操作系统实验报告

课程名称	:	操作系统
姓　　名	:	沈一芃
学　　院	:	计算机学院
专　　业	:	计算机科学与技术
学　　号	:	3220101827
指导教师	:	李环 柳晴

2024 年　12　月　2　日

Lab 4 Report

实验名称：RV64 用户态程序

联系方式：1669335639@qq.com 18310211513

1. 实验目的与要求

1.1. 实验目的

- 了解用户态程序的概念和实现原理
- 了解ELF程序的原理与加载

1.2. 实验要求

- 创建**用户态进程**，并完成内核态与用户态的转换
- 正确设置用户进程的**用户态栈**和**内核态栈**，并在异常处理时正确切换
- 补充异常处理逻辑，完成指定的**系统调用**（SYS_WRITE, SYS_GETPID）功能
- 实现用户态 ELF 程序的解析和加载

2. 实验环境

本机CPU	本机操作系统	实验环境	实验操作系统	配置环境
Mac Silicon M3	CentOS Sequoia 15.0.1	Docker 4.33.0	Ubuntu 24.10	Qemu-system-riscv64 9.0.2

3. 实验过程

3.1. 实验环境搭建

- 整个实验建立在lab3的基础上。从仓库克隆实验4代码到本地，跟随实验流程搭建文件结构如下：

```
├── arch
│   └── riscv
│       ├── include
```

```
│       │   ├── mm.h        # already modify; lead in the buddy system to manage memory
pages
│       │   ├── proc.h
│       │   ├── clock.h
│       │   ├── syscall.h   # need to accomplish; the headfile of the 2 unimplemented
syscall
│       │   ├── defs.h      # need to modify; add macro of user mode
│       │   └── sbi.h
│       ├── kernel
│       │   ├── vm.c
│       │   ├── mm.c        # already modify; lead in the buddy system to manage memory
pages
│       │   ├── proc.c      # need to accomplish; modify and properly init the thread
struct
│       │   ├── syscall.c   # need to accomplish; implement 2 syscall
│       │   ├── sbi.c
│       │   ├── trap.c      # need to modify; enable u-mode ecall and calls the syscall
│       │   ├── clock.c
│       │   ├── head.S      # need to modify; annotate the sstatus's sie
│       │   ├── entry.S     # need to modify; modify _traps, __dummy, __switch_to
│       │   ├── Makefile
│       │   └── vmlinux.lds # need to modify; add "uapp" to the data part
│       └── Makefile        # need to modify; add "uapp" to the makefile
├── include
│   ├── elf.h               # already accomplish; the definition of elf structures
│   ├── printk.h
│   ├── stddef.h
│   ├── stdint.h
│   ├── stdlib.h
│   └── string.h
├── init
│   ├── main.c              # need to modify;
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── printk.c
├── user
│   ├── Makefile            # already accomplish;
│   ├── getpid.c            # already accomplish; user mode program
│   ├── link.lds            # already accomplish;
│   ├── printf.c            # already accomplish; here use the syscall "sys_write"
│   ├── start.S             # already accomplish; entrance of the user mode
│   ├── stddef.h            # already accomplish;
│   ├── stdio.h             # already accomplish;
│   ├── syscall.h           # already accomplish; provide the ID of the two syscall
│   └── uapp.S              # already accomplish; extract the binary content
└── Makefile                # need to modify; compile the user directory
```

3.2. RV64 用户态程序

3.2.0. 总括

- 本次实验的目的是实现用户态进程。lab2中我们实现了多线程的切换与调度，但这些线程其实均运行在内核态线程，调度则是依靠时间中断进入异常态后处理的。本次实验中，要实现运行在用户态的线程。
  - 同一个线程，既可以运行在内核态，又可以运行在用户态。当一个线程进入中断时，系统(sbi)会自动将线程拔高到S态运行。所以一个线程是内核线程还是用户线程，取决于非中断时程序运行在什么状态。
  - sstatus中的`SPP`位记录了本次中断前线程运行在什么状态。这样当中断结束后sret时，系统会根据`SPP`决定返回U态(1'b0)还是S态（1'b1）。故而SPP的置位决定了线程的运行状态。
  - 当计算机启动时，所有代码都运行在最高权限，此实验中即S态，因为内核引导代码和操作系统内核需要获取对硬件的权限。我们将用户态的程序加载进内存并将对应的线程实例化时，他们其实也"运行"在S态。为了让用户态程序运行在用户态，我们就需要将其`SPP`位置0；当线程第一次被调度时，会运行在S态，第一次时钟中断后，SPP的置位会使其回到U态。这样，当U态的线程发起ecall时，就会触发`0x8`号`Environment call from U-mode`异常，下一次中断时进入我们在`trap_handler`中写好的对应处理逻辑，如`sys_getpid`或`sys_write`。
- 整体逻辑如上。具体实现中，不同于lab2所有的进程共用同一段地址上的代码，仅切换寄存器等上下文状态；本实验中用户态进程应该做到相互隔离，有各自的虚拟地址空间；加之用户态进程多了一个用户态运行的栈空间，本次实验在初始化时要对上述内容分配空间并做虚拟地址映射。同样，由于新增了状态的切换，在切换上下文内容、线程调度时，也有新的变量要保存。
- 此外，本次实验中用户态运行的程序并非直接写在内核代码中，而是通过读入的方式加载进来。

3.2.1. 创建用户态进程

3.2.1.1. 修改结构体

- 创建用户态进程，要对每个进程的`sepc`，`sstatus`，`sscratch`做操作。同时，用户态进程相对隔离则要求每个进程有自己的虚拟地址空间，即页表映射。故需修改线程结构体如下：

```
struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];
    uint64_t sepc, sstatus, sscratch;
};

struct task_struct {
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;

    struct thread_struct thread;
    uint64_t *pgd;  // 用户态页表
```

```
};
```

3.2.1.2. 线程初始化

- 需要填充上述新增的内容。
  - **处理寄存器**：`sepc`，`sscratch`分别置为U态的开始(USER_START)以及U态的sp，即栈顶(USER_END)。`sstatus` 中置 `SPP`为0，使sret返回U态；置`SUM`为1，使得S态可以访问User页面；置`SPIE`为1，允许下一次中断

```
task[i]->thread.sepc = USER_START;
task[i]->thread.sstatus = 1 << 18 | 1 << 5;
task[i]->thread.sscratch = USER_END;
```

  - **处理页表**：U态地址空间为`0x0-0x4000000000000000`，S态地址空间为`0xffffffe000000000-0xffffffff000000000`。由于线程在U态和M态切换，需要线程根据页表实现对这两块空间的访问。为了省去在两种状态切换时切换页表，就将S态的页表也写入U态的页表中。对于U态的页表，则要先alloc_pages分配适当内存地址，将 `uapp` 二进制文件内容**拷贝**过去，之后再完成页表的U态部分，将内存地址和虚拟地址做映射。

> 关于用户态程序uapp及其拷贝：过去我们整个内核是一起编译链接，且所有线程共享一段内存中的代码，运行在同一块物理内存上。本次实验中的用户态程序，则要有各自的地址空间，每个线程的代码都单独运行在一块物理内存上，虽然这些代码依然是相同的。而操作流程，则是在user中单独编译出二进制strip文件，在vmlinux.lds中写好位置，把这个二进制文件加载到内存的对应位置，该位置为`_sramdisk`。再把从这个位置开始的内容拷贝到给这个线程分配的物理空间，对该物理内存和虚拟地址做create_mapping映射。

```
  task[i]->pgd = alloc_page();
  uint64_t *a = task[i]->pgd;
  for(uint64_t i = 0; i < 512;i++){
    a[i] = swapper_pg_dir[i];
  }
  uint64_t bin_size = (uint64_t)_eramdisk - (uint64_t)_sramdisk;
  uint64_t page_num = (bin_size + PGSIZE - 1)/PGSIZE;
  uint64_t *addr = alloc_pages(page_num);
  char *phy_addr = (char*)addr;
  for(uint64_t i = 0; i < bin_size; i++){
    phy_addr[i] = _sramdisk[i];
  }
  create_mapping(task[i]->pgd, USER_START, (uint64_t)addr - PA2VA_OFFSET,
bin_size, 0x1f);
```

  - **处理栈空间**：每个用户态进程拥有两个栈：内核态栈已经在lab3中设置好，即`thread.sp`。用户态栈则需要申请一块空间并进行映射。这个栈在虚拟地址中的布局位于用户空间的末页。

```
  uint64_t *user_stack = alloc_page();
  create_mapping(task[i]->pgd, USER_END - PGSIZE, (uint64_t)user_stack -
PA2VA_OFFSET, PGSIZE, 0x1f);
```

3.2.2. 修改`__switch_to`

- 由于新增了页表和寄存器，在线程调度切换上下文时，也要切换这些内容。

> 1. ⚠️注意栈的保存逻辑。对于先移动sp再操作的方法，假设最后一个操作是sd 180(sp)，则sp实际应该先挪动188位。
> 2. 更新页表的过程同lab3中写satp；记得用sfence.vma刷新。

3.2.3. 更新中断处理逻辑

- RV64只有一个栈指针sp，故而在S态和U态切换（进入/退出中断）时，需要切换栈指针指向对应的栈空间。
  - **修改`_dummy`**：从第一次中断退出时，会进入__dummy，之后通过SPP+sret的逻辑返回U态。故而此处应该切换栈指针为U态栈指针。由于目前S态指针存在thread.sp，U态指针存在sscratch，此处需要交换两个寄存器的值。
  - **修改`_traps`**：由于_traps处理了中断的触发和退出，我们需要在保存寄存器/恢复寄存器的前/后交换栈指针，即thread.sp与sscratch。注意，由于内核程序，即一直运行在S态的程序，并没有用户栈，故而此处需要做判断。
  - **修改`trap_handler`**：修改其借口，新接受一个参数regs。由于保存的寄存器值在trap_handler中需要被用到，而其连续存储可以被视作一个结构体，故实现regs以方便寄存器的存取。

```
__dummy:
    csrr a0, sscratch
    csrw sscratch, sp
    addi sp, a0, 0          # change sp from kernel to user
    sret

_traps:                     # the part that determines whether it's U mode or S mode
    csrr t0, sscratch
    li t1, 0
    beq t0, t1, _ssp
    csrw sscratch, sp
    addi sp, t0, 0          # change sp to kernel_sp if it's a user thread
_ssp:
...
    csrr a0, scause         # store CSRs while prepare the parameters for
trap_handler by updating a0-a2
    # sd a0, 240(sp)
    csrr a1, sepc
    sd a1, 240(sp)
    csrr a2, sstatus
    sd a2, 248(sp)
    addi a2, sp, 0          # store sp in a2, as the parameter "structure pt_regs
*regs"
    jal x1, trap_handler
...
```

3.2.4. 添加系统调用

- 本实验中有有两个系统调用，一个是`getpid`，出现在`getpid.c`中；另一个则是`write`，出现在`printf.c`中，分别负责获得调用的进程号，以及将输出重定向到终端。系统调用在S态才能起作用，但此处为U态触发，故其作用是触发编号为`0x8`的`ecall from U-mode`，进入中断提高权限后，再根据寄存器中存下的系统调用号和相关的值，实现对应的功能。此处即修改trap_handler中的逻辑，使得在识别`ecall from U-mode`触发的中断中调用实现的系统调用。
- 具体实现方式则为增加`syscall.c`，`syscall.h`文件，在其中实现 `getpid()` 以及 `write()` 逻辑
- 系统调用的返回参数放置在 `a0` 中，修改 regs 中保存的内容
- 针对系统调用这一类异常，我们需要手动完成 `sepc + 4`

> 手动sepc+4的原因见思考题

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    if (scause & (1ull << 63)) {
      ...
    } else {
        if (scause & 0x8) {
            if (regs->a7 == SYS_GETPID){
                regs->a0 = sys_getpid();
                regs->sepc += 4;                // go to the next instruction
            } else if (regs->a7 == SYS_WRITE){
                regs->a0 = sys_write((uint64_t)regs->a0, (char*)regs->a1,
(uint64_t)regs->a2);
                regs->sepc += 4;                // go to the next instruction
            }
        }
        return;
    }
    return;
}
```

3.2.5. 调整时钟中断

- 在之前的lab 中，在OS boot之后，我们需要等待一个时间片，才会进行调度，我们现在更改为OS boot完成之后立即调度uapp，即在 `start_kernel()` 中，`test()` 之前调用 `schedule()`
- 将 `head.S` 中设置 sstatus.SIE 的逻辑注释掉，确保 schedule 过程不受中断影响

```
int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");
    schedule();
    test();
    return 0;
}
```

- 在处理完上述内容后，内核已经可以加载strip二进制文件正常运行

3.2.6. elf的加载

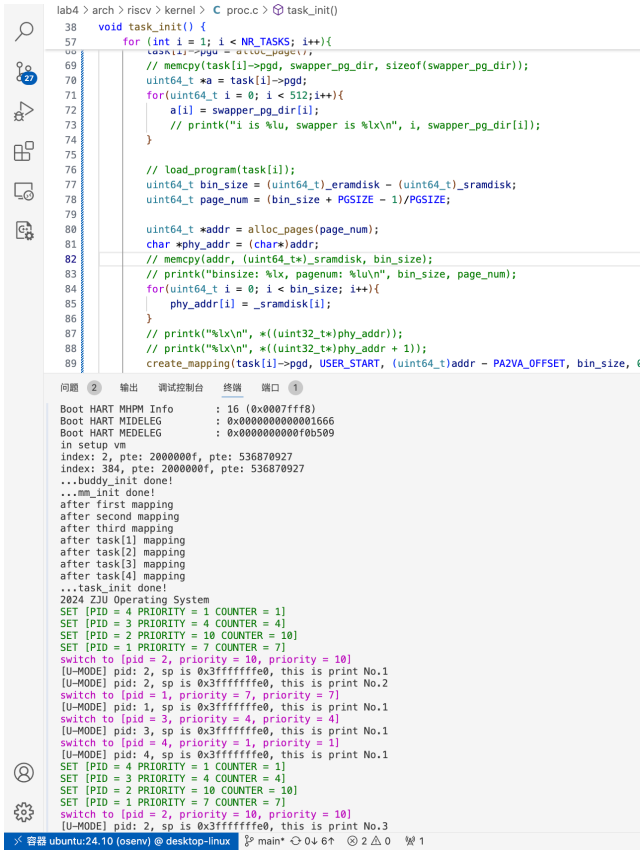
- 对于elf文件的加载，和纯二进制文件的区别只在于elf文件格式更规整，具有metadata部分记录了整个elf文件的内容以及分布；最直观的一点就是可以根据定位segment来确定具体的段位置。直观来看就是，同样通过vmlinux.lds加载uapp.bin / uapp到`_sramdisk`,对strip而言，`_sramdisk`处已经是要运行的指令，而对uapp而言，`_sramdisk`处是metadata信息。
- 本部分要通过metadata，找到线程运行时需要加载到内存中的部分以及地址，将其按照处理strip文件的方式进行相同操作，即为线程分配内存空间，将找到的需要加载的部分从内核代码部分拷贝到分配好的内存中，再进行虚拟地址的映射。
    - elf文件头部的定义较为简单直观，直接阅读即可。本次实验中在`elf.h`内部写好了结构体`Elf64_Ehdr`和`Elf64_Phdr`，可以直接用这两个结构体来访问elf文件中记录的对应内容。前者是整个文件的metadata，后者是每一个segment段的metadata。
    - elf文件中，虚拟地址在编译时确定，所以做映射时应该映射到对应的地址，而非从0x0开始。此外，由于metadata的存在，物理地址和虚拟地址并非页对齐的，但分配的物理空间和虚拟空间做映射时是以页为单位进行的，故而要考虑将两者对齐，确保指令不会出现跨页的现象。

> 对这部分，我的理解是，假设elf文件中要加载的地址是0x110ef，对于pgsize=0x1000，存在一个大小为0xef的offset。由于elf文件也是按页存储的，此时这个elf保证了不会出现指令跨页的现象。但如果分配一块内存空间0x20000，直接将0xef映射到0x20000，则elf对应的内容都前提了offset，不能保证挪移后不会出现指令跨页现象。所以应该将加载地址的offset在分配的空间中对齐，分配空间中向后偏移offset再拷贝，保证不出现跨页现象。

```
void load_program(struct task_struct *task) {
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk;
    Elf64_Phdr *phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->e_phoff);
    for (int i = 0; i < ehdr->e_phnum; ++i) {
        Elf64_Phdr *phdr = phdrs + i;
        if (phdr->p_type == PT_LOAD) {
            uint64_t page_num = (phdr->p_memsz + phdr->p_offset + PGSIZE - 1) /
PGSIZE;
            uint64_t *paddr = alloc_pages(page_num);
            uint64_t *addr = (uint64_t*)(_sramdisk + phdr->p_offset);
            uint64_t phy_offset = (uint64_t)addr & (PGSIZE - 1);
            for(uint64_t i = 0; i < phdr->p_memsz; i++){
                *((char*)paddr + i + phy_offset) = *((char*)addr + i);
            }
            for(uint64_t i = phdr->p_filesz; i < phdr->p_memsz; i++){
```

```
                *((char*)paddr + i + phy_offset) = 0;
            }
            create_mapping(task->pgd, phdr->p_vaddr - phy_offset, (uint64_t)paddr -
PA2VA_OFFSET, phdr->p_memsz, 0x1f);
        }
    }
    task->thread.sepc = ehdr->e_entry;
}
```

- 替换后，内核也能正常运行。下面分别列出运行strip和elf的结果：

运行strip二进制文件	运行elf文件
	

4. 讨论与心得

4.1. 用户态线程的本质

- 其实这一部分在`3.2.0`总括节中已经阐述清楚。过去的实验中，所有的程序其实都是内核代码，既运行在内核代码空间中，又一直运行在S态；一切线程切换只不过是寄存器的上下文切换。而一个用户态程序的本质，其一是独立运行在内存中，其二是**在非中断情况下**运行在U-mode。本次实验的用户态程序实现方式，分别对应：1. 在创建进程时为其分配物理内存，存储其运行时的代码；即用户态程序独立运行在这块分配给他的物理内存上。 2. 在调度运行后的第一次中断返回时，通过SPP置位使sret返回U态。

4.2. 内存分配系统

- 在对用户态线程分配物理空间并进行虚拟地址映射时，我有一个疑惑：lab3中的create_mapping建立在程序运行在`early_pg_dir`的基础上，即当时物理地址和虚拟地址的关系满足`PA2VAOFFSET`的关系。但当进入到lab4，当程序已经运行在swapper_pg_dir时，物理地址和虚拟地址的关系满足的是三级页表关系，而非`PA2VAOFFSET`；既然如此，为什么对于alloc的空间，依然采用`PA2VAOFFSET`得到正确的物理地址呢?
    - 我的理解是，内存分配系统本来就是用堆或其他方式管理**物理内存**。若不开启虚拟地址，则alloc得到的地址其实就是物理内存的地址。但当开启虚拟地址后，整个内核运行在虚拟地址上，故而内存分配系统分配的也应该是一个虚拟地址
    - 在create_mapping中，建立虚拟地址和物理地址的联系需要手动计算分配的空间的虚拟地址对应的物理地址；在设计某些内核代码时，也需要手动计算。这是因为我们在进行开发和设计，而不是在直接应用，所以才觉得手动计算的过程是个冗余，内存分配系统应该设计的更智能些，直接返回给我们一个正确的物理地址。
    - 但既然我们明确了内存分配系统返回的是一个虚拟地址，为什么它遵循的转换关系是`PA2VAOFFSET`而不是三级页表呢？归根结底，页表只是一个映射关系，比如在lab4中，切换线程时要切换页表，我们不光可以只切换satp的ppn部分；切换页表策略其实也没问题。不同的部分完全可以采用不同的映射方式。所以显然内存分配系统一直采用的就是一种direct mapping的方式。
    - 为什么内存分配系统一直采用direct mapping的方式呢？对比下这种方式与三级页表，三级页表优势在于面对巨大的虚拟空间，可以需要用时再分配，节省了页表占据的空间；其劣势则是三级页表查找会带来更慢的性能。而内存本来就是一块不大的空间，其特点又是快速，故内存分配显然用direct mapping更经济简单有效。

4.3. 其他

> 这里记录了实验的几个小tips/我的愚蠢错误

- 关于栈，一定要注意sp挪动位置的意义。sp挪动的位置是栈空间的结尾，而非最后一块可用地址的开头。
- 关于寄存器，虽然同为caller寄存器，尽量使用t0，t1而非a0，a1；因为a类寄存器用来传递参数。本次实验中，trap_handler中我能正确读到a2-a7但无法正确读到a0和a1；最后发现是因为进入_trap后，要切换用户/内核栈，这个过程中我借助了a0和a1，于是之前ecall传入a0和a1的值就被覆盖掉了。。。

5. 思考题

5.1. 我们在实验中使用的用户态线程和内核态线程的对应关系是怎样的?

- 应该是1对1关系。本次实验是一个用户态线程可以在用户态和内核态之间切换运行，运行在U态和运行在S态一一对应

5.2. 系统调用返回为什么不能直接修改寄存器?

- 从实现上讲，因为直接修改寄存器后，当从trap_handler返回指trap中，会恢复寄存器，未修改的regs中的值会把对应寄存器中的值覆盖掉。从原理上讲，则是应该修改栈空间而不是直接修改寄存器。

5.3. 针对系统调用，为什么要手动将 sepc + 4?

- 系统调用是通过ecall实现的，本实验中ecall会触发异常。正常来讲，异常是因为运行指令出现错误导致的；处理异常就是在修复问题，之后自然要试着重新运行这条指令。但系统调用只是为了借用ecall处理异常来实现功能，并非运行的指令出现了错误，不需要重新运行。若不手动讲sepc+4，程序就会永远卡死在这个系统调用上反复执行系统调用功能。

5.4. 为什么 Phdr 中，p_filesz 和 p_memsz 是不一样大的，它们分别表示什么?

- Phdr中，p_filesz和p_memsz分别表示Segment段在文件中的尺寸和在内存中的尺寸。如果是单独一段代码，自然在文件中和在内存中的尺寸是相同的。但对于一段程序，其代码有布局，就像.lds中描述的那样。比如，有些地方需要页对齐，有些地方需要预留空间。在文件中，空白不需要存储，预留空间也只需要描述；但在内存中，却要真实的预留出空白或空间。故而两者大小不同。

5.5. 为什么多个进程的栈虚拟地址可以是相同的？用户有没有常规的方法知道自己栈所在的物理地址?

- 多个进程各自的虚拟地址空间是相互隔离的，虽然虚拟地址相同，但其映射的物理地址是不同的。
- 用户没有常规方法。但程序员可以在gdb调试的时候打印出来栈虚拟地址和satp页表，之后手动计算得到栈的物理地址。