



中山大學  
SUN YAT-SEN UNIVERSITY

# 操作系统实验报告

## 实验二：中断处理

姓 名： 刘家祥  
学 号： 23336152  
教学班号： 计科二班  
专 业： 计算机科学与技术  
院 系： 计算机学院

2024~2025 学年第二学期

# 中断处理

## 一. 合并实验代码

在 `~/ysos` 目录下创建 `0x02` 目录，将 `0x01` 实验的代码复制到 `0x02` 目录下，然后在实验总仓库下执行 `git pull` 命令，再将实验仓库中最新的 `0x02` 下的代码复制到 `~/ysos/0x02` 目录下，以替换掉与 `0x01` 中同名的代码文件。

## 二. GDT 与 TSS

1. 在 `src/memory/gdt.rs` 中补全 TSS 的中断栈表，为 `Double Fault` 和 `Page Fault` 准备独立的栈。

```
// Fill interrupt_stack_table for Double Fault and Page Fault handlers
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 IST : 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 IST    : 0x{:016x}-0x{:016x}",
        stack_start.as_u64(),
        stack_end.as_u64()
    );
    stack_end
};

```

## 2. 合并后的代码修复

### 2.1. pkg/kernel/src/interrupts/exception.rs

添加 `use x86_64::VirtAddr;` 添加 `unsafe` 块

```

// 修改第7-9行
unsafe {
    idt.double_fault
        .set_handler_fn(double_fault_handler)
        .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX);
}

// 修改第10-12行
unsafe {
    idt.page_fault
        .set_handler_fn(page_fault_handler)
        .set_stack_index(gdt::PAGE_FAULT_IST_INDEX);
}

```

### 2.2. pkg/kernel/src/interrupts/apic/xapic.rs

添加适当的返回值

```

// 修复 support 函数返回类型错误
fn support() → bool {
    // 添加适当的返回值
    true // 或根据实际需求返回false
}

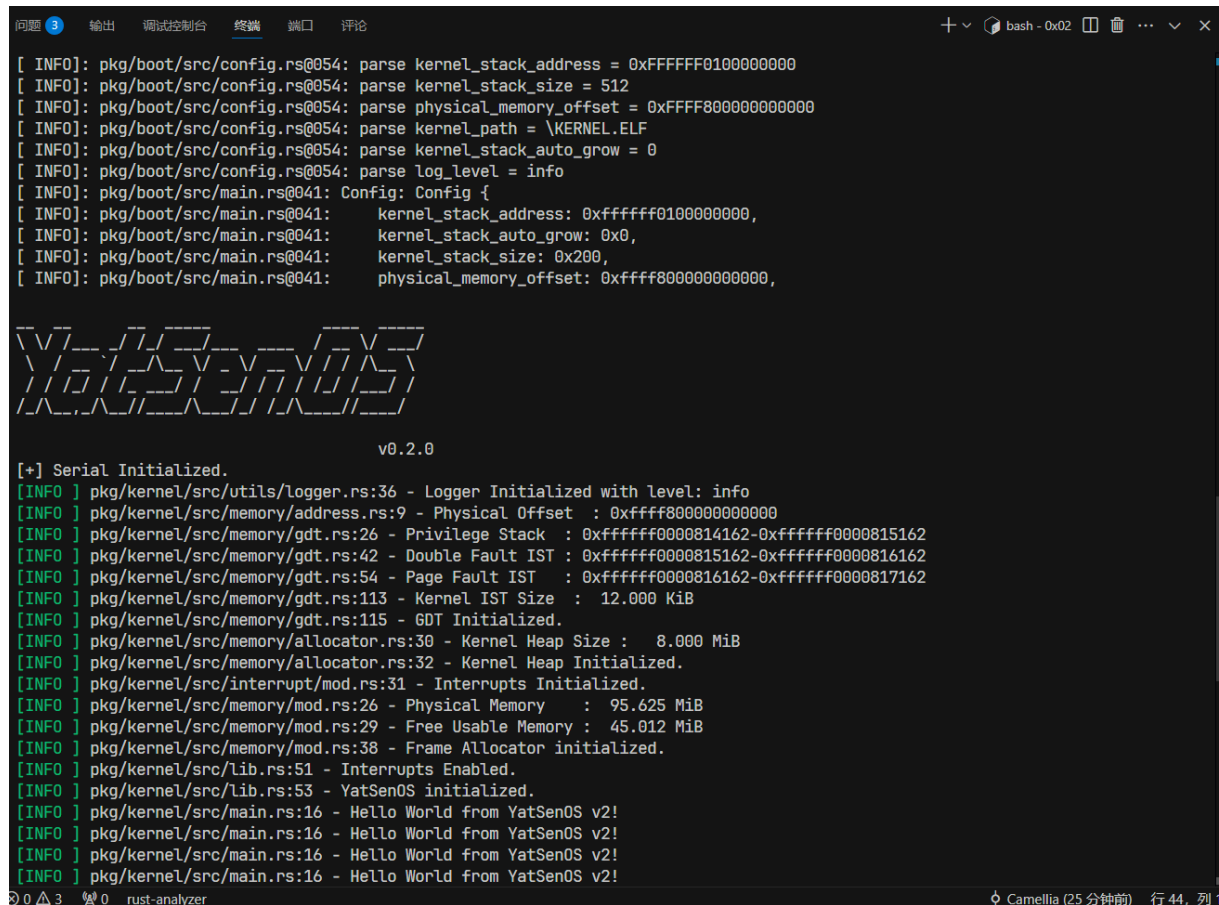
```

## 2.3. pkg/kernel/src/lib.rs

添加适当的参数

```
// 修改第43行
logger::init("info"); // 提供适当的日志级别
```

## 2.4. python3 ysos.py run 测试



```
[ INFO]: pkg/boot/src/config.rs@054: parse kernel_stack_address = 0xFFFFFFFF0100000000
[ INFO]: pkg/boot/src/config.rs@054: parse kernel_stack_size = 512
[ INFO]: pkg/boot/src/config.rs@054: parse physical_memory_offset = 0xFFFFF8000000000000
[ INFO]: pkg/boot/src/config.rs@054: parse kernel_path = \KERNEL.ELF
[ INFO]: pkg/boot/src/config.rs@054: parse kernel_stack_auto_grow = 0
[ INFO]: pkg/boot/src/config.rs@054: parse log_level = info
[ INFO]: pkg/boot/src/main.rs@041: Config: Config {
[ INFO]: pkg/boot/src/main.rs@041:     kernel_stack_address: 0xffffffff0100000000,
[ INFO]: pkg/boot/src/main.rs@041:     kernel_stack_auto_grow: 0x0,
[ INFO]: pkg/boot/src/main.rs@041:     kernel_stack_size: 0x200,
[ INFO]: pkg/boot/src/main.rs@041:     physical_memory_offset: 0xfffff8000000000000,

v0.2.0

[+] Serial Initialized.
[INFO ] pkg/kernel/src/utils/logger.rs:36 - Logger Initialized with level: info
[INFO ] pkg/kernel/src/memory/address.rs:9 - Physical Offset : 0xfffff8000000000000
[INFO ] pkg/kernel/src/memory/gdt.rs:26 - Privilege Stack : 0xffffffff0000814162-0xffffffff0000815162
[INFO ] pkg/kernel/src/memory/gdt.rs:42 - Double Fault IST : 0xffffffff0000815162-0xffffffff0000816162
[INFO ] pkg/kernel/src/memory/gdt.rs:54 - Page Fault IST : 0xffffffff0000816162-0xffffffff0000817162
[INFO ] pkg/kernel/src/memory/gdt.rs:113 - Kernel IST Size : 12.000 KiB
[INFO ] pkg/kernel/src/memory/gdt.rs:115 - GDT Initialized.
[INFO ] pkg/kernel/src/memory/allocator.rs:30 - Kernel Heap Size : 8.000 MiB
[INFO ] pkg/kernel/src/memory/allocator.rs:32 - Kernel Heap Initialized.
[INFO ] pkg/kernel/src/interrupt/mod.rs:31 - Interrupts Initialized.
[INFO ] pkg/kernel/src/memory/mod.rs:26 - Physical Memory : 95.625 MiB
[INFO ] pkg/kernel/src/memory/mod.rs:29 - Free Usable Memory : 45.012 MiB
[INFO ] pkg/kernel/src/memory/mod.rs:38 - Frame Allocator initialized.
[INFO ] pkg/kernel/src/lib.rs:51 - Interrupts Enabled.
[INFO ] pkg/kernel/src/lib.rs:53 - YatSenOS initialized.
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
```

可以看出 GDT、Interrupts 等模块均已成功初始化。

# 三. 注册中断处理程序

## 1. 扩展 pkg/kernel/src/interrupt/exception.rs 中的 register\_idt 函数。

```

// 这个文件负责处理 x86_64 CPU 架构中的所有异常
// 异常是 CPU 内部产生的事件，用于通知操作系统需要处理的特殊情况
use crate::memory::*;
use x86_64::registers::control::Cr2; // 用于访问 CR2 寄存器，其中存储了页错误的地址
use x86_64::structures::idt::{InterruptDescriptorTable, InterruptStackFrame,
PageFaultErrorCode};
use x86_64::VirtAddr;

/// 将所有 CPU 异常处理程序注册到中断描述符表(IDT)
///
/// x86_64 架构定义了 20 多种不同的异常：
/// - 0-31 号向量被保留用于 CPU 异常
/// - 其余的可由操作系统用于硬件中断
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    // 基本错误：数学和基本操作相关的异常
    idt.divide_error.set_handler_fn(divide_error_handler); // 除零错误 (#DE)
    idt.debug.set_handler_fn(debug_handler); // 调试异常 (#DB)
    idt.non_maskable_interrupt.set_handler_fn(nmi_handler); // 不可屏蔽中断，硬件级
    的严重错误
    idt.breakpoint.set_handler_fn(breakpoint_handler); // 断点异常 (#BP)，用于调试
    idt.overflow.set_handler_fn(overflow_handler); // 溢出异常 (#OF)
    idt.bound_range_exceeded.set_handler_fn(bound_range_exceeded_handler); // 数
    组访问越界 (#BR)
    idt.invalid_opcode.set_handler_fn(invalid_opcode_handler); // 无效操作码 (#UD)
    idt.device_not_available.set_handler_fn(device_not_available_handler); // 设
    备不可用 (#NM)

    // 双重错误需要特殊处理，发生在处理一个异常时又触发了另一个异常
    // 使用单独的栈以防止栈溢出导致三重错误（系统重置）
    unsafe {
        idt.double_fault
            .set_handler_fn(double_fault_handler)
            .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX); // 使用专用栈处理双重错误
    }

    // 内存和段错误相关的异常
    idt.invalid_tss.set_handler_fn(invalid_tss_handler); // 无效的 TSS (#TS)
    idt.segment_not_present.set_handler_fn(segment_not_present_handler); // 段不
    存在 (#NP)
    idt.stack_segment_fault.set_handler_fn(stack_segment_fault_handler); // 栈段
    错误 (#SS)

    idt.general_protection_fault.set_handler_fn(general_protection_fault_handler); //
    通用保护错误 (#GP)

```

```

// 页错误也需要特殊处理，使用专用栈防止在处理页错误时发生栈溢出
unsafe {
    idt.page_fault
        .set_handler_fn(page_fault_handler)
        .set_stack_index(gdt::PAGE_FAULT_IST_INDEX); // 使用专用栈处理页错误
}

// 浮点和 SIMD 相关的异常
idt.x87_floating_point.set_handler_fn(x87_floating_point_handler); // x87
FPU 错误 (#MF)
idt.alignment_check.set_handler_fn(alignment_check_handler); // 对齐检查错误
(#AC)
idt.machine_check.set_handler_fn(machine_check_handler); // 机器检查异常
(#MC)，可能是硬件故障
idt.simd_floating_point.set_handler_fn(simd_floating_point_handler); // SIMD
浮点异常 (#XF)

// 虚拟化和安全相关的异常
idt.virtualization.set_handler_fn(virtualization_handler); // 虚拟化异常 (#VE)
idt.security_exception.set_handler_fn(security_exception_handler); // 安全异常
(#SX)
}

/// 处理除零错误（向量 0）
/// 当程序尝试除以零时触发
pub extern "x86-interrupt" fn divide_error_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: DIVIDE ERROR\n\n{:#?}", stack_frame);
}

/// 处理调试异常（向量 1）
/// 当启用调试功能并触发断点条件时触发
pub extern "x86-interrupt" fn debug_handler(stack_frame: InterruptStackFrame) {
    panic!("EXCEPTION: DEBUG\n\n{:#?}", stack_frame);
}

/// 处理不可屏蔽中断（向量 2）
/// 通常由硬件故障引起，无法被 CLI 指令屏蔽
pub extern "x86-interrupt" fn nmi_handler(stack_frame: InterruptStackFrame) {
    panic!("EXCEPTION: NON-MASKABLE INTERRUPT\n\n{:#?}", stack_frame);
}

/// 处理断点异常（向量 3）
/// 由 INT3 指令触发，常用于调试器实现

```

```

pub extern "x86-interrupt" fn breakpoint_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: BREAKPOINT\n\n{:#?}", stack_frame);
}

/// 处理溢出异常 (向量 4)
/// 当 INTO 指令检测到溢出时触发
pub extern "x86-interrupt" fn overflow_handler(stack_frame: InterruptStackFrame)
{
    panic!("EXCEPTION: OVERFLOW\n\n{:#?}", stack_frame);
}

/// 处理数组边界超出异常 (向量 5)
/// 当 BOUND 指令检测到数组索引超出范围时触发
pub extern "x86-interrupt" fn bound_range_exceeded_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: BOUND RANGE EXCEEDED\n\n{:#?}", stack_frame);
}

/// 处理无效操作码异常 (向量 6)
/// 当 CPU 遇到无法识别的指令时触发
pub extern "x86-interrupt" fn invalid_opcode_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: INVALID OPCODE\n\n{:#?}", stack_frame);
}

/// 处理设备不可用异常 (向量 7)
/// 当尝试使用 x87 FPU 而它不存在或被禁用时触发
pub extern "x86-interrupt" fn device_not_available_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: DEVICE NOT AVAILABLE\n\n{:#?}", stack_frame);
}

/// 处理双重错误 (向量 8)
/// 当异常处理过程中发生第二个异常时触发
/// 这是一个严重错误, 如果不处理会导致系统重置
/// 返回 '!' 表示此函数永不返回
pub extern "x86-interrupt" fn double_fault_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) → ! {
    panic!(
        "EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

```

```

/// 处理无效 TSS 异常（向量 10）
/// 当任务状态段(TSS)中包含无效数据时触发
pub extern "x86-interrupt" fn invalid_tss_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) {
    panic!(
        "EXCEPTION: INVALID TSS, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

/// 处理段不存在异常（向量 11）
/// 当程序尝试使用不存在的段或描述符时触发
pub extern "x86-interrupt" fn segment_not_present_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) {
    panic!(
        "EXCEPTION: SEGMENT NOT PRESENT, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

/// 处理栈段错误（向量 12）
/// 当栈操作导致段限制被超过时触发
pub extern "x86-interrupt" fn stack_segment_fault_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) {
    panic!(
        "EXCEPTION: STACK SEGMENT FAULT, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

/// 处理通用保护错误（向量 13）
/// 这是最常见的异常之一，当程序违反保护机制时触发
/// 例如：访问非法内存地址、执行特权指令等

```



```

pub extern "x86-interrupt" fn general_protection_fault_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) {
    panic!(
        "EXCEPTION: GENERAL PROTECTION FAULT, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

/// 处理页错误（向量 14）
/// 当内存分页操作失败时触发，如访问未映射的页面
/// PageFaultErrorCode 包含具体错误信息：是否因写操作触发、是否是特权级访问等
/// CR2 寄存器中存储了导致页错误的内存地址
pub extern "x86-interrupt" fn page_fault_handler(
    stack_frame: InterruptStackFrame,
    err_code: PageFaultErrorCode,
) {
    panic!(
        "EXCEPTION: PAGE FAULT, ERROR_CODE: {:?}\n\nTrying to access:
{:#x}\n\n{:#?}",
        err_code,
        Cr2::read().unwrap_or(VirtAddr::new_truncate(0xdeadbeef)),
        stack_frame
    );
}

/// 处理 x87 浮点异常（向量 16）
/// 当 x87 FPU 操作发生错误时触发
pub extern "x86-interrupt" fn x87_floating_point_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: X87 FLOATING POINT\n\n{:#?}", stack_frame);
}

/// 处理对齐检查异常（向量 17）
/// 当启用对齐检查时，访问未对齐的内存地址时触发
pub extern "x86-interrupt" fn alignment_check_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) {
    panic!(
        "EXCEPTION: ALIGNMENT CHECK, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

```

```

/// 处理机器检查异常（向量 18）
/// 这是硬件级别的严重错误信号，如内存或总线错误
/// 返回 '!' 表示此函数永不返回
pub extern "x86-interrupt" fn machine_check_handler(stack_frame:
InterruptStackFrame) → ! {
    panic!("EXCEPTION: MACHINE CHECK\n\n{:#?}", stack_frame);
}

/// 处理 SIMD 浮点异常（向量 19）
/// 当 SSE/AVX 等 SIMD 指令执行错误时触发
pub extern "x86-interrupt" fn simd_floating_point_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: SIMD FLOATING POINT\n\n{:#?}", stack_frame);
}

/// 处理虚拟化异常（向量 20）
/// 与虚拟机操作相关的异常
pub extern "x86-interrupt" fn virtualization_handler(stack_frame:
InterruptStackFrame) {
    panic!("EXCEPTION: VIRTUALIZATION\n\n{:#?}", stack_frame);
}

/// 处理安全异常（向量 30）
/// 与 CPU 安全机制相关的异常
pub extern "x86-interrupt" fn security_exception_handler(
    stack_frame: InterruptStackFrame,
    error_code: u64,
) {
    panic!(
        "EXCEPTION: SECURITY EXCEPTION, ERROR_CODE: 0x{:016x}\n\n{:#?}",
        error_code, stack_frame
    );
}

```

## 四. 初始化 APIC

1. 在 `pkg/kernel/src/interrupts/apic/xapic.rs` 中实现 `init` 函数。

```

use super::LocalApic;
use bit_field::BitField;
use core::fmt::{Debug, Error, Formatter};
use core::ptr::{read_volatile, write_volatile};
use x86::cpuid::CpuId;

/// Default physical address of xAPIC
pub const LAPIC_ADDR: u64 = 0xFEE00000;

pub struct XApic {
    addr: u64,
}

impl XApic {
    pub unsafe fn new(addr: u64) → Self {
        XApic { addr }
    }

    unsafe fn read(&self, reg: u32) → u32 {
        unsafe {
            read_volatile((self.addr + reg as u64) as *const u32)
        }
    }

    unsafe fn write(&mut self, reg: u32, value: u32) {
        unsafe {
            write_volatile((self.addr + reg as u64) as *mut u32, value);
            self.read(0x20);
        }
    }
}

impl LocalApic for XApic {
    /// If this type APIC is supported
    fn support() → bool {
        CpuId::new().get_feature_info().map(|f| f.has_apic()).unwrap_or(false)
    }

    /// Initialize the xAPIC for the current CPU.
    fn cpu_init(&mut self) {
        // 定义 APIC 寄存器地址常量
        const REG_ID: u32 = 0x0020;
        const REG_VERSION: u32 = 0x0030;
        const REG_EOI: u32 = 0x00B0;
        const REG_ESR: u32 = 0x0280;
    }
}

```

```

const REG_ICR_LOW: u32 = 0x0300;
const REG_ICR_HIGH: u32 = 0x0310;
const REG_LVT_TIMER: u32 = 0x0320;
const REG_LVT_PERF: u32 = 0x0340;
const REG_LVT_LINT0: u32 = 0x0350;
const REG_LVT_LINT1: u32 = 0x0360;
const REG_LVT_ERROR: u32 = 0x0370;
const REG_TIMER_INIT_CNT: u32 = 0x0380;
const REG_TIMER_DIV: u32 = 0x03E0;
const REG_SVR: u32 = 0x00F0;

// 定义配置位常量
const APIC_ENABLE: u32 = 1 << 8;
const MASKED: u32 = 1 << 16;
const TIMER_PERIODIC: u32 = 1 << 17;
const BCAST: u32 = 1 << 19; // 广播到所有处理器
const INIT: u32 = 5 << 8; // INIT De-assert 模式
const TMLV: u32 = 1 << 15; // TM=1, LV=0
const DS: u32 = 1 << 12; // 传递状态位

// 假设的中断向量常量 - 实际项目中应使用真实定义
const IRQ_BASE: u32 = 32;
const IRQ_SPURIOUS: u32 = 31;
const IRQ_TIMER: u32 = 0;
const IRQ_ERROR: u32 = 19;

unsafe {
    // 1. 启用本地 APIC 并设置虚假中断向量
    let mut svr = self.read(REG_SVR);
    svr |= APIC_ENABLE; // 设置 EN 位
    svr &= !0xFF; // 清除向量字段
    svr |= IRQ_BASE + IRQ_SPURIOUS;
    self.write(REG_SVR, svr);

    // 2. 配置定时器 - 周期模式
    let mut lvt_timer = self.read(REG_LVT_TIMER);
    lvt_timer &= !0xFF; // 清除向量字段
    lvt_timer |= IRQ_BASE + IRQ_TIMER;
    lvt_timer &= !MASKED; // 清除屏蔽位
    lvt_timer |= TIMER_PERIODIC; // 设置周期模式
    self.write(REG_LVT_TIMER, lvt_timer);

    // 设置分频系数为 1
    self.write(REG_TIMER_DIV, 0b1011);
}

```

```

// 设置初始计数值
self.write(REG_TIMER_INIT_CNT, 0x40000);

// 3. 禁用逻辑中断线 LINT0, LINT1
self.write(REG_LVT_LINT0, MASKED);
self.write(REG_LVT_LINT1, MASKED);

// 4. 禁用性能计数器溢出中断
self.write(REG_LVT_PERF, MASKED);

// 5. 映射错误中断
let mut lvt_error = self.read(REG_LVT_ERROR);
lvt_error &= !0xFF; // 清除向量字段
lvt_error |= IRQ_BASE + IRQ_ERROR;
self.write(REG_LVT_ERROR, lvt_error);

// 6. 清除错误状态寄存器（需要连续两次写入）
self.write(REG_ESR, 0);
self.write(REG_ESR, 0);

// 7. 确认未处理的中断
self.eoi();

// 8. 发送 Init Level De-Assert 以同步仲裁 ID
self.write(REG_ICR_HIGH, 0); // 设置高位
self.write(REG_ICR_LOW, BCAST | INIT | TMLV); // 设置低位

// 等待传递完成
while self.read(REG_ICR_LOW) & DS != 0 {}
}

fn id(&self) → u32 {
    // NOTE: Maybe you can handle regs like `0x0300` as a const.
    unsafe { self.read(0x0020) >> 24 }
}

fn version(&self) → u32 {
    unsafe { self.read(0x0030) }
}

fn icr(&self) → u64 {
    unsafe { (self.read(0x0310) as u64) << 32 | self.read(0x0300) as u64 }
}

```

```

fn set_icr(&mut self, value: u64) {
    unsafe {
        while self.read(0x0300).get_bit(12) {}
        self.write(0x0310, (value >> 32) as u32);
        self.write(0x0300, value as u32);
        while self.read(0x0300).get_bit(12) {}
    }
}

fn eoi(&mut self) {
    unsafe {
        self.write(0x00B0, 0);
    }
}

impl Debug for XApic {
    fn fmt(&self, f: &mut Formatter) → Result<(), Error> {
        f.debug_struct("Xapic")
            .field("id", &self.id())
            .field("version", &self.version())
            .field("icr", &self.icr())
            .finish()
    }
}

```

## 五. 时钟中断

### 1. 为 Timer 设置中断处理程序

```

use super::consts::*;
use core::sync::atomic::{AtomicU64, Ordering};
use x86_64::structures::idt::InterruptDescriptorTable;
use x86_64::structures::idt::InterruptStackFrame;

```

```
// ... 省略部分代码
#[inline]
pub fn read_counter() → u64 {
    // 加载计数器值，使用Relaxed顺序即可
    COUNTER.load(Ordering::Relaxed)
}

#[inline]
pub fn inc_counter() → u64 {
    // 原子递增并返回新值
    COUNTER.fetch_add(1, Ordering::SeqCst) + 1
}
```

为调节时钟中断的频率，需要修改修 APIC 初始化代码中的：

- 定时器分频系数： `REG_TIMER_DIV`
- 初始计数值： `REG_TIMER_INIT_CNT`

要减半时钟中断的频率，可以将分频系数设置为 `0b1011`，或者初始计数值设置为 `0x40000`。

输出如下图：

```
v0.2.0

[+] Serial Initialized.
[INFO ] pkg/kernel/src/utils/logger.rs:36 - Logger Initialized with level: info
[INFO ] pkg/kernel/src/memory/address.rs:9 - Physical Offset : 0xfffff80000000000
[INFO ] pkg/kernel/src/memory/gdt.rs:26 - Privilege Stack : 0xffffffff0000817128-0xffffffff0000818128
[INFO ] pkg/kernel/src/memory/gdt.rs:42 - Double Fault IST : 0xffffffff0000818128-0xffffffff0000819128
[INFO ] pkg/kernel/src/memory/gdt.rs:54 - Page Fault IST : 0xffffffff0000819128-0xffffffff000081a128
[INFO ] pkg/kernel/src/memory/gdt.rs:113 - Kernel IST Size : 12.000 KiB
[INFO ] pkg/kernel/src/memory/gdt.rs:115 - GDT Initialized.
[INFO ] pkg/kernel/src/memory/allocator.rs:30 - Kernel Heap Size : 8.000 MiB
[INFO ] pkg/kernel/src/memory/allocator.rs:32 - Kernel Heap Initialized.
[INFO ] pkg/kernel/src/interrupt/mod.rs:35 - APIC initialized.
[INFO ] pkg/kernel/src/interrupt/mod.rs:42 - Interrupts Initialized.
[INFO ] pkg/kernel/src/memory/mod.rs:26 - Physical Memory : 95.625 MiB
[INFO ] pkg/kernel/src/memory/mod.rs:29 - Free Usable Memory : 45.000 MiB
[INFO ] pkg/kernel/src/memory/mod.rs:38 - Frame Allocator initialized.
[INFO ] pkg/kernel/src/lib.rs:51 - Interrupts Enabled.
[INFO ] pkg/kernel/src/lib.rs:53 - YatSenOS initialized.
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
[INFO ] pkg/kernel/src/interrupt/clock.rs:15 - Tick! @4096
[INFO ] pkg/kernel/src/main.rs:16 - Hello World from YatSenOS v2!
[INFO ] pkg/kernel/src/interrupt/clock.rs:15 - Tick! @8192
```

## 六. 串口输入中断

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

```
// 启用接收数据中断
unsafe {
    interrupt_enable.write(0x01_u8);
}
/// 发送退格控制符序列（用于删除一个字符）
pub fn backspace(&mut self) {
    self.send(0x08); // 后退
    self.send(0x20); // 空格覆盖
    self.send(0x08); // 再次后退
}
```

## 2. 补全 src/drivers/input.rs 驱动代码

### 2.1. 选择 `crossbeam_queue::ArrayQueue` 作为缓冲区的实现

```
/// 定义输入类型
pub type Key = u8;

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

### 2.2. 实现数据结构的初始化

```
/// 将按键添加到缓冲区
#[inline]
pub fn push_key(key: Key) {
    if INPUT_BUF.push(key).is_err() {
        warn!("Input buffer is full. Dropping key '{}:~'", key);
    }
}

/// 尝试从缓冲区获取一个按键，如果没有则返回 None
#[inline]
pub fn try_pop_key() → Option<Key> {
    INPUT_BUF.pop()
}
```



## 2.3. 实现并暴露 `pop_key` 函数,实现并暴露 `get_line` 函数。

```
/// 阻塞直到有按键可用
pub fn pop_key() → Key {
    loop {
        if let Some(key) = try_pop_key() {
            return key;
        }
        // 使用 hint::spin_loop 优化等待循环
        core::hint::spin_loop();
    }
}

/// 读取一行输入，直到遇到换行符
pub fn get_line() → String {
    // 创建一个预分配容量的字符串
    let mut line = String::with_capacity(64);
    // 使用 COM1 端口创建串口实例
    let mut serial = SerialPort::<0x3F8>::new();
    loop {
        let key = pop_key();
        match key {
            // 回车键 - 结束输入
            b'\r' | b'\n' => {
                serial.send(b'\r');
                serial.send(b'\n');
                break;
            }
            // 退格键 - 删除最后一个字符
            0x08 | 0x7F => {
                if !line.is_empty() {
                    line.pop();
                    serial.backspace();
                }
            }
            // 其他可打印字符
            _ => {
                // 回显字符
                serial.send(key);
                line.push(key as char);
            }
        }
    }
    line
}
```

### 3. 在 `src/interrupt/serial.rs` 中补全代码，为 `IRQ4 Serial0` 设置中断处理程序：

注意：这部分内容实验仓库代码已经给出不再赘述

### 4. 在 `src/interrupt/serial.rs` 补全 `receive` 函数，利用刚刚完成的 `input` 驱动，将接收到的字符放入缓冲区。

```
/// 从串口接收字符并放入输入缓冲区
/// 在每次中断时调用
fn receive() {
    // 创建串口实例
    let mut serial = SerialPort::<0x3F8>::new();

    // 读取可用的字符并放入输入缓冲区
    while let Some(byte) = serial.receive() {
        input::push_key(byte);
    }
}
```

## 七. 实现用户交互

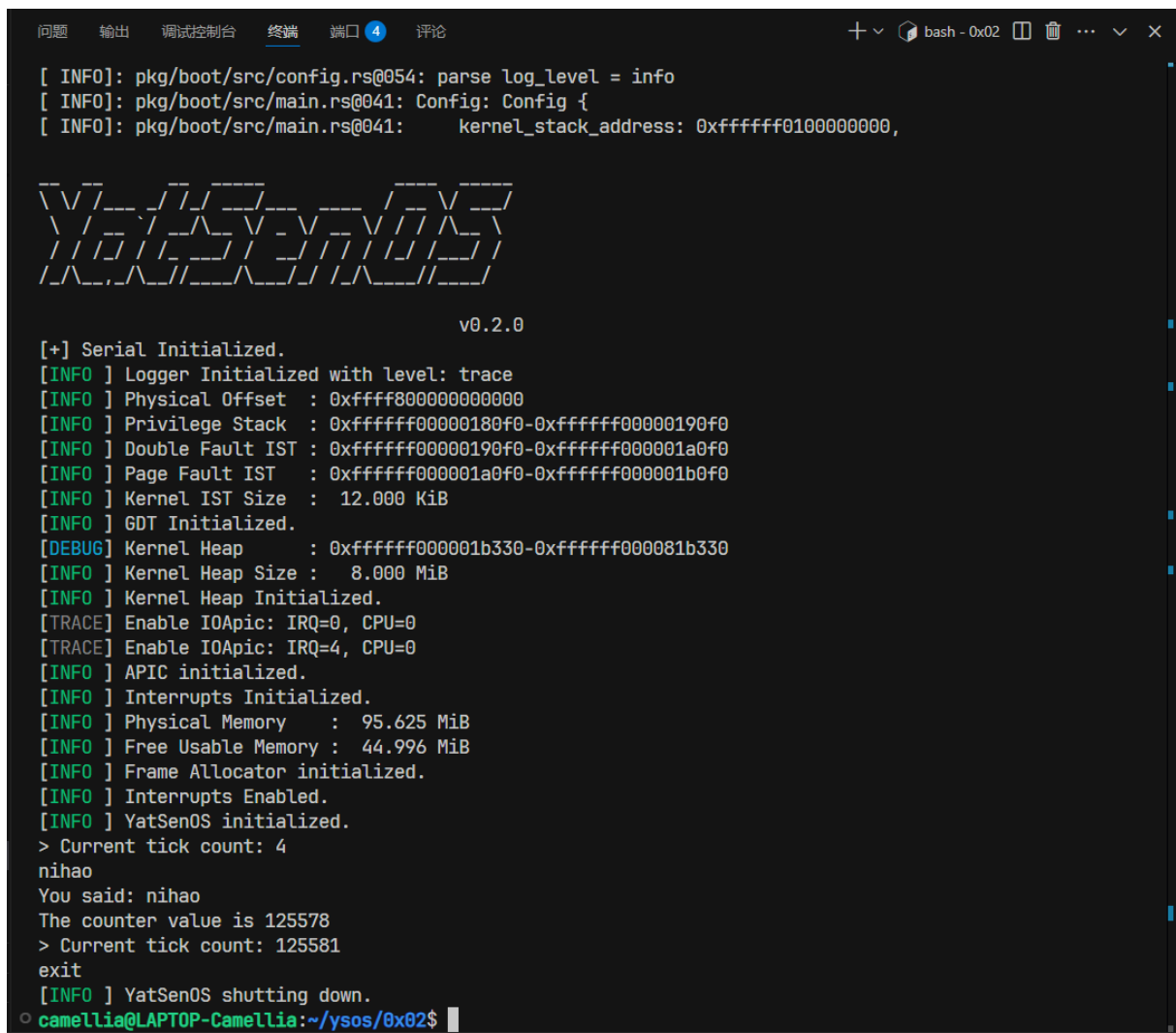
### 1. 为了避免时钟中断频繁地打印日志，删除 `clock_handle` 中的日志输出部分，仅保留计数器增加部分。

```
pub extern "x86-interrupt" fn clock_handler(_sf: InterruptStackFrame) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        // 注意：按照指导要求，直接删除日志输出，只保留计数器增加
        inc_counter();
        super::ack();
    });
}
```

## 2. 在 `src/drivers/input.rs` 的 `get_Line` 函数中增加打印当前计数器的值。

```
pub fn get_line() → String {  
    // 新增：打印当前计数器值，证明时钟中断正在工作  
    println!("Current tick count: {}",  
    // ...  
}
```

输出结果展示：



```
问题 输出 调试控制台 终端 端口 4 评论  
+ bash - 0x02  
[ INFO]: pkg/boot/src/config.rs@0054: parse log_level = info  
[ INFO]: pkg/boot/src/main.rs@041: Config: Config {  
[ INFO]: pkg/boot/src/main.rs@041: kernel_stack_address: 0xffffffff00000000,  
  
v0.2.0  
[+] Serial Initialized.  
[INFO ] Logger Initialized with level: trace  
[INFO ] Physical Offset : 0xffffffff800000000000  
[INFO ] Privilege Stack : 0xffffffff00000180f0-0xffffffff00000190f0  
[INFO ] Double Fault IST : 0xffffffff00000190f0-0xffffffff000001a0f0  
[INFO ] Page Fault IST : 0xffffffff000001a0f0-0xffffffff000001b0f0  
[INFO ] Kernel IST Size : 12.000 KiB  
[INFO ] GDT Initialized.  
[DEBUG] Kernel Heap : 0xffffffff000001b330-0xffffffff000001b330  
[INFO ] Kernel Heap Size : 8.000 MiB  
[INFO ] Kernel Heap Initialized.  
[TRACE] Enable IOApic: IRQ=0, CPU=0  
[TRACE] Enable IOApic: IRQ=4, CPU=0  
[INFO ] APIC initialized.  
[INFO ] Interrupts Initialized.  
[INFO ] Physical Memory : 95.625 MiB  
[INFO ] Free Usable Memory : 44.996 MiB  
[INFO ] Frame Allocator initialized.  
[INFO ] Interrupts Enabled.  
[INFO ] YatSenOS initialized.  
> Current tick count: 4  
nihao  
You said: nihao  
The counter value is 125578  
> Current tick count: 125581  
exit  
[INFO ] YatSenOS shutting down.  
camellia@LAPTOP-Camellia:~/ysos/0x02$
```

可以看出用户交互已经完成，而且避免了时钟中断频繁打印日志干扰用户交互。

## 八. 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？

- 为什么用？
  - ▶ 防止中断嵌套：时钟中断处理过程中，如果不禁用中断，可能会被另一个中断（包括另一个时钟中断）打断，导致中断嵌套。在嵌套中断的情况下，内核栈可能会不断增长，甚至导致栈溢出。
  - ▶ 保护日志操作的完整性：日志输出通常涉及多步操作（格式化字符串、访问输出设备等），这些操作需要作为一个原子单元完成。如果在日志输出过程中被中断打断，可能会导致日志输出不完整或混乱。
  - ▶ 确保中断确认的及时性：`super::ack()` 调用用于确认中断处理完成，这一步骤需要在整个处理过程之后立即执行，以便系统能继续接收新的中断。如果在确认前被打断，可能会导致中断控制器状态不一致。
- 不使用 `without_interrupts` 可能发生的问题
  - ▶ 日志输出混乱：多个时钟中断处理程序同时执行会导致日志消息交错混合，使得日志难以阅读和理解。
  - ▶ 状态不一致：如果在关键操作（如计数器递增和日志输出）过程中被打断，可能导致读取到的计数器值与实际输出日志时不一致。
  - ▶ 性能问题：中断嵌套会增加内核栈的使用和上下文切换的开销，降低系统性能。
  - ▶ 中断处理延迟：嵌套的中断处理可能会延长整体的中断响应时间，影响系统的实时性能。

## 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。

### 2.1. 时钟中断频率与进程调度

时钟中断是操作系统实现抢占式多任务的关键机制，其频率设置对系统性能有重要影响。

### 2.2. 合适的时钟频率

- 在进程调度中，时钟中断频率应根据系统类型和应用需求来设置：
  - ▶ 通用操作系统：通常在 100Hz 到 1000Hz 之间（即每秒 100 到 1000 次中断），以平衡响应时间和 CPU 使用率。
  - ▶ 服务器系统：可能使用较低频率（如 100Hz），以减少上下文切换和 CPU 开销。
  - ▶ 实时系统：可能需要更高频率（如 1000Hz 或更高），以确保及时响应外部事件。
  - ▶ 嵌入式系统：频率可能更低（如 10Hz 到 100Hz），以节省功耗和资源。

## 2.3. 频率太快的问题

- 上下文切换开销过大
  - 每次中断可能触发进程切换，包括保存/恢复寄存器、切换页表、刷新 TLB 等
  - 系统会将更多时间花在上下文切换上，而不是实际执行进程。
- CPU 缓存效率低
  - 频繁切换导致 CPU 缓存命中率下降。
  - 进程的工作集在缓存中难以保持，增加内存访问延迟。
- 中断处理开销
  - 处理中断本身占用 CPU 时间
  - 过高频率会使大量 CPU 时间浪费在中断处理上
- 能耗增加
  - 频繁唤醒 CPU 处理中断会增加能源消耗
  - 对移动设备和服务器尤为重要

## 2.4. 频率太慢的问题

- 系统相应性降低
  - 进程可能需要较长时间才可能被调度
  - 用户会感知到交互延迟和系统不流畅
- 进程公平性问题
  - CPU 密集型任务可能长时间占用处理器
  - I/O 密集型或者交互式任务得不到及时响应
- 实时性需求无法满足
  - 无法为需要及时时间保证的任务提供足够的调度精度
  - 可能导致截止时间失效
- 定时器精度不足
  - 降低系统计时精度
  - 影响依赖精确时间的应用程序

**注意：** 现代操作系统通常采用动态时钟(tickless)技术，可以根据系统负载自动调整中盾频率，平衡性能与能耗之间的关系。

3. 在进行 `receive` (位于 `src/interrupt/serial.rs`) 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。

## 3.1. 原因分析

- 资源冲突
  - 日志输出会调用 `pkg/kernel/src/drivers/uart16550.rs` 中的 `SerialPort::send()` 函数
  - 中断处理程序 `receive()` 使用相同的串口输出设备
- 死锁风险
  - 如果串口访问有互斥锁保护，会导致对同一个锁的重入请求
  - 中断处理程序已经在访问串口，再次请求会永久等待自己释放锁

## 3.2. 强行输出日志的后果

- 系统死锁：如果日志系统使用了互斥锁保护串口资源，会因尝试重复获取已持有的锁而死锁
- 递归中断：日志输出可能触发新的串口操作，导致新的中断，形成无限递归
- 数据损坏：串口接收和发送的数据可能混淆，导致接受数据不完整或发送数据错误
- 系统崩溃：由于上述任何原因，最终导致整个系统不稳定或崩溃

## 3.3. 对串口和互斥锁的认识

### 3.3.1. 串口 (UART16550)

- 共享资源：串口是一个物理硬件，同一时间只能执行一种操作（要么接收，要么发送）
- 中断驱动：从代码可见，系统使用中断方式处理串口接收，能够更高效地处理输入
- 缓冲区管理：代码中使用了 `ArrayQueue` 缓冲区存储接收到的输入，这是一种常见的中断-主线程数据交换方式

### 3.3.2. 互斥锁

- 防止竞争条件：互斥锁用于保护共享资源，确保同一时间只有一个执行上下文能访问资源
- 中断上下文的风险：在中断处理程序中使用互斥锁特别危险，因为：
  - 中断可能打断正常代码的锁操作
  - 如果中断处理程序尝试获取已被普通代码持有的锁，会导致死锁
  - 如果中断处理程序拿着锁时被其他优先级更高的中断打断，也可能导致优先级反转
- 替代方案：对于中断处理程序，更适合使用：
  - 自旋锁或无锁数据结构
  - 中断禁用/启用作为同步手段
  - 采用上半部/下半部机制分离时间关键型代码和可能阻塞的代码

## 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？

#### 4.0.1. 输入缓冲区何时会满？

- 输入缓冲区 `INPUT_BUF` 是一个 `ArrayQueue`，在初始化时被设置为固定大小 128：  
`ArrayQueue::new(128)`。
- 这意味着它最多只能存储 128 个按键（`Key` 类型，即 `u8`）。
- 当连续向缓冲区推送（`push_key`）了 128 个按键，而没有通过 `pop_key` 或者 `try_pop_key` 函数消费掉它们时，缓冲区就会达到其容量上限，即变满。这种情况通常发生在输入（例如来自串口中断）的速度超过了应用程序处理输入（例如 `get_line` 函数消耗按键）的速度时。

#### 4.0.2. 如果缓冲区满了，用户输入的数据会发生什么？

- 在 `push_key` 函数中，有以下逻辑：

```
if INPUT_BUF.push(key).is_err() {  
    warn!("Input buffer is full. Dropping key '{}'", key);  
}
```

- `INPUT_BUF.push(key)` 尝试将新的按键 `key` 添加到队列中。
- 如果队列已满，`push` 方法会返回一个 `Err`。
- 代码检查到 `is_err()` 为 `true` 时，会执行 `warn!` 宏，打印一条警告日志，内容是 “Input buffer is full. Dropping key ‘...’” （输入缓冲区已满。丢弃按键 ‘...’）。
- 最关键的是：这个未能成功推入缓冲区的按键数据会被丢弃，不会存储在缓冲区中，也就丢失了。后续调用 `pop_key` 或 `try_pop_key` 也无法获取到这个被丢弃的按键。

### 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。

#### 5.1. triple Fault

- 修改 `pkg/kernel/src/interrupt/mod.rs` 文件
  - 在 `init()` 函数内部，找到 `IDT.load();` 这一行。
  - 紧接着在 `IDT.load();` 之前，插入以下代码行：

```
// --- ADD THIS LINE ---

// Intentionally cause a divide-by-zero fault before IDT is loaded

unsafe { core::arch::asm!("mov dx, 0; div dx", options(nostack, nomem)); }

// --- END OF ADDED LINE ---
```

**注意：**这段汇编代码会尝试执行 `div dx`，其中 `dx` 为 0，这将触发一个除零异常（#DE）。由于此时 `IDT.load()` 尚未执行，CPU 无法找到 #DE 的处理程序，也无法找到 Double Fault 的处理程序，从而导致 Triple Fault。

- 重新编译内核 `python3 ysos.py build`，添加 `intdbg` 参数运行 `qemu`  
`python3 ysos.py run --intdbg`

得到如下结果：

```
GS =0000 0000000000000000 00000000 00000000
LDT=0000 0000000000000000 0000ffff 00008200 DPL=0 LDT
TR =0018 ffffffff0000017004 00000067 00008900 DPL=0 TSS64-avl
GDT=      ffffffff0000017070 00000027
IDT=      0000000005472018 00000fff
CR0=80010033 CR2=0000000000000000 CR3=0000000005c01000 CR4=00000668
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=00
0000000000000000
DR6=00000000ffff0fff DR7=0000000000000400
CCS=0000000000000088 CCD=ffffff01001ffe28 CCO=SUBQ
EFER=00000000000000d0
check_exception old: 0x8 new 0xd
Triple fault
camellia@LAPTOP-Camellia:~/ysos/0x02$
```

## 5.2. Double Fault

- 预期使用堆栈

```
[INFO ] Double Fault IST : 0xffffffff00000180c8-0xffffffff00000190c8
```

此处不再沿用(1)中代码，将其修改为：



```
// 触发页错误
unsafe {
    asm!(
        "mov rax, 28",
        "mov [0xdeadbeef], rax",
        options(nostack)
    );
}
```

- 通过访问一个无效地址（这里的无效地址有多种设置方法）先触发 Pagefault，并注释掉 Pagefault 处理函数，这样就能直接触发 Doublefault 了。

```
FS =0000 0000000000000000 00000000 00000000
GS =0000 0000000000000000 00000000 00000000
LDT=0000 0000000000000000 0000ffff 00000200 DPL=0 LDT
TR =0018 ffffffff0000017004 00000067 00000900 DPL=0 TSS64-avl
GDT= ffffffff0000017070 00000027
IDT= ffffffff000001a350 000000ff
CR0=80010033 CR2=ffffffffdeadbeef CR3=0000000005c01000 CR4=00000668
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000000
DR6=00000000ffff0fff DR7=0000000000000400
CCS=0000000000000000 CCD=0000000000000000 CC0=EFLAGS
EFER=0000000000000d00
[ERROR] ERROR: panic!

PanicInfo {
  message: EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x0000000000000000

  InterruptStackFrame {
    instruction_pointer: VirtAddr(
      0xffffffff00000405a,
    ),
    code_segment: SegmentSelector {
      index: 1,
      rpl: Ring0,
    },
    cpu_flags: RFlags(
      INTERRUPT_FLAG | SIGN_FLAG | 0x2,
    ),
    stack_pointer: VirtAddr(
      0xffffffff01001fff48,
    ),
    stack_segment: SegmentSelector {
      index: 0,
      rpl: Ring0,
    },
  },
  location: Location {
    file: "pkg/kernel/src/interrupt/exceptions.rs",
    line: 112,
    col: 5,
  },
}
```

- 预期使用的堆栈

[INFO] Double Fault IST : 0xffffffff00000180c8-0xffffffff00000190c8

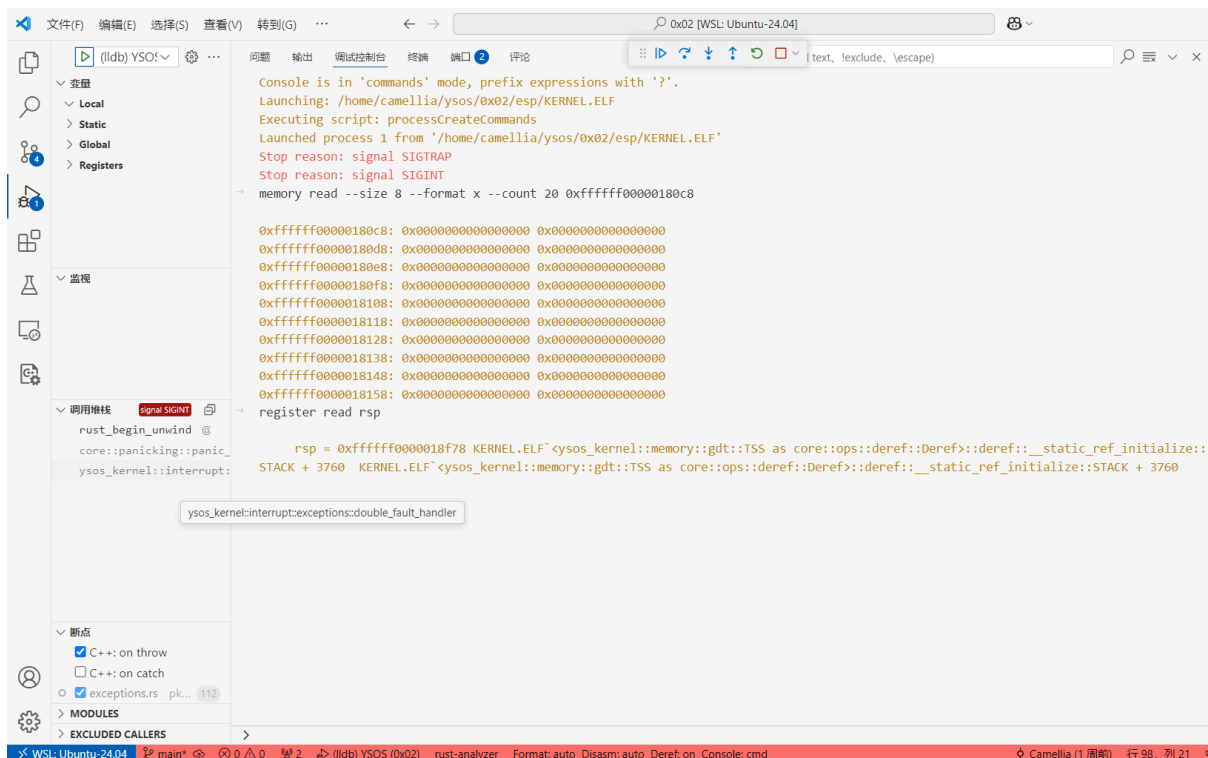
- gdb 调试结果
  - 断点设置: b ysos\_kernel::interrupt::exceptions::double\_fault\_handler

```
0xffffffff0000018ea8 +0x0000: 0xffffffff0000018f58 → 0xffffffff0000018fa0 → 0xffffffff0000014b00 → 0xffffffff00000107b → "EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x\n\nEXCEPTI[...]" ← $rsp
0xffffffff0000018eb0 +0x0008: 0xffffffff0000018ea8 → 0xffffffff0000018f58 → 0xffffffff0000018fa0 → 0xffffffff0000014b00 → 0xffffffff00000107b → "EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x\n\nEXCEPTI[...]"
0xffffffff0000018eb8 +0x0010: 0xffffffff0000044e0 → <&T as core::fmt::Debug>::fmt+0000 push rbp
0xffffffff0000018ec0 +0x0018: 0xffffffff0000000218 → "ysos_kernel::utils::macrosBKiBMiBGiBpkg/kernel/src[...]"
0xffffffff0000018ec8 +0x0020: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff0000018ed0 +0x0028: 0xffffffff0000000218 → "ysos_kernel::utils::macrosBKiBMiBGiBpkg/kernel/src[...]"
0xffffffff0000018ed8 +0x0030: 0x0000000000000000 → 0x0000000000000000 → [loop detected]
0xffffffff0000018ee0 +0x0038: 0xffffffff00000141d8 → 0xffffffff00000001eb → "pkg/kernel/src/utils/macros.rsERROR: panic!\n\nnyso[...]"
```

- 可以看出发生 DoubleFault 时使用的堆栈和预期一致，即 0xffffffff00000180c8-0xffffffff00000190c8

- CodeLLDB 调试结果

- 在触发断点时使用 `memory read --size 8 --format x --count 20 0xffffffff00000180c8` 命令查看栈内存，并使用 `register read rsp` 查看寄存器状态。



- 可以很直观的看出发生 `DoubleFault` 时调用的堆栈有 `double_fault_handler`，并且使用的栈地址和预期一致。
- 通过 `gdb` 和 `CodeLLDB` 都能很好的调试到发生 `DoubleFault` 时使用的栈地址，且都符合预期。

当访问 `0xdeadbeef` 无效地址时触发了异常 处理第一个异常时发生了第二个异常 CPU 自动切换到专用 IST 栈处理双重错误

### 5.3. 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。

- 代码与 5.2 保持一致，只是解除了 PageFault 处理函数的注释，随后执行 `python3 ysos.py run --intdbg`。得到如下结果：
  - 从图中可以看出成功触发了 PageFault, Cr2 寄存器的值正是我们先前设置的无效地址,这是因为当发生页错误 ( `Page Fault` ) 时, CPU 会将引起错误的线性地址存储在 CR2 寄存器中。

```
GDT= ffffffff0000017070 00000027
IDT= ffffffff0000081a350 00000fff
CR0=80010033 CR2=ffffffffdeadbeef CR3=0000000005c01000 CR4=00000668
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000000
DR6=00000000ffff0fff DR7=00000000000000400
CCS=0000000000000080 CCD=0000000000000000 CC0=EFLAGS
EFER=0000000000000d00
[ERROR] ERROR: panic!

PanicInfo {
  message: EXCEPTION: PAGE FAULT, ERROR_CODE: PageFaultErrorCode(CAUSED_BY_WRITE)

  Trying to access: 0xffffffffdeadbeef
  InterruptStackFrame {
    instruction_pointer: VirtAddr(
      0xffffffff000000405a,
    ),
    code_segment: SegmentSelector {
      index: 1,
      rpl: Ring0,
    },
    cpu_flags: RFlags(
      INTERRUPT_FLAG | SIGN_FLAG | 0x2,
    ),
    stack_pointer: VirtAddr(
      0xffffffff01001fff48,
    ),
    stack_segment: SegmentSelector {
      index: 0,
      rpl: Ring0,
    },
  },
}
```

- PageFault 通常属于可恢复的异常，是因为它不仅用于报告错误，更是现代操作系统实现虚拟内存（如按需加载页面、写时复制、内存交换）的关键机制。当 PageFault 代表的时这些合法的内存管理操作时，操作习堂内核的处理程序可以通过加载数据、分配内存、修改页表等方式来解决问题，并让程序无缝地继续执行。只有当页错误指示的是真正的程序错误（如访问非法地址或违反内存保护规则）时，它才变得不可恢复（从当前进程执行的角度看）。

## 6. 如果在 TSS 中为中断分配的栈空间不足, 会发生什么情况? 请分析 CPU 异常的发生过程, 并尝试回答什么时候会发生 Triple Fault。

### 6.1. TSS 和中断栈

在 x86 架构（特别是保护模式和长模式）中，TSS 扮演着关键角色，尤其是在中断或异常发生导致权限级别（Privilege Level）变化时（例如，从用户态 Ring 3 进入内核态 Ring 0）。

1. **栈切换**: 当一个中断或者异常发生，并且目标处理程序的权限级别（由 IDT 中的描述符指定）高于当前自行代码的权限级别时，CPU 需要切换到一个新的、属于更高级别的栈。
2. **TSS 提供栈指针**: TSS 结构中包含了指向不同权限级别栈顶的指针（例如，RSP0, RSP1, RSP2 用于长模式，或 SS0:ESP0, SS1:ESP1, SS2:ESP2 用于保护模式）。当需要切换到 Ring0 时，CPU 会从 TSS 中读取 RSP0（或 ESP:ESP0）的值，将其加载到 RSP（或 ESP）和 SS 寄存器中。作为新的栈指针。

3. **状态压栈**：在切换到新栈之后，CPU 会自动将一些重要的状态信息压入这个新栈，以便中断处理程序结束后能正确返回。这些信息通常包括：
- 原始（用户态）的栈指针
  - 原始的标志寄存器
  - 原始（用户态）的代码段选择子和指针指令
  - 某些异常还会压入一个错误码。

## 6.2. 栈空间不足时 CPU 异常的发生过程

假设一个中断发生，需要从 Ring 3 切换到 Ring 0，并且 TSS 中为 Ring 0 指定的栈（由 `RSP0` 指向）空间非常小，不足以容纳 CPU 要自动压入的所有状态信息。

1. 初始中断/异常发生：一个外部中断（如键盘中断）或内部异常（如除零错误）发生。CPU 查找 IDT，确定需要调用 Ring 0 的处理程序。
2. 栈切换尝试：CPU 从 TSS 加载 `RSP0` 的值到 `RSP`，准备使用内核栈。
3. 压栈操作：CPU 开始将上述状态信息（旧 SS, RSP, RFLAGS, CS, RIP, 可能还有错误码）依次压入由 `RSP0` 指向的内存区域。
4. 栈溢出：由于分配给 `RSP0` 的栈空间不足，当 CPU 尝试压入某个数据时（比如压入旧 CS:RIP 后，再压旧 RFLAGS 时空间就不够了），写操作会超出该栈内存区域的下限边界（栈向下增长）。
5. 栈段错误（Stack Segment Fault, #SS）：CPU 检测到写操作超出了栈段（Stack Segment）的界限。这会立即触发一个新的异常：#SS (Stack Segment Fault)，中断向量为 12。

### 6.2.1. 从 #SS 到双重故障（Double Fault, #DF）

现在 CPU 停止处理原始的中断/异常，转而尝试处理刚刚发生的 #SS 异常。

6. 尝试调用 #SS 处理程序：CPU 再次查询 IDT，找到 #SS 异常的处理程序描述符。#SS 通常也配置为在 Ring 0 执行。
7. 再次需要内核：因为 #SS 处理程序也在 Ring 0 运行，CPU 理论上应该已经在使用 Ring 0 的栈了（就是那个空间不足的栈）。但处理异常本身也需要压栈（至少要压入导致 #SS 的指令的 CS:RIP, RFLAGS 等）。
8. 再次栈溢出（或无法调用）：CPU 尝试为调用 #SS 处理程序而压栈。但它仍然在使用那个空间不足的 `RSP0` 栈！因此，尝试为 #SS 处理程序压栈时，很可能再次发生栈溢出。
9. 双重故障（Double Fault, #DF）：当 CPU 在尝试调用一个异常（如 #SS）的处理程序期间，发生了另一个异常（如再次发生 #SS 或 #GP - General Protection Fault，如果栈段描述符本身有问题），CPU 无法处理这种情况，于是它会放弃调用 #SS 处理程序，转而尝试调用预定义的 #DF (Double Fault) 处理程序，中断向量为 8。

### 6.2.2. 从 #DF 到三重故障（Triple Fault）

双重故障是处理严重错误的最后一道防线。#DF 处理程序的设计目标是捕获那些连普通异常处理都失败的情况。

10. **尝试调用 #DF 处理程序**：CPU 查询 IDT，找到 #DF 处理程序的描述符。根据 x86 架构规定，#DF 处理程序**必须**在 Ring 0 运行，并且**强烈推荐**为其在 TSS 中设置一个**独立且保证可用的栈**（虽然有时会复用 `RSP0`，但这很危险，如此例所示）。
11. **再次需要内核栈（可能是同一个）**：CPU 需要为调用 #DF 处理程序压栈（至少包括 EFLAGS, CS, EIP, #DF 的错误码固定为 0）。它会查找用于 #DF 的栈。如果 #DF 被配置为使用与普通 Ring 0 中断相同的 `RSP0` 栈（那个空间不足的栈），那么灾难将继续。
12. **第三次栈溢出（或无法调用）**：CPU 尝试将状态压入为 #DF 指定的栈。如果这个栈就是那个有问题的 `RSP0` 栈，或者 #DF 处理程序本身或其栈的设置（如 IDT 描述符、TSS 条目）存在问题（例如指向无效内存、权限错误），导致 CPU **无法成功调用 #DF 处理程序**。
13. **三重故障（Triple Fault）**：当 CPU 在尝试调用 **Double Fault（#DF）处理程序**期间，再次发生任何阻止 #DF 处理程序执行的异常时，CPU 就进入了 **Triple Fault** 状态。

### 6.2.3. 什么是三重故障（Triple Fault）？

- 三重故障不是一个有特定中断向量号的“异常”，而是一种 CPU 状态。
- 它表示 CPU 的异常处理机制已经彻底崩溃，无法再调用任何处理程序来应对当前的错误链。
- 发生三重故障时，CPU 通常会停止执行指令，并触发一个硬件机制，最常见的就是系统复位（System Reset）。这就是为什么这类底层错误会导致机器重启。

### 6.2.4. 总结发生三重故障的条件（与栈相关的情况）：

最典型的由 TSS 栈空间不足导致三重故障的链条是：

1. 初始中断/异常 -> 尝试切换到 Ring 0 栈（`RSP0`）-> 压栈时空间不足 -> **#SS 异常**。
2. 尝试调用 #SS 处理程序 -> 仍使用不足的 `RSP0` 栈 -> 压栈时再次空间不足 -> **#DF 异常**。
3. 尝试调用 #DF 处理程序 -> 如果仍使用那个不足的 `RSP0` 栈（或者 #DF 的 IDT/TSS 设置本身有问题）-> 压栈时第三次失败或无法调用 -> **Triple Fault** -> **系统重启**。

### 6.2.5. 其他可能导致三重故障的情况：

除了栈空间问题，其他破坏了异常处理机制基础的情况也可能导致三重故障，例如：

- IDT（中断描述符表）本身损坏或其指针（IDTR）无效。
- GDT（全局描述符表）损坏或其指针（GDTR）无效，导致无法加载中断处理程序所需的段描述符（代码段、数据段、TSS 段）。
- #DF 处理程序自身的代码段描述符、栈段描述符或 TSS 描述符无效或权限错误。
- 用于 #DF 处理程序的栈段本身无效或不存在。

总之，三重故障是 x86 异常处理机制的终点，表明系统遇到了无法通过软件（异常处理程序）解决的严重底层问题。

## 7. 在未使用 set\_stack\_index 函数时，中断处理程序的栈可能哪里？尝试结合 gdb 调试器，找到中断处理程序的栈，并验证你的猜想是否正确。

- 注释掉 PageFault 的异常处理函数，并注释掉 DoubleFault 的 set\_stack\_index 函数。重新编译内核 `python3 ysos.py build -p debug`，运行 `qemu python3 ysos.py run --debug`。
- gdb 调试得到以下结果：

```
0xffffffff1001ffd28|+0x0000: 0xffffffff1001ffd8 → 0xffffffff1001ffe20 → 0xffffffff0000014b00 → 0xffffffff000000107b → "EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x\n\nEXCEPTI[...]" ← $rsp
0xffffffff1001ffd30|+0x0008: 0xffffffff1001ffd28 → 0xffffffff1001ffd8 → 0xffffffff1001ffe20 → 0xffffffff0000014b00 → 0xffffffff000000107b → "EXCEPTION: DOUBLE FAULT, ERROR_CODE: 0x\n\nEXCEPTI[...]"
0xffffffff1001ffd38|+0x0010: 0xffffffff00000044e0 → <&T as core::fmt::Debug>::fmt+0000 push rbp
0xffffffff1001ffd40|+0x0018: 0xffffffff0000000218 → "ysos_kernel::utils::macrosBKiBMiB6iBpkg/kernel/src[...]"
0xffffffff1001ffd48|+0x0020: 0x0000000000000001a → 0x0000000000000000 → [loop detected]
0xffffffff1001ffd50|+0x0028: 0xffffffff0000000218 → "ysos_kernel::utils::macrosBKiBMiB6iBpkg/kernel/src[...]"
0xffffffff1001ffd58|+0x0030: 0x0000000000000001a → 0x0000000000000000 → [loop detected]
0xffffffff1001ffd60|+0x0038: 0xffffffff00000141d8 → 0xffffffff00000001eb → "pkg/kernel/src/utils/macros.rsERROR: panic!\n\nnyso[...]"
```

根据 x86\_64 的规则，当 IST 字段为 0 时，CPU 会切换到目标代码段特权级（Ring 0）对应的栈。但是并非我的日志中出现的特权栈的内存地址：

**[INFO ] Privilege Stack : 0xffffffff00000170c8-0xffffffff00000180c8**

注意：最终没有搞清楚上面调试中显示的栈地址是什么地址😓。

## 九. 加分项

### 1. 🤗 为全部可能的 CPU 异常设置对应的处理程序，使用 panic! 输出异常信息。

- 见前面代码。

### 2. 😊 你如何定义用于计数的 COUNTER，它能够做到线程安全吗？如果不能，如何修改？

```
static COUNTER: AtomicU64 = AtomicU64::new(0); 能做到线程安全。
```

- read\_counter() 函数使用了 COUNTER.load(Ordering::Relaxed)。
- inc\_counter() 函数使用了 COUNTER.fetch\_add(1, Ordering::Relaxed)。
- load 和 fetch\_add 是原子类型特有的方法，用于在多线程环境下安全地读取和修改值，防止数据竞争。Ordering 参数进一步指定了内存序，确保了操作的原子性。

### 3. 🤔 你的串口输入驱动是否能正确的处理中文甚至 emoji 输入? 如何能够正确处理?

- 目前的串口输入驱动仅支持 ASCII 字符集, 无法正确处理中文和 emoji 等多字节字符。

#### 3.1. 核心思想转变

从 基于字节 (Byte) 的处理转变为 基于字符 (Unicode Scalar Value / `char`) 的处理。

#### 3.2. 具体修改点

##### 3.2.1. 中断处理层 ( `pkg/kernel/src/interrupt/serial.rs` )

- 引入 UTF-8 解码缓冲:
  - 添加了 `static mut UTF8_BUF: [u8; 4]` 和 `static mut UTF8_LEN: usize`。用于暂存从串口接收的字节, 因为一个 UTF-8 字符可能由 1 到 4 个字节组成。
  - 之前的代码: 很可能直接将接收到的单个字节 `byte` 推入输入队列。
- 实现 UTF-8 解码逻辑:
  - 在 `receive` 函数的循环中, 不再直接推送字节, 而是:
    - 将接收到的 `byte` 存入 `UTF8_BUF`。
    - 使用 `core::str::from_utf8()` 尝试解码 `UTF8_BUF` 中当前积累的字节序列 (`current_bytes`)。
    - 成功 ( `Ok(s)` ): 表明已形成一个有效的 UTF-8 字符。提取这个 `char` (`s.chars().next().unwrap()`)。根据字符是普通字符、回车符 (`\r`) 还是退格符 (`\x08`, `\x7f`), 调用 `input` 模块相应的推送函数 (`push_char`, `push_newline`, `push_backspace`)。清空解码缓冲区 (`UTF8_LEN = 0`)。
    - 失败 - 不完整 ( `Err(e) where e.error_len().is_none()` ): 表明当前字节序列还不足以构成一个完整字符, 需要等待更多字节。继续循环。
    - 失败 - 无效 ( `Err(e) where e.error_len().is_some()` ): 表明接收到了无效的 UTF-8 字节序列。推送一个替代字符 `\u{FFFD}` (`input::push_char('\u{FFFD}')`) 并清空缓冲区。
- 调用更具体的输入函数:
  - 不再是统一调用 `input::push_key(byte)`, 而是根据解码结果调用 `input::push_char(c)`, `input::push_newline()`, 或 `input::push_backspace()`。

##### 3.2.2. 输入驱动层 ( `pkg/kernel/src/drivers/input.rs` )

- 定义语义化的输入类型 ( `InputKey` ):

- ▶ 用 `enum InputKey { Char(char), Backspace, NewLine }` 替换了之前的 `type Key = u8`。这使得输入缓冲区存储的不再是原始字节，而是具有明确含义的输入事件。`InputKey::Char` 可以直接容纳任何 Unicode 字符。
- 更新输入缓冲区类型:
  - ▶ `INPUT_BUF` 的类型从 `ArrayQueue<u8>` 改为 `ArrayQueue<InputKey>`。
- 提供专门的 Push 函数:
  - ▶ 用 `push_char(char)`，`push_backspace()`，`push_newline()` 替换了原来的 `push_key(u8)`，分别用于推送不同类型的 `InputKey`。
- 更新 Pop 函数:
  - ▶ `try_pop_key()` 和 `pop_key()` 现在返回 `Option<InputKey>` 和 `InputKey`。
- 修改 `get_line` 处理逻辑:
  - ▶ `pop_key()` 获取的是 `InputKey`。
  - ▶ 处理 `InputKey::Char(c)` :
    - 将 `char c` 直接 `push` 到 `line` 这个 `String` 中。`String` 在 Rust 中本身就是 UTF-8 编码的，可以正确存储中文和 emoji。
    - 回显 (Echo): 关键点 - 为了在终端上正确显示输入的字符，需要将 `char c` 重新编码为 UTF-8 字节序列 (`c.encode_utf8(&mut buf).as_bytes()`)，然后将这些字节逐个发送回串口 (`serial.send(*byte)`)。
    - 之前的代码: 可能只是简单地 `serial.send(key)` 发送原始字节。
  - ▶ 处理 `InputKey::Backspace` :
    - 调用 `line.pop()`。这个方法会从 `String` 的末尾移除一个完整的 `char`，无论它占多少字节，确保了对多字节字符的正确删除。
    - 调用 `serial.backspace()` 在终端上执行视觉删除。
    - 之前的代码: 可能只是简单地弹出最后一个字节。
  - ▶ 处理 `InputKey::Newline` : 触发换行并结束输入。

### 3.2.3. 总结

主要改动在于:

1. 在中断层增加了 **UTF-8 解码** 步骤，将原始字节流转换为 `char`。
2. 在输入驱动层将缓冲和处理的基本单位从 `u8` 升级为 `InputKey` 枚举（特别是 `InputKey::Char(char)`）。
3. 修改了 `get_line` 函数，使其能正确地追加 `char` 到 `String`、处理基于 `char` 的退格，以及将 `char` 重新编码为 UTF-8 字节序列进行回显。

重新编译内核输出日志，并进行串口输入输出测试，结果如下:



```
问题 1 输出 调试控制台 终端 端口 2 评论 python3 - 0x02

v0.2.0

[+] Serial Initialized.
[INFO ] Logger Initialized with level: info
[INFO ] Physical Offset : 0xfffff80000000000
[INFO ] Privilege Stack : 0xfffff000008193e0-0xfffff0000081a3e0
[INFO ] Double Fault IST : 0xfffff0000081a3e0-0xfffff0000081b3e0
[INFO ] Page Fault IST : 0xfffff0000081b3e0-0xfffff0000081c3e0
[INFO ] Kernel IST Size : 12.000 KiB
[INFO ] GDT Initialized.
[INFO ] Kernel Heap Size : 8.000 MiB
[INFO ] Kernel Heap Initialized.
[INFO ] APIC initialized.
[INFO ] Interrupts Initialized.
[INFO ] Physical Memory : 95.625 MiB
[INFO ] Free Usable Memory : 44.992 MiB
[INFO ] Frame Allocator initialized.
[INFO ] Interrupts Enabled.
[INFO ] YatSenOS initialized.
> Current tick count: 6
你好
You said: 你好
The counter value is 25484
> Current tick count: 25486
😁
You said: 😁
The counter value is 61946
> Current tick count: 61946
你好啊 😁
You said: 你好啊 😁
The counter value is 122905
> Current tick count: 122906
```

可以看出正确处理了中文和 emoji 的输入输出。