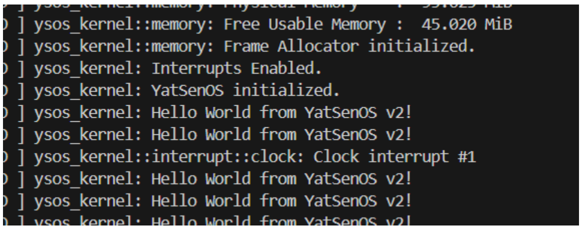
Lab2实验报告

> 姓名： 谭栋泽
>
> 学号：23336011

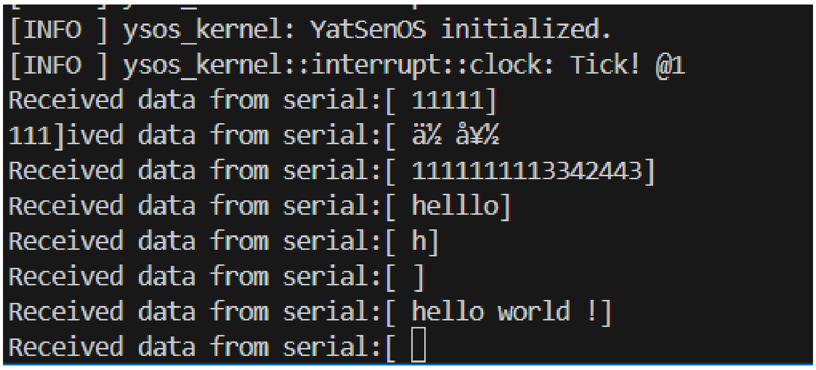
实验结果

正确实现xapic，并且实现时钟中断后，第一次查看到信息



```
] ysos_kernel::memory: Free Usable Memory :  45.020 MiB
] ysos_kernel::memory: Frame Allocator initialized.
] ysos_kernel: Interrupts Enabled.
] ysos_kernel: YatSenOS initialized.
] ysos_kernel: Hello World from YatSenOS v2!
] ysos_kernel: Hello World from YatSenOS v2!
] ysos_kernel::interrupt::clock: Clock interrupt #1
] ysos_kernel: Hello World from YatSenOS v2!
] ysos_kernel: Hello World from YatSenOS v2!
```

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



```
[INFO ] ysos_kernel: YatSenOS initialized.
[INFO ] ysos_kernel::interrupt::clock: Tick! @1
Received data from serial:[ 11111]
111]ived data from serial:[ ä½ å¥½
Received data from serial:[ 1111111113342443]
Received data from serial:[ helllo]
Received data from serial:[ h]
Received data from serial:[ ]
Received data from serial:[ hello world !]
Received data from serial:[ 
```

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

```rust
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

```rust
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) {
```

```rust
    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

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

当前的串口实现

```rust
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

```
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。