Writing an OS in Rust

Manisha kumari

Code: https://github.com/iit2018062/blog_os
Video: https://drive.google.com/file/d/1K6hq24VAZIQ3e0uODSTbnECe6-_2EQ9K/view?usp=sharing

1. Introduction

With time, the Rust programming language has gained much popularity in recent years. The key features of rest involve memory safety, concurrency, and performance, making it an excellent choice for system programmers. Creating an operating system(OS) from scratch is one of the most challenging and rewarding projects in system programming. With this report, we outline my journey in building a simple yet functional operating system using RUST. The project is based on Philipp Oppermann's blog series "Writing an OS in Rust."

The blog provides a comprehensive guide and a step-by-step tutorial on implementing OS features and source code. For this project, we will work on concepts such as memory management, interrupts, heap allocation, and multitasking using Rust's async/await features. The goal is to create a 64-bit kernel for x86 architecture capable of running Rust code on bare metal without relying on an underlying operating system.

2. Project Goals

- Create a Freestanding Rust Binary
- Construct a 64-bit kernel for the x86 architecture that can be booted using a disk image and is capable of printing output to the screen.
- Provide an interface for safe and efficient text output using VGA text mode. Encapsulate unsafe code within dedicated modules and support Rust's formatting macros.
- Integrate unit and integration testing for the kernel using Rust’s custom test frameworks, and report test results through QEMU emulation.
- Handle CPU and Hardware Interrupts
- Implement Memory Management
- Support Heap Allocation
- Explore Allocator Designs
- Introduce Multitasking with Async/Await
- Apply Rust’s safety guarantees, ownership model, and concurrency support in a low-level programming context, addressing real-world challenges associated with kernel and OS development.

The reason why I chose the above goals is to learn more about system programming, Rust, low-level OS development, and energy-efficient programming.

3. Methodology

3.1 Started the project by making a freestanding "**Rust Binary**".

- In the start, I made a Rust executable a minimal Rust-based OS which runs without an underlying operating system. For this, we disabled the "no_std" library of rust and defined a custom panic handler to handle any undefined behaviour. To prevent unwinding of the stack on panic, we specified panic = "abort" in cargo.toml. Now that we don't have a standard "main" function, we needed to define an entry point using "#[no_mangle] pub extern "C" fn _start() -> !". As we don't have a default linker we compiled for a bare-metal target which helped in resolving this issue.
- These steps helped in creating a custom target for x86_64 bare metal, combining the binary with a bootloader and printing the output on the screen.

3.2 Setting Up a Minimal Rust Kernel

- To build a Rust kernel, a custom target specification file x86_64-blog_os.json was created which has specifications like disabling SMD(to avoid save/restore operation during interrupts), enabling Soft-Float(for software-emulated floating point operation), and setting rustc-abi to x86-softfloat.
- To build the kernel we modified the entry_point from the previous step. The kernel is compiled and a rust "core" library is added for custom targets. To enable required memory functions like memset, memcpy, and memcmp, the compiler_builtins-mem feature is enabled.
- A VGA text buffer at 0xb8000 is used for simple screen output. To write characters we used raw memory access in an unsafe block.
- To make the kernel bootable we used a bootloader crate (bootloader = "0.9"). The bootimage tool helps to compile and link the kernel with the bootloader.
- We tested the kernel image using QEMU using the command "qemu-system-x86_64 -drive format=raw,file=target/x86_64-blog_os/debug/bootimage-blog_os.bin"

3.3 VGA Text Mode:

- The VGA (video graphics Array) is a way of outputting text to the screen, this was most commonly used in early operating systems and simple kernel implementations. In this step, we analysed how to interact with the VGA text buffer directly in Rust to print characters and messages on the screen. The VGA text buffer is located at the physical memory address 0xB8000. This contains a 2d array (ScreenChar structure), each representing a character displayed on the screen along with the colour code.
- To manage the position of the cursor(column_position) and the colour of the text (color_code) we used Writer. It helps in writing characters and strings to the screen buffers.
- To handle the ASCII characters we use the write_string method which ensures that printable characters are displayed on the screen while the unprintable ones (such as special UTF-8 characters) are replaced with a placeholder.
- To prevent optimization of the writes to the VGA buffer by the compiler, the volatile crate is used. This makes sure that the writes have the intended side effects, ensuring that the output is correctly displayed on the screen.
- To print various data types like integers and floating point numbers we used fmt::write trait implementation which helps in using formatting macros like write! And writeln!.

- The new_line method helps in moving the screen content up by one line when the cursor reaches the end of the screen ensuring that next continues on the next line. To clear the last row when a newline is written we used clear_row.
- The lazy_static crate is used to create a globally accessible WRITER which allows any part of the code to print to the screen without passing a Writer instance around.
- To synchronize the access to WRITER by multiple places we used spinlock. This helps in making sure that Writer can be accessed safely without blocking, even in the absence of OS-level thread management.
- To print text with automatic newlines we define println macros. This utilizes the global WRITER to output formatted messages to the screen.

3.4 Testing

Testing plays a vital role in any kind of setup, for our implementation we included unit tests(for isolated components) and integration tests(for system-wide functionality) using Rust's custom test framework.

- Unit testing in a no_std kernel: Rust's default test framework doesn't work in no_std, we use custom test frameworks. In this case, we defined test_runner that collects and runs all test cases. For the output we used serial_println! for output since VGA output may be unreliable in tests. At the end of the test, call exit_qemu() to terminate QEMU.
- For panic handling, we defined test_panic_handler which prints errors and exits QEMU with a failure code. This also helps in preventing infinite loops during panic.
- To make the testing easier we moved core functionality from main.rs into lib.rs.
  Finally, lib.rs had : test_runner: runs test cases and exits QEMU, test_panic_handler: handles panics in tests, exit_qemu() - exits QEMU with a specific status code and serial_println! - enables serial output for test logs.
  This helped both main.rs and integration tests to reuse the same test infrastructure.
- Integration testing in tests/This helps in integration testing and system-wide behaviour verification. We implemented the following integration tests:
  1. basic_boot.rs: ensures that kernel boots and println works
  2. should_panic.rs: verifies expected panics

3. 5 CPU Exceptions

An exception in CPU happens when various erroneous situations like invalid memory access or division by zero. Handling these exceptions is important and to handle it an interrupt Descriptor Table (IDT) must be set up to define handler functions. There are the following kinds of exceptions:

1. Page fault: accessing unmapped memory
2. Invalid opcode: when executing an unsupported instructions
3. General protection fault: when violating access permissions.
4. Double fault: an exception which happens while handling other exceptions.
5. Triple fault: fatal error leading to system reset.

Interrupt Descriptor Table (IDT): the IDT consists of a function pointer to the handler, code segment selector, flags defining interrupt/trap gates and privilege levels.
The **x86_64 Rust crate** provides an InterruptDescriptorTable struct that abstracts the IDT setup.

3.6 Interrupt Handling and Stack Management: the x86-interrupt calling convention makes sure that all registers are preserved. The essential state information like stack pointer, instructions pointer, etc are saved by the CPU before jumping to a handler. Some exceptions push error codes.
For the above implementation, we created an InterruptDescriptorTable and assigned handlers. Then on defined handlers e.g. a breakpoint_handler which prints debug information. Enabled the unstable x86-interrupt ABI in Rust. Loaded the IDT to make it active using the load method. This helped the kernel to catch exceptions like breakpoints and resume normal executions.

3.7 Double Faults
For this part of our journey, we focused on handling double faults safely by setting up the Global Descriptor Table (GBT), configuring a Task State Segment (TSS) and modifying the interrupt Descriptor Table (IDT). In addition to this, we implemented an integration test to ensure that a stack overflow correctly triggers the double fault handler instead of causing a triple fault.

- The GDT is a core structure of x86_64 systems that define memory segments. In our project, we created a new GDT and added a kernel code segment, a Task State Segment(TSS) which handles special cases like stack switching during exceptions. Once our GDT was ready we loaded it into the CPU.
- Configuring the Interrupt Descriptor Table (IDT): The IDT helps to define how the CPU should handle exceptions and interrupts. To avoid system crashes from double faults we assigned an interrupt stack Table(IST) index. We made sure double fault uses a separate stack using the set_stack_fault() function. This prevents kernel stack corruption and avoids triple faults.
- Testing Stack Overflow Handling: To make sure that the double fault handler works as expected we implemented an interrupt test which deliberately triggers a stack overflow.
- Custom IDT for Testing: we created a test-specific IDT with a custom double fault handler for testing. This helps in making sure that our test double fault handler is used instead of the default one.
- Custom Double Fault Handler: Instead of panicking on a double fault, the test handler exits QEMU successfully, marking the test as passed.

3.8 Hardware Interrupts
For this section, we explored handling deadlocks, race conditions, energy-efficient CPU operation and keyboard input.

- In our case, a deadlock occurs when the println! Macro which locks a global WRITER is interrupted by an interrupt handler. In the scenario, the handler tries to print but it cannot acquire the already locked WRITER which leads to a system hang. To fix this interrupts are disabled while the WRITER is locked using interrupts::without_interrupts() to prevent the interrupt handler from trying to print while the WRITER is locked.
- The race conditions fail test_println_output. This test writes the string to the VGA buffer but the timer interrupt might trigger in between which leads to inconsistent results. To fix this test is modified to lock the WRITER during the entire process and disable interrupts during the test to avoid the timer interrupt from this issue.
- In our code, the kernel uses an empty loop (loop{}) to keep the CPU running which consumes a lot of power. To make it energy efficient we introduced halt instructions which are used to halt the CPU until the next interrupt arrives.

- Keyboard input: The inputs from the keyboard are processed by reading scancodes from the PS/2 keyboard controller. The keyboard_interrupt_handler is added to handle the keyboard interrupt and print a "k" when a key is pressed. The scancode is mapped to key presses (e.g. “1” for 0x02, “2” for 0x03). A match statement is used to translate these scancodes to their corresponding characters. The pc-keyboard crate is used to handle scancode set 1 and translate scancodes into actual characters, handling key events like presses and releases more accurately.

3.9 Paging

For this, we have a bootloader which has set up a 4-level paging hierarchy which ensures that every memory access is virtual. To safeguard the kernel by making any out-of-bound memory access a page fault exception. The bootloader also ensures that pages have the correct access permissions, such as read-only for code and writable for data.

We implemented how to handle page faults and access page tables within the kernel. To summarize:

- Paging setup: The bootloader helps in setting up a 4-level paging hierarchy for the kernel which ensures that every address used is virtual. Memory protection is enforced when we access out-of-bounds memory which invokes a page fault instead of corrupting random physical memory. Pages have proper access permissions.
- Page Fault Handler: The function “page_fault_handler” was created to handle page fault and was registered in the interrupt Descriptor Table(IDT). This handler uses the CR2 register to print the accessed virtual address which caused the fault, along with an error code detailing the type of memory access. The kernel is halted in case of page fault using hlt_loop().
- Testing Page Faults: To test the functionality, the kernel deliberately accesses invalid memory which triggers a page fault. The error codes (CAUSED_BY_WRITE, PROTECTION_VIOLATION) are printed to indicate a write access violation.
- Accessing Page Tables: The kernel accesses the level 4 page table by reading the CR3 register with the help of Cr3::read(). The output of this shows the physical address of the level 4 page table, however, the direct access to physical memory is restricted due to active paging.

3.10 Heap Allocation

- Setting Up the Heap: imported the blog_os::allocator module and initialized the heap using allocator::init_heap. For determining the physical memory offset we used boot_info and set up the memory mapper and frame allocator to manage heap memory.
- Using an External Allocator: To use an allocator we relied on linked_list_allocator (a crate designed for no_std environment) instead of writing an allocator from scratch. The allocator is used to track free memory using a linked list of available blocks.
- Defining a Global Allocator: In the file src/allocator.rs, we declared a global allocator using the LockedHeap struct. To initialize the allocator without any blocking memory we used LockedHeap::empty().
- Initializing the Heap Memory: Mapped a heap region and initialized the allocator to allocate memory. The translation of virtual memory to physical memory is done using a memory mapper which is also used to set up a heap.
- Testing Heap Allocation: To ensure everything worked correctly, we wrote tests in tests/heap_allocation.

3.11 Allocator Designs

We implemented FixedSizeBlockAllocator which allocated memory using fixed-size blocks (with a set of predefined sizes in BLOCK_SIZES). In our code, the FixedSizeBlockAllocator has two components list_heads (an array of linked lists to store free blocks of each size) and a fallback_allocator(a linked list allocator for allocation that doesn't fit the predefined block sizes).

Methods:

new: it initializes an empty allocator

init: help in setting up the allocator with heap bounds(unsafe)

Alloc: this allocates memory by checking the appropriate block size and uses either the block list or the fallback allocator.

dealloc: helps in freeing the memory either by adding it to the list or using the fallback allocator.

This kind of allocator offers a fast allocation by reducing fragmentation but may waste memory for sizes that aren't a perfect match to the block sizes.

Lazy initialization: the FixedSizeBlockAllocator only initializes block lists when allocation of corresponding block sizes is made. This improves efficiency by reducing overhead.

Synchronization: we wrapped the allocator in a Locked type to allow safe mutable access across different threads or contexts.

Alignment and size matching: we carefully calculated the block size and alignment to avoid wasting memory.

3.12 Async/Await:

Executor: implemented a custom Executor type to manage the scheduling of tasks. The tasks are stored in a shared queue tast_queue and the executor then runs the method continuously to poll tasks in a busy loop.

Futures and async/await: We have used Rust's async/await syntax where async functions return a Future which is a task that can be scheduled and executed. The async functions are transformed into state machines using a compiler where each .await point represents a potential pause point in a function.

Task wakers: The implementation has introduced a TaskWaker type which uses a queue to notify the executor when a task needs to be resumed. To poll a task at an appropriate time Waker is used to notify.

Asynchronous Keyboard Task: An asynchronous task which listens to the inputs from the keyboard is created. This uses the ArrayQueue (from the crossbeam crate) to store keypress events. To process the input asynchronously the interrupt handler places scancodes in the queue. Then these scancodes are converted into a future that the executor will poll to print using ScancodeStream.

CPU Power Efficiency: Just as we saw before to make the CPU efficient we push it into a sleep state using halt instruction.

Arc-Sharing for Task Queue: To share the tasks between task_queue and executor safely we use Arc(atomic reference-counted). This helps in running the tasks concurrently as tasks are woken and added to the queue.

Basic Scheduling: So this is a form of round-robin scheduling where tasks are polled in a first-in-first-out manner. In this, we don't consider any specific prioritization of tasks but we leave space for future enhancements such as priority scheduling or multi-core support.

4. Challenges

1. New to Rust language: My biggest challenge right now is understanding Rust. Working on this project requires me to read the documentation and watch videos to understand the basics of rust.
2. Handling memory directly with unsafe code, handling paging and managing heap allocation without considering standard libraries added a steep learning curve and debugging the errors took a long time.
3. Implementing CPU exceptions, managing the IDT and ensuring no deadlocks or race conditions happen while interrupting required me to go through a deep understanding of low-level OS concepts.
4. For this project, we could not use Rust testing frameworks because of the no_std environment, which made debugging and testing difficult.
5. Debugging any error in Rust took me longer than expected, I went through several Stackoverflow comments to resolve issues like while async/await implementation my keyboard interrupt printed each text twice. Figuring out where these second prints of characters coming from took me a couple of hours.

5. Accomplishment

1. Learning Rust: This was my first time experience with a system programming language, and I can certainly that I enjoyed working with Rust. This journey has given me a new language to work with and I hope to work with Rust in future on more projects.
2. Learning about OS: I have taken classes in OS but never learned things which I learned doing this project. For example, this project helped me understand the red zone and how that is used in different mode.
3. Learning about low-level architecture: The project helped me to understand how to build a minimal kernel from scratch using Rust, and helped me understand paging, memory allocation, deadlocks, race conditions, keyboard interrupts and many more.
4. Through this project I learned the importance of energy-efficient coding, initially, I was using infinite loop {} for the main function, and then on I learned to halt the CPU to save power.
5. **Working with no_std**: This project introduced me to work without the standard library (no_std), which required a deeper understanding of low-level hardware interactions. Although I previously found hardware and systems challenging, this experience has sparked a genuine interest in learning more.

6. Future Scope

Multi-core support: As of now my OS is running on a single-core processor. The project can be stretched to implement support for multi-core processing.

File System Integration: My current OS, doesn't have any file support. It's an important part of any operating system so stretching the project to have file support is something important for the future.
User Mode and Process Management: My project runs in kernel mode with no distinct user space, Implementation of process management involves creating user-space processes that run with different privileges.
Advanced Memory Management: Advanced Memory Management like swapping, memory-mapped files or shared memory for process needs to be implemented in the OS.

8. RELEVANCE

Rust's Safety and Performance: The reason why Rust is an ideal candidate for OS is its unique combination of memory safety and performance. Unlike other languages C or C++, the Rust model ensures safe memory management. These features of Rust are important in low-level programming where the bugs can cause catastrophic system failure. This approach of Rust makes it fit for OS development making it more secure and reliable while still maintaining high efficiency.

Learning Low-Level System Programming: This project is an excellent way to learn low-level OS and Rust. This helps in understanding how hardware works, managing the memory manually, tackling CPU exceptions, working with interrupts and interacting with the kernel. This will prepare a developer for tackling complex system-level problems in real-world applications.

Practical Application of Rust's no_std: The use of a no_std environment forces us to handle hardware directly instead of relying on Rust's standard library which is a perfect example of software development in constrained environments. This project will teach how to create efficient, reliable software for embedded systems, microcontrollers and an environment where minimum runtime support is required. Overall it encourages exploring system programming potentially making rust a more common option for embedded and bare-metal systems.

Sustainability and Power Efficiency: With time we have seen the importance of energy efficiency particularly in embedded systems, the project works with power-saving techniques like CPU halt instructions which is highly relevant. This way we can learn to work for efficient operating systems development which directly impacts how hardware can be used more sustainably.

7. Conclusion

This project successfully developed a minimal operating system kernel in Rust, it focused on key components such as memory management, interrupt handling and task scheduling. Through this project, we leveraged safety and performance features, at the end of this project we successfully created a bare-metal Rust based on an OS which runs without the underlying OS. The important achievements we got from this project are setting up a custom target for x86_64, making a VGA text buffer for output, handling CPU exceptions, paging and establishing heap allocation using a linked list allocator.

For this project, we adopted iterative development techniques where we developed and tested complex scenarios such as double faults, page faults and hardware interrupts making sure that our kernel remains robust and stable. In addition to this, the implementation of asynchronous tasks and scheduling mechanisms shows an early approach to concurrency and task management.

Through this project, we tried to highlight the need for testing, specifically in a no_std environment where standard Rust testing frameworks are not used. For testing, we relied on custom test infrastructure to run unit and integration tests which presented the reliability of the kernel's core functionalities.

To conclude, the project has given us a valuable foundation for building a minimal and efficient OS in Rust, presenting the potential of a language for low-level system programming.

8. References

- **Philipp Oppermann's Blog:** Writing an OS in Rust
  Oppermann, Philipp. Writing an OS in Rust. Available at: https://os.phil-opp.com/.

- **YouTube Video on Writing an OS in Rust**
  Philipp Oppermann. Writing an OS in Rust [YouTube Series]. Available at: https://www.youtube.com/watch?v=rH5jnbJ3tL4&list=PLib6-zlkjfXkdCjQgrZhmfJOWBk_C2FTY.

- **ChatGPT for Understanding and Debugging**
  OpenAI. ChatGPT Model (version 4), 2025. Available at: https://chat.openai.com.