Sri Lanka Institute of Information Technology

RUST

Name - S.A.D.H. Vishwajith
IT Number - IT19952376

Table of Contents

1. <u>Introduction</u>

Rust is a programming language for open-source systems that focuses on speed, memory protection and parallelism. Rust is being used by developers to build a broad variety of new software applications, such as gaming engines, operating systems, file systems, browser components and virtual reality modelling engines.

The Rust code base is managed by an active group of volunteer coders and continues to incorporate new improvements. The Rust open source project is funded by Mozilla.

Rust was developed from scratch and integrates components from the programming languages of tried-and - true structures and modern programming language architecture. It blends high-level languages' expressive and intuitive syntax with the control and efficiency of a low-level language. It also avoids faults in segmentation and assures thread integrity. This empowers developers to write ambitious, simple, and accurate code.

By mixing power with ergonomics, Rust makes the programming of systems open. Using it, software that is less vulnerable to glitches and security exploits may be created by programmers. It entails powerful features under the hood, such as zero-cost abstractions, stable memory storage, fearless competitiveness, and more. For a list of Rust capabilities, visit the Rust open-source project website. Big and small businesses, including Mozilla, Dropbox, npm, Postmates, Braintree and others, use Rust in manufacturing all over the world.

2. Bare Bones

2.1. A Minimal Rust Kernel

For the x86 architecture, we can build a minimal 64-bit Rust kernel. To produce a bootable disk image, we build on the freestanding Rust binary that prints something on the computer.

- **The Boot Process**
  The firmware code that is stored in the motherboard ROM begins executing when you turn the machine on. This algorithm conducts a power-on auto-test, detects available RAM, and pre-initializes the CPU and hardware. It then searches for a bootable disk and attempts to boot the operating system's kernel.
  Two firmware specifications on x86 are the "Basic Input / Output Architecture" (BIOS) and the newer "Unified Extensible Firmware Interface" (UEFI). The BIOS specification is old and outdated on any x86 computer since the 1980s, but simple and well-supported. On the other hand, UEFI is more modern and has a lot more features, but it is more difficult to set up.

- **A Minimal Kernel**
  Now that we know approximately how a machine boots up, it's time to build a lightweight kernel of our own. Our purpose is to create a disk image to print a "Hello World!" "If booted, to the phone. We depend on the freestanding Rust binary from the previous article for that.

  We designed the freestanding binary through freight, as you might recall, but we needed numerous entry point names and compile flags based on the operating system. This is because the *cargo* is designed by default for the host device, i.e. the device on which you are running. For our kernel, this isn't what we like, since a kernel that runs on top of e.g. Doesn't make any sense about Windows. Instead, for a precisely defined goal framework, we want to compile it.

2.2. The VGA Text Mode

In order to print a character on a screen in VGA text mode, it must be written to the VGA hardware text buffer. A two-dimensional array of usually 25 rows and 80 columns is the VGA text buffer, which is rendered directly to the screen. First write or paste anything here and then click Paraphrase button.

Bit(s)	Value
0-7	ASCII code point
8-11	Foreground color
12-14	Background color
15	Blink

The text mode of the VGA is a convenient way to print text on the screen. In this article, by encapsulating all uncertainties in a separate module, we create an interface that renders its use secure and simple. We are also adding support for formatting macros from Rust.

2.3. Testing

In *no_std* executables, this here discusses unit and integration checking. To perform test functions within our kernel, we can use Rust 's support for custom test frameworks. We can use various features of QEMU and the *bootimage* system to record the outcomes of QEMU.

- ### Testing in RUST

Rust has a built-in test structure that can run unit tests without the need to set up anything. Simply construct a function that tests those outcomes with assertions and apply the # [test] tag to the header of the function. Then the cargo search will locate and perform all of the crate's test functions automatically.

Unfortunately, for no-std apps like our kernel, it's a little more difficult. The issue is that the built-in test library, which relies on the standard library, is indirectly used by Rust's test system. This suggests that we can't use our # [no_std] kernel's default test system.

3. Interrupts

3.1. CPU Exceptions

In different erroneous cases, CPU exceptions exist, such as when reading an invalid memory address or when dividing by zero. We have to set up an interrupt descriptor table that provides handler functions to respond to them. Our kernel will be able to grab breakpoint exceptions at the end of this article and to restore regular execution afterwards.

An exception shows that the present instruction is wrong for something. For starters, if the current instruction attempts to break by 0, the CPU issues an exception. When an error occurs, the Kernel stops its current job and based on the error type, calls a particular exception handling function immediately.

There are approximately 20 independent CPU exception types on x86. The most critical ones are,

- **Page Fault -** A page error happens when accessing unauthorized memory. For instance, if the current instruction attempts to read from an unmapped page, or if it attempts to write from a read-only page.
- **Invalid Opcode -** For example, when we attempt to use newer SSE instructions on an old CPU that does not accept them, this exception happens when the current instruction is invalid.
- **General Protection Fault -** For the broadest spectrum of reasons, this is the exception. Different types of permission breaches exist, such as attempting to execute a privileged user level code instruction or writing reserved fields in configuration registers.
- **Double Fault -** The CPU attempts to call the corresponding controller function anytime an exception happens. The CPU raises a double fault exception when another exception happens when calling the exception handler. This exception often exists where an exception does not have a handler feature registered with it.
- **Triple Fault -** If an error arises when the CPU attempts to call the feature of the double fault handler, a fatal triple fault is given. We can't catch a triple fault or control it. By resetting themselves and rebooting the operating system, most processors react.

3.2. Double Faults

A double fault is a rare exception in simpler language, which happens when the CPU fails to invoke an exception handler. For example, it happens when a page fault is caused, but the Interrupt Descriptor Table (IDT) does not have a page fault handler recorded. So, in programming languages with exceptions, it is kind of equivalent to catch-all blocks, e.g. catch (...) in C++ or catch (Exception e) in Java or C #.

As a regular exception, a double fault act. It has the number 8 vector, and, in the IDT, we may specify a standard handler function for it. It is very necessary to have a double fault handler since a fatal triple fault happens if a double fault is unhandled. It is difficult to capture triple faults and most hardware responds with a device reset.

- **Causes of Double Faults**

  We need to know the precise causes of double faults before we look at these special situations. We used a pretty ambiguous description above:

  A double fault is a rare exception that happens when an exception handler is not invoked by the CPU.

  Exactly what does "fails to evoke" mean? Isn't the handler present? Is the handler been switched out? And what happens if the handler itself triggers exceptions?

  What happens, for instance, if:

I. There is a breakpoint exception, but is the appropriate handler feature switched out?
II. A page flaw exists, nor is the page flaw operator switched out?
III. A breakpoint exception is generated by a divide-by-zero handler, but is the breakpoint handler swapped out?
IV. Does our kernel overflow its stack and get hit by the guard page?

Fortunately, there is an identical description in the AMD64 manual (PDF) (in Section 8.2.9). According to him, when a second exception occurs during the handling of a previous (first) exception handler, a' double fault exception can occur.' "It is important to" can.

3.3. Hardware Interrupts

Interrupts offer a means for connected hardware devices to alert the CPU. That the keyboard should alert the kernel of each keystroke instead of asking the kernel to regularly search the keyboard for new characters (a method called polling). This is even more powerful, since when anything happens, the kernel just has to function. It also makes for quicker response times, as the kernel is able to respond quickly and not only at the next poll. It is not feasible to connect all hardware devices straight to the CPU. Instead, the interrupts are aggregated from both machines by a separate interrupt controller and the CPU is then alerted.

The majority of interrupt controllers are programmable, meaning they accept various interrupt priority levels. For starters, to ensure reliable timekeeping, this makes it easier to assign timer interrupts a higher priority than keyboard interrupts.

Hardware interrupts, unlike exceptions, occur asynchronously. This suggests that they are totally independent of the code executed and can arise at any moment. Thus, in our kernel, we unexpectedly have a form of concurrency with all the possible bugs linked to the concurrency. The strict ownership model of Rust assists us here because it excludes a mutable global state. However, as we can see later in this article, deadlocks are still possible.

- **The 8259 PIC**

  A Programmable Interrupt Controller (PIC) introduced in 1976 is the Intel 8259. It has long been replaced by the newer APIC, but for backward compatibility purposes, its gui is still supported on existing systems. The 8259 PIC is much simpler to set up than the APIC, so before we turn to the APIC in a later post, we can use it to add interrupts.

  For contact with the CPU, the 8259 has 8 interrupt lines and several lines. Two instances of the 8259 PIC, one main and one secondary PIC linked to one of the main interrupt lines, were fitted with the standard systems back then.

4. Memory Management

4.1. Introduction to Paging

This post introduces paging, a very common memory management scheme used for our operating system as well. It explains why memory isolation is needed, how fragmentation works, what virtual memory is, and how paging solves memory fragmentation problems. It also explores the layout of multi-level page tables in X86_64 architecture.

• Memory Protection

One of an operating system's key duties is to separate programs from each other. For one, your web browser should not be able to mess with your text editor. Operating systems employ hardware features to accomplish this purpose, to ensure the memory regions of one process are not accessible by other processes. Depending on the hardware and the implementation of the OS, there are numerous methods.

For instance, certain ARM Cortex-M processors (used for embedded systems) have a Memory Protection Unit (MPU) that allows you to specify a limited number of different access allowed memory regions (e.g. no access, read-only, read-write). The MPU guarantees that the address is in an area with valid access rights for each memory access and throws an exception elsewhere. The operating system will ensure that each process only accesses its own memory by modifying the regions and access permissions on each process switch, thus isolating processes from each other.

• Segmentation

Segmentation, initially to expand the amount of addressable memory, was already implemented in 1978. The case back then was that only 16-bit addresses were used by CPUs, which restricted the amount of addressable memory to 64KiB. Additional section registers, each having an offset address, were added to make more than these 64 KiBs usable. This offset was automatically applied by the CPU for each memory entry, so that up to 1MiB of memory was available.

Depending on the method of memory entry, the section register is selected automatically by the CPU: the code segment CS is used for fetching instructions and the stack segment SS is used for stack operations (push / pop). The data segment DS or the extra segment ES are used for other instructions. Two new section registers were introduced later, FS and GS, which can be freely used.

In the first segmentation version, the segment registers contained the offset explicitly and no access control was done. With the implementation of the secure mode, this was later modified. The segment descriptors include an index into a local or global descriptor table while the CPU operates in this mode, which includes the segment size and access permissions in addition to an offset address. The OS will distinguish processes from each other by loading different global / local descriptor tables for each process that confines memory access to the process's own memory areas.

Segmentation has already employed a strategy that is already found virtually everywhere by changing the memory addresses before the physical access: virtual memory.

- **Virtual Memory**

Abstracting the memory addresses from the underlying physical storage unit is the concept behind virtual memory. A translation stage is done first, rather than simply accessing the storage unit. For segmentation, the translation step is to add the active segment's offset address. Imagine a machine reading a 0x1234000 memory address in a 0x1111000 offset segment: 0x2345000 is the address that is currently accessed.

Addresses before the conversion are called interactive to distinguish the two address types and addresses after the conversion are called actual. One major contrast between these two types of addresses is that physical addresses are identical and often refer to the same, different place of memory. On the other hand, virtual addresses depend on the translation function. It is entirely conceivable that the same physical address is responded to by two separate virtual addresses.

4.2. Heap Allocation

This post provides heap allocation support to our kernel. Second, it introduces dynamic memory and illustrates how typical allocation errors are avoided by the borrowing checker. It then implements Rust 's fundamental allocation interface, builds a region of heap memory, and sets up an allocator crate. At the end of this article, our kernel will be able to control both allocation and array forms of the built-in allocation crate.

- **Local and Static Variables**

In our kernel, we use two kinds of variables: local variables and static variables. On the call stack, local variables are stored and are valid only before the surrounding function returns. At a fixed memory spot, static variables are stored and still live for the program's entire lifespan.

4.3. Allocator Designs

This post explains how to implement heap allocators from scratch. It presents and discusses different allocator designs, including bump allocation, linked list allocation, and fixed-size block allocation. For each of the three designs, we will create a basic implementation that can be used for our kernel.

- **Design Goals**

The duty of an allocator is to control the memory of the heap available. On allow calls, it needs to return free memory and retain track of memory released by deallocate so that it can be reused again. Most notably, since this will trigger undefined actions, memory that is still in use somewhere else must never be passed out.

There are also secondary feature objectives, aside from correctness. The allocator, for example, can use the available memory efficiently and keep fragmentation minimal. In addition, for concurrent implementations, it can perform well and scale to any number of processors. It may also customize the memory configuration with respect to the CPU caches for optimal efficiency to maximize cache location and prevent false sharing.

Such specifications will make good allocators very complex. Jemalloc has over 30,000 lines of code, for instance. In kernel code, where a single error may contribute to major security vulnerabilities, this difficulty is sometimes undesired. Fortunately, it is, compared to userspace code, the allocation patterns in kernel code are often somewhat easier, such that relatively straightforward allocator designs are often necessary.

5. Multitasking

5.1 Async/Await

Multitasking, which is the ability to perform several functions simultaneously, is one of the core characteristics of most operating systems. For starters, when looking at this article, you probably have other programs open, such as a text editor or a terminal window. There are probably different background tasks for managing your desktop windows, checking for updates, or indexing files, even if you just have a single browser window open. Although all tasks appear to be running in parallel, only a single operation can be executed at a time on a CPU core. The operating system easily transitions between active tasks to create the illusion that the tasks run in parallel, so that each one can make a bit of progress. As computers are fast, most of the time, we don't notice these switches. While single-core CPUs can only perform a single task at a time, multi-core CPUs can perform many tasks in a truly parallel manner. A CPU with 8 cores, for example, will run 8 tasks at the same time. In a future post, we will clarify how to configure multi-core CPUs. For this article, for convenience, we will concentrate on single-core CPUs. (It should be remembered that all multi-core CPUs begin with just one active core, so for now we should consider them as single-core CPUs.) There are two types of multitasking: Cooperative multitasking allows tasks to give up CPU power on a regular basis so that other tasks should advance. Preemptive multitasking uses operating system capabilities to transfer threads by forcibly pausing them at random points in time.

6. Conclusion

In this report, we can knowledge about rust language and how to use rust to build OS. And also, can know about rust's secure, reliability, usability. And we can get and in-depth knowledge about how modern language like rust can be used to develop an operating system.