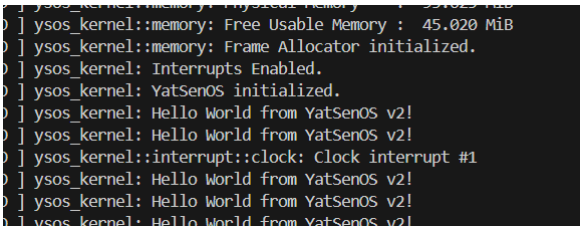
Lab2实验报告

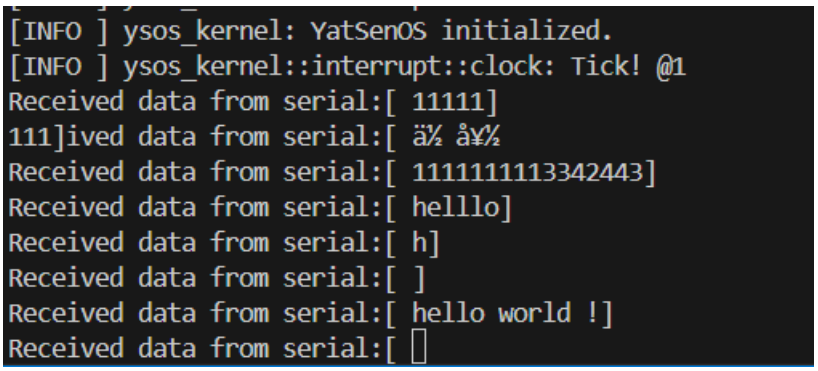
> 姓名： 谭栋泽
>
> 学号：23336011

实验结果

正确实现xapic，并且实现时钟中断后，第一次查看到信息



解决自旋锁错误占用导致info忽略的神秘问题（实验过程有），成功实现串口输入和中断显示



成功input用户输入，输入过程可以正确回显和退格

```
__ __      __  _____           ____  _____
\ \/ /___ _/ /_/ ___/___  ____ / __ \/ ___/
 \  / __ `/ __/\__ \/ _ \/ __ \/ / / /\__ \
 / / /_/ / /_ ___/ /  __/ / / / /_/ /___/ /
/_/\__,_/\__//____/\___/_/ /_/\____//____/

                                    v0.2.0
[+] Serial Initialized.
[INFO ] ysos_kernel::utils::logger: Logger Initialized.
[INFO ] ysos_kernel::memory::address: Physical Offset  : 0xffff800000000000
[INFO ] ysos_kernel::memory::gdt: Privilege Stack  : 0xffffff0000014398-0xffffff0000015398
[INFO ] ysos_kernel::memory::gdt: interrupt Stack  : 0xffffff0000015398-0xffffff0000016398
[INFO ] ysos_kernel::memory::gdt: Kernel IST Size  :  12.000 KiB
[INFO ] ysos_kernel::memory::gdt: GDT Initialized.
[INFO ] ysos_kernel::memory::allocator: Kernel Heap Size :   8.000 MiB
[INFO ] ysos_kernel::memory::allocator: Kernel Heap Initialized.
[INFO ] ysos_kernel::interrupt: Interrupts Initialized.
[INFO ] ysos_kernel::memory: Physical Memory    :  95.625 MiB
[INFO ] ysos_kernel::memory: Free Usable Memory :  45.016 MiB
[INFO ] ysos_kernel::memory: Frame Allocator initialized.
[INFO ] ysos_kernel: Interrupts Enabled.
[INFO ] ysos_kernel: YatSenOS initialized.
> 11111
You said: 11111
> hello yatos
You said: hello yatos
> exit
[INFO ] ysos_kernel: YatSenOS shutting down.
```

实验过程

> 一些实验中遇到的问题和思考

LVT向量操作

其实就是操作一大堆表示配置信息的二进制位，构造正确的配置， 写入正确的位置。

那就要一大堆二进制运算，本来我是打算这样实现的

```
bitflags! {
    /// LVT (Local Vector Table) flags
    struct LvtFlags: u32 {
        const MASKED = 1 << 16;       // 中断屏蔽位
        const LEVEL_TRIGGERED = 1 << 15;  // 电平触发(vs边沿触发)
        const ACTIVE_LOW = 1 << 13;  // 低电平有效(vs高电平)
        const DELIVERY_STATUS = 1 << 12;  // 传送状态
        const VECTOR = 0xFF;  // 中断向量号
    }
}
```

这部分参考代码是这样，感觉比它好一点，多一层封装

```
let mut lvt_timer = self.read(0x320);
// clear and set Vector
lvt_timer &= !(0xFF);
lvt_timer |= Interrupts::IrqBase as u32 + Irq::Timer as u32;
lvt_timer &= !(1 << 16); // clear Mask
lvt_timer |= 1 << 17; // set Timer Periodic Mode
self.write(0x320, lvt_timer);
```

但感觉用起来还是很奇怪，既然rust都零成本抽象了，我为什么一定要这样做呢，我开始尝试更面向对象一点。

我们直接定义一个编辑器

```rust
impl LVT_Editor {
    pub fn new() -> Self {
        LVT_Editor { data: 0 }
    }

    pub fn with_data(data: u32) -> Self {
        LVT_Editor { data }
    }

    pub fn bits(&self) -> u32 {
        self.data
    }

    // 终端屏蔽
    pub fn write_masked(mut self, masked: bool) -> Self {
        if masked {
            self.data |= LvtFlags::MASKED.bits();
        } else {
            self.data &= !LvtFlags::MASKED.bits();
        }
        self
    }

    // 设置电平触发
    pub fn write_level_triggered(mut self, level_triggered: bool) -> Self {
        if level_triggered {
            self.data |= LvtFlags::LEVEL_TRIGGERED.bits();
        } else {
            self.data &= !LvtFlags::LEVEL_TRIGGERED.bits();
        }
        self
    }

    // 设置低电平有效
    pub fn write_active_low(mut self, active_low: bool) -> Self {
        if active_low {
            self.data |= LvtFlags::ACTIVE_LOW.bits();
        } else {
            self.data &= !LvtFlags::ACTIVE_LOW.bits();
        }
        self
    }

    // 设置中断向量号
    pub fn write_vector(mut self, vector: u32) -> Self {
        self.data &= !LvtFlags::VECTOR.bits(); // 清除原有向量
        self.data |= vector & LvtFlags::VECTOR.bits(); // 设置新向量
        self
    }

```

```
}

```

那它用起来应该是这样的

```
self.write(APIC_LVT_TIMER, LVT_Editor::new()
                .write_masked(false) // 禁用计时器中断
                .write_level_triggered(false) // 边沿触发
                .write_active_low(false) // 高电平有效
                .write_vector(Interrupts::IrqBase as u32 + Irq::Timer as
u32)// 设置处理 set Timer Periodic Mode
                .write_bit(17, true)
                .bits());
```

哦很好，可读性比一堆位运算好太多，至少看起来这样。

你可以直观看到这三个版本的代码比较

```
// 3. 禁用逻辑中断线 (LINT0, LINT1)
self.write(reg: APIC_LVT_LINT0, value: LVT_Editor::new().write_masked(true).bits());
self.write(reg: APIC_LVT_LINT1, value: LvtFlags::MASKED.bits());
self.write(reg: 0x350, value: 1 << 16); // set Mask
```

我觉得第一种更好，更进一步，甚至可以把几个子类型都实现了， 这里只抽取公共部分抽象，所以仍然存在write_bit这种本身没有实际含义的操作。没有过分的抽象， 有需求可以友好重构，这是一种折中的编程方法。

时钟中断

这个地方由于多线程安全，必须用原子操作, 因为存在一个用于时钟计数的静态变量

```
extern "x86-interrupt" fn clock_interrupt_handler(_stack_frame:
InterruptStackFrame) {
    // 使用原子操作
    static COUNTER: AtomicUsize = AtomicUsize::new(0);

    let count = COUNTER.fetch_add(1, Ordering::Relaxed);

    // 偶尔输出一次
    if count % 1 == 0 {
        info!("Clock interrupt #{}", count + 1);
    }

    // 确认中断
    crate::interrupt::ack();
}
```

改成和教程差不多的结构

```
// 使用原子操作
static COUNTER: AtomicUsize = AtomicUsize::new(0);
extern "x86-interrupt" fn clock_interrupt_handler(_stack_frame:
InterruptStackFrame) {
```

```
    x86_64::instructions::interrupts::without_interrupts(|| {
        if inc_counter() % 0x10 == 0 {
            info!("Tick! @{}", read_counter());
        }
        super::ack();
    });
}


// FIX-ME
#[inline]
pub fn read_counter() -> usize {
    COUNTER.load(Ordering::Relaxed)
}

#[inline]
pub fn inc_counter() -> usize {
    COUNTER.fetch_add(1, Ordering::Relaxed)
}
```

串口中断

出现double fault错误， 初步认定是IDT设置错误， 然后发现忘记注册串口中断处理了

```
serial::register_idt(&mut idt);
```

当前的串口实现

```
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    idt[Interrupts::IrqBase as u8 + interrupt::Irq::Serial0 as u8]
        .set_handler_fn(serial_interrupt_handler);
}


extern "x86-interrupt" fn serial_interrupt_handler(_stack_frame:
InterruptStackFrame) {
    // 处理串口中断

    let mut serial = get_serial_for_sure();
    print!("Received data from serial:[ ");
    while let Some(byte) = serial.receive() {
        print!("{}", byte);
    }
    println!("]");

    // 确认中断
    crate::interrupt::ack();
}
```

很诡异，没有正确输出，尝试调试

哦！我突然醒悟，这个地方应该是正确中断了，也正确print了， 但是print的锁被我占用了， 它获取不到锁，自动忽略了输出，导致最后什么也没有发生。

找到原因就很容易解决了， 利用rust的语法特性？把用到锁的位置包裹起来， 等用完会被自动回收。

```
extern "x86-interrupt" fn serial_interrupt_handler(_stack_frame: InterruptStackFrame) {
    // 处理串口中断
    x86_64::instructions::interrupts::without_interrupts(|| {

        print!("Received data from serial:[ ");
        loop
        {
            let result: Option<u8>;
            {
                let mut serial: MutexGuard<'_, SerialPort> = get_serial_for_sure();
                result = serial.receive();
            }
            if let Some(byte: u8) = result {
                print!("{}", byte as char);
            } else {
                break;
            }
        }
        println!("]");
        super::ack();
    });
}
```

那只后要写的东西就比较显然了， 缓冲区缓冲一下。

```
pub extern "x86-interrupt" fn serial_interrupt_handler(_st:
InterruptStackFrame) {
    receive();
    super::ack();
}

/// Receive character from uart 16550
/// Should be called on every interrupt
fn receive() {
    // FIX-ME: receive character from uart 16550, put it into INPUT_BUFFER
    let result;
    {
        let mut serial = get_serial_for_sure();
        result = serial.receive();
    }
    if let Some(key) = result {
        input::push_key(key);
        print!("{}", key as char);
    } else {
    }
}
```

这样还不能正确回显退格，改进一下

我们在这里处理， 又直接回到串口确实有点不太好了

```rust
pub fn push_key(key: Key) {
    if INPUT_BUF.push(key).is_err() {
        warn!("Input buffer is full. Dropping key '{:?}'", key);
    }else{
        match key {
            b'\n' => {
                if let Some(mut serial) = get_serial() {
                    serial.send(b'\n');
                }}
            b'\r' => {
                if let Some(mut serial) = get_serial() {
                    serial.send(b'\n');
                }}
            // 退格键 (Backspace: 0x08, Delete: 0x7F)
            0x08 | 0x7F => {
                    if let Some(mut serial) = get_serial() {
                        serial.backspace();
                    }}
            key if key >= 32 && key <= 126 => {
                if let Some(mut serial) = get_serial() {
                    serial.send(key);
                }
            }
            _ => {

            }
        }
        // println!("Key pushed: {}", key as char);
    }
}
```

非常好，可以正确回显，包括退格和换行， getline也应该可以正常接入

接下来把用户交互的代码复制进去，就可以正常运行。

思考题

> **为什么需要在 clock_handler 中使用 without_interrupts函数?**

如果在处理时钟中断时又发生了其他中断，可能导致栈溢出、数据竞争或死锁。这确保在关键代码段执行期间临时禁用中断，避免这些问题。

> **max_phys_addr 是如何计算的，为什么要这么做?**

max_phys_addr通过遍历 UEFI 内存映射表中的所有内存区域，计算每个区域的结束地址（起始地址 + 页数 * 页大小），然后取最大值。代码中 `.max(0x1_0000_0000)` 确保至少包含 4GB 以上的地址空间，这是为了覆盖 IOAPIC 等 MMIO 设备的内存映射区域。这样做是为了确定系统的物理内存范围，以便后续建立虚拟内存映射时能够正确映射所有可用的物理内存和设备地址空间。

> **串口驱动启用前的输出是如何实现的?**

在进入内核之前，输出是通过 UEFI 固件提供的服务实现的。在 bootloader 阶段，代码使用了 info! 宏和 UEFI 的标准输出服务来显示信息。UEFI 固件本身提供了控制台输出功能，可以直接向屏幕输出文本。只有在退出 UEFI boot services 并进入内核后，才需要自己实现串口驱动来处理输出。

> **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。**

在APIC里面的寄存器设置。太快可能影响性能，毕竟每次都要系统中断，太慢会导致调度的时候粒度不够，因为最快也是每一个时钟调度一次， 太慢可能导致每个程序运行时间不均匀。

> **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。**

可以参考前面实验过程中，INFO丢失的问题， 我可能没有完全按实验文档来做，所以提前遇到了这个问题。串口的主要部分是一个缓冲去，不能混用，所以有锁的存在，不能同时获得一个互斥锁的写权限，rust中有借用的概念我觉得很像（同时要么有一个写，要么没有写只有多个读入）。我的教训是：少用临时变量，用方法直接替换对应的变量，不知道是否合理。（rust不是零成本抽象来着，不知道这还算不算）

> 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？

两边速度差太大，或者一边停止了，满了要阻塞，不阻塞只能丢弃数据了。