscv virtual machine.

Conclusion

The project aimed to build a kernel that runs wasm modules in kernel mode and also exposes its system calls using the WASI specifications. This was in a bid to prove that wasm modules could run as fast as native apps in the embedded space. The research conducted provided valuable insights into different kernel designs and implementations, guiding the project in making informed trade-offs when choosing which features to implement. The designs from the Tock embedded operating system were particularly helpful in this regard.

However, the exportation of WASI system calls proved to be more challenging than anticipated. The implementor underestimated the time required to achieve this feat, resulting in incomplete implementation within the project's timeline. As a result, no definitive conclusions can be made about the performance of wasm modules in the embedded space based on this project .

Despite the challenges faced, several components were successfully built and tested as part of the project. These include the bootloader, the memory initializer, the Page Manager, the Byte Manager, the interrupt handler, the UART driver, the PLIC driver, and the MMU component. The block driver was also successfully ported to the project. These components were designed and implemented using a monolithic kernel architecture, which facilitated seamless communication between different modules and expedited development and debugging.

To fully realize the design outlined in the project, further work is needed to complete the exportation of WASI system calls and integrate them into the kernel. This could be done by first porting WASI functions whose implementations can be satisfied by the kernel as is. The kernel can easily handle WASI functions under memory management, I/O and Timer control.

Additionally, the project would benefit from the implementation of other planned components, such as the networking module and the file system module. These additions would enhance the functionality and versatility of the embedded operating system.

In conclusion, while the project encountered challenges in exporting WASI system calls and did not achieve its initial goal of fully demonstrating the performance of wasm modules in the embedded space, it provided valuable insights into kernel design and implementation. The components that were successfully built and tested lay the foundation for future development and integration, paving the way for further exploration of wasm in the embedded operating system domain.

Recommendations

While the project has made progress, certain fundamental goals remain unmet, indicating potential areas for improvement. In light of this, the following recommendations are put forward:

Integration of File System: A pivotal step towards enhancing the project's functionality would involve integrating a comprehensive file system. This integration would streamline file referencing by enabling the system to access files through their designated path names, a feature that could significantly elevate the user experience.

Exporting System Calls as WASI Calls: Despite the unfinished nature of the project, it remains feasible to pursue the objective of exporting current system calls as WebAssembly System Interface (WASI) calls. By aligning the kernel's functionality with the WASI standard, the project could tap into the advantages of standardized interfaces, ultimately enhancing compatibility and interoperability.

Enhancing Memory Abstractions: A notable observation pertains to the efficiency of memory abstractions within the project. Particularly, the process of descriptor parsing has been observed to introduce latency concerns. To address this, the adoption of more advanced algorithms for memory parsing could yield improvements, effectively mitigating parsing latency and optimizing overall system performance.

These recommendations are aimed at further refining the project, fostering its completion, and potentially augmenting its functionality and efficiency.

Appendix

[1] : “Isomorphic Internet of Things Architectures With Web Technologies” by Mikkonen, Tommi. Available online [https://ieeexplore.ieee.org/document/9473238] last accessed 14th Aug 2023.

[4]: Wasmachine: Bring IoT up to Speed with A WebAssembly OS By Elliott Wen and Gerald Weber

[5]: Github List of Webassembly Runtimes. Available Online [https://github.com/appcypher/awesome-wasm-runtimes] last accessed 14th Aug 2023.

[6]: Li Clark’s Article on WebAssembly System Interface. Available online at [https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/] last accessed 14th Aug 2023.

[7]: Official WASi website : https://wasi.dev/

[9]: Official Tock Website on its design :Available online at [https://tockos.org/documentation/design] last accessed 14th Aug 2023.

[11]: More Documentation on this project : https://kiarie404.github.io/CSC-416-4TH/main_site/Documentation_books/Developer%20Documentation/book/index.html

[12]: The RISCV – GNU toolchain : https://github.com/riscv-collab/riscv-gnu-toolchain

[13]: Stephen Marz block driver : Available online at [https://github.com/sgmarz/osblog/blob/master/risc_v/chapters/ch9/src/block.rs] last accessed 21st Aug 2023