中山大學

SUN YAT-SEN UNIVERSITY

实验报告

实验二：中断处理

姓　　名	元朗曦
学　　号	23336294
班　　级	计算机八班
专　　业	计算机科学与技术
学　　院	计算机学院

1 合并实验代码

在 pkg/kernel/src/memory 文件夹中，增量代码补充包含了如下的模块：

- address.rs：定义了物理地址到虚拟地址的转换函数，这一模块接受启动结构体提供的物理地址偏移，从而对物理地址进行转换；
- frames.rs：利用 bootloader 传入的内存布局进行物理内存帧分配，实现 x86_64 的 FrameAllocator trait。本次实验中不会涉及，后续实验中会用到；
- gdt.rs：定义 TSS 和 GDT，为内核提供内存段描述符和任务状态段；
- allocator.rs：注册内核堆分配器，为内核堆分配提供能力。从而能够在内核中使用 alloc 提供的操作和数据结构，进行动态内存分配的操作，如 Vec、String、Box 等。

在 pkg/kernel/src/interrupt 文件夹中，增量代码补充包含了如下的模块：

- apic.rs：有关 XAPIC、IOAPIC 和 LAPIC 的定义和实现；
- consts.rs：有关于中断向量、IRQ 的常量定义；
- exceptions.rs：包含了 CPU 异常的处理函数，并暴露 register_idt 用于注册 IDT；
- mod.rs：定义了 init 函数，用于初始化中断系统，加载 IDT。

2 GDT 与 TSS

在本实验的操作系统中，GDT、TSS 和 IDT 均属于全局静态的数据结构，因此需要将它们定义为 static 类型，并使用 lazy_static 宏来实现懒加载，其本质上也是通过 Once 来保护全局对象，但是它的初始化函数无需参数传递，因此可以直接声明，无需手动调用 call_once 函数来传递不同的初始化参数。

在 src/memory/gdt.rs 中补全代码如下：

```rs
lazy_static! {
    static ref TSS: TaskStateSegment = {
        // ...

        tss.interrupt_stack_table[DOUBLE_FAULT_IST_INDEX as usize] = {
            const STACK_SIZE: usize = IST_SIZES[1];
            static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
            let stack_start = VirtAddr::from_ptr(addr_of_mut!(STACK));
            let stack_end = stack_start + STACK_SIZE as u64;
            info!(
                "Double Fault Stack: 0x{:016x}-0x{:016x}",
                stack_start.as_u64(),
                stack_end.as_u64()
            );
            stack_end
        };
        tss.interrupt_stack_table[PAGE_FAULT_IST_INDEX as usize] = {
            const STACK_SIZE: usize = IST_SIZES[2];
            static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
            let stack_start = VirtAddr::from_ptr(addr_of_mut!(STACK));
            let stack_end = stack_start + STACK_SIZE as u64;
            info!(
                "Page Fault Stack: 0x{:016x}-0x{:016x}",
```

```
                stack_start.as_u64(),
                stack_end.as_u64()
            );
            stack_end
        };

        tss
    };
}
```

3 注册中断处理程序

在 `src/interrupt/mod.rs` 中将中断描述符表的注册委托给各个模块：

```rs
mod clock;
mod serial;
mod exceptions;

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

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        unsafe {
            exceptions::register_idt(&mut idt);
            clock::register_idt(&mut idt);
            serial::register_idt(&mut idt);
        }
        idt
    };
}
```

之后我们需要在 `src/interrupt` 目录下创建 `exceptions.rs`、`clock.rs` 和 `serial.rs` 三个文件：

- `exceptions.rs` 中描述了 CPU 异常的处理，这些异常由 CPU 在内部生成，用于提醒正在运行的内核需要其注意的事件或情况。`x86_64` 的 `InterruptDescriptorTable` 中为这些异常处理函数提供了定义，如 `divide_error`、`double_fault` 等。
- 对于中断请求（IRQ）和硬件中断，我们将在独立的文件中进行处理。`clock.rs` 中描述了时钟中断的处理，`serial.rs` 中描述了串口输入中断的处理。
- 对于软件中断，如在 x86 架构中的系统调用 `int 0x80`，我们将在 `syscall.rs` 中进行处理，从而统一地对中断进行代码组织。这部分内容将在后续实验中进行实现。

按照项目规范，为 `interrupt` 模块添加 `pub fn init()` 函数，将中断系统的初始化工作统一起来：

```rs
/// init interrupt system
pub fn init() {
    IDT.load();

```

```
    // DONE: check and init APIC
    unsafe {
        let mut lapic = XApic::new(physical_to_virtual(LAPIC_ADDR));
        lapic.cpu_init();
    }

    // ...

    info!("Interrupts Initialized.");
}
```

在 exception.rs 中为各种 CPU 异常注册中断处理程序：

```rs
#[allow(unsafe_op_in_unsafe_fn)]
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    idt.debug
        .set_handler_fn(debug_handler);

    idt.divide_error
        .set_handler_fn(divide_error_handler);

    // ...
}
```

由于中断处理函数需要遵循相应的调用约定，我们需要使用 `extern "x86-interrupt"` 修饰符来声明函数，例如：

```rs
pub extern "x86-interrupt" fn debug_handler(
    stack_frame: InterruptStackFrame
) {
    panic!("EXCEPTION: DEBUG\n\n{:#?}", stack_frame);
}

pub extern "x86-interrupt" fn divide_error_handler(
    stack_frame: InterruptStackFrame
) {
    panic!("EXCEPTION: DIVIDE ERROR\n\n{:#?}", stack_frame);
}
```

4 初始化 APIC

可编程中断控制器（PIC）是构成 x86 架构的重要组成部分之一。得益于这一类芯片的存在，x86 架构得以实现中断驱动的操作系统设计。中断是一种处理外部事件的机制，允许计算机在运行过程中响应异步的、不可预测的事件。PIC 的引入为处理中断提供了关键的硬件支持。

最初，x86 架构使用的是 8259 可编程中断控制器，它是一种级联的、基于中断请求线（IRQ）的硬件设备。随着计算机体系结构的发展和性能需求的提高，单一的 8259 PIC 逐渐显露出瓶颈，无法满足现代系统对更高级别中断处理的需求。

为了解决这个问题，高级可编程中断控制器（APIC）被引入到 x86 架构中。APIC 提供了更灵活的中断处理机制，支持更多的中断通道和更先进的中断处理功能。它采用了分布式的架构，允许多个处理器在系统中独立处理中断，从而提高了整个系统的并行性和性能。

补全 `src/interrupt/apic/xapic.rs` 中 APIC 的初始化代码，以便在后续实验中使用 APIC 实现时钟中断和 I/O 设备中断；我们通过 `bitflags` 对寄存器进行位操作：

```rs
// Default physical address of xAPIC
pub const LAPIC_ADDR: u64 = 0xFEE00000;
// Local APIC Registers
pub struct LapicRegister;
impl LapicRegister {
    const TPR: u32 = 0x080;
    const SVR: u32 = 0x0F0;
    const ESR: u32 = 0x280;
    const LVT_TIMER: u32 = 0x320;
    const LVT_PCINT: u32 = 0x340;
    const LVT_LINT0: u32 = 0x350;
    const LVT_LINT1: u32 = 0x360;
    const LVT_ERROR: u32 = 0x370;
    const ICR: u32 = 0x380;
    const DCR: u32 = 0x3E0;
}
// Local APIC BitFlags
bitflags! {
    pub struct SpuriousFlags: u32 {
        const ENABLE = 0x00000100;
        const VECTOR = 0x000000FF;
        const VECTOR_IRQ = Interrupts::IrqBase as u32 + Irq::Spurious as u32;
    }
    pub struct LvtFlags: u32 {
        const MASKED = 0x00010000;
        const PERIODIC = 0x00020000;
        const VECTOR = 0x000000FF;
        const VECTOR_IRQ_TIMER = Interrupts::IrqBase as u32 + Irq::Timer as u32;
        const VECTOR_IRQ_ERROR = Interrupts::IrqBase as u32 + Irq::Error as u32;
    }
}

// ...

impl LocalApic for XApic {
    /// If this type APIC is supported.
    fn support() -> bool {
        // ...
    }

```

```
    /// Initialize the xAPIC for the current CPU.
    fn cpu_init(&mut self) {
        unsafe {
          // ...
        }
    }

    // ...
}
```

1. 检测系统中是否存在 APIC：

```rs
fn support() -> bool {
    // DONE: Check CPUID to see if xAPIC is supported.
    CpuId::new().get_feature_info().map(|f| f.has_apic()).unwrap_or(false)
}
```

2. 操作 SPIV 寄存器，启用 APIC 并设置 Spurious IRQ Vector：

```rs
fn cpu_init (&mut self) {
    unsafe {
        // DONE: Enable local APIC; set spurious interrupt vector.
        let mut spiv = SpuriousFlags::from_bits_truncate(self.read(LapicRegister::SVR));
        spiv.insert(SpuriousFlags::ENABLE);
        spiv.remove(SpuriousFlags::VECTOR);
        spiv.insert(SpuriousFlags::VECTOR_IRQ);
        self.write(LapicRegister::SVR, spiv.bits());

        // ...
    }
}
```

3. 设置计时器相关寄存器。APIC 中控制计时器的寄存器包括 TDCR、TICR 和 LVT Timer。其中，TDCR 用于设置分频系数，TICR 用于设置初始计数值，LVT Timer 用于设置中断向量号和触发模式：

```rs
fn cpu_init (&mut self) {
    unsafe {
        // ...

        // Set Initial Count.
        self.write(LapicRegister::ICR, 0x00002000);
        // Set Timer Divide.
        self.write(LapicRegister::DCR, 0x0000000B);
        // DONE: The timer repeatedly counts down at bus frequency.
        let mut timer = LvtFlags::from_bits_truncate(self.read(LapicRegister::LVT_TIMER));
        timer.remove(LvtFlags::MASKED);
```

```
            timer.insert(LvtFlags::PERIODIC);
            timer.remove(LvtFlags::VECTOR);
            timer.insert(LvtFlags::VECTOR_IRQ_TIMER);
            self.write(LapicRegister::LVT_TIMER, timer.bits());

            // ...
        }
    }
```

4. 禁用 PCINT、LINT0、LINT1 寄存器：

```rs
fn cpu_init (&mut self) {
    unsafe {
        // ...

        // DONE: Disable performance counter overflow interrupts (PCINT).
        self.write(LapicRegister::LVT_PCINT, LvtFlags::MASKED.bits());
        // DONE: Disable logical interrupt lines (LINT0, LINT1).
        self.write(LapicRegister::LVT_LINT0, LvtFlags::MASKED.bits());
        self.write(LapicRegister::LVT_LINT1, LvtFlags::MASKED.bits());

        // ...
    }
}
```

5. 设置错误中断 LVT Error 到对应的中断向量号：

```rs
fn cpu_init (&mut self) {
    unsafe {
        // ...

        // DONE: Map error interrupt to IRQ_ERROR.
        let mut error = LvtFlags::from_bits_truncate(self.read(LapicRegister::LVT_ERROR));
        error.remove(LvtFlags::MASKED);
        error.remove(LvtFlags::VECTOR);
        error.insert(LvtFlags::VECTOR_IRQ_ERROR);
        self.write(LapicRegister::LVT_ERROR, error.bits());

        // ...
    }
}
```

6. 连续写入两次 0 以清除错误状态寄存器；向 EOI 寄存器写入 0 以确认任何挂起的中断；设置 ICR 寄存器；设置 TPR 寄存器为 0，允许接收中断：

```rs
fn cpu_init (&mut self) {
    unsafe {
        // ...
```

```
4
5           // DONE: Clear error status register (requires back-to-back writes).
6           self.write(LapicRegister::ESR, 0);
7           self.write(LapicRegister::ESR, 0);
8
9           // DONE: Ack any outstanding interrupts.
10          self.eoi();
11
12          // DONE: Send an Init Level De-Assert to synchronise arbitration
            ID's.
13          self.set_icr(0x00088500);
14
15          // DONE: Enable interrupts on the APIC (but not on the processor).
16          self.write(LapicRegister::TPR, 0);
17      }
18  }
```

5 时钟中断

在顺利配置好 XAPIC 并初始化后，APIC 的中断就被成功启用了。为了响应时钟中断，我们需要为 IRQ0 Timer 设置中断处理程序。

创建 `src/interrupt/clock.rs` 文件，为 Timer 设置中断处理程序：

```rs
use super::consts::*;

use core::sync::atomic::{AtomicU64, Ordering};
use x86_64::structures::idt::*;

pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    idt[Interrupts::IrqBase as u8 + Irq::Timer as u8]
        .set_handler_fn(clock_handler);
}

pub extern "x86-interrupt" fn clock_handler(_sf: InterruptStackFrame) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        if inc_counter() % 0x10000 == 0 {
            info!("Tick! @{}", read_counter());
        }
        super::ack();
    });
}

static COUNTER: AtomicU64 = AtomicU64::new(0);

#[inline]
pub fn read_counter() -> u64 {
    // DONE: load counter value
```

```rust
    COUNTER.load(Ordering::SeqCst)
}

#[inline]
pub fn inc_counter() -> u64 {
    // DONE: read counter value and increase it
    COUNTER.fetch_add(1, Ordering::SeqCst)
}
```

仅仅开启 APIC 的中断并不能触发中断处理，这是因为 CPU 的中断并没有被启用。在 `src/lib.rs` 中，所有组件初始化完毕后，需要为 CPU 开启中断：

```rust
pub fn init(boot_info: &'static BootInfo) {
    // ...

    x86_64::instructions::interrupts::enable();
    info!("Interrupts Enabled.");

    // ...
}
```

6 串口输入中断

遵循 I/O 中断处理的 Top half & Bottom half 原则，在中断发生时，仅仅在中断处理中做尽量少的事：读取串口的输入，并将其放入缓冲区。而在中断处理程序之外，选择合适的时机，从缓冲区中读取数据，并进行处理。

在 `src/drivers/uart16550.rs` 的 `init` 函数末尾为串口设备开启中断：

```rust
impl SerialPort {
    // ...

    /// Initializes the serial port.
    #[allow(clippy::identity_op)]
    pub fn init(&self) {
        // DONE: Initialize the serial port
        const PORT: u16 = 0x3F8; // COM1

        unsafe {
            // ...

            // Enable interrupts.
            let mut port: Port<u8> = Port::new(PORT + 1);
            port.write(0x01);
        }
    }

    /// Sends a byte on the serial port.
```

```rust
    #[allow(clippy::identity_op)]
    pub fn send(&mut self, data: u8) {
        // DONE: Send a byte on the serial port
        const PORT: u16 = 0x3F8; // COM1

        unsafe {
            let mut rdi: Port<u8> = Port::new(PORT + 5);
            let mut rax: Port<u8> = Port::new(PORT + 0);

            while (rdi.read() & 0x20) == 0 {}
            rax.write(data);
        }
    }

    /// Receives a byte on the serial port no wait.
    #[allow(clippy::identity_op)]
    pub fn receive(&mut self) -> Option<u8> {
        // DONE: Receive a byte on the serial port no wait
        const PORT: u16 = 0x3F8; // COM1

        unsafe {
            let mut rdi: Port<u8> = Port::new(PORT + 5);
            let mut rax: Port<u8> = Port::new(PORT + 0);

            if (rdi.read() & 0x01) == 0 {
                None
            } else {
                Some(rax.read())
            }
        }
    }

    pub fn backspace(&mut self) {
        self.send(0x08);
        self.send(0x20);
        self.send(0x08);
    }
}

impl fmt::Write for SerialPort {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for byte in s.bytes() {
            self.send(byte);
        }
        Ok(())
    }
```

```
}
```

同时，为了能够接收到 IO 设备的对应中断，我们需要在 `src/interrupt/mod.rs` 中为 IOAPIC 启用对应的 IRQ：

```rs
/// init interrupts system
pub fn init() {
    // ..

    // DONE: enable serial irq with IO APIC (use enable_irq)
    enable_irq(Irq::Serial0 as u8, 0);

    // ...
}
```

为了承接全部（可能的）用户输入数据，并将它们统一在标准输入，我们需要为输入准备缓冲区，并将其封装为一个驱动，在 `src/drivers/input.rs` 中实现：

```rs
use alloc::string::String;
use crossbeam_queue::ArrayQueue;

type Key = u8;

lazy_static! {
    static ref INPUT_BUF: ArrayQueue<Key> = ArrayQueue::new(128);
}

#[inline]
pub fn push_key(key: Key) {
    if INPUT_BUF.push(key).is_err() {
        warn!("Input buffer is full. Dropping key '{:?}'", key);
    }
}

#[inline]
pub fn try_pop_key() -> Option<Key> {
    INPUT_BUF.pop()
}

#[inline]
pub fn pop_key() -> Key {
    loop {
        if let Some(key) = try_pop_key() {
            return key;
        }
    }
}
```

```rust

#[inline]
pub fn get_line() -> String {
    let mut pos: usize = 0;
    let mut line = String::with_capacity(128);
    loop {
        // Print the prompt line.
        print!("\r\x1B[K> {line}");
        // Print the cursor (with offset for "> ").
        print!("\r\x1B[{}C", pos + 2);

        match pop_key() {
            0x0A | 0x0D => { // break on newline (LF|CR)
                print!("\n");
                break
            }
            0x08 | 0x7F => { // backspace (BS|DEL)
                if pos > 0 {
                    line.remove(pos - 1);
                    pos -= 1;
                }
            }
            0x1B => { // escape
                // Skip a '['.
                let _ = pop_key();

                match pop_key() {
                    0x43 => pos += (pos < line.len()) as usize,
                    0x44 => pos = pos.saturating_sub(1),
                    _ => {}
                }
            }
            c => {
                line.insert(pos, c as char);
                pos += 1;
            }
        }
    }
    line
}
```

> **注意**
>
> 我们把 `\n` 和 `\r` 都视为换行，而非按照文档描述的 `\n`。后者在实际运行时无法成功换行。

7 用户交互

完善输入缓冲区后，我们在 `src/main.rs` 中使用 `get_line` 函数来获取用户输入的一行数据，并将其打印出来、或进行更多其他的处理，实现响应用户输入的操作：

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

use ysos::*;
use ysos_kernel as ysos;

extern crate alloc;

#[macro_use]
extern crate log;

boot::entry_point!(kernel_main);

pub fn kernel_main(boot_info: &'static boot::BootInfo) -> ! {
    ysos::init(boot_info);
    info!("Hello World from YatSenOS v2!");

    loop {
        let input = input::get_line();

        match input.trim() {
            "exit" => break,
            _ => {
                println!("😜: no such command!");
                println!("Current clock: {} ticks\n",
                interrupt::clock::read_counter());
            }
        }
    }

    ysos::shutdown();
}
```

为了避免时钟中断频繁地打印日志，我们在 `clock_handler` 中，删除输出相关的代码，只保留计数器的增加操作。之后在 `get_line` 中打印计数器的值，以便证明时钟中断的正确执行。

8 思考题

1. 为什么需要在 `clock_handler` 中使用 `without_interrupts` 函数？如果不使用它，可能会发生什么情况？

> **解答**
>
> `without_interrupts` 函数用于禁用中断，以确保在处理时钟中断时不会发生其他中断的干扰。这样可以避免在处理中断时发生递归调用或竞争条件，从而导致系统不稳定或崩溃。

2. 考虑时钟中断进行进程调度的场景，时钟中断的频率应该如何设置？太快或太慢的频率会带来什么问题？请分别回答。

> **解答**
>
> 时钟中断的频率应该根据系统的需求进行设置。一般来说，频率过高会导致 CPU 频繁地处理中断，从而浪费 CPU 时间和资源，影响系统性能；而频率过低则可能导致系统响应不及时，影响用户体验。
>
> 通常情况下，时钟中断的频率设置在 100Hz 到 1000Hz 之间是比较合适的，这样可以在保证系统响应速度的同时，避免过多的 CPU 开销。

3. 在进行 `receive` 操作的时候，为什么无法进行日志输出？如果强行输出日志，会发生什么情况？谈谈你对串口、互斥锁的认识。

> **解答**
>
> 在进行 `receive` 操作时，串口的输入输出是通过中断来实现的，因此在中断处理程序中进行日志输出可能会导致死锁或竞争条件。因为日志输出本身也可能会触发中断，从而导致系统不稳定或崩溃。
>
> 串口是一种常见的串行通信接口，用于连接计算机和外部设备。它通常使用异步传输方式进行数据传输，具有简单、可靠等优点。互斥锁是一种用于保护共享资源的同步机制，可以确保在同一时刻只有一个线程可以访问共享资源，从而避免数据竞争和不一致性的问题。

4. 输入缓冲区在什么情况下会满？如果缓冲区满了，用户输入的数据会发生什么情况？

> **解答**
>
> 输入缓冲区满的情况通常发生在用户输入数据的速度超过了系统处理数据的速度时。例如，当用户快速输入数据时，缓冲区可能会很快填满。
>
> 如果缓冲区满了，新的输入数据将无法被存储到缓冲区中，这可能导致数据丢失或覆盖。为了避免这种情况，可以考虑增加缓冲区的大小，或者优化系统处理数据的速度。

5. 进行下列尝试，并在报告中保留对应的触发方式及相关代码片段：
   - 尝试用你的方式触发 Triple Fault，开启 `intdbg` 对应的选项，在 QEMU 中查看调试信息，分析 Triple Fault 的发生过程。
   - 尝试触发 Double Fault，观察 Double Fault 的发生过程，尝试通过调试器定位 Double Fault 发生时使用的栈是否符合预期。
   - 通过访问非法地址触发 Page Fault，观察 Page Fault 的发生过程。分析 `Cr2` 寄存器的值，并尝试回答为什么 Page Fault 属于可恢复的异常。

解答

参考 Writing an OS in Rust。

删除或注释 `src/interrupt/exceptions.rs` 中的 `page_fault_handler` 和 `double_fault_handler` 函数及相关调用，之后在 `src/main.rs` 中加入

```rs
pub fn kernel_main(boot_info: &'static boot::BootInfo) -> ! {
  // ...

  unsafe {
    *(0xDEADBEEF as *mut u8) = 42;
  }

  // ...
}
```