

北京理工大学
计算机学院
毕 业 设 计 （ 论 文 ）
中 期 报 告

选题名称 计算机系统综合实验集成和改进

学 号 1120205029

姓 名 李 雯

专 业 计算机科学与技术（第二学位）

班 级 07712001

导 师 陆慧梅、向勇

学 院 计算机学院

研究所（系、中心）

2022 年 4 月 12 日

一 毕业设计（论文）进展情况

（一）课题相关背景与目的

2006 年，MIT 的 Frans Kaashoek 等人参考 PDP-11 上的 UNIX Version 6 写了一个可在 x86 指令集架构上运行的操作系统 xv6（基于 MIT License）。

2010 年，清华大学操作系统教学团队参考 MIT 的教学操作系统 xv6，开发了在 x86 指令集架构上运行的操作系统 ucore，多年来作为操作系统课程的实验框架使用，已经成为了大家口中的“祖传实验”。

而近年来，随着 riscv 指令集的完善和发展以及开源优势，国内外各大高校的教学操作系统都有向 risc 指令集移植的趋势。清华大学操作系统教学团队的基于 [riscv 的 uCore 64bit](#) 操作系统，MIT 的基于 riscv 的 [MIT-6.S081-2020 实验（xv6-riscv64）](#)，以及清华大学操作系统教学团队的基于 riscv 的 rCore（Rust、riscv 汇编）。

对于初学者来说，Rust 语言并不友好，因此针对对 Rust 语言和操作系统感兴趣的同学来说，用 Rust 写操作系统有些吃力。作为 uCore 和 rCore 之间的过渡，我们打算在看懂 [rCore-Tutorial-Book-v3](#) 实验教程的基础上，用 C 语言照着实现一遍，虽然说是“照猫画虎”，目前存在很多不足之处，边实验边记录边修改。

（二）毕设要求与指标

rCore 教学操作系统是用 Rust 语言实现的具备 批处理、分时多任务、内存管理、进程管理、文件系统、进程间通信、I/O 重定向、并发处理等功能完备而微小的 RISC-V 64 位操作系统，拟打算在读懂 Rust 代码的基础上用 C 语言“照猫画虎”实现 rCore，应至少完成五个实验的修改移植，撰写相应的实验指导书。

（三）目前进度

截止到 4.12 日，完成 rCore 教学操作系统实验（C 改写）的实验零（RV64 裸机应用实验）、实验一（简单批处理）、实验二（分时多任务），以及实验三（内存管理）的部分内容，并撰写了相应章节的实验指导书。

二 取得成果和存在问题

(一) 已完成的研究成果

实验零 RV64 裸机应用

假设我们拥有一台模拟计算机，包含：CPU、存储设备（包含断电保存数据的内存 和 断电遗失数据的硬盘）、输入输出设备、总线等，那么如何在这台模拟设备上启动操作系统呢？操作系统内核的内存布局又是怎样设置的呢？这是实验零完成的内容。这一章节我们主要完成实验环境部署以及内核启动。首先是关于环境部署，按照实验指导书依次安装虚拟环境（ubuntu）、开发工具（编译器等，会涉及到环境变量的添加）、硬件模拟器（qemu）和调试工具。

这一基础实验主要包含的代码有：entry.S、linker script、os.c、printf.c、makefile、rustsbi-qemu.bin 等。其中 **entry.S** 汇编文件指定了操作系统内核的 `_start` 入口点，栈顶栈底 以及栈的空间大小。**linker script** 则指定了输出文件的架构（riscv）、程序入口点、各段数据存放的位置（如：代码段、只读数据段、可读写数据段等）。**os.c** 文件声明并调用了 sbi 接口（`putchar` 和 `shutdown`），并最终实现 `main` 函数——在模拟器上打印“Hello World!”。由于我们是从零开始实现操作系统，也就是 不能依赖标准库函数，因此借鉴 MIT xv6 的操作系统实验 实现 `printf` 函数。最后是 **makefile** 文件，其中第二行的 `-ffreestanding` 的意思是“允许重新定义标准库里已经有的函数”，比如我自己定义了一个 `printf` 函数（主要内容是从 xv6 借鉴的，这里就显现出 Rust 的好了，Rust 的格式化输出是定义在语言内部的，只需要重写字符串的输出方式，C 的整个格式化都得重写），不加这个编译选项就会报错。在编译完后，需要用 `objcopy` 把程序的 elf 元信息去掉，因为裸机只能理解代码，不能解析 elf 格式，经过 `objcopy` 后整个文件上来就是二进制代码，裸机可以直接执行。

代码树和（代码目录下在终端 `make`）运行结果如下：

├─ entry.S (定义程序入口点_start)

├─ linker.ld (链接脚本)

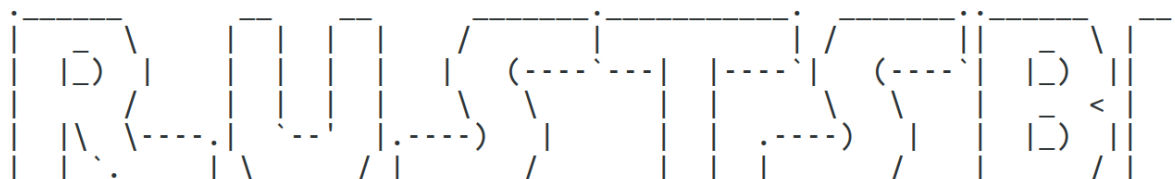
├─ makefile (告诉make工具如何编译和链接程序)

├─ os.c (调用sbi_putchar和sbi_shutdown)

├─ printf.c (直接用的函数参考xv6-riscv)

└─ rustsbi-qemu.bin (可运行在 qemu 虚拟机上的RustSBI预编译二进制版本)

```
[rustsbi] RustSBI version 0.1.1
```



```
[rustsbi] Platform: QEMU (Version 0.1.0)
```

```
[rustsbi] misa: RV64ACDFIMSU
```

```
[rustsbi] mideleg: 0x222
```

```
[rustsbi] medeleg: 0xb1ab
```

```
[rustsbi-dtb] Hart count: cluster0 with 1 cores
```

```
[rustsbi] Kernel entry: 0x80200000
```

```
Hello, world!
```

实验一 简单批处理

这个实验中我们拟打算实现简单批处理，也就是 能输入好几个程序，然后对程序依次执行的操作系统。要求用户程序与系统有隔离，需要实现用户态和内核态的切换。这一部分的代码主要分成两部分（用户态和内核态），大概说一下功能。

用户态文件包括 `hello_world.c`、`lib.c`、`user.h`、`linker.ld`：

`lib.c` 是包含用户程序的入口函数，在函数中执行 `main` 函数，然后调用 `exit()` 系统调用退出，另外还包含了 清零 `bss` 段 和 系统调用函数 在用户态的实现（即设置对应寄存器然后 `ecall` 的过程）。

`linker.ld` 是 用户态链接脚本，在其中我们做的事情是：将程序的起始物理地址调整为 `0x80400000`，应用程序会被加载到这个物理地址上运行；将 `_start` 所在的 `.text.entry` 放在整个程序的开头，也就是说批处理系统只要在加载之后跳转到 `0x80400000` 就已经进入了 用户库的入口点，并会在初始化之后跳转到应用程序主逻辑；提供了最终生成可执行文件的 `.bss` 段的起始和终止地址，方便 `clear_bss` 函数使用。

`user.h` 则包含一些类型、宏定义和函数头。

内核态文件则包括 `batch.c`、`entry.S`、`kernel.h`、`sbicall.c`、`syscall.c`、`link_app.S`、`printf.c`、`mod.c`、`trap.S` 等：

`batch.c` 是批处理系统实现的主要程序，其中指定了最大可顺序执行的应用程序数量、应用程序装载入口、用户栈空间、内核栈空间。这个程序里包含的方法主要有 `load_all`、`load_app`、`app_init_context`、`run_next_app`、`trap_handler` 等。

`sbicall.h` 包含了对 SBI 的调用，和第零章中的 `os.c` 一样。

`syscall.c` 则包含了对写系统调用、退出系统调用的实际处理。

`trap.S` 包含了进出内核态时保存现场的汇编代码。

`mod.c` 包含了对 `trap` 的处理，如果是系统调用则执行 `syscall.c` 里那些函数，否则说明是其他异常，直接 `panic`。

`entry.S` 和实验零一样，是内核的入口点，此时的程序入口点是 `load_all`。

`kernel.h` 包含一些类型、宏定义和函数头。

`printf.c` 和实验一一样，实现了输出函数和 `panic` 函数。

本章节目目前实现的功能是系统调用 写、退出函数，依次执行用户程序（即批处理）。

[rustsbi] RustSBI version 0.1.1



[rustsbi] Platform: QEMU (Version 0.1.0)

[rustsbi] misa: RV64ACDFIMSU

[rustsbi] mideleg: 0x222

[rustsbi] medeleg: 0xb1ab

[rustsbi-dtb] Hart count: cluster0 with 1 cores

[rustsbi] Kernel entry: 0x80200000

Hello World from User Mode program!

Application exited with code 0

Hello World from User Mode program!

Application exited with code 0

Hello World from User Mode program!

Application exited with code 0

Hello World from User Mode program!

Application exited with code 0

实验二 分时多任务

这一实验要实现的是一个分时多任务的系统，也就是能够在多个任务运行期间进行切换，让一个程序在等待 IO 时其他程序能执行而不是傻等。一个程序的一次完整执行过程称为一次 **任务** (Task)，把一个程序在一个 **时间片** (Time Slice) 上占用处理器执行的过程称为一个 **任务片** (Task Slice)。操作系统对不同程序的执行过程中的 **任务片** 进行调度和管理，即通过平衡各个程序在整个时间段上的任务片数量，就能达到一定程度的系统公平和高效的系统效率。在一个包含多个时间片的时间段上，会有属于不同程序的多个任务片在轮流占用处理器执行，这样的操作系统就是支持 **分时多任务** 或 **抢占式多任务** 的抢占式操作系统。

简单说，除了前面我们已经有的东西外，我们还需要实现的东西有：**定时器**、计时、触发、**任务调度**、**任务的内存布局**。在 RISC-V 64 处理器中，天然拥有一个频率略低于 CPU 的计时器，我们只需要利用 sbi 接口调用相应的控制状态寄存器 (CSR) 即可。

sbicall.c 中增加了 set_timer 调用，它是一个由 SEE 提供的标准 SBI 接口函数，它可以用来设置 mtimecmp 的值。在 RISC-V 64 架构中，有一个用来保存计数器的 64 位的 CSR mtime；另一个 64 位的 CSR mtimecmp 的作用是：一旦计数器 mtime 的值超过了 mtimecmp，就会触发一次时钟中断。这样就可以方便的通过设置 mtimecmp 的值来决定下一次时钟中断何时触发。

timer.c 中的关键函数是 get_time (读取时间寄存器)、set_t_next_trigger (计时触发)、get_time_ms (计时：统计应用的运行时长)。

loader.c 中关键函数 task_init (初始化)、__restore (保存 TrapContext 和 TaskContext task.c 将记录应用运行状态 TaskStatus，任务控制块 TaskControlBlock。

switch.S 控制任务切换 (涉及指针操作)。

以及应用 task0.c、task1.c、task2.c，分别实现每 10ms\20ms\30ms 循环打印，相应的 linker0.ld 将定向加载应用的内存布局。

- ├─ batch.c (和第一个实验一样)
- ├─ entry.S (指定程序入口点、栈空间)
- ├─ kernel.h (专门放宏定义、类型、方法)
- ├─ lib.c (和批处理实验中的os.c)
- ├─ link_app.S ()
- ├─ linkers.ld (内核内存布局)
- ├─ linker0.ld (应用程序的内存布局)
- ├─ linker1.ld
- ├─ linker2.ld
- ├─ linkeru.ld
- ├─ **loader.c** (新增：将应用加载到内存并进行管理 TrapContext和TaskContext)
- ├─ makefile
- ├─ mod.c (修改：时钟中断相应处理)
- ├─ printf.c (参考xv6的printf函数)
- ├─ rustsbi-qemu.bin
- ├─ sbicall.c (修改：引入新的 sbi call set_timer)
- ├─ **switch.S** (控制任务切换)
- ├─ syscall.c (修改：含四个系统调用函数 写、退出、暂停、定时器)
- ├─ task0.c (每 10ms 循环打印，打印10次)
- ├─ task1.c (每 20ms 循环打印，打印5次)
- ├─ task2.c (每 30ms 循环打印，打印3次)
- ├─ **task.c** (新增：task 子模块，主要负责任务管理)
- ├─ timer.c (新增：计时器相关)
- ├─ trap.S
- └─ user.h (新增：宏定义yield、get_time)

```
[rustsbi] RustSBI version 0.1.1
```

```

┌───────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐
│  ( )  \    │ │  ( )  \    │ │  ( )  \    │ │  ( )  \    │
│  \    \    │ │  \    \    │ │  \    \    │ │  \    \    │
│  \    \    │ │  \    \    │ │  \    \    │ │  \    \    │
└───────────┘ └───────────┘ └───────────┘ └───────────┘

```

```
[rustsbi] Platform: QEMU (Version 0.1.0)
```

```
[rustsbi] misa: RV64ACDFIMSU
```

```
[rustsbi] mideleg: 0x222
```

```
[rustsbi] medeleg: 0xb1ab
```

```
[rustsbi-dtb] Hart count: cluster0 with 1 cores
```

```
[rustsbi] Kernel entry: 0x80200000
```

```
Task[0] print0
```

```
Task[1] print0
```

```
Task[2] print0
```

```
Task[0] print1
```

```
Task[1] print1
```

```
Task[2] print1
```

```
Task[0] print2
```

```
Task[0] print3
```

```
Task[1] print2
```

```
Task[2] print2
```

```
Task[0] print4
```

```
Task[1] print3
```

```
Application has done and exited. 0
```

```
Task[0] print5
```

```
Task[1] print4
```

```
Task[0] print6
```

```
Task[0] print7
```

```
Application has done and exited. 0
```

```
Task[0] print8
```

```
Task[0] print9
```

```
Application has done and exited. 0
```

实验三 内存管理

操作系统的内存管理，核心是**虚拟地址**和**物理地址**空间的管理。

将涉及到：动态内存分配支持、物理页帧管理、虚拟地址管理、地址空间管理和分时管理的融合。

（二）当前研究存在的问题

中断处理 Trap（包括硬件中断、软件中断、时钟中断）只实现了时钟中断，软件中断还在探索中。

异常捕获和处理，缺少相应的应用设计，例如某个用户态应用程序访问了非法地址等。

三 导师意见

导师对中期报告的审阅意见

指导教师签字：

2022 年 4 月 12 日