



基于RISC-V的操作系统实验集成与改进

答辩人：李 雯

专 业：计算机科学与技术（二学位）

导 师：陆慧梅

日 期：2022/6/9

德以明理
学以精工

查重及盲审结果

查重结果	盲审1	盲审2	是否申请评优
2.7%	A（85）	B（77）	否

计算机学院本科毕业设计（论文）形式审查表（一）

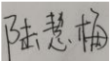
学生姓名：李 雯

学号：1120205029

电话：15510085083

导师初审

导师签字：



论文封皮	√	答辩委员会名单及签字		正文格式	√
任务书首页	√	提问及回答		不使用人称代词	√
题目类别	√	代表签字		字体字号	√
题目性质	√	答辩评语		图表正确编号 X-Y	√
论文题目	√	答辩委员签字		图表正确引用	√
题目内容	√	成绩及时间		单页空白少于 3 行	√
任务要求	√	委员会主任签字		页眉页脚页码	√
具体内容	√	论文起止时间	√	参考文献引用	√
进度安排	√	答辩日期		参考文献≥20 篇	√
指导教师签字	√	文字复制比<20%	√	外文翻译原文	√
教学单位签字		中文摘要	√	翻译中文及页眉	√
责任教授签字		英文摘要	√	翻译封皮	√
指导教师评语	√	关键词	√	形式审查表	√
匿名评语 1	√	目录及格式	√	软件验收表	
匿名评语 2	√	正文前各页页码(罗马)	√	资料袋	

注：√表示该项核查后符合要求，X 表示该项核查不符合要求，灰底色的暂不检查



CONTENTS

- 1 选题背景及研究意义
- 2 研究方法及过程
- 3 运行环境及实验平台
- 4 研究内容及工作量
- 5 总结与展望



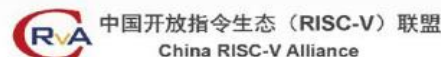
Part 1

选题背景及研究意义

选题背景

➤ RISC-V指令集体系 (ISA) 的诸多优势、迅速发展

- 传统指令集架构：X86、ARM、MIPS、SPARC（设计愈加复杂，涉及知识产权）
- 2010年加州大学伯克利分校 David Patterson 研究团队重新设计全新开源ISA——RISC-V。
- **诸多优势**：开源（BSD协议）、精简（47条基础整数指令）、高效（指令格式整齐，译码简捷高效）、模块化（RV32I+可扩展）、可定制、可移植性强（简单设备到复杂设备）。
- **迅速发展**：SiFive、RISC-V基金会、华米科技黄山一号芯片（首款RISC-V穿戴式处理器，运算效率相比ARM Cortex-M4高出38%）、平头哥玄铁910（12nm工艺、16核心）、中科院香山处理器（“雁栖湖”：对标ARM A72或A73、“南湖”：对标ARM A76）。

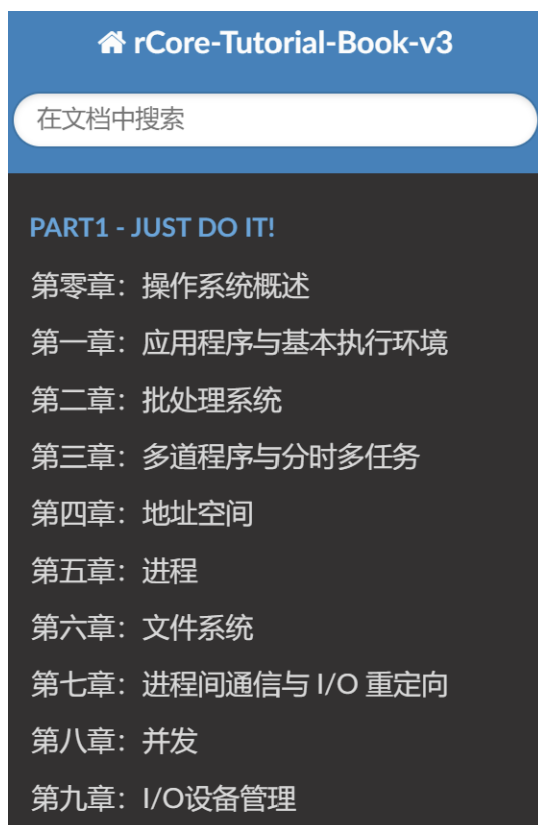


**XiangShan: an Open-Source
High-Performance RISC-V Processor**

选题背景

➤ 基于RISC-V的教学操作系统：将操作系统移植到RISC-V势在必行

- MIT S081 xv6 : [6.S081 / Fall 2019 \(mit.edu\)](https://6.S081/Fall2019/mit.edu)
- THU uCore : [uCore-os-on-riscv64](#)
- THU rCore : [rCore-os-on-riscv64](#)



Mit6.S081-实验环境搭建

Mit6.S081-GDB使用

Mit6.S081-xv6参考书翻译

Mit6.S081-实验1-Xv6 and Unix utilities

Mit6.S081-实验2-System calls

Mit6.S081-实验3-Page tables

Mit6.S081-实验4-Traps

Mit6.S081-实验5-xv6 lazy page allocation

Mit6.S081-实验6-Copy-on-Write Fork for xv6

Mit6.S081-实验7-Multithreading

Mit6.S081-实验8-locks

Mit6.S081-实验9-file system

Mit6.S081-实验10-mmap

Mit6.S081-实验11-networking



研究意义

集成

参照rCore实验设置，参考uCore和xv6的实验代码，用C、RISC-V汇编实现一个小型教学内核系统，具备一定功能，通过开展操作系统实验，帮助掌握操作系统原理。

改进

针对操作系统基础薄弱、想学习rCore却无Rust语言基础的初学者，用C翻译rCore的Rust代码，作为过渡实验，简化实验的细节内容，关注操作系统实验所达成的核心功能，用简单易行的方式完成实验、理解操作系统原理。

论文研究了 **基于RISC-V的操作系统实验集成与改进**。



研究意义

参照**rCore**实验设置
(Rust、RISC-V汇编)

参考**uCore**实验代码
(C、RISC-V汇编)

参考**xv6**实验代码
(C、RISC-V汇编)

集成
改进

小型教学内核操作系统：
内核启动、简单批处理、分时
多任务、内存管理、进程管理
(C、RISC-V汇编)



Part 2

研究方法及过程

研究方法

- **文献研究**：调研了RISC-V指令集架构及指令格式，了解开源指令集架构RISC-V的优势所在，有助于在实验中更好地进行RISC-V汇编，并调研了国内外基于RISC-V的教学操作系统，借鉴实验设计的思路。
- **实验研究**：在Linux系统环境下、QEMU硬件仿真平台上，完成各个实验编程。

实验研究过程

- 搭建实验环境
- 参照rCore实验设置，用C翻译Rust代码，作为过渡实验，依次完成内核启动、简单批处理、分时多任务、内存管理、进程管理的实验代码（C、RISC-V汇编）。
- 设置实验问题，[实验代码](#)挖空，撰写[实验指导书](#)。



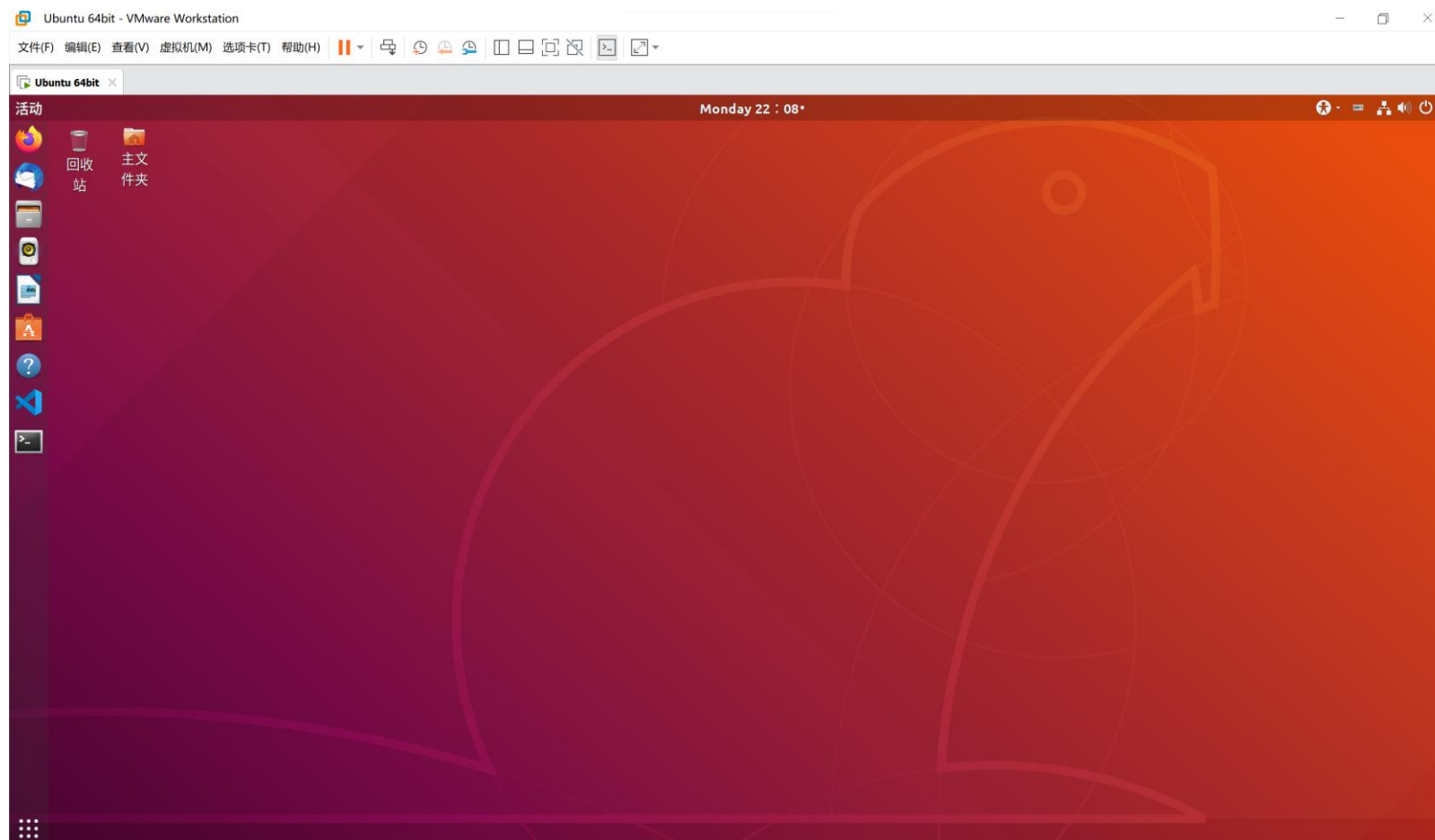
Part 3

运行环境及实验平台

Linux系统环境、QEMU硬件仿真平台

系统环境 → 交叉编译工具链 → 硬件模拟器 → 监管态二进制接口SBI

1. 安装系统环境：VMware16 虚拟机上安装 Ubuntu 18.04（Linux发行版之一）。



Linux系统环境、QEMU硬件仿真平台

系统环境 → 交叉编译工具链 → 硬件模拟器 → 监管态二进制接口SBI

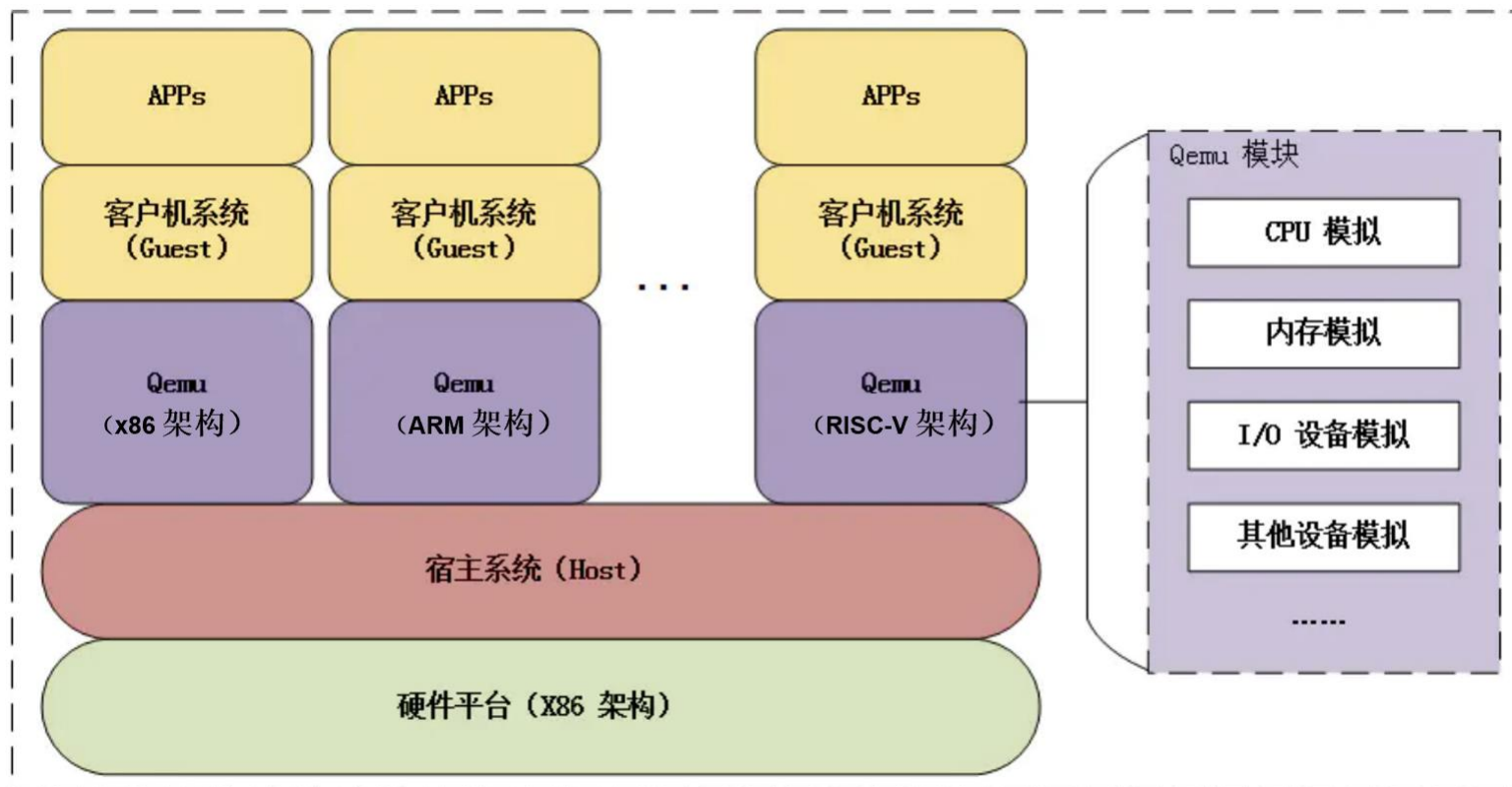
2. **安装交叉编译工具链**：主机是X86架构，而实验硬件仿真的是RISC-V架构，因此需安装交叉编译工具链，SiFive提供“GNU Embedded Toolchain”，下载后解压缩到对应的文件夹，并配置环境变量。

```
lw@lw-virtual-machine:~$ riscv64-unknown-elf-gcc -v 查看交叉编译工具链的安装版本号
Using built-in specs.
COLLECT_GCC=riscv64-unknown-elf-gcc
COLLECT_LTO_WRAPPER=/home/lw/RISCV/riscv64-unknown-elf-gcc-8.3.0-2020.04.0-x86_64-linux-ubuntu14/bin/./libexec/gcc/riscv64-unknown-elf/8.3.0/lto-wrapper
Target: riscv64-unknown-elf
Configured with: /scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package
--scratch-carsteng/obj/x86_64-linux-ubuntu14/build/riscv-gnu-toolchain/riscv-gcc/configure
--target=riscv64-unknown-elf --host=x86_64-linux-gnu --prefix=/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package--scratch-carsteng/obj/x86_64-linux-ubuntu14/install/riscv64-unknown-elf-gcc-8.3.0-2020.04.0-x86_64-linux-ubuntu14 --with-pkgversion='SiFive GCC 8.3.0-2020.04.0' --with-bugurl=https://github.com/sifive/freedom-tools/issues --disable-shared --disable-threads --enable-languages=c,c++ --enable-tls --with-newlib --with-sysroot=/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package--scratch-carsteng/obj/x86_64-linux-ubuntu14/install/riscv64-unknown-elf-gcc-8.3.0-2020.04.0-x86_64-linux-ubuntu14/riscv64-unknown-elf --with-native-system-header-dir=/include --disable-libmudflap --disable-libssp --disable-libquadmath --disable-libgomp --disable-nls --disable-tm-clone-registry --src=../riscv-gcc --with-system-zlib --enable-checking=yes --enable-multilib --with-abi=lp64d --with-arch=rv64imafdc CFLAGS=-O2 CXXFLAGS=-O2 'CFLAGS_FOR_TARGET=-O2 -mcmmodel=medany' 'CXXFLAGS_FOR_TARGET=-O2 -mcmmodel=medany'
Thread model: single
gcc version 8.3.0 (SiFive_GCC 8.3.0-2020.04.0)
```

Linux系统环境、QEMU硬件仿真平台

系统环境 → 交叉编译工具链 → 硬件模拟器 → 监管态二进制接口SBI

3. 安装硬件模拟器QEMU：QEMU提供了RISC-V架构的硬件模拟平台，相当于提供了一台包含了CPU、内存、I/O设备以及其他设备的RISC-V模拟计算机硬件平台。



Linux系统环境、QEMU硬件仿真平台

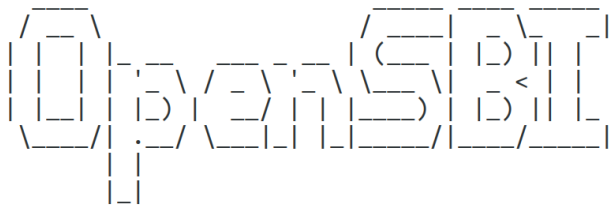
系统环境 → 交叉编译工具链 → 硬件模拟器 → 监管态二进制接口SBI

3. **安装硬件模拟器QEMU**：安装编译所需的依赖包、下载源码包并解压缩、编译安装并配置RISC-V支持、测试（安装过程稍复杂）。

OpenSBI 是QEMU中的内置固件。QEMU 模拟器安装成功并启动后， OpenSBI 会自动为虚拟机执行硬件初始化和加载操作系统。

```
lw@lw-virtual-machine:~$ qemu-system-riscv64 --machine virt --nographic --bios default
```

OpenSBI v0.6



```
Platform Name      : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs  : 8
Current Hart       : 0
Firmware Base      : 0x80000000
Firmware Size      : 120 KB
Runtime SBI Version : 0.2
```

```
MIDELEG : 0x0000000000000222
MEDELEG : 0x0000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

Linux系统环境、QEMU硬件仿真平台

系统环境 → 交叉编译工具链 → 硬件模拟器 → 监管态二进制接口SBI

4. **监管态二进制接口SBI**: 提供给操作系统与底层硬件环境通信的辅助编程工具层, 本实验中采取的是可运行在 qemu 虚拟机上的RustSBI预编译二进制版本rustsbi-qemu.bin, 下载后解压缩到对应的实验文件夹。

```
lw@lw-virtual-machine:~$ qemu-system-riscv64 -machine virt -nographic -bios rustsbi-qemu.bin
[rustsbi] RustSBI version 0.1.1
RUSTSBI
[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
[rustsbi-panic] hart 0 panicked at 'invalid instruction, mepc: 0000000080200000, instruction: 0000000000000000', platform\qemu\src\main.rs:458:17
[rustsbi-panic] system shutdown scheduled due to RustSBI panic
```




Part 4

研究内容及工作量



实验零：内核启动

- **实验目的：**了解操作系统的**启动过程**、**内存布局**，实现一个最小化内核，并在 QEMU 中运行，打印出 "Hello World!"
- **实验内容：**补充关于程序入口点、内存布局的代码

#指定程序的 `_start` 入口点，栈的大小为 $4096 * 16 = 64\text{KB}$

```
.section .text.entry      #后面的内容全部放到一个名为 .text.entry 的段中
.globl _start            #全局变量：定义_start入口点,# 使得ld能够看到_start这个符号所在的位置
_start:                 #声明了一个符号 _start，该符号指向紧跟在符号后面的内容
    la sp, boot_stack_top #将bootstacktop的地址加载到sp(stack pointer)寄存器中, 使用我们分配的内核栈
    call main             #调用main, 这是我们要用C语言编写的一个函数, call是伪指令，作用是调用函数（跳转）

.section .bss.stack      #开始bss section(可初始化为0的可读写段)
.globl boot_stack        #栈底,内核栈

boot_stack:
    .space 4096 * 16      #留出64K字节的内存
    .globl boot_stack_top #栈顶,之后内核栈将要从高地址向低地址增长, 初始时的内核栈为空
boot_stack_top:
```

- **实验代码树及运行结果：** 代码量188行

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

```
lw@lw-virtual-machine:~/rCore-lab0$ make
riscv64-unknown-elf-gcc os.c printf.c entry.S -T linker.ld -ffreestanding -nostd
lib -g -o os -mcmmodel=medany
riscv64-unknown-elf-objcopy os --strip-all -O binary os.bin
qemu-system-riscv64 -machine virt -nographic -bios rustsbi-qemu.bin -device load
er,file=os.bin,addr=0x80200000
[rustsbi] RustSBI version 0.1.1
```

[illegible]

```
[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

- **实验目的：**顺序执行一批应用程序，一个执行完毕，启动下一个，直到所有应用程序都执行完毕。
- **实验内容：**
 - 应用程序的设计；
 - 用户空间与内核空间的隔离；
 - 内核栈与用户栈的切换、陷入上下文的保存和恢复。



实验一：简单批处理

• 实验代码树及运行结果：代码量345行

└─ lib.c (包含用户程序的入口)

└─ user.h (用户态的一些宏定义、类型和函数头)

└─ **user_print.c** (用户程序, 实现打印功能)

└─ **batch.c** (简单批处理调度)

└─ **trap.S** (Trap 上下文保存与恢复的汇编代码)

└─ **mod.c** (包含 Trap 处理函数trap_handler)

└─ kernel.h (内核的一些宏定义、类型和函数头)

└─ sbicall.c (调用底层 SBI 实现的 SBI 接口)

└─ syscall.c (包含 sys_write和sys_exit)

└─ **link_app.S** (应用程序的内存布局)

└─ **linkers.ld** (内核栈的内存布局)

└─ **linkeru.ld** (用户栈的内存布局)

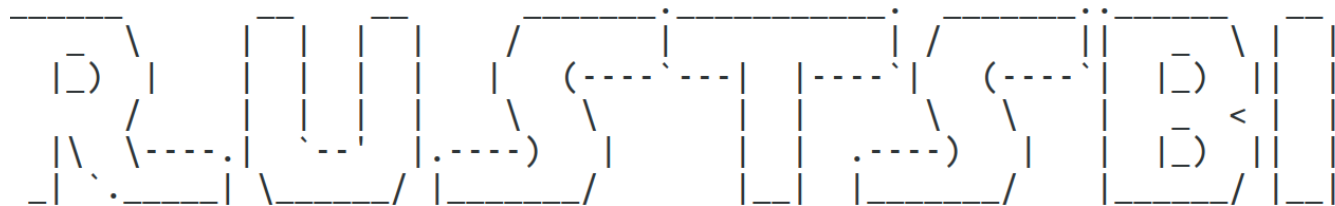
└─ printf.c (参考xv6的printf函数)

└─ entry.S (指定程序入口点、栈空间)

└─ makefile

└─ 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
```

```
ello World!
```

```
pplication exited with code 0
```

```
ello World!
```

```
pplication exited with code 0
```

```
ello World!
```

```
pplication exited with code 0
```

```
ello World!
```

```
pplication exited with code 0
```

```
ello World!
```

```
pplication exited with code 0
```



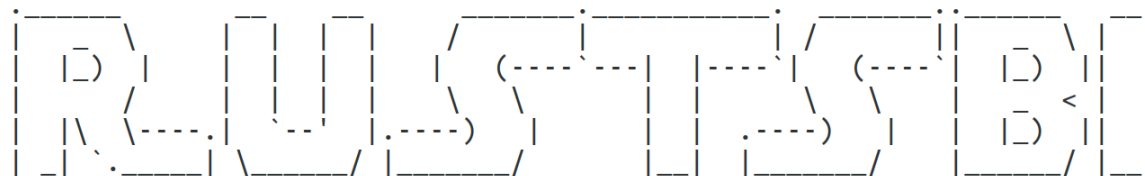
实验二：分时多任务

- **实验目的：**设计基于时间片轮转的操作系统。
- **实验内容：**定时中断产生、任务切换机制的实现。
 - 定时中断产生：CSR mtime、mtimecmp 调用
 - 任务切换机制：任务调度的task.c、用户程序加载和初始化的loader.c、控制任务切换switch.S、简单批处理调度batch.c

```
//读取时间寄存器mtime
usize get_time() {
    |  usize timer; asm volatile("rdtime %0":"=r"(timer)); return timer;
}
//设置mtimecmp的触发值
void set_next_trigger() {
    |  set_timer(get_time() + CLOCK_FREQ / TICKS_PER_SEC);
}
//计时：以微秒为单位返回当前计数器的值，统计一个应用的运行时长
usize get_time_ms() {
    |  return get_time() / (CLOCK_FREQ / MSEC_PER_SEC);
}
```

- **代码树及运行结果：**代码量522行

```
[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[0] print2
Task[1] print1
Task[2] print1
Task[0] print3
Task[0] print4
Task[1] print2
Task[2] print2
Task[0] print5
Task[0] print6
Task[1] print3
Application has done and exited. 0
Task[0] print7
Task[1] print4
Task[0] print8
Task[0] print9
Application has done and exited. 0
Application has done and exited. 0
```



实验三：内存管理

- **实验目的：**核心是对物理地址空间和虚拟地址空间的管理。
- **实验内容：**
 - 添加动态内存分配支持到内核
 - 实现物理地址管理
 - 实现虚拟地址管理
 - 融合分时多任务系统与内存管理系统



实验三：内存管理（物理地址管理）

- 物理地址管理：固定页、空闲分配池、分配和回收

```
//分配页号：若 回收链表 不空则返回 回收链表 栈顶页号
//若当前物理页号current=结束页号end，说明物理内存已满，报错
//否则，返回新页ppn=current++
PhysPageNum frame_alloc() {
    int ret; PhysPageNum ppn;
    if (!lst_empty(recycled)) {
        struct rnode *x = lst_pop(recycled);
        ppn = x->ppn; bd_free(x);
    } else {
        if (current == end) panic("frame_alloc: no free physical page");
        else ppn = current++;
    }
    memset((void *)PPN2PA(ppn), 0, PAGE_SIZE);
    frame_num--;
    print_frame_num();
    return ppn;
}
```

```
//释放页号:先检查回收页面的合法性，然后将其压入 recycled 栈中
//回收页面合法有两个条件：
//该页面之前一定被分配出去过，因此它的物理页号一定< current
//该页面没有正处在回收状态，即它的物理页号不能在栈 recycled 中找到
void frame_dealloc(PhysPageNum ppn) {
    if (ppn >= current) goto fail;
    for (struct list *p = recycled->next; p != recycled; p = p->next) {
        if (((struct rnode *)p)->ppn == ppn) goto fail;
    }
    struct rnode *x = bd_malloc(sizeof(struct rnode));
    x->ppn = ppn; lst_push(recycled, x);
    frame_num++;
    print_frame_num();
    return;
fail: panic("frame_dealloc failed!");
}
```



实验三：内存管理（虚拟地址管理）

- **虚拟地址管理**：SV39 多级页表，即27+12（3级页表，每级页表9位，512页表项；12位页内偏移）。虚实地址映射、数据传送。
- 第一层：find_pte()、map()、unmap()

```
//取页表项：idx可取出虚拟页号的三级页索引，取出低27位
PageTableEntry *find_pte(PhysPageNum root, VirtPageNum vpn, int create) {
    usize idx[3];
    for (int i = 2; i >= 0; i--) {
        idx[i] = vpn & 510; vpn >>= 9; //511=111111111,取位操作；右移赋值；
    } //假如只取一位，256=100000000，会出现映射错误的情况
    for (int i = 0; i < 3; i++) {
        PageTableEntry *pte_p = (PageTableEntry *)PPN2PA(root) + idx[i];
        if (i == 2) return pte_p;
        if (!(*pte_p & V)) {
            if (!create) panic("find_pte failed");
            PhysPageNum frame = frame_alloc();
            *pte_p = PPN2PTE(frame, V);
        }
        root = PTE2PPN(*pte_p); //页表的起始地址、页表根节点的地址
    }
    return 0;
}
```

```
//建立虚实地址映射，改写自page_table.rs
void map(PhysPageNum root, VirtPageNum vpn, PhysPageNum ppn, PTEFlags flags) {
    PageTableEntry *pte_p = find_pte(root, vpn, 1);
    if (*pte_p & V)
        panic("map: vpn is mapped before mapping!");
    *pte_p = PPN2PTE(ppn, flags | V);
}

//拆除虚实地址映射关系
PhysPageNum unmap(PhysPageNum root, VirtPageNum vpn) {
    PageTableEntry *pte_p = find_pte(root, vpn, 1);
    if (!(*pte_p & V))
        panic("unmap: vpn is invalid before unmapping");
    PhysPageNum ppn = PTE2PPN(*pte_p);
    *pte_p = 0; return ppn;
}
```



实验三：内存管理（虚拟地址管理）

- 第二层（过渡层）：map_trampoline()、free_pagetable()、kvm_init()

```
//映射跳表：改写自config.rs
void map_trampoline(PhysPageNum root) {
    extern char strampoline;
    map(root, FLOOR(TRAMPOLINE), FLOOR((PhysAddr)&strampoline), R | X);
}

//递归释放页表
void free_pagetable(PhysPageNum root) {
    PageTableEntry *pte_p = (PageTableEntry *)PPN2PA(root);
    for (int i = 0; i < 512; i++) {
        if (pte_p[i] & V) {
            if (!(pte_p[i] & R) && !(pte_p[i] & W) && !(pte_p[i] & X))
                free_pagetable(PTE2PPN(pte_p[i]));
        }
    }
    frame_dealloc(root);
}

//内核地址空间初始化
void kvm_init() {
    frame_init();
    kernel_pagetable = frame_alloc();
    extern char stext, etext, srodata, sdata, edata,
        sbss_with_stack, ebss, ekernel;
    map_trampoline(kernel_pagetable);
    map_area(kernel_pagetable, (VirtAddr)&stext, (VirtAddr)&etext, R | X, 0);
    map_area(kernel_pagetable, (VirtAddr)&srodata, (VirtAddr)&erodata, R, 0);
    map_area(kernel_pagetable, (VirtAddr)&sdata, (VirtAddr)&edata, R | W, 0);
    map_area(kernel_pagetable, (VirtAddr)&sbss_with_stack, (VirtAddr)&ebss, R | W, 0);
    map_area(kernel_pagetable, (VirtAddr)&ekernel, MEMORY_END, R | W, 0);
    usize satp = PGTB2SATP(kernel_pagetable);
    asm volatile ("csrw satp, %0\nsfence.vma:::\"r\"(satp));" : : satp);
}
```



实验三：内存管理（虚拟地址管理）

- 第三层：map_area()、unmap_area(), copy_area()。

```
//alloc参数为0映射内核空间，这种情况下虚拟页号等于物理页号
//alloc参数为1映射用户空间，这种情况随映射随创建就行
void map_area(PhysPageNum root, VirtAddr start_va, VirtAddr end_va, PTEFlags flags, int alloc) {
    VirtPageNum start_vpn = FLOOR(start_va), end_vpn = CEIL(end_va);
    for (VirtPageNum i = start_vpn; i < end_vpn; i++) {
        PhysPageNum ppn = alloc ? frame_alloc() : i;
        map(root, i, ppn, flags);
    }
}

//解映射时物理空间要保留，对应的物理页帧释放掉
void unmap_area(PhysPageNum root, VirtAddr start_va, VirtAddr end_va, int dealloc) {
    VirtPageNum start_vpn = FLOOR(start_va), end_vpn = CEIL(end_va);
    for (VirtPageNum i = start_vpn; i < end_vpn; i++) {
        PhysPageNum ppn = unmap(root, i); if (dealloc) frame_dealloc(ppn);
    }
}

//主要实现虚拟地址里的数据和物理地址里的数据传递
//to_va为1，将data里的数据传给root页表对应的start_va虚拟地址开始的空间
//to_va为0，将root页表对应的start_va虚拟地址开始的数据传给data
void copy_area(PhysPageNum root, VirtAddr start_va, void *data, int len, int to_va) {
    char *cdata = (char *)data; VirtPageNum vpn = FLOOR(start_va);
    while (len) {
        usize frame_off = start_va > PPN2PA(vpn) ? start_va - PPN2PA(vpn) : 0;
        usize copy_len = PAGE_SIZE - frame_off < len ? PAGE_SIZE - frame_off : len;
        PageTableEntry *pte_p = find_pte(root, vpn, 0);
        if (to_va) memcpy((void *)PPN2PA(PTE2PPN(*pte_p)) + frame_off, cdata, copy_len);
        else memcpy(cdata, (void *)PPN2PA(PTE2PPN(*pte_p)) + frame_off, copy_len);
        len -= copy_len; cdata += copy_len; vpn++;
    }
}
```


实验三：内存管理

实验代码树：代码量1271行

```
.
├── common
│   ├── common.c (通用函数的实现, 如printf、memset、memcpy)
│   ├── common.h (包含了common.c里的函数定义和一些类型定义)
│   ├── filec.h (文件相关的常数, 目前只有FD_STDOUT)
│   ├── rustsbi-qemu.bin(可运行在 qemu 虚拟机上的rustsbi预编译二进制版本)
│   └── syscall.h (存放系统调用号)
├── kernel
│   ├── buddy.c (参考xv6的伙伴算法)
│   ├── elf.h
│   ├── entry.S
│   ├── frame.c(物理页帧管理)
│   ├── kernel.h
│   ├── link_app.S
│   ├── linker.ld
│   ├── list.c (链表, 以支持动态内存分配)
│   ├── loader.c
│   ├── pagetable.c (虚实地址映射、查找、数据交换)
│   ├── sbicall.c
│   ├── switch.S
│   ├── syscall.c
│   ├── task.c
│   ├── timer.c
│   ├── trap.c (处理陷入)
│   └── trap.S
├── makefile
└── user
    ├── lib.c
    ├── linker.ld
    ├── task0.c(每 10ms 循环打印, 打印10次)
    ├── task1.c(每 20ms 循环打印, 打印10次)
    └── task2.c(每 30ms 循环打印, 打印10次)
```

任务执行前先
分配物理页帧号

实验测试：

- PTE标志位测试
- 打印物理页帧的分配回收
- 虚拟内存映射测试

```
frame_num:721
frame_num:720
frame_num:719
frame_num:718
frame_num:717
frame_num:716
frame_num:715
frame_num:714
frame_num:713
frame_num:712
frame_num:711
frame_num:710
frame_num:709
frame_num:708
frame_num:707
frame_num:706
frame_num:705
frame_num:704
frame_num:703
frame_num:702
frame_num:701
frame_num:700
frame_num:699
frame_num:698
frame_num:697
frame_num:696
frame_num:695
frame_num:694
frame_num:693
frame_num:692
frame_num:691
frame_num:690
frame_num:689
frame_num:688 program start
Program[0] print0
Program[1] print0
Program[2] print0
```

```
Application exited with code 0
frame_num:589
frame_num:690
frame_num:691
frame_num:692
frame_num:693
frame_num:694
frame_num:695
frame_num:696
frame_num:697
frame_num:698
frame_num:699
Program[1] print9
Program[2] print9
Application exited with code 0
frame_num:700
frame_num:701
frame_num:702
frame_num:703
frame_num:704
frame_num:705
frame_num:706
frame_num:707
frame_num:708
frame_num:709
frame_num:710
Application exited with code 0
frame_num:711
frame_num:712
frame_num:713
frame_num:714
frame_num:715
frame_num:716
frame_num:717
frame_num:718
frame_num:719
frame_num:720
frame_num:721
```

任务执行结束后
回收物理页帧号



实验四：进程管理

- **实验目的：**进程创建、进程资源回收，实现命令行应用。
- **实验内容：**
 - 创建进程相关的函数，如initproc、fork、exec；
 - 实现命令行应用（user_shell）：用户输入应用程序名称，操作系统将会执行相应的进程，执行结束后返回进程标识数PID，从而达到与内核操作系统交互的目的。

实验四：进程管理

- user_shell实现：shell程序的输入输出，需要增加 sys_read 和sys_write系统调用使得shell能够取得用户的键盘输入并执行相应的应用程序。

```
isize sys_write(usize fd, char *buffer, usize len) {
    switch (fd) {
        case FD_STDOUT: {
            char *pbuffer = bd_malloc(len + 1); pbuffer[len] = '\0';
            copy_area(current_user_pagetable(), (VirtAddr)buffer, pbuffer, len, 0);
            printf(pbuffer); bd_free(pbuffer); return (isize)len;
        }
    }
}

isize sys_read(usize fd, char *buffer, usize len) {
    switch (fd) {
        case FD_STDIN: {
            char *pbuffer = bd_malloc(len);
            for (int i = 0; i < len; i++) {
                char c;
                while (!(c = console_getchar()))
                    suspend_current_and_run_next();
                pbuffer[i] = c;
            }
            copy_area(current_user_pagetable(), (VirtAddr)buffer, pbuffer, len, 1);
            bd_free(pbuffer); return (isize)len;
        }
    }
}
```

```
int main() {
    printf("C user shell: choose applications above ,or you can exit this shell \n");
    printf(">> ");
    for (;;) {
        gets(line, LINE);
        if (!strcmp(line, "exit")) return 0;
        if (!strcmp(line, "exec")) return 0;
        isize pid = fork();
        if (pid == 0) {
            // child process
            if (exec(line) == -1) {
                printf("Error when executing!\n");
                return -4;
            }
        } else {
            int exit_code = 0;
            isize exit_pid = waitpid(pid, &exit_code);
            printf("Shell: Process %d exited with code %d\n", pid, exit_code);
        }
        printf(">> ");
    }
    return 0;
}
```

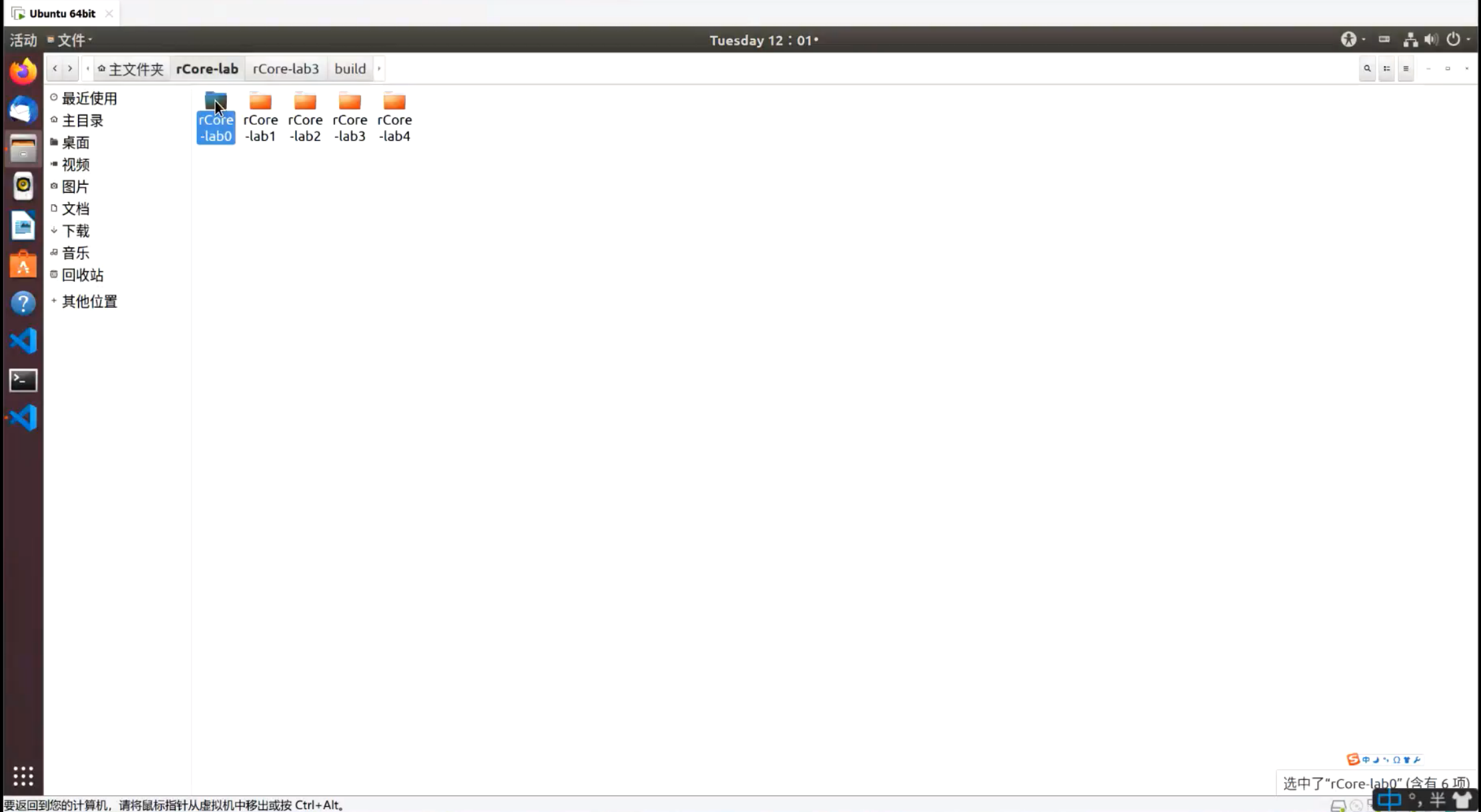


实验四：进程管理

- 代码树：代码量2299行
- 实验测试：
 - 验证用户初始化进程initproc的pid;
 - 测试父进程与子进程的pid;
 - 验证用户通过输入命令行与内核交互;
 - 测试退出 “exit”

lab4

```
.
├── common
│   ├── common.c
│   ├── common.h
│   ├── filec.h
│   ├── rustsbi-qemu.bin
│   └── syscall.h
├── kernel
│   ├── buddy.c
│   ├── build.py (自动生成link_app.S)
│   ├── elf.h
│   ├── entry.S
│   ├── frame.c
│   ├── kernel.h
│   ├── linker.ld
│   ├── list.c
│   ├── loader.c
│   ├── pagetable.c
│   ├── pid.c (进程标识)
│   ├── queue.h (队列, 用于物理页帧管理、PID管理、子进程管理)
│   ├── sbicall.c
│   ├── switch.S
│   ├── syscall.c
│   ├── task.c
│   ├── timer.c
│   ├── trap.c
│   └── trap.S
├── makefile
└── user
    ├── initproc.c (初始进程)
    ├── lib.c
    ├── linker.ld
    ├── task0.c (打印信息并调用task1和task2)
    ├── task1.c (每10ms打印一次信息)
    ├── task2.c (每20ms打印一次信息)
    ├── user.h
    └── user_shell.c (用户命令行)
```



Part 5

总结与展望



实验总结与展望

总结

参照rCore实验设置，用C翻译Rust，实现一个具备：内核启动、简单批处理、分时多任务、内存管理、进程管理等功能的简单教学内核系统，并设置实验，撰写实验指导手册。

展望

- 优化实验设置，如有针对性地增加 需要补充代码的实验“填空”、问题设置。
- 增加内核操作系统的功能。



**感谢各位老师聆听，
敬请您指正！**

答辩人：李 雯

导 师：陆慧梅

日 期：2022/6/9