TiOS - A Modern Kernel

Jianzhong Liu

***Abstract*—The development of modern systems programming languages have provided numerous opportunities at improving the safety and architecture of current operating systems while maintaining the same level of performance. In this project I wrote a modern operating system using Rust to explore what these languages had in store for low level systems programming.**

***Index Terms*—Rust, Operating Systems, kernel.**

I. Introduction

CURRENT mainstream operating systems are mainly programmed using classical systems programming languages such as C. Though these languages offer low level hardware access and maintains portability to some degree, the cost of not having a safe programming paradigm results in the large numbers of bugs and vulnerabilities found in current kernels and system libraries. These insecure sources include the freedom of dereferencing any pointer, a lack of array boundary checks and a lack of initialization checks on variables. In addition, languages like C were not prepared for concurrent programming, thus requiring manual explicit locking to ensure mutual exclusion and prevent race conditions.

The introduction of high level programming languages was intended to relieve the programmer and system architect of having to keep track of low level details and focus on higher level design. Therefore, since languages such as C requires extra human effort in ensuring memory and parallel safety, it is increasing difficult to program modern kernels and other systems in C.

New developments in systems programming languages have provided better ways of finding prone mistakes in operating systems before their distribution. Languages such as Rust and Swift[1] provide modern language features while offering no runtime overhead for memory management[2]. These characteristics make these languages especially Rust increasingly suitable for programming low level systems such as operating system kernels, system libraries and system daemons.

In this project, I made an attempt to program a modern kernel for the x86-64 architecture using Rust to test its potential in ensuring safety while maintaining a comparable level of performance. Though this project is currently unfinished, there are numerous issues to address while using Rust for systems programming.

School of Information Science and Technology, ShanghaiTech University, 201210

[1]Swift systems programming language introduction https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

[2]Swift aims at implementing these features in future releases.

II. Overview of a Modern Kernel

Operating system kernels have come a long way since Ken Thompson and Dennis Ritchie developed the first version of UNIX on a PDP-7 at Bell Labs. The overall landscape of computing as a whole has shifted towards microprocessors and personal computers. CPUs now have hardware protection mechanisms in addition to being exponentially faster and less power consuming. VGA displays have long since replaced the teletype terminals of the past.

It has also been twenty-plus years since the debate of micro-kernels versus monolithic kernels. Monolithic kernels such as Linux have integrated concepts from micro-kernels while the opposite has been true as well.

Here are the important aspects of modern kernels so that much of the issues related with writing a kernel are understood.

A. Memory Management Model

The memory model of x86-64 is mainly a flat memory model, while retaining relics from previous iterations of the x86 line of instruction set architectures. Unlike most architectures, the x86-64 architecture implements both segmentation and paging for protection purposes. There are two address spaces present, one the virtual memory address space and the other physical memory address space. The details are listed below.

1) Virtual Memory Space: The x86-64 uses 64-bit virtual addresses, but current implementations only utilize 48 bits of the address. Canonical addresses are used in the system, requiring sign extensions on the 48th to 63th binary digits. This limits the virtual address space from `0x0000000000000000` to `0x00007fffffffffff` and `0xffff800000000000` to `0xffffffffffffffff`. This reduction of address space hardly affects any usage, since a 48-bit virtual addresse space already spans 256 TiBs of addresses which are already more than sufficient for modern hardware.

2) Physical Memory Space: The x86-64 uses 52-bit physical addresses, though currently no processor supports this amount of memory. Physical memory mapping is provided by the BIOS. Some addresses are reserved for memory mapped devices such as text consoles and I/O.

3) Segmentation: Segmentation on x86-64 is largely defunct therefore presenting a flat memory model, however some aspects are retained, such as a requirement of segment descriptor tables, including global descriptor tables and local descriptor tables. The usage of task state segments also rely on segmentation.

4) Paging: Paging is enabled on x86-64 enabling memory protection but requires many workarounds when modifying the page structures themselves. PAE (Physical Address Extension) must be enabled. Current x86-64 processors uses a 4-level page table structure. The physical address of the level 4 page table (PML4) should be present in the `cr3` register.

B. *Interrupt Driven Model*

Modern architectures are interrupt driven, meaning external or internal events can trigger interrupts, which will interrupt the normal execution flow and execute an interrupt service routine (ISR) and, after processing the interrupt, resume normal execution.

The current implementation of x86-64 processors requires the use of a 256-entry Interrupt Descriptor Table (IDT) that contains function pointers to the corresponding ISRs. Software exceptions are predefined by the architecture and can be found in the systems programming manual provided by Intel[3] or AMD[4], while external interrupts and systems calls should be defined by the operating system.

External interrupts are handled by Programmable Interrupt Controllers (PICs), though current processors use the Advanced Programmable Interrupt Controller (APIC) providing more flexibility and functionality.

C. *Processes and Threads*

Modern kernels have different interpretations and implementations regarding the status of processes and threads. Linux for instance do not differentiate between the two and use a unified task concept. Processes and threads on Linux mainly differentiate by what they share. On Windows NT kernels however, threads and processes are different concepts. This gives us a wide range of options to select from for implementing our process and thread model.

For this project I used the Linux model, which does not differentiate between processes and threads. This allows for page frame sharing between processes, thus lowering the overhead for process creation.

III. The Boot Process

The boot process of an operating system is critical for the systems' performance and stability. On modern operating systems, bootloaders and kernels are separate and are tasked with different things.

For this project, I used GRUB 2, a Multiboot 2[5] compliant bootloader, to boot my kernel for modularity and simplicity. The kernel has to include a multiboot header than can be identified by the bootloader. The bootloader can load the kernel into memory according to the memory sections provided in the executable.

A. *From Bootloader to Kernel*

The kernel is compiled into an ELF-64 format, thus providing a predefined entry point that the bootloader can identify and jump to.

The entry point for the kernel is the `start` label in `boot.asm`, written in x86 assembly under the `arch/amd64/boot` directory in the source files. Upon transferring control to the kernel, the bootloader leaves the boot information section address in the `ebx` register.

We can save the multiboot information address in the `edi` register for later use when we jump to sections written in Rust.

These are the steps we should do to initialize an x86-64 environment for our kernel.

1) Test CPU Features: Currently the CPU characteristics are largely unknown to our kernel. From information provided on OSDev.org forums, we can check if the CPU supports long mode by using some architecture specific instructions. The details can be found in the source code.

2) Setup Rudimentary Page Tables: The x86-64 architecture mandates that paging should be turned on in long mode. Therefore we can reserve several pages in the `.bss` section and temporarily identity map the first physical 1GiB so we can directly address everything as usual.

Modern kernels are mostly higher half kernels, leaving the lower half of the memory space for user space programs while reserving the higher half of the memory for the kernel. Since x86-64 supports addressing 48 bits in the virtual memory space, we can reserve `0xffff800000000000` to `0xffffffffffffffff` for the kernel and leave `0x0000000000000000` to `0x00007fffffffffff` for user space programs. Therefore we will need to map the kernel two times, one is the identity mapping and the other is the higher half mapping. After entering Rust code, we can unmap the lower half of the memory space.

In practice, we can reserve three pages in the `.bss` section, each for the level 4, 3 and 2 page tables. The 0th and the 256th entry on the level 4 page table points to the level 3 page table. The 0th entry on the level 3 page table points to the level 2 page table. On the level 2 page table, we can use huge pages and identity map the lower 1GiB of the physical memory space.

3) Setup Mandatory GDT: As a relic of the x86 architecture, we need to setup a basic GDT to make the jump to long mode. Since segmentation is largely ignored in 64-bit mode, we need two entries, one for the mandatory zero entry, the other an identity mapped segment with all access permissions.

4) Long Jump to 64-bit Code: After setting up all required structures, we still need to conduct a long jump to clear the code segment register of the processor to enter 64-bit mode. For convenience purposes, I used a very short piece of C code to move the stack pointers to the higher half and bootstrap the Rust code.

B. *Kernel Subsystem Initialization*

After entering Rust code, we can utilize all the features provided in the `core` crate, including but not limited to, iterators, closures and integrated traits. The kernel currently has memory management, I/O and interrupt handling functions.

[3]Intel 64 and IA-32 Architectures Developer's Manual https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[4]AMD64 Architecture Programmers Manual Volume 2: System Programming http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf

[5]Multiboot Specifications: https://www.gnu.org/software/grub/manual/multiboot/multiboot.html

1) Memory System Initialization: For a self-sustained kernel, we need an expandable and modifiable page table structure. Therefore we need to create a new page table and remap the kernel solely to the higher half of the address space. However since paging cannot be turned off in 64-bit mode, we need a way of addressing page tables so we can modify their structure. Therefore we need to implement a trick called *recursive mapping*[6], which directs the 511th entry in the level 4 page table to point to itself. This way, we can access the page tables through the addresses given below:

1) Level 4 `0o177777_777_777_777_777_0000`
2) Level 3 `0o177777_777_777_777_XXX_0000`
3) Level 2 `0o177777_777_777_XXX_YYY_0000`
4) Level 1 `0o177777_777_XXX_YYY_ZZZ_0000`

where `XXX` is the level 3 page table index, `YYY` is the level 2 page table index and `ZZZ` is the level 1 page table index.

After remapping the kernel, the kernel has to steal some memory before initializing a dynamically allocated physical frame allocator and virtual page allocator, then initialize a kernel heap and provide a kernel malloc and free function for the `alloc` crate.

For this project I chose to use a buddy allocator[7] like the one in Linux. It provides fast allocation and frees while reasonable amounts of memory overhead.

2) Interrupt Handler Initialization: For the systems to correctly handle interrupts, the interrupt descriptor table has to be initialized before enabling interrupts. For convenience purposes, I chose to use the IDT functionality provided in the `x86_64` crate[8].

Since the prologue and epilogue of the interrupt service routines are largely similar, we can use macros to help us save and restore all required registers. As there are two types of interrupts, one with error codes and one without, therefore we need two macros. This functionality has been integrated into the `x86_64` crate, so we can directly use the macros to generate prologues and epilogues for our ISRs.

Since the IDT must reside in memory for an indefinite amount of time, we would need the lifetime of the IDT to be static. For the IDT to initialize correctly, we would need the help of the `lazy_static` crate which allows for loose static variable initialization.

The x86-64 architecture also supports kernel stack switching via a Task State Segment (TSS) during interrupts so we can prevent kernel stack overflow from triggering a triple fault resulting in a complete system reset. However since the TSS is based on segments, we need to initialize a new GDT to accommodate new entries. This functionality has also been provided by the `x86_64` crate so we may directly use its methods.

After Interrupt handler initialization, the interrupt descriptor table and basic interrupt service routines have been initialized, but requires other subsystems to add in their own service routines.

[6]Recursive Mapping Briefly Explained: http://wiki.osdev.org/Page_Tables#Recursive_mapping

[7]The buddy allocator in Linux is explained here: https://www.kernel.org/doc/gorman/html/understand/understand009.html

[8]Crate information at https://crates.io/crates/x86_64

3) I/O System Initialization: The operating system communicates with external devices via ports as predefined in the architecture manual. The instructions `outb`, `inb` and other similar instructions for different length operands are the main method of communication.

A naive approach would be to write a simple wrapper for these instructions, but opening ports for devices requires authorization and are considered unsafe operations. Therefore we can create a better interface by creating a trait called `InOut` and creating a struct called `Port`. The `InOut` trait defines two unsafe functions `port_in()` and `port_out` and is implemented for types `u8`, `u16` and `u32`. The struct `Port` contains the port number and a `PhantomData<T>` where `T` implements the `InOut` trait. Then we can define the `new()` function for a `Port` to be unsafe, and conduct permission checks before hand.

Thus we have implemented a port interface that prevents sending the wrong type of data to a specific port. Device drivers can be written on top of the port system.

IV. Sections Under Work

Currently this project is still under work and aims to functional in a few months. Currently the kernel is able to boot and initialize and relocate to the higher half of the virtual memory address space. The aforementioned subsystems are also largely functional, being able to initialize interrupts and I/O systems. However the following subsystems and functions should be implemented for this kernel to be fully functional and meaningful.

A. Process Management

As mentioned before, this kernel aims to provide a process and thread model similar to that of Linux, using a unified "task" concept to describe both and differentiating them by the amount they share between other tasks.

B. Process Scheduling

This kernel aims to implement simple multi-queue feedback scheduler for simplicity and educational purposes.

C. File System

This kernel aims at providing FAT and Ext2 file system support for basic file handling and mounting floppy disks and hard drives.

D. User Mode Programs

Currently two user mode programs are under work, one is an init process that takes control from the kernel and initialize the user space, while the second is a basic shell that features basic commands and syntax.

V. Advantages of Using Rust in Systems Programming

Using Rust in systems programming presents many benefits over traditional programming languages such as C and C++ as well as emerging systems programming languages such as Go and Swift.

A. *Safety Assurance in Bare Metal Programming*

The type system and borrow checker in Rust ensures that in safe Rust, mistakes such as memory corruption, wild pointer usage and uninitialized variables that are commonly made in C and C++ can be largely avoided.

The use of unsafe blocks for unsafe operations on low level interfaces also make debugging straightforward, since the safety of many low level operations also depend on external factors and cannot be assessed by the type system and borrow checker in Rust.

B. *Powerful Macros*

Rust macros allows for improved expressiveness of the language as Rust is a strong typed language. They allow for simpler and more readable expressions, for instance variable length function calls can be implemented using Rust macros, while repetitive work can be wrapped in macro invocations.

C. *Foreign Function Interface*

The Rust Foreign Function Interface (FFI) is highly efficient and fully compatible with the C calling convention. Calling into assembly or C routines are straightforward while callbacks are also supported. This lets Rust replace C easily in systems development.

D. *Well Structured Module System*

In comparison with C and C++, the implementation of a module system in Rust has made code management for large project easier,

E. *Tricks to Prevent Problems at Compile Time*

Rust also provides the capability of using its type system and borrow checker to prevent potential runtime problems at compile time, by using a combination of traits and `PhantomData` in structs and other methods.

VI. Pitfalls of Using Rust

Though Rust is a potential replacement for C and other systems programming languages, it has some drawbacks that should be solved.

A. *API Instability*

Rust still has some instable APIs either within its standard or core library or within other necessary crates, that limits portability and backwards compatibility and increases the workload of code maintaining.

B. *Bare Metal Programming Features Experimental*

Bare metal programming features in Rust are still largely experimental, requiring the use of a nightly compiler to compile kernel level executables. Combined with an instable API, this involves more work for the programmer.

VII. Conclusion

Writing TiOS has provided me with experiences in low level systems programming as well as the use of Rust as a systems programming language.

Acknowledgment

Thanks to Philipp Oppermann for his work and tutorial on a basic kernel in Rust *blog-os* and utility crates such as *x86-64*. Website for *blog-os*: https://os.phil-opp.com/.

Also many thanks to Eric Kidd for his work on implementing port controllers in Rust. Project URL: http://www.randomhacks.net/bare-metal-rust/.

In addition, many thanks to the people who worked on the *OSDev.org* wiki. The entries on operating systems development have been very helpful for starters on OS development. Website: http://wiki.osdev.org/

References

[1] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th ed. Beijing, China: Publishing House of Electronics Industry, 2012.9