






# Rucore 课程设计报告

g15 杨国炜 乔一凡

# 课程设计目标

- 使用 Rust 重新实现 ucore，目标平台为 x86\_64
- 为移植好的 ucore 适配相应的驱动，包括如下几部分：
  - 存储设备驱动，ATA/SATA
  - 显示设备驱动，VGA图形驱动
  - PS2 键盘/鼠标驱动
- 在 64 位 ucore 上实现可加载内核模块（LKM），并实现驱动的模块化

# 完成工作

- 使用 Rust 重新实现 ucore, 目标平台为 x86\_64 
- 为移植好的 ucore 适配相应的驱动, 包括如下几部分:
  - 存储设备驱动, ATA/SATA 
  - 显示设备驱动, VGA图形驱动 
  - PS2 键盘/鼠标驱动 
- 在 64 位 ucore 上实现可加载内核模块 (LKM) , 并实现驱动的模块化 

# 支持

- 使用 Rust 在 x86\_64 下重新实现 ucore
- 启动并进入长模式；
- 底层驱动支持，VGA 字符显示，串口
- 内存管理
- 内核进程创建，Round Robin 调度
- 文件系统
- IDE 硬盘驱动，PS2 键盘鼠标驱动

# 不支持

- 用户进程
- VGA 图形驱动
- LKM 内核可加载模块

# 底层驱动支持

- PIT（计时器芯片），PIC（中断控制器），串口驱动直接使用 Redox 的驱动
- ACPI（高级配置和电源管理接口）驱动：要求能够检测到 RSDP，并根据 RSDP 找到 RSDT(Root System Description Table) 和 XSDT（如果 ACPI version  $\geq$  2.0），并逐项获取 RSDT 内容
- 我们参考了 Redox 的 ACPI 驱动实现，但是过于复杂，有很多我们不需要的东西：如 AML

# ACPI

- 我们参考了 Redox 的 ACPI 驱动实现，但是过于复杂，有很多我们不需要的东西：
- 如 AML，仅在 DSDT 和 SSDT 表中存在，但是我们只需要获得 FADT 和 Local APIC，IO APIC
- 其中 FADT 用于获取 i8042 芯片存在信息

# 物理内存管理

- 最原始的框架只支持每次分配一个物理页帧，且不支持释放；
- 改进算法，在遍历连续空闲空间时判断空间是否足够，实现  $n$  个帧的分配方法；
- 对于页帧释放的问题，参考 Redox 的实现，增加一个 Recycler



# 物理内存管理

- 采用装饰器模式，使用 Recycler 装饰最初的 Allocator，负责回收并重新分配页帧
- 当释放页帧时 Recycler 会回收并记录这些页帧；
- 分配页帧时先由 Recycler 尝试分配，无法分配后才交给内部 Allocator 进行分配

# 虚拟内存管理

- 对于 x86\_64 建立 4 级页表
- 采用自映射机制实现对页表的修改，以及页表的切换
- 段机制的设置：设置 GDT 时我使用了一个外部 crate，将 GNULL 段描述符放在了第一个表项处，结果所有的段都偏了一位

# 中断处理

- 主要建立 IDT 表以及设置相应中断处理例程
- 开始时使用了一个外部库，方便但是封装的很好限制很多，无法控制
- 最后我们参考了 G11 组王润基的实现，实现了类似 ucore 的中断分发机制和 IDT 表

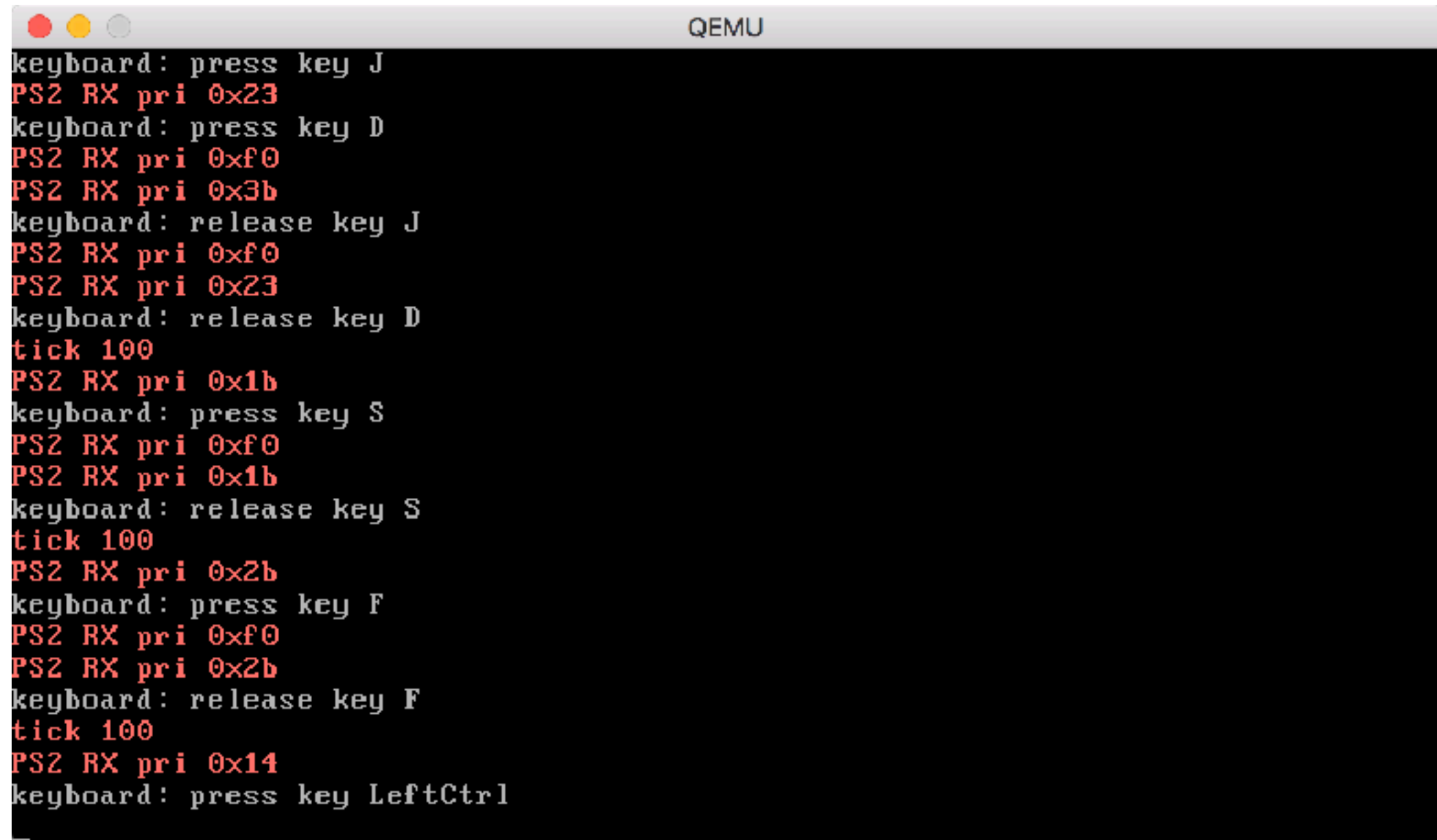
# PS2 驱动

- 主要分为两步，首先要完成 i8042 芯片的驱动
- i8042 有两个 port，分别对应键盘和鼠标
- 初始化的时候使用 inb/outb 按照协议进行初始化，确认 port 的工作状态

# 鼠标键盘驱动

- 鼠标键盘有多种标准设备
- 采用模块化设计方法，使用 PS2Dev 抽象设备
- port 会根据收到的数据判断外设的具体类型并动态设置设备
- 采用状态机进行请求的响应

# 运行结果



```
QEMU
keyboard: press key J
PS2 RX pri 0x23
keyboard: press key D
PS2 RX pri 0xf0
PS2 RX pri 0x3b
keyboard: release key J
PS2 RX pri 0xf0
PS2 RX pri 0x23
keyboard: release key D
tick 100
PS2 RX pri 0x1b
keyboard: press key S
PS2 RX pri 0xf0
PS2 RX pri 0x1b
keyboard: release key S
tick 100
PS2 RX pri 0x2b
keyboard: press key F
PS2 RX pri 0xf0
PS2 RX pri 0x2b
keyboard: release key F
tick 100
PS2 RX pri 0x14
keyboard: press key LeftCtrl
```

# 进程管理

- 实现进程控制块，trapframe
- 实现内核进程创建
- 实现一个简单的调度器，RR
- 用户态没有成功
  - rip没有改变
  - 页表切换，新页表不能工作

# IDE 驱动

- 参考ucore实现了ide硬盘驱动
- 能实现将磁盘某位位置连续的n个扇区大小的数据读入到dst数组中，同时能将dst数组写入到磁盘某位位置后连续的地址中
- 在实现中rust的x86\_64::port提供了如inb、outb等函数，因此相较使用c实现更加简单



# 文件系统

- 参考rust\_os及ucore完成了简单的文件系统
- rust\_os实现的是ramfs，ramfs是一种基于RAM做存储的文件系统
- ramfs的实现就相当于把RAM作为最后一层的存储，所以在ramfs中不会使用swap。因此ramfs有一个很大的缺陷就是它会吃光系统所有的内存，同时它也只能被root用户访问

# 文件系统

- 在文件系统等代码中我们大量用到了枚举类型enum，该类型相较于c的enum的优点是，rust的enum不同元素可以为不同的类型，而c的enum只能是数字

```
pub enum NodeType<'a>{  
    File, //常规文件类型  
    Dir, //文件夹类型  
    Symlink(&'a super::Path), //链接类型，允许读取链接内容  
}
```

# 文件系统

- 对三种不同类型文件的基本操作包括

File

```
pub trait File: NodeBase {  
    /// 返回此文件的大小（以字节为单位）  
    fn size(&self) -> u64;  
    /// 更新文件的大小（零填充或截断）  
    fn truncate(&self, newsize: u64) -> Result<u64>;  
    /// 清除文件的指定范围（用零替换）  
    fn clear(&self, ofs: u64, size: u64) -> Result<()>;  
    /// 从文件中读取数据  
    fn read(&self, ofs: u64, buf: &mut [u32]) -> Result<usize>;  
    /// 将数据写入文件  
    fn write(&mut self, id: InodeId, buf: &[u32]) -> Result<usize>;  
}
```

# 文件系统

Dir

```
pub trait Dir: NodeBase {  
    /// 获取给定名称的节点  
    fn lookup(&self, name: &ByteStr) -> Result<InodeId>;  
  
    /// 读取条目  
    /// 返回:  
    /// - Ok(Next Offset)  
    /// - Err(e): 错误  
    fn read(&self, start_ofs: usize, callback: &mut ReadDirCallback) -> Result<usize>;  
  
    /// 在该目录下创建一个新文件, 返回新创建的节点编号  
    fn create(&self, name: &ByteStr, nodetype: NodeType) -> Result<InodeId>;  
}
```

# 文件系统

Symlink

```
pub trait Symlink: NodeBase {  
    /// 将符号链接的内容读入一个字符串  
    fn read(&self) -> ByteString;  
}
```

# 文件系统

- 内存索引节点结构，描述了文件的inode等信息，用于引用计数、同步互斥等操作

```
pub struct CacheHandle{
    mountpt: usize, //挂载点编号
    inode: InodeId, //inode编号
    ptr: *const CachedNode,
}

struct CachedNode{
    refcount: AtomicUsize, //引用计数
    node: CacheNodeInt, //inode, 用枚举类型表示, 有3种不同的inode
}
```

# 总结

- 总结来说，使用 rust 写 os 是一个痛苦而又有收获的过程。首先，由于我们对 rust 本身不够熟悉，同时 rust 本身编译要求十分严格，经常出现死活编译不过的情况；另一方面，对 rust 的不熟悉也使我们对一些 rust 的高级特性不够了解，降低了开发效率的同时也一定程度上损失了 rust 提供的高等级安全保证的特性。

# 总结

- rust 本身对于资源申请和使用所有权的严格要求也大大降低了代码出错的概率；可能的不安全代码段会强制程序员使用 `unsafe` 进行声明；对于权限和可见性的更加严格的要求，等等语言特性在编译器就通过严格的约束帮助程序员及时发现 bug，降低 debug 成本



# 总结

- rust 作为比 C 更加高级的语言，在语言层面提供了更多描述能力更强的特性和实现。同时，rust 的包管理器 cargo 可以让我们更方便地指定工程使用的外部库以及版本，从而可以方便地利用各种现成的轮子。如在实现中我们大量使用的 `spin::Mutex` 等

**谢谢大家！**