Sri Lanka Institute of Information Technology

BSc (Hons) in Information TechnologySpecializing in

Cyber Security

Year 2, Semester 1

IE2032 – Secure Operating System

RUST Language Assignment

Group Members Details

No	Student ID	Student Name	E-mail	Contact Number
01	IT21195402	Gunathilaka D.J.V	it21195402@my.sliit.lk	+94 714911894
02	IT21172496	Suhaif A.M.	it21172496@my.sliit.lk	+94 763816708
03	IT21163890	W.C.Gihan	it21163890@my.sliit.lk	+94 753146404
04	IT21261978	Sriskandarajah J.P	it21261978@my.sliit.lk	+94 76 1140271

Student ID	Topics
IT21195402	• Testing • Paging Implementation • Async/Await
IT21172496	• VGA Text Mode • CPU Exceptions • Hardware Interrupts
IT21163890	• A Freestanding Rust Binary • A Minimal Rust Kernal • Heap Allocation
IT21261978	• Double Faults • Introduction to Paging • Allocator Designs

Contents

Bare-Metal Rust Binary

Introduction

OS kernel can't depend on another OS or software because when the operating system is executing, no software is already executed. When programmers write the OS, they should use limited libraries to write it, so no predefined libraries are in there. To solve this problem RUST has a feature, can use without using RUST Standard Libraries as instances **Iterators**, **option**, **result** and **ownership system.**
By design, all RUST crates use standard library

Disabling the RUST Standard Library

By design, all RUST crates use the standard library; it's dependent on the OS, as I mentioned, so the programmer needs to stop it. Because when the kernel is running, there is no standard library there. To avoid this problem, programmers should use the ***no_std*** attribute.

This is how it is use

```
#![no_std]
fn main_funtion() {}
```

But the issue is not over there. When we stop using the std-library, it creates another two issues.

The first one is leaving the ***"panic_handler"*** function. Panic implementation should be there to handle panic when it occurs, but it usually comes with the standard library. So, programmers need to define it themselves.

The other issue is leaving the "***eh_personality***" function with std-lib; a programmer can fix it by doing custom implementation. It is a highly unstable implementation. Because of that we keep it as a last resort, "***eh_personality***" is used to implement stack unwinding. After use, this used memory is freed and allows the parent thread to continue its execution. However, unwinding needs some libraries. So, we don't use it for writing an OS.

And we can disable unwinding by adding this code to the "***.toml***" file.

```
[profile.dev]
panic = "abort"
[profile.release]
panic = "abort"
```

And finally, there is one more thing we need, called the "***start attribute***". Execution starts in crt0, which creates the environment for a C application by creating a stack and setting up the proper registers for the arguments. The "***start attribute***" is used to mark the point of the run time in the C-library, and points are given to RUST runtime.

After that runtime, call the main function. Now that we've put our own entry point at the start of the code, this is how it looks.
The Rust compiler is informed that we don't wish to use the standard entry point chain by adding the ***#![no_main]*** attribute. We do not add **main** because it is unnecessary without an underlying runtime.

```
#![no_std]
#![no_main]


Use core::panic::PanicInfo;
#[panic_handler]
Fn panic9_info: &PanicInfo) -> ! {
        Loop { }
}
```

Ower writing **_start** function

We use ***#[no_mangle]*** attribute to make sure Rust compiler output function with the name ***_ start*** by disabling the "name mangling".

```
#[no_mangle]
pub extern "C" fn _start() -> ! {

        loop {}
}
```

Linker Errors

After compiling it there is another problem called ***Linker Error*** . linker is kind of program it generated code into an executable. In this code don't have C runtime , So programmer need to say to linker there is C runtime and there are two ways to do it one is pass the certain set of arguments, and other one is building the bare metal target

Build Bare Metal Target

When the compilation is done the linker assuming an underpinning operating system is there. This is the reason for the link error because these OS usually use C runtime. To avoid this compile without regard to the underlying operating system for a wide variety of environments. To do it run this command on system command prompt/terminal.

```
rustup target add thumbv7em-none-eabihf
```

```
cargo build --target thumbv7em-none-eabihf
```

The final freestanding RUST binary code looks like this:

src/main.rs file

```
#![no_std]  // stop linking the Rust std-lib
#![no_main] // disable all Rust-level entry points

use core::panic::PanicInfo;

#[no_mangle] // stopping the mangle the name of this function
pub extern "C" fn _start() -> ! {
    // named `_start` by default
    loop {}
}

// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

cargo.toml file

```
[package]
name = "name_of_creater"
version = "0.1.0"
authors = ["Name of author <author_mail@sliit.lk>"]

#this the profiles used for `cargo build`
[profile.dev]
panic = "abort" # disable stack unwinding on panics

# the profile used for `cargo build --release`
[profile.release]
panic = "abort" # disable stack unwinding on panics
```

- run this command

```
cargo build --target thumbv7em-none-eabihf
```

- To compile this, you should pass additional arguments like these:

```
# Linux
cargo rustc -- -C link-arg=-nostartfiles
# Windows
cargo rustc -- -C link-args="/ENTRY:_start /SUBSYSTEM:console"
# macOS
cargo rustc -- -C link-args="-e __start -static -nostartfiles"
```

Minimal Rust Kernel

At this point, we're going to make a 64-bit x86 architectural minimal Rust kernel. So, we will see how it does. Before starting, let's talk about what the boot process is. Basically, it is the process of a computer starting up. There are two types of firmware here: the BIOS and the UEFI. The BIOS standard is a bit more outdated but still usable, and UEFI is the modern one with more features.

BIOS Boot

BIOS is a firmware that runs at the start of the computer, looking for the bootable disk and loading it into memory. Usually, the bootloader is identified by the 512-bit executable file that is stored at the beginning of the disk. After it is loaded, it determines the kernel image and adds it to the memory, switches the CPU bits from 16-bit to 64-bit, and then passes certain information from the BIOS to the OS kernel.

Target Specification

Now we going to make kernel, So we need to do some modification on Ower system if you are fallow the reference link you can get more details about this modification [1]
Make JASON file to explain the objective looks like

```
{
   "llvm-target": "x86_64-unknown-none",
   "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
   "arch": "x86_64",
   "target-endian": "little",
   "target-pointer-width": "64",
   "target-c-int-width": "32",
   "os": "none", //it runs on bare metal
   "executables": true, //make code executable
   "linker-flavor": "ld.lld", //ships with RUST the kernel
   "linker": "rust-lld",  //ships with RUST the kernel
   "panic-strategy": "abort", //if keep using the Cargo.toml option, include this option
   "disable-redzone": true, // can handle interrupts signal safely
   "features": "-mmx,-sse,+soft-float"
}
```

Creating the Kernel

Now time to passe JASON file as - - target by typing this command

```
> cargo build --target Name_Of_File_You_Created_JASON_File.json
```

But before we run this, we should add build-std to cargo.toml file like this.

```
# in config.toml

[unstable]
build-std = ["core", "compiler_builtins"]
```

This code can fix the can't find crate for 'core' error, and it tells cargo that it has to recompile the core and compiler_builtins. After doing this, you can pass JASON file and correctly run it.

Memory-Related Intrinsics

Usually, we don't need memory related function for kernel but when add another code to it these functions are usable, usually these functions are enabled by C-Library, but problem is this we can't use C library. So now use "comiler_builtins" to do this.

```
# in cargo.toml

[unstable]
build-std-features = ["compiler-builtins-mem"] //add memory related function
build-std = ["core", "compiler_builtins"]
```

Set a Default Target

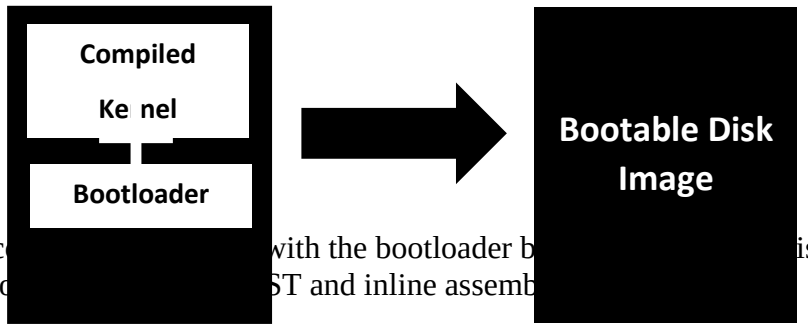
To avoid code more complex we can add cargo configuration file to cargo.toml by doing this , tell cargo to use user created Jason file when -target argument is passed

Printing to Screen

VGA text buffer is a special memory area mapped to VGA hardware that contains 25 lines. More details are provided in upcoming slides.

Running our kernel

Creating a Bootimage



The c with the bootloader b isk image. Using bootloadre crate make own bootlo ST and inline assemb

```
# in Cargo.toml

[dependencies]
bootloader = "0.9.8"
```

We don't need memory-related functions for the kernel, but when we add another piece of code to it, these memory functions become usable. Usually, these functions are enabled by the C-Library, but the problem is that we can't use the C library. So now use "comiler_builtins" to do this.

```
cargo install bootimage  //add bootimage tool to cargo
```

After that can create a bootable **disk-image-compiling** this command.

```
> cargo bootimage
```

Booting it in QEMU

Now our disk image is in boot step so we can execute using the following command.

```
> qemu-system-x86_64 -drive format=raw,file=target/x86_64-blog_os/debug/bootimage-blog_os.bin
```

But be careful to run this because everything on that device is going to be overwritten. It is more secure to install this on a virtual machine and finally we can make it easier to run kernel by adding these codes.

```
# in cargo.toml

[target.'cfg(target_os = "none")']
runner = "bootimage runner"
```

2.Minimal Rust Kernel

At this point, we're going to make a 64-bit x86 architectural minimal Rust kernel. So, we will see how it does. Before starting, let's talk about what the boot process is. Basically, it is the process of a computer starting up. There are two types of firmware here: the BIOS and the UEFI. The BIOS standard is a bit more outdated but still usable, and UEFI is the modern one with more features.

BIOS Boot

BIOS is a firmware that runs at the start of the computer, looking for the bootable disk and loading it into memory. Usually, the bootloader is identified by the 512-bit executable file that is stored at the beginning of the disk. After it is loaded, it determines the kernel image and adds it to the memory, switches the CPU bits from 16-bit to 64-bit, and then passes certain information from the BIOS to the OS kernel.

Target Specification

Now we going to make kernel, So we need to do some modification on Ower system if you are fallow the reference link you can get more details about this modification [1]
Make JASON file to explain the objective looks like

```
{
  "llvm-target": "x86_64-unknown-none",
  "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
  "arch": "x86_64",
  "target-endian": "little",
  "target-pointer-width": "64",
  "target-c-int-width": "32",
  "os": "none", //it runs on bare metal
  "executables": true, //make code executable
  "linker-flavor": "ld.lld", //ships with RUST the kernel
  "linker": "rust-lld",  //ships with RUST the kernel
  "panic-strategy": "abort", //if keep using the Cargo.toml option, include this option
  "disable-redzone": true, // can handle interrupts signal safely
  "features": "-mmx,-sse,+soft-float"
}
```

Creating the Kernel

Now time to passe JASON file as - - target by typing this command

```
> cargo build --target Name_Of_File_You_Created_JASON_File.json
```

But before we run this, we should add build-std to cargo.toml file like this.

```
# in config.toml

[unstable]
build-std = ["core", "compiler_builtins"]
```

This code can fix the can't find crate for 'core' error, and it tells cargo that it has to recompile the core and compiler_builtins. After doing this, you can pass JASON file and correctly run it.

Memory-Related Intrinsics

Usually, we don't need memory related function for kernel but when add another code to it these functions are usable, usually these functions are enabled by C-Library, but problem is this we can't use C library. So now use "comiler_builtins" to do this.

```
# in cargo.toml

[unstable]
build-std-features = ["compiler-builtins-mem"] //add memory related function
build-std = ["core", "compiler_builtins"]
```

Set a Default Target

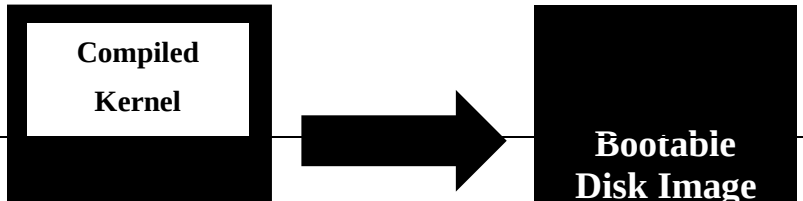
To avoid code more complex we can add cargo configuration file to cargo.toml by doing this , tell cargo to use user created Jason file when -target argument is passed

Printing to Screen

VGA text buffer is a special memory area mapped to VGA hardware that contains 25 lines. More details are provided in upcoming slides.

Running our kernel

Creating a Bootimage



Bootloader

The compiled kernel link with the bootloader becomes a bootable disk image. Using bootloadre crate make own bootloader only using RUST and inline assembly

```
# in Cargo.toml

[dependencies]
bootloader = "0.9.8"
```

We don’t need memory-related functions for the kernel, but when we add another piece of code to it, these memory functions become usable. Usually, these functions are enabled by the C-Library, but the problem is that we can’t use the C library. So now use "comiler_builtins" to do this.

```
cargo install bootimage  //add bootimage tool to cargo
```

After that can create a bootable **disk-image-compiling** this command.

```
> cargo bootimage
```

Booting it in QEMU

Now our disk image is in boot step so we can execute using following command.

```
> qemu-system-x86_64 -drive format=raw,file=target/x86_64-blog_os/debug/bootimage-blog_os.bin
```

But be careful to run this because everything on that device is going to be overwritten. It is more secure to install this on a virtual machine and finally we can make it easier to run kernel by adding these codes.

```
# in cargo.toml

[target.'cfg(target_os = "none")']
runner = "bootimage runner"
```

3.VGA Text Mode

IBM launched VGA text mode in 1987 as part of the VGA (Video Graphic Array) standard for its IBM Second generation personal Computers. And In 1990 it was started to spread widely as well as now in the modern computers also it is used in some applications. The Important features in the VGA text mode are colored (programmable 16 color palette) it means set of available colors from which an image can be made. Not only that the background, blinking, different shapes of the cursor and fonts. Actually, it is a way to print text to the screen.

VGA Text buffer

Every screen character is described by two bytes with 16-bit word accessible for a single work. If a person writes something in the text buffer on VGA hardware, then only it will be print characters. Two-dimensional array with 25 rows and 80 columns. All the array entries describe a single screen character.

1st byte shows character it will be printed in ASCII encoding. But not exactly but with some additional characters.2nd byte represent how the character is displayed. Foreground color displays with first 4 bits. And Background color represented by next three bits. Last bit defines blink details of the characters.

Description of the arrays.

```
// in src/main.rs
mod vga_buffer;
```

During reading and writing it is not accessing the RAM however from the i/o it will access the buffer which is in the VGA hardware. And not possible at all RAM operations.

Rust module

Rust module used to handle printing.

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

Colors

It is specifying the number for each color. And the color code Struct contain full color byte foreground color background color.

Number	Color	Number + Bright Bit	Bright Color
0x0	Black	0x8	Dark Gray
0x1	Blue	0x9	Light Blue
0x2	Green	0xa	Light Green
0x3	Cyan	0xb	Light Cyan
0x4	Red	0xc	Light Red
0x5	Magenta	0xd	Pink
0x6	Brown	0xe	Yellow
0x7	Light Gray	0xf	White

```rust
// in src/vga_buffer.rs

#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
pub enum Color {
    Black = 0,
    Blue = 1,
    Green = 2,
    Cyan = 3,
    Red = 4,
    Magenta = 5,
    Brown = 6,
    LightGray = 7,
    DarkGray = 8,
    LightBlue = 9,
    LightGreen = 10,
    LightCyan = 11,
    LightRed = 12,
    Pink = 13,
    Yellow = 14,
    White = 15,
}
```

Representing the color code

Text Buffer

Text buffer used as a state machine to set foreground color, background color and shakiness of the cursor as well as we can clear the screen by moving the cursor and be able to write with the cursor. Although it is only efficient method when we have a structured way of display things. However, For the simple writings we can use parser. In the 0.2.0 version it gives the chance to draw in multiple text buffers as on top of each other. Not only that it is allowed to write 16-bit characters also.

How Text buffer Works

```rust
// in src/vga_buffer.rs

impl Writer {
    pub fn write_byte(&mut self, byte: u8) {
        match byte {
            b'\n' => self.new_line(),
            byte => {
                if self.column_position >= BUFFER_WIDTH {
                    self.new_line();
                }

                let row = BUFFER_HEIGHT - 1;
                let col = self.column_position;

                let color_code = self.color_code;
                self.buffer.chars[row][col] = ScreenChar {
                    ascii_character: byte,
                    color_code,
                };
                self.column_position += 1;
            }
        }
    }
```

Printing

In this section we can get start to print, if the byte is new line byte writer don't print anything. Otherwise, if it calls a newline method then it will be implemented on later. However, except that byte's others will be printed to the screen. As well as to print all the strings we can transmit them into bytes and can get the output one by one. By creating functions can write some characters to the screen.

```rust
// in src/vga_buffer.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(C)]
struct ScreenChar {
    ascii_character: u8,
    color_code: ColorCode,
}

const BUFFER_HEIGHT: usize = 25;
const BUFFER_WIDTH: usize = 80;

#[repr(transparent)]
struct Buffer {
    chars: [[ScreenChar; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

Volatile

Actually, it is allowed to creating read only and write only Volatile values. It provides the wrapper type Volatile, which covers a reference to any copy-able type and allows for volatile access to wrapped value.

```
// in src/vga_buffer.rs

use volatile::Volatile;

struct Buffer {
    chars: [[Volatile<ScreenChar>; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

Spinlocks

Spin locks can be used to get synchronized mutability. As well as it gives mutual exclusion by blocking threads when the resource is already locked.

```
# in Cargo.toml
[dependencies]
spin = "0.5.2"
```

4.Testing

I. Testing in Rust

The built-in test infrastructure in Rust allows for the immediate execution of unit tests. Simply build a function that utilizes assertions to validate some results, then include the #[test] property in the function header. Then, all of your crate's test operations will be found and executed right away by cargo test.

A little more intricacy than our kernel is not needed by any of the common applications, though. The built-in test library, which is reliant on the standard library, is used by default by the test framework in Rust, which is problematic. As a result, our #[no std] kernels cannot be tested on the standard test platform.

See the following when trying to do a cargo test for the task:

```
> cargo test
   Compiling blog_os v0.1.0 (/.../blog_os)
error[E0463]: can't find crate for `test`
```

Due to its reliance on the standard library for our bare metal target, the test crate is not usable. It is feasible to move the test crate to a #[no std] context, but doing so requires multiple hacks, including rewriting the panic macro, and is quite unstable.

- **Custom Test Frameworks**

The built-in test framework can be changed thanks to Rust's support for unreliable custom test frameworks. This capability may be used in #[no std] circumstances because it doesn't require any external libraries. Every function with a #[test case] property is gathered, and after that, a user-specified runner function is called with the list of tests as input. As a result, the testing procedure is entirely under the implementation's control.

When compared to the default test framework, a disadvantage is the absence of several complex features, such as should panic tests. Instead, the implementation must take care of this on its own, if necessary. This is ideal for us because such sophisticated capabilities' default implementations would most likely not work. This is ideal for us since, given the execution environment that we employ, the default implementations of such complicated capabilities would undoubtedly not work.

For example, the #[should panic] attribute uses stack unwinding to identify panics, which the kernel prevents.

Including the following code in our main.rs will enable us to develop a unique test framework for our kernel:

```
// in src/main.rs

#![feature(custom_test_frameworks)]
#![test_runner(crate::test_runner)]

#[cfg(test)]
fn test_runner(tests: &[&dyn Fn()]) {
    println!("Running {} tests", tests.len());
    for test in tests {
        test();
    }
}
```

Before calling each of the test functions, the kernel first delivers a brief debug message. The Fn() trait's object references are categorized by the parameter type &[&dyn Fn()]. Essentially, it is a collection of pointers to types that may be called in a similar way to functions. Use the #[cfg(test)] property to only include the procedure in testing runs; otherwise, it is pointless.

Make that the cargo test has been successful. Get the test runner's "Hello World" greeting replaced with ours. This is true since we still utilize an entry point in our _start function. The #[no main] parameter is used by the custom test frameworks feature to create the main function that launches the test runner and provides the entry point for it.

To fix this, you must first change the generated function's name from main using the reexport test harness main attribute. The renamed function may then be called using our _start function:

```
// in src/main.rs

#![reexport_test_harness_main = "test_main"]

#[no_mangle]
pub extern "C" fn _start() -> ! {
    println!("Hello World{}", "!");

    #[cfg(test)]
    test_main();

    loop {}
}
```

From our _start entry point, call test main, the test framework's main entry method. Only include the call to test main in test contexts using conditional compilation as the function is not generated on a regular run.

The message "Executing 0 tests" will appear on the screen when the cargo test has been completed running. We are now ready to write our first test function:

```
// in src/main.rs

#[test_case]
fn trivial_assertion() {
    print!("trivial assertion... ");
    assert_eq!(1, 1);
    println!("[ok]");
}
```

At this time, the results of the cargo test are as follows:

```
Hello World!
Running 1 tests
trivial assertion... [ok]
```

The test slice that was sent to our test runner method now has a reference to the minimal assertion function. The trivial assertion... [ok] presented on the screen indicates that the test was called and passed.

After finishing the tests, our test runner goes back to the test main method, which then goes back to our _start entry point function. At the end of _start, start an endless loop because the entry point function is unable to return. The cargo test should stop when all tests have been completed, thus this is a problem.

Testing the VGA Buffer

Under the test framework, in a simple test PrintIn works without any error.

```
// in src/vga_buffer.rs

#[test_case]
fn test_println_simple() {
    println!("test_println_simple output");
}
```

Taking care of VGA buffer facts dealing with PrintIn, not to worries working in order.
The way can be satisfied is there will be no shift of the type lines are in the correct order or no changes.

```rust
// in src/vga_buffer.rs

#[test_case]
fn test_println_many() {
    for _ in 0..200 {
        println!("test_println_many output");
    }
}
```

```rust
// in src/vga_buffer.rs

#[test_case]
fn test_println_output() {
    let s = "Some test string that fits on a single line";
    println!("{}", s);
    for (i, c) in s.chars().enumerate() {
        let screen_char = WRITER.lock().buffer.chars[BUFFER_HEIGHT - 2][i].read();
        assert_eq!(char::from(screen_char.ascii_character), c);
    }
}
```

5.CPU Exception

CPU Exception can happen in numerous situations. As a example if when we access a invalid memory address or the dividing occurs by zero. We can identify that by setting an interrupt descriptor table. Which contains a handler functions. Another way we ca say that if something happened wrong there will be an exception signal.

We can classify the CPU exceptions into 20 types. How there are five main exceptions are mentioned as following,

Invalid Opcode

These types of exceptions are happening during when the current command is in invalid condition. As a example before enabling the SSE if we tried to use SSE instructions that period the CPU can't understand the movups and movaps instructions. During that time there will be an exception.

Page Fault

Page faults occurs during an illegal memory access happened. If the running command tries to read from an unmapped page or tries to write to a read only page.

General Protection fault

This is the rare case with the most diverse set of reasons. It happens as a result of several types of access breaches, such as attempting to launch a privileged instruction in user program or altering allocated fields in configuration registers.

Double fault

If an exception happened the CPU expect to call the appropriate handler function. Meanwhile if another exception happened during calling the exception handler CPU display a double fault exception as well as this will happen When no handler function has been registered for an exception.

Triple Fault

If an error displayed during a fatal triple fault is generated by the CPU if an error occurs while it is attempting to call the double fault handler code. A triple error escapes our grasp or control. When this happens, the majority of CPUs reset themselves and restart the operating system.

The interrupt Descriptor Table

The table is used for identifying the errors. All the CPU errors are specified by a function and the hardware utilize the table directly. Every input should be entered by 16-byte structure.

Type	Name	Description
u16	Function Pointer [0:15]	The lower bits of the pointer to the handler function.
u16	GDT selector	Selector of a code segment in the global descriptor table.
u16	Options	(see below)
u16	Function Pointer [16:31]	The middle bits of the pointer to the handler function.
u32	Function Pointer [32:63]	The remaining bits of the pointer to the handler function.
u32	Reserved	

Bits	Name	Description
0-2	Interrupt Stack Table Index	0: Don't switch stacks, 1-7: Switch to the n-th stack in the Interrupt Stack Table when this handler is called.
3-7	Reserved	
8	0: Interrupt Gate, 1: Trap Gate	If this bit is 0, interrupts are disabled when this handler is called.
9-11	must be one	
12	must be zero	
13-14	Descriptor Privilege Level (DPL)	The minimal privilege level required for calling this handler.
15	Present	

Format of the fields

If an error happened CPU will do the following things,

1. The command pointer and the RFLAGS register are among the registers that are pushed into the stack.
2. Read the Interrupt Descriptor Table entry that corresponds (IDT). For instance, when a page failure happens, the CPU reads the 14th item.

3. Verify the existence of the entry. If not, raise a double fault.
4. Block the interrupts if the input is an interrupt gate (bit 40 not set)
5. Fill the CS segment with the supplied GDT selection.
6. Navigate to the chosen handler function.

The interrupt calling Convention

Calling conventions specifics of a function call. For instance, they define how results are returned and where function parameters are put (such as in registers or on the stack). There are some rules used for c functions,

1.first six integer arguments are passes in registers. (rdi, rsi, rdx, rcx, r8, r9)

2.other arguments are passed on the stack.

3. results are shown as rax and rdx

Preserved and Scratch Registers

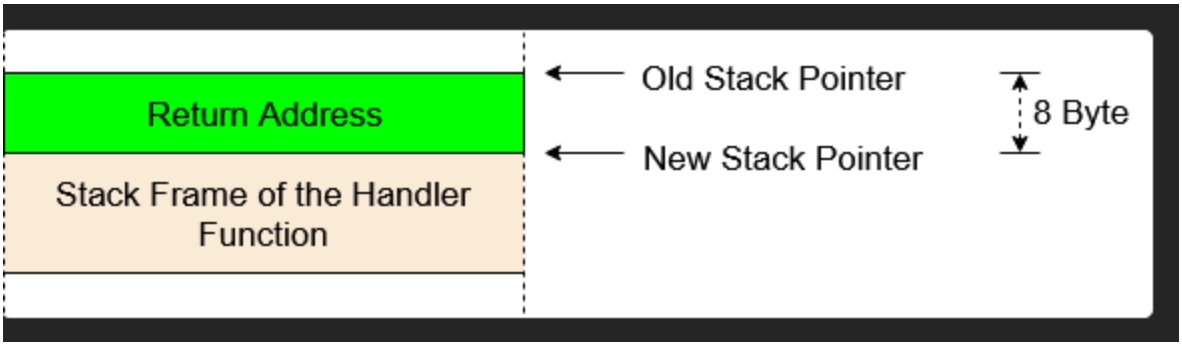
Calling convention can be divided as preserved registers and Scratch registers. In here the values of the registers cannot be change during function calls. So that the change can be made only by the called function as well as the value will be restoring before returning the original value.

X86_64 C_calling conventions

<u>The interrupt Stack Frame</u>

Normal function call performs to push the address by CPU before moving to the exact function. If the function returns any value by the ret instruction the CPU pops and return the current address, then move to it.

preserved registers	scratch registers
rbp , rbx , rsp , r12 , r13 , r14 , r15	rax , rcx , rdx , rsi , rdi , r8 , r9 , r10 , r11
callee-saved	*caller-saved*



Stack Frame of a function

If any interrupt happened then CPU performs some steps as follows,

1.Aligning the stack pointer- pointer will be aligned on a 16-byte boundary

2.Switching Stacks-used for special interrupts with the so-called interrupt table

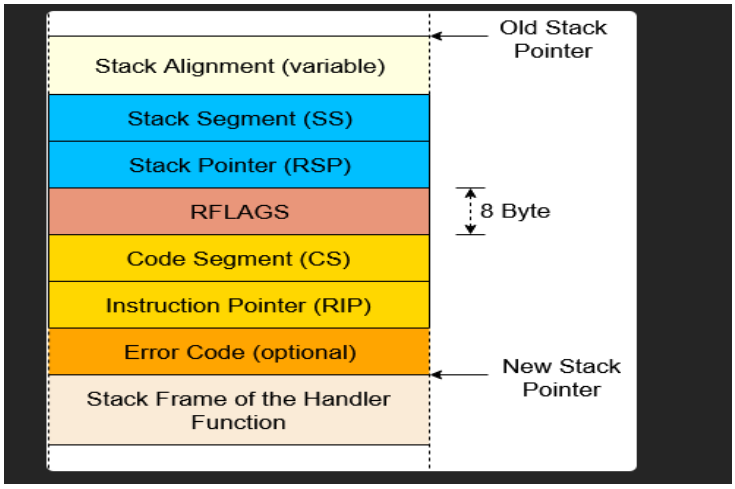
3.Pushing the old stack pointer-Get the chance to restore the exact stack pointer

4.Pushing and updating the RFLAGS register-When the interrupt starts CPU changes some bits and push the older value.

5.Pushing the Instruction pointer-CPU send the instruction pointer and codes. And it can be comparable for the return address push of a normal function call.

6.Pushing an error code-Sometimes when CPU compiles an error code then it will describe as cause of the exception.

7.Invoking the interrupt handler-Load the values into the rip and cs registers.



Implementation

```
// in src/lib.rs

pub mod interrupts;

// in src/interrupts.rs

use x86_64::structures::idt::InterruptDescriptorTable;

pub fn init_idt() {
    let mut idt = InterruptDescriptorTable::new();
}
```

Creating a new interrupt model.

6.Double Faults

When the CPU neglects to contact an exception handler, a specific exception called a double fault occurs. A double fault behaves in the same way as any other exception. Its vector number

is 8, and the IDT (Interrupt Descriptor Table) allows us to construct a standard handler function for it. A double fault handler is essential because, if ignored, a double fault might develop into a fatal triple fault. Since triple failures are impossible to identify, most hardware resets the system as a response..

- Triggering a Double Fault

When we use unsafe to write an invalid address, here the virtual address is not mapped to the physical address so, the page fault will occur. And if we have not registered a page fault handler in Interrupt Descriptor Table, so a double fault will occur.

A Double Fault Handler

To prevent the triple fault, we can use the double fault handler

Steps:

- ✓ The CPU writes to the IP address, so a page fault will occur
- ✓ If there is no handler code declared, the CPU will look at the item in the Interrupt Descriptor Table and cause a double fault.
- ✓ The CPU then switches to the handler for double faults.

Switching Stacks

In the event of an exception, the x86 64 architecture can move to a predetermined, known-good stack. The CPU does not need to push the exception stack frame since this transition occurs at the hardware level.

- The IST and TSS

The Task State Segment, an outdated legacy structure, contains the Interrupt Stack Table (IST) (TSS). The TSS was used, for instance, for hardware context switching, to store different bits of information (such as the status of the processor registers) about a job in 32-bit mode. However, hardware context switching is no longer allowed in 64-bit mode, and the TSS format has changed.

- Creating a TSS

The const evaluator in Rust is not yet capable of performing this at build time. The top address of a single fault stack is then written to the 0th entry after defining that the IST entry at position 0 is the double fault stack (any other IST index would also work). This is because, on x86, stacks expand from high to low addresses, or downwards. Because mutable statics are accessed, the compiler is unable to ensure race freedom, necessitating the usage of the unsafe.

- The Global Descriptor Table

Before paging became the de facto norm, the Global Descriptor Table (GDT) was utilized for memory segmentation. In 64-bit mode, segmentation is no longer supported, but the GDT is still there. The two main functions it performs are loading a TSS structure and switching

between kernel space and user space.

- The Final Steps
  The issue is that the GDT segments are not yet active since the previous GDT values are still present in the segment and TSS registers. Additionally, we must change the double fault IDT entry so that it makes advantage of the new stack.

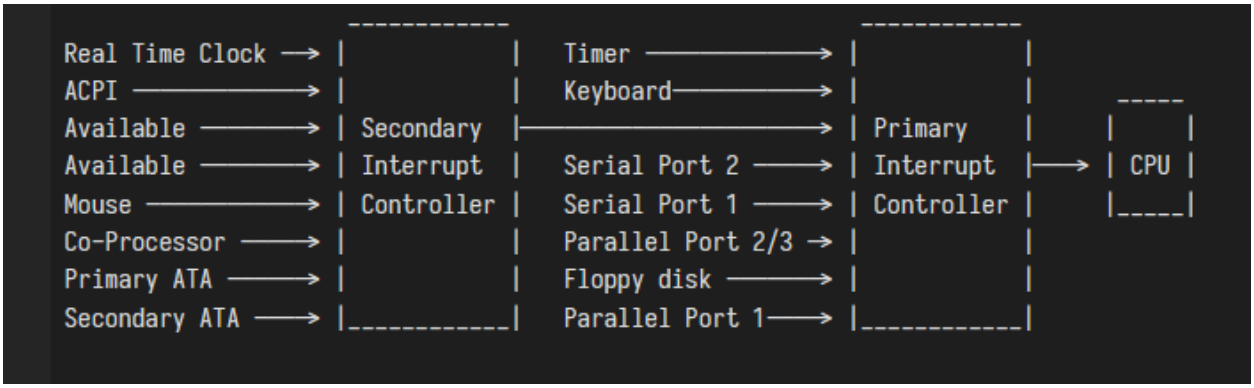
  Steps
  - ✓ We need to reload cs, the code segment register because our GDT has changed. As a result, this is necessary since the previous segment selector might now correspond to a new GDT descriptor.
  - ✓ Although a GDT with a TSS selection was loaded, we still need to instruct the CPU to utilize that TSS.
  - ✓ The CPU immediately gets access to a functioning interrupt stack table after loading our TSS (IST). Next, by making changes to our double-fault IDT entry, we can instruct the CPU to use our new double-fault stack.

7.Hardware Interrupts

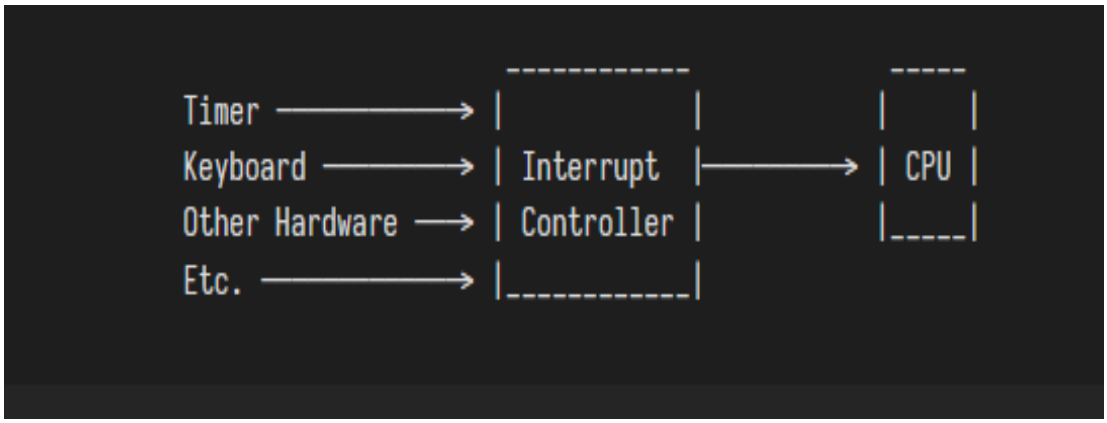
Hardware interrupts gives a chance to notify the CPU from attached hardware devices. When a processor stops and saves the current process it executes a function called Interrupt Service Routine this is used to deal with Interrupts. This function is only for a moment and then when the interrupt handler finishes, or during are there any occurs displayed then the processor will work on normal activities.

We can't connect all hardware devices directly to CPU. So, a special interrupt controller aggregates the interrupts from all devices and then informs to CPU. Devices convey their need for attention through hardware interrupts, or in the absence of an operating system, a bare-metal program executing on the CPU. Not only that these interrupt controllers are programmable, and they allow to achieve precise timekeeping, give timer interrupts a higher priority than keyboard interrupts. Hardware interrupts can be happened asynchronously. It means they are totally independent from the executed code and at all they can occur. Now in

this technological world there is another way to conduct Interrupts which is Message Signaled Interrupts (MSI).



8259 PIC



This intel type programmable interrupt controller invoked in 1976. As well as still it is using in backwards, according to the (APIC) it is more than easier to use. And it includes eight interrupt lines with some other lines.

All the controllers can be verified by the two input/output ports ’one command port and one data port’. Actually, for primary controller it has 0x20 command ports and for data 0x21 ports. for the secondary controller 0xa0 command ports and 0x1 data ports.

Implementation

The usual configuration is not possible the reason is the sending range of the interrupt vector

numbers 0-15 to CPU. Already these numbers are inside the CPU

```rust
// in src/interrupts.rs

use pic8259::ChainedPics;
use spin;

pub const PIC_1_OFFSET: u8 = 32;
pub const PIC_2_OFFSET: u8 = PIC_1_OFFSET + 8;

pub static PICS: spin::Mutex<ChainedPics> =
    spin::Mutex::new(unsafe { ChainedPics::new(PIC_1_OFFSET, PIC_2_OFFSET) });
```

exceptions. How ever the PICs configuration can be happened by writing special values to their command and data ports. But already there is crate pic as 8259 so no need to write the initialization part.

Crate is used for representing the primary/secondary pic layout of ChainedPics struct. The offsets can be set from 32-47 of the PICs range. Safe mutable access can be delivered by using chainedpics struct in a mutex. However chainedPics:new function also not a suitable because it can be deliver undefined function when the PIC is configured in wrong manner.

Enabling Interrupts

x86_64

double

been

```
# in Cargo.toml

[dependencies]
pic8259 = "0.10.1"
```

crate function enables the external interrupts by executing the special Sti instructions. How ever there will be a fault shown because automatically the hardware timer of the intel (8253) has enabled. So that we get the timer notifications quickly when we enable the interrupts.

```rust
// in src/lib.rs

pub fn init() {
    gdt::init();
    interrupts::init_idt();
    unsafe { interrupts::PICS.lock().initialize() };
    x86_64::instructions::interrupts::enable();     // new
}
```

Handling the timer interrupt

Primary PICs line number 0 is used by the timer. It is indicating that arrives to the CPU as an interrupt at 32(0+offset32). Without hardcoding it we can store in a InterruptIndex enum. Although we can specify the index as a variant. All the variants can be sown as a (u8) as well as it is specified by the repr(u8) attribute.

```rust
// in src/interrupts.rs

#[derive(Debug, Clone, Copy)]
#[repr(u8)]
pub enum InterruptIndex {
    Timer = PIC_1_OFFSET,
}

impl InterruptIndex {
    fn as_u8(self) -> u8 {
        self as u8
    }

    fn as_usize(self) -> usize {
        usize::from(self.as_u8())
    }
}
```

End of Interrupt

From the interrupt handler there will be a signal expected by the PIC as end of interrupt. Then the signal shows that the controller which interrupt was happened and now the system is ready to get another interrupt. Because of that Pic thinks that still it is in a busy processing with first timer interrupt so that it will be wait until the EOI signal before sending another one.

Deadlocks

Deadlocks happen during when a thread tries to obtain a lock that can never be released, a deadlock results. Consequently, the thread continues to hang. Using a spinlock, we can provoke a deadlock in the kernel.

```
// in src/vga_buffer.rs

[...]

#[doc(hidden)]
pub fn _print(args: fmt::Arguments) {
    use core::fmt::Write;
    WRITER.lock().write_fmt(args).unwrap();
}
```

```
extern "x86-interrupt" fn timer_interrupt_handler(
    _stack_frame: InterruptStackFrame)
{
    print!(".");

    unsafe {
        PICS.lock()
            .notify_end_of_interrupt(InterruptIndex::Timer.as_u8());
    }
}
```

Timestep	_start	interrupt_handler
0	calls `println!`	
1	`print` locks `WRITER`	
2		**interrupt occurs**, handler begins to run
3		calls `println!`
4		`print` tries to lock `WRITER` (already locked)
5		`print` tries to lock `WRITER` (already locked)
...		...
never	*unlock* `WRITER`	

How Interrupts happen

Fixing a Race condition

Race condition can be happened because of timer interrupt rotating inside the println and the reading of the screen characters. However, in this situation it is not a big issue because during the compile time rust prevent.

```
// in src/vga_buffer.rs

#[test_case]
fn test_println_output() {
    use core::fmt::Write;
    use x86_64::instructions::interrupts;

    let s = "Some test string that fits on a single line";
    interrupts::without_interrupts(|| {
        let mut writer = WRITER.lock();
        writeln!(writer, "\n{}", s).expect("writeln failed");
        for (i, c) in s.chars().enumerate() {
            let screen_char = writer.buffer.chars[BUFFER_HEIGHT - 2][i].read();
            assert_eq!(char::from(screen_char.ascii_character), c);
        }
    });
}
```

The hlt Instruction

Due to some functions CPU will be run in a infinite way. So that always the CPU run in full speed it will not consider about the works. Because of that we need to stop the CPU until the next interrupt arrives. By the hlt instruction we can move the CPU to a Sleep mode it saves the energy.

8.Introduction to Paging

Memory Protection

Operating systems achieve this goal by ensuring that memory portions of one process are inaccessible to other processes using hardware capabilities. Various methods exist based on the hardware and OS configuration.
Example for memory region with access permissions: read-only, read-write, no access

Segmentation

To expand the amount of accessible memory, segmentation was first proposed in 1978. Additional segment registers were created, each of which had an offset address, to make memory larger than this 64 KB Accessible Up to 1 MB of memory was available since the CPU automatically inserted this offset on each memory access.

- ✓ For fetching instruction: CS (Code Segment)
- ✓ For Stack operation: SS (Stack Segment)
- ✓ For other instruction: DS (Data Segment), ES (Extra Segment)
- ✓ Other Segments: FS, GS

```
// in src/lib.rs

pub fn hlt_loop() -> ! {
    loop {
        x86_64::instructions::hlt();
    }
}
```

- Virtual Memory

The idea behind virtual memory

is to separate the memory locations from the underlying physical storage device. The storage device is not accessed directly; rather, a translation step is carried out first. The translation stage in segmentation entails adding the offset address of the active segment.
There are 2 types of addresses

1. Virtual- Addresses before translation
2. Physical- Addresses after translation

The physical address is unique and refers same distinct and the Virtual address depends on the translation function

- Fragmentation

When a process is loaded and unloaded from memory repeatedly, the free memory space becomes fragmented, which is an undesirable OS issue known as fragmentation. Due to their small size, the memory blocks cannot be assigned to the processes. As a result, the memory blocks are never used.

Paging

The plan is to create small, fixed-size blocks in both the physical and virtual memory spaces. While the blocks of the virtual memory space are referred to as pages, the blocks of the physical address space are known as frames. It is feasible to divide bigger memory areas across non-continuous physical frames since each page may be independently mapped to a frame.

- Hidden fragmentation
  Few large and variable-sized regions are used by Segment, but many small and fixed-sized regions are used by Paging. So, no fragmentation occurs in Paging. But some hidden fragmentations are there
  1. Internal Fragmentation- occurs due to some part of memory being left unused
  2. External Fragmentation- occurs when total space is available to process but still not able to add that process in the memory because that space is not contiguous.

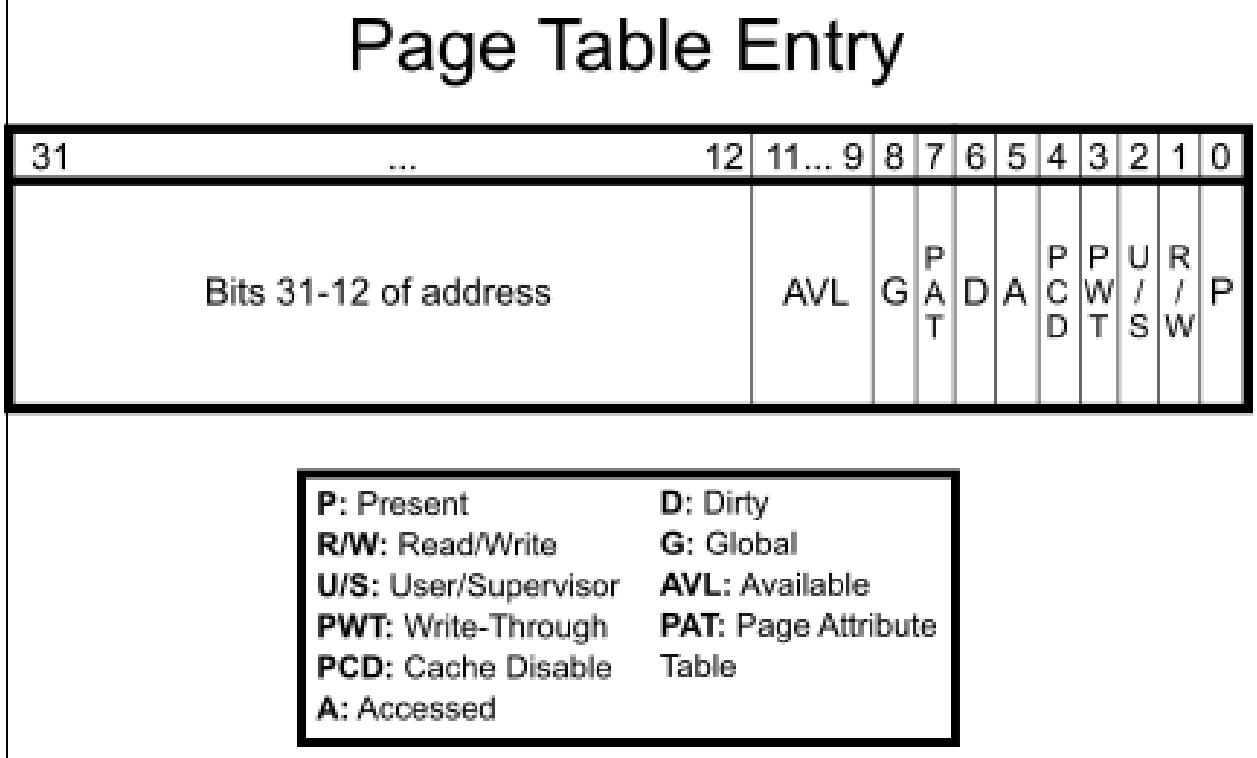
- Page Tables
  A page table is the data structure offered by the virtual memory architecture of an operating system for a computer to keep track of the mapping between virtual addresses and physical addresses.
- Multilevel Page Tables
  We can utilize a two-level page table to minimize wasted Memory. For various address areas, the aim is to employ various page tables. The mapping between address regions and (level 1) page tables is stored in a separate table called the level 2 page table.

Three, four, or more levels can be added to the two-level page table approach. Following that, the page table registration points to the highest-level table, which points to the next lower-level table, and so forth.

Paging on x86_64

A 4-level page table and a 4 KiB page size are used in the x86 64 architecture. There are 512 items in each page table, regardless of the level. Each table is precisely one page long because each item is 8 bytes in size, making each table 512 * 8 B = 4 KiB large.

- Page Table Format



- The Translation Lookaside Buffer (TLB)

The page table entries are tracked in a high-speed cache called Translation Lookaside Buffer. The page number is utilized as an index when processing a page table if an entry is missing from the TLB. The CPU checks the TLB after receiving a virtual address to determine if the requested page is already present in the main memory. The frame number is obtained, and the address is created when a page entry is reached.

Implementation

By causing a page fault exception instead of writing to arbitrary physical memory whenever memory access is outside of boundaries, paging makes the kernel already quite secure. Even better, only pages containing code are executable, and only data pages are readable because the bootloader sets the appropriate access rights for each page.

- Page faults

When a program tries to use memory that is a part of its working set but can't locate it, a page fault is produced.

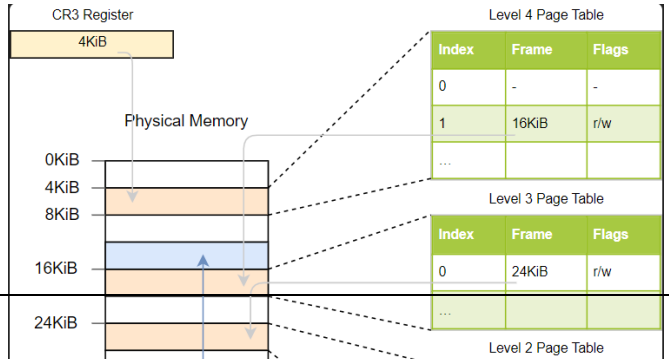
- Accessing the page tables
  When paging is enabled, it is not feasible to access physical memory directly since doing so would allow applications to easily get around memory protection and access the memory of other programs. Therefore, using a virtual page that is linked to the physical location is the only option to access the table. Since the kernel often accesses the page tables, the difficulty of establishing mappings for page table frames is a general issue.

9.Paging Implementation

I. Introduction

This note explores how the kernel stores page tables in physical memory, the inability of the kernel to access page tables already running on virtual addresses, and the different approaches the kernel has to access page table frames.

II. Accessing Page Tables

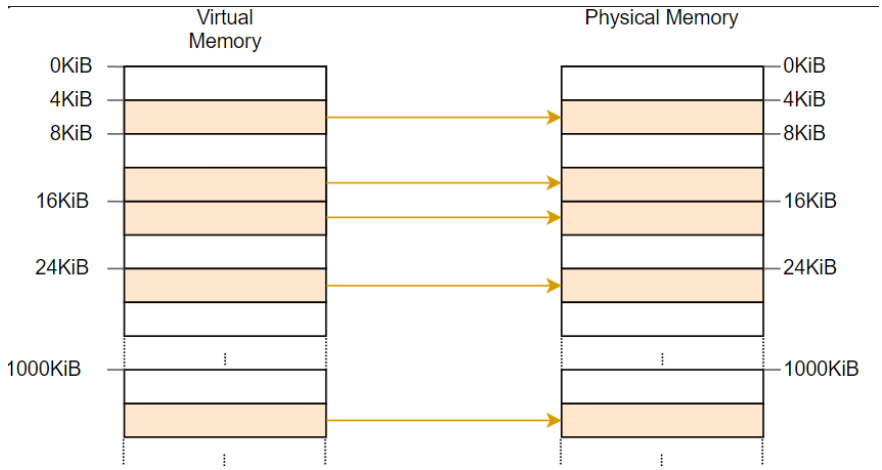


It's important to point out that each page item includes the precise location of the next table. It is not essential to undertake translations for these sites as well as a result of difficulties and the possibility of endless translation cycles.

Our kernel can use virtual addresses to operate, but it can't directly access physical addresses, which is the problem. For instance, instead of accessing the physical address 4 KiB, use the virtual address 4 KiB to go to the level 4 page table. It can only be accessed using a virtual address with a physical address of 4 KiB.

To access them, some virtual pages must be mapped to page table frames. These mappings can be created in many ways and allow access to any page table frame.

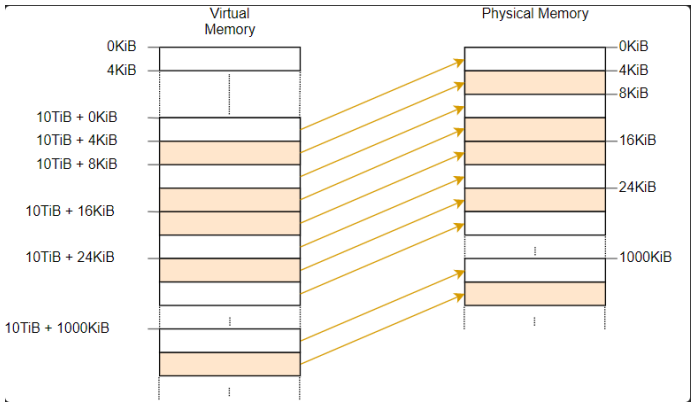
- **Identify Mapping**



This example displays many identity-mapped page tables, all of which are accessible at any level, beginning with the CR3 register, and whose physical addresses are both unique and valid unique addresses..

- **Map the Complete Physical Memory**

Using this method, the kernel may access page table frames from different address spaces as well as any physical memory. Except for the absence of unmapped pages, the size of the reserved virtual memory region remains the same.
This method requires additional page tables to store the physical memory mapping. On systems with minimal memory, the necessity of using physical memory to hold these page tables might lead to issues.



On x86 64, the mapping may be done with large sites that are two MiB in size rather than the typical 4 KiB pages. Only one level 3 table and 32 level 2 tables are required to map 32 GiB of physical memory, resulting in 132 KiB of page tables. In addition, bigger pages take up less space in the translation lookaside buffer, enhancing cache efficiency (TLB)

III. Bootloader Support

The bootloader has access to the page tables and may construct any required mappings as a result. The following two techniques, which are controlled by cargo features, are now supported by the bootloader crate:

- The map physical memory feature places a position in the virtual address space where it maps the whole physical memory. The kernel can thus map all of the physical memory since it has access to it.
- The recursive page table feature is used by the bootloader to recursively map a level 4 page table item.

➢ Boot Information

In a BootInfo struct, all the information that the bootloader crate provides to our kernel is specified. When the map_physical_memory feature is enabled, the two fields that are now available are memory_map and physical_memory_offset.

The BootInfo struct is sent from the kernel to the bootloader through a &'static BootInfo parameter to _start function. Since it hasn't yet been defined in the method, we must add the following argument.

```
// in src/main.rs

use bootloader::BootInfo;

#[no_mangle]
pub extern "C" fn _start(boot_info: &'static BootInfo) -> ! { // new argument
    [...]
}
```

➢ The enrtry_point Macro

The function signature of the _start function is not validated when it is called from outside the bootloader. This indicates that, while it may accept any number of arguments without encountering any compilation issues, at runtime it might malfunction or behave in an undefinable way**.**

You may type-check a Rust function declaration as the entry point by using the bootloader crate's entry point macro. By modifying the entry point function, use this macro:

```
// in src/main.rs

use bootloader::{BootInfo, entry_point};

entry_point!(kernel_main);

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    [...]
}
```

The entry point should be extern "C" or no mangle since the macro supplies the real lower-level _start entry point. Now that the kernel main function is a fully working Rust function, you are free to assign it any name.

```
// in src/lib.rs

#[cfg(test)]
use bootloader::{entry_point, BootInfo};

#[cfg(test)]
entry_point!(test_kernel_main);

/// Entry point for `cargo test`
#[cfg(test)]
fn test_kernel_main(_boot_info: &'static BootInfo) -> ! {
    // like before
    init();
    test_main();
    hlt_loop();
}
```

To all objects, add the attribute #[cfg(test)]. To distinguish our test entry point from our main's kernel main, we give it a unique name: test kernel main. rs. We prefix the parameter name with a_ to hide the warning about a variable that is now useless because we don't utilize the BootInfo parameter.

Heap Allocation

Local and Static Variables

Both local and static variables have fixed size

Local Variables

This type of variable is stored on the call-stack, and these are valid till adjacent function returns. When we deal with rust compiler it enforces these lifetimes and put error because of use value for too long. And these variables are only available until the function is returned.

Static Variables

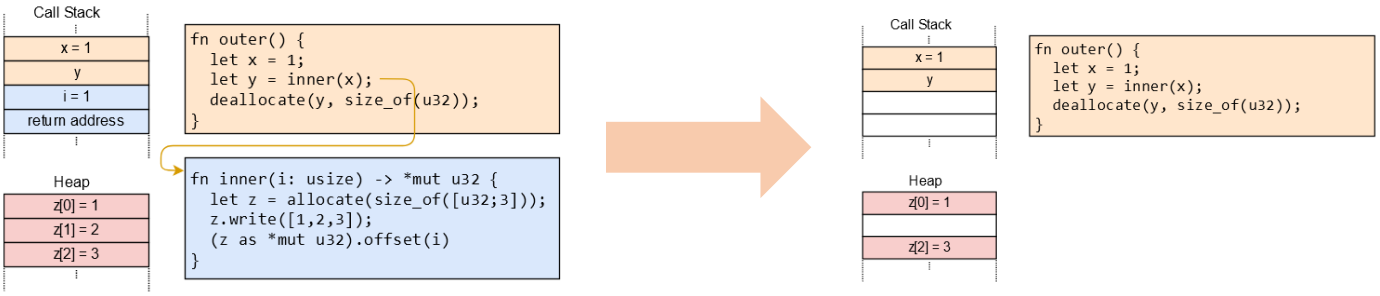
We use fixed memory location to store static variables separate from the stack and the **linker** and **encoded** assigned it to the memory at compile time. This type of variable is always in the Ready position and read only automatically. Static variables don't reduce the space even there no longer needed.

Dynamic Memory

heap support to dynamic memory allocation it constrains two functions called allocate and deallocate , the allocate use to return free memory space and deallocate use to freed up memory space when it stored variable is called

Allocation in RUST

Rust provides a service that can call allocate and deallocate function implicitly. By using box::new we can allocate values, call into the heap with data size. If want to freed up memory the box executes drop attribute to call deallocate.



The allocator Interface

To allocate the interface the first step is add dependency on the alloc crate like this

```
// in src.rs

extern crate alloc;
```

```
# in cargo.toml

#[global_allocate] // the alloc needs heap allocator but in rust the heap allocators are represent by the GlobalAlloc trait so we need to add it

#[alloc_error_handler] // to handle the memory efficiency


[unstable]
build-std = ["core", "compiler_builtins", "alloc"]
```

The GlobalAlloc Trait

This trait describes the functions that a heap allocator should deliver. This is how it declaration of it

```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);

    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 { ... }
    unsafe fn realloc(
        &self,
        ptr: *mut u8,
        layout: Layout,
        new_size: usize
    ) -> *mut u8 { ... }
}
```

A DummyAllocator

we can create a simple dummy allocator. For that, we created a new allocator module like this

```
// in lib.rs

pub mod allocator;
```

Our dummy allocator does the perfect minimum to execute the trait and always returns an error when the alloc is called. It looks like this:

```rust
// in allocator.rs

use alloc::alloc::{GlobalAlloc, Layout};
use core::ptr::null_mut;

pub struct Dummy;

unsafe impl GlobalAlloc for Dummy {
    unsafe fn alloc(&self, _layout: Layout) -> *mut u8 {
        null_mut()
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        panic!("dealloc should be never called")
    }
}
```

The #[global_allocator] Attribute

This attribute tells that to rust compiling program which allocator command it should apply as a global heap allocator. This is how enter an example of Dummy allocator as the global allocator

```rust
// in allocator.rs

#[global_allocator]
static ALLOCATOR: Dummy = Dummy;
```

The #[alloc_Error_handler] Attribute

This is the special function that calls when allocator error occurs. Like panic handler

```rust
// in src/lib.rs

#![feature(alloc_error_handler)] // at the top of the file

#[alloc_error_handler]
fn alloc_error_handler(layout: alloc::alloc::Layout) -> ! {
    panic!("allocation error: {:?}", layout)
}
```

```rust
// in src/main.rs

extern crate alloc;

use alloc::boxed::Box;

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // [...] print "Hello World!", call `init`, create `mapper` and `frame_allocator`

    let x = Box::new(41);

    // [...] call `test_main` in test mode

    println!("It did not crash!");
    blog_os::hlt_loop();
}
```

Creating a Kernel Heap

Now we are going to make a heap memory region this step should do before the create allocation. To do this we need to make a virtual memory range for the heap fist step is make virtual memory this how we do it

```rust
// in src/allocator.rs

pub const HEAP_START: usize = 0x_4444_4444_0000;
pub const HEAP_SIZE: usize = 100 * 1024; // 100 KiB
```

Then we need to map physical memory to resolve this we can create an init_heap function its look like this

```rust
// in src/allocator.rs

use x86_64::{
    structures::paging::{
        mapper::MapToError, FrameAllocator, Mapper, Page, PageTableFlags, Size4KiB,
    },
    VirtAddr,
};

pub fn init_heap(
    mapper: &mut impl Mapper<Size4KiB>,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
) -> Result<(), MapToError<Size4KiB>> {
    let page_range = {
        let heap_start = VirtAddr::new(HEAP_START as u64);
        let heap_end = heap_start + HEAP_SIZE - 1u64;
        let heap_start_page = Page::containing_address(heap_start);
        let heap_end_page = Page::containing_address(heap_end);
        Page::range_inclusive(heap_start_page, heap_end_page)
    };
    for page in page_range {
        let frame = frame_allocator
            .allocate_frame()
            .ok_or(MapToError::FrameAllocationFailed)?;
        let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE;
        unsafe {
            mapper.map_to(page, frame, flags, frame_allocator)?.flush()
        };
    }

    Ok(())
}
```

```rust
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    use blog_os::allocator; // new import
    use blog_os::memory::{self, BootInfoFrameAllocator};

    println!("Hello World{}", "!");
    blog_os::init();

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = unsafe {
        BootInfoFrameAllocator::init(&boot_info.memory_map)
    };

    // new
    allocator::init_heap(&mut mapper, &mut frame_allocator)
        .expect("heap initialization failed");

    let x = Box::new(41);

    // [...] call `test_main` in test mode

    println!("It did not crash!");
    blog_os::hlt_loop();
}
```

Using an Allocator Crate

We begin by using an exterior allocator box because implementing an allocator is somewhat complicated.

Adding a Test

```
// in tests/heap_allocation.rs

#![no_std]
#![no_main]
#![feature(custom_test_frameworks)]
#![test_runner(blog_os::test_runner)]
#![reexport_test_harness_main = "test_main"]

extern crate alloc;

use bootloader::{entry_point, BootInfo};
use core::panic::PanicInfo;

entry_point!(main);

fn main(boot_info: &'static BootInfo) -> ! {
    unimplemented!();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    blog_os::test_panic_handler(info)
}
```

```
// in tests/heap_allocation.rs

fn main(boot_info: &'static BootInfo) -> ! {
    use blog_os::allocator;
    use blog_os::memory::{self, BootInfoFrameAllocator};
    use x86_64::VirtAddr;

    blog_os::init();
    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = unsafe {
        BootInfoFrameAllocator::init(&boot_info.memory_map)
    };
    allocator::init_heap(&mut mapper, &mut frame_allocator)
        .expect("heap initialization failed");

    test_main();
    loop {}
}
```

```rust
// in tests/heap_allocation.rs
use alloc::boxed::Box;

#[test_case]
fn simple_allocation() {
    let heap_value_1 = Box::new(41);
    let heap_value_2 = Box::new(13);
    assert_eq!(*heap_value_1, 41);
    assert_eq!(*heap_value_2, 13);
}
```

```rust
// in tests/heap_allocation.rs

use blog_os::allocator::HEAP_SIZE;

#[test_case]
fn many_boxes() {
    for i in 0..HEAP_SIZE {
        let x = Box::new(i);
        assert_eq!(*x, i);
    }
}
```

11.Allocator Designs

Bump Allocator

A bump allocator is the most basic type of allocator (also known as a stack allocator). It basically maintains track of the total amount of bytes and allocations while allocating memory linearly. Because it has a serious restriction—it can only release full memory at once—it is only usable in extremely restricted use situations. A bump allocator allocates memory by growing the next variable that corresponds to the beginning of the free memory. The heap's start address is equal to the next at the beginning. Next is constantly raised by the allocation size for each allocation, pointing to the line separating occupied and unused memory.

- Implementation

We can implement by declaring a new allocator::bump submodule

Linked List Allocator

This is such that even when the stored data is no longer required, the regions are nonetheless mapped to virtual addresses and supported by physical frames. We can maintain track of an infinite number of liberated areas without using extra memory if we store the information about the freed region in the region itself.

- Implementation

We can create our simple LinkedListAllocator type
We can create a private ListNode struct in a new allocator::linked_list submodule.

Fixed-Size Block Allocator

A fixed-size block allocator works on the following principles: We set a finite number of block sizes and round up each allocation to the next block size instead of allocating exactly the amount of RAM that is required. When employing blocks of 16, 64, and 512 bytes, for example, an allocation of 4 bytes would produce a block of 16, 48 bytes in a block of 64, and 128 bytes in a block of 512 bytes.
We maintain a list of the available memory by building a linked list in the available memory, much like the linked list allocator. We, therefore, generate a distinct list for each size class rather than use a single list with various block sizes. Then, each list only contains blocks of a single size.

- Implementation

We can create a ListNode type in a new allocator::fixed_size_block and we can implement it

12.Async / Await

I. Multitasking

The capacity to multitask, or manage many things at once, is one of the core features of most operating systems. While reading this article, you could, for instance, have a text editor or a terminal window open. The background operations that keep your desktop windows maintained, check for updates, or index data continue to operate even when there is just one active browser window.

Even though it could seem as though all jobs are active at once, each CPU core can only handle one task at a time. For each active process to make some progress, the operating system switches between them quickly, providing the impression that the processes are running simultaneously. As a result of how swiftly computers run, we usually ignore these transitions.

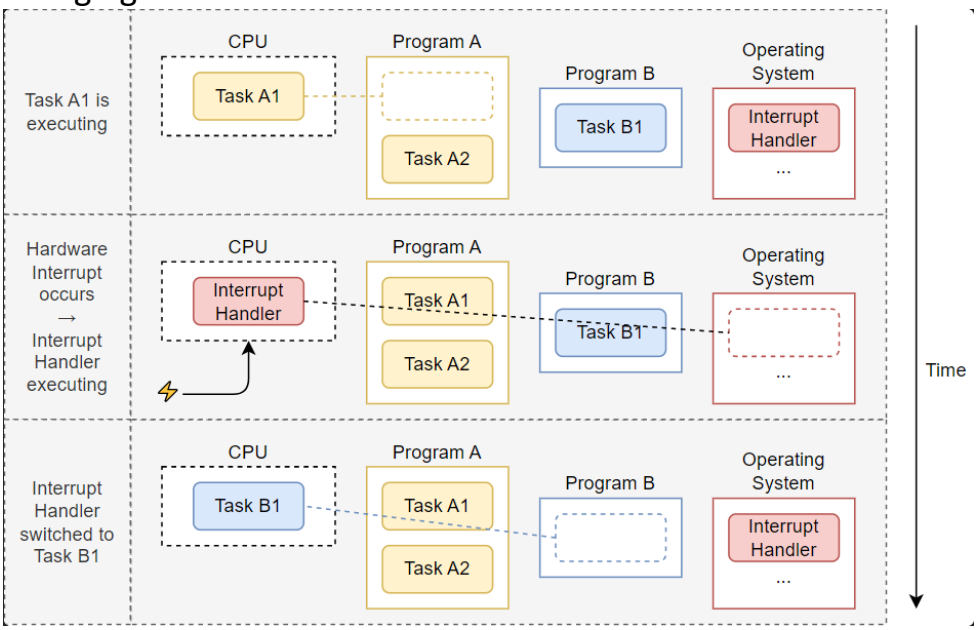
Multi-core CPUs have more processing cores than single-core CPUs, which can only do one activity at once. A CPU with 8 cores, for instance, can handle 8 tasks concurrently. Check out a subsequent article that goes into multi-core CPU setup. For the sake of simplicity, this course will solely address single-core CPUs.

There are two ways that one can multitask. Cooperative multitasking frequently demands the handover of CPU control from certain activities to others for other processes to proceed. Preemptive multitasking switches between them at random moments by forcibly suspending threads and using operating system capabilities. The two forms of multitasking, as well as their advantages and disadvantages, will be more thoroughly discussed in the sections that follow.

- **Preemptive Multitasking**

When using preemptive multitasking, the operating system chooses the best moment to switch between tasks. It does this by regaining control of the CPU after each interrupt. As soon as fresh input is made available to the system, this enables task switching. For instance, it could change jobs in reaction to incoming network packets or mouse movement. A hardware timer set up by the operating system to broadcast an interrupt after a certain amount of time may also be used to determine the precise amount of time a process is permitted to operate.

The task-switching procedure during a hardware interrupt is depicted in the following figure:



The CPU executes task A1 from program A in the top row. The remaining work has been delayed. A hardware interrupt is sent to the second row's CPU. The CPU immediately stops task A1's progress, as described in the documentation's section on hardware interrupts, and switches to the interrupt handler indicated in the interrupt descriptor table (IDT). Task B1 is given priority over Process A1 thanks to this interrupt handler, which restores control of the CPU to the operating system.

- **Cooperative Multitasking**

Instead of forcibly halting jobs at random intervals, cooperative multitasking enables each work to continue running until it freely gives up control of the CPU. As a result, tasks can halt themselves when necessary, such as when they must wait for I/O operations.

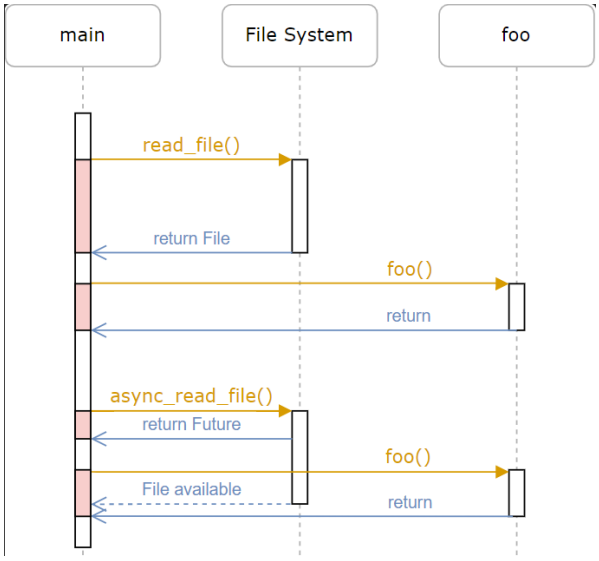
At the linguistic level, two typical instances of cooperative multitasking are coroutines and async/await. The goal is to include yield operations, which are determined either by the programmer or the compiler, and relinquish CPU control to allow for the execution of other processes. For instance, a complicated loop may have a yield appended to it after each iteration.

Asynchronous tasks and cooperative multitasking are commonly mixed. Instead of delaying an operation's completion and prohibiting other activities from running during this time, asynchronous operations return a "not ready" state if an operation isn't yet finished. As a result, the waiting job may take a yield action to release resources for other tasks.

II. Async / Await in Rust

The Rust programming language's async/await paradigm is extremely advantageous for cooperative multitasking. We must first comprehend how asynchronous programming and Rust's futures function before we can analyze what async/await is and how it functions.

The greatest way to explain the idea of the future is with a simple example:



Future traits in Rust look like this to express futures:

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

The associated type Output specifies the type of the asynchronous value. For instance, in the example above, the async read file method would produce a Future example with the Output attribute set to File.

The polling mechanism enables checking whether the value is already available. The following is the Poll enum that is returned:

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Executors and Wakers

When using futures, async/await enables ergonomic asynchronous operation. As a result, for the asynchronous code to execute, it has to be polled at some point.

i. **Executors** - A program called an executor controls each future in Rust from a central location. Distributing the burden among them might leverage many CPU cores and work theft. Additionally, memory overhead and low latency are optimized in embedded executor implementations.

ii. **Wakers** - For the API to function, each poll call must supply a unique Waker type wrapped in a Context type. To announce the task's (partial) completion, the executor constructs this type.