浙江大学

本科实验报告

RV64 用户态程序

课程名称：	操作系统
姓　　名：	陈杰伟
学　　院：	地球科学学院
专　　业：	地理信息科学
学　　号：	3200101205
指导老师：	寿黎但

2022 年 12 月 7 日

浙江大学实验报告

专业：地理信息科学
姓名：陈杰伟
学号：3200101205
日期：2022 年 12 月 7 日
地点：玉泉曹光彪-西 503

课程名称：操作系统　　指导老师：寿黎但　　成绩：
实验名称：RV64 用户态程序　　实验类型：编程实验　　同组学生姓名：无

一、 实验目的

- 创建用户态进程，并设置 sstatus 来完成内核态转换至用户态。
- 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。

二、 实验环境

Ubuntu 20.04

三、 实验步骤

1. 创建用户态进程

由于多个用户态进程需要保证相对隔离，因此不可以共用页表。为每个用户态进程都创建一个页表。修改 task_struct 如下。

task_struct

```
// proc.h

typedef unsigned long* pagetable_t;

struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];

    uint64_t sepc, sstatus, sscratch;
};

struct task_struct {
    struct thread_info* thread_info;
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;
```

```
19
20      struct thread_struct thread;
21
22      pagetable_t pgd;
23  };
```

对 task_init 做如下修改

1. 通过 alloc_page 接口申请一个空的页面来作为 U-Mode Stack

2. 为每个用户态进程创建自己的页表并将 uapp 所在页面，以及 U-Mode Stack 做相应的映射，同时为了避免 U-Mode 和 S-Mode 切换的时候切换页表，将内核页表（swapper_pg_dir ）复制到每个进程的页表中

3. 用户文件需拷贝至内存（uapp 段是在模拟硬盘）

4. 对每个用户态进程将 sepc 修改为 USER_START，配置修改好 sstatus 中的 SPP （使得 sret 返回至 U-Mode ），SPIE （sret 之后开启中断），SUM （S-Mode 可以访问 User 页面），sscratch 设置为 U-Mode 的 sp，其值为 USER_END （即 U-Mode Stack 被放置在 user space 的最后一个页面）。

task_init

```
1   void task_init()
2   {
3       // idle settings
4
5       int i = 0;
6       for (i = 1; i < NR_TASKS; ++i)
7       {
8           task[i] = (struct task_struct *)kalloc();
9
10          task[i]->state = TASK_RUNNING;
11
12          task[i]->counter = 0;
13          task[i]->priority = rand();
14
15          task[i]->pid = i;
16
17          task[i]->thread.ra = (uint64)&__dummy;
18          task[i]->thread.sp = (uint64)task[i] + PGSIZE;
19
20          task[i]->pgd = (pagetable_t)alloc_page();
21          //用户态页表
22          task[i]->pgd = (uint64 *)memcpy(task[i]->pgd, swapper_pg_dir, 512 * sizeof(
                uint64));
23          //复制内核页表到用户页表
24
25          create_mapping(task[i]->pgd, (uint64)USER_END - (uint64)PGSIZE, (uint64)
                alloc_page() - (uint64)PA2VA_OFFSET, (uint64)PGSIZE, (uint64)
                VM_USER_PERM_R_W_X);
26          //对user stack的映射
27
28          uint64_t uapp_pg_size = (uint64)(((uint64)uapp_end - (uint64)uapp_start) / (
                uint64)PGSIZE);
29          if ((uint64)uapp_end - (uint64)uapp_start != (uint64)PGSIZE * uapp_pg_size)
```

```
30              uapp_pg_size++;
31
32          task[i]->uapp = alloc_pages(uapp_pg_size);
33          task[i]->uapp = (uint64 *)memcpy(task[i]->uapp, (uint64)uapp_start, (uint64)
                uapp_end - (uint64)uapp_start); //拷贝user space至自己的空间
34
35          Elf64_Ehdr *ehdr = (Elf64_Ehdr *)task[i]->uapp;
36
37          if (ehdr->e_ident[EI_MAG0] != ELFMAG0 || ehdr->e_ident[EI_MAG1] != ELFMAG1 ||
                ehdr->e_ident[EI_MAG2] != ELFMAG2 || ehdr->e_ident[EI_MAG3] != ELFMAG3)
38          {
39              task[i]->thread.sepc = (uint64)USER_START;
40              task[i]->thread.sscratch = (uint64)USER_END;
41              task[i]->thread.sstatus = (uint64)0 + ((uint64)1 << 5) + ((uint64)0 << 8) +
                    ((uint64)1 << 18);
42
43              create_mapping(task[i]->pgd, (uint64)USER_START, (uint64)task[i]->uapp - (
                    uint64)PA2VA_OFFSET, (uint64)uapp_end - (uint64)uapp_start, (uint64)
                    VM_USER_PERM_R_W_X);
44              //对user space的映射
45          }
46
47          else
48              load_program(task[i]);
49      }
```

注：为拷贝内存方便起见在 mm 中加入 memcpy 接口

memcpy

```
1   void *memcpy(void *dst, const void *src, uint64 n)
2   {
3
4       if (dst == NULL || src == NULL || n <= 0)
5           return NULL;
6
7       char *pdst = (char *)dst;
8       char *psrc = (char *)src;
9
10      if (pdst > psrc && pdst < psrc + n)
11      {
12          pdst = pdst + n - 1;
13          psrc = psrc + n - 1;
14
15          while (n--)
16              *pdst-- = *psrc--;
17      }
18
19      else
20      {
21          while (n--)
```

```
22              *pdst++ = *psrc++;
23          }
24
25          return dst;
26      }
```

修改 ___switch_to，加入保存/恢复 sepc，sstatus，sscratch 以及切换页表的逻辑。在切换了页表之后，通过 fence.i 和 vma.fence 来刷新 TLB 和 ICache。

___switch_to

```
1   __switch_to:
2   # save state to prev process
3
4   # restore state from next process
5
6   #切换页表
7   li t0, 0x8000000000000000
8   ld t1, 8*22(a1)
9   li t2, PA2VA_OFFSET
10  sub t1, t1, t2
11  srl t1, t1, 12
12  add t0, t0, t1
13
14  csrw satp, t0
15
16  # flush tlb
17  sfence.vma zero, zero
18
19  # flush icache
20  fence.i
21
22  #指针移动
23  la t0, current
24  sd a1, 0(t0)
25
26  ret
```

2. 修改中断入口/返回逻辑（__trap）以及中断处理函数（trap__handler）

1. 修改 ___dummy，在 S-Mode -> U->Mode 的时候，交换 sp 和 sscratch 的值

___dummy

```
1  __dummy:
2      #转换到U mode的stack，交换运行栈指针
3      csrr t0, sscratch
4      csrw sscratch, sp
5      add sp, zero, t0
6
7      #la t0, dummy
```

```
    #csrw sepc, t0
    sret
```

2. 修改 _trap 。同理在 _trap 的首尾我们都需要做类似的操作

_trap

```
    _traps:
    sd t0, -8(sp)
    addi sp, sp, -8
    csrr t0, sscratch
    beqz t0, _traps_skip_one
    #转换到U mode的stack，交换运行栈指针
    #栈指针切换，如果是内核线程不需要切换栈指针
    csrw sscratch, sp
    add sp, zero, t0

_traps_skip_one:
    #该段将32个寄存器和spec的值写入栈中，每个寄存器64位，并随时更新栈顶的地址

    csrr t0, sscratch
    beqz t0, _traps_skip_two
    #转换到U mode的stack，交换运行栈指针
    #栈指针切换，如果是内核线程不需要切换栈指针
    csrw sscratch, sp
    add sp, zero, t0
    ld t0, 0(sp)
    addi sp, sp, 8

_traps_skip_two:
    sret #一定要用sret返回sepc地址！！！
    }
```

添加 regs 结构体，便于方便地读取中断时寄存器值
trap_handler 中补充处理 SYSCALL 的逻辑。

trap_handler

```
    else // exception
    {
        switch (trap_type)
        {
        case 8: // Environment call from U-mode
            switch (regs->a7)
            {
            case SYS_WRITE:
                regs->a0 = sys_write(regs->a0, regs->a1, regs->a2);
                break;
            case SYS_GETPID:
                regs->a0 = sys_getpid();
                break;

            default:
```

```
            break;
        }

        regs->sepc += 4; //使sepc返回异常触发的后一条指令
        break;
    default:
        break;
    }
}
```

3. 添加系统调用

sys_write 将用户态传递的字符串打印到屏幕上

sys_write

```
uint64_t sys_write(unsigned int fd, const char *buf, size_t count)
{
    int i = 0;
    uint64 output_count = 0;

    for (i = 0; i < count; i++)
    {
        sbi_ecall(fd, 0, (uint64) * (buf + i), 0, 0, 0, 0, 0);
        output_count++;
    }

    return output_count;
}
```

sys_getpid 获取当前的 pid 放入 a0 中返回

sys_getpid

```
uint64_t sys_getpid()
{
    return (uint64)current->pid;
}
```

系统调用后手动将 sepc + 4

4. 修改 head.S 以及 start_kernel

在 start_kernel 中调用 schedule()

start_kernel

```
int start_kernel()
{
    schedule();
    test(); // DO NOT DELETE !!!

    return 0;
```

```
}
```

将 head.S 中 enable interrupt sstatus.SIE 逻辑注释

5. 添加 ELF 支持

检测到 elf 头后，按照分段做虚拟内存映射，并且存储正确的指令地址到 sepc

load_program

```
void load_program(struct task_struct *task)
{
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)task->uapp;

    uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;

    Elf64_Phdr *phdr;
    int load_phdr_cnt = 0;
    for (int i = 0; i < phdr_cnt; i++)
    {
        phdr = (Elf64_Phdr *)(phdr_start + sizeof(Elf64_Phdr) * i);
        if (phdr->p_type == PT_LOAD)
            switch (phdr->p_flags)
            {
            case PF_R:
                create_mapping(task->pgd, (uint64_t)phdr->p_vaddr, (uint64)ehdr - (
                    uint64)PA2VA_OFFSET + (uint64)phdr->p_offset, (uint64)phdr->p_memsz,
                     VM_USER_PERM_R);
                break;
            case PF_R + PF_W:
                create_mapping(task->pgd, (uint64_t)phdr->p_vaddr, (uint64)ehdr - (
                    uint64)PA2VA_OFFSET + (uint64)phdr->p_offset, (uint64)phdr->p_memsz,
                     VM_USER_PERM_R_W);
                break;
            case PF_R + PF_X:
                create_mapping(task->pgd, (uint64_t)phdr->p_vaddr, (uint64)ehdr - (
                    uint64)PA2VA_OFFSET + (uint64)phdr->p_offset, (uint64)phdr->p_memsz,
                     VM_USER_PERM_R_X);
                break;
            case PF_R + PF_W + PF_X:
                create_mapping(task->pgd, (uint64_t)phdr->p_vaddr, (uint64)ehdr - (
                    uint64)PA2VA_OFFSET + (uint64)phdr->p_offset, (uint64)phdr->p_memsz,
                     VM_USER_PERM_R_W_X);
                break;
            default:
                break;
            }
    }

    // pc for the user program
```

```
34          task->thread.sepc = ehdr->e_entry;
35          // sstatus bits set
36          task->thread.sstatus = (uint64)0 + ((uint64)1 << 5) + ((uint64)0 << 8) + ((uint64)
                1 << 18);
37          // user stack for user program
38          task->thread.sscratch = (uint64)USER_END;
39      }
```

四、 运行结果

纯二进制和 ELF 文件都可以正确在其用户态虚拟内存上运行并触发系统调用的中断进行正确系统调用，如图一

五、 思考题

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？（一对一，一对多，多对一还是多对多）

一对一的

2. 为什么 Phdr 中，p_filesz 和 p_memsz 是不一样大的？

p_filesz 是 segment 在文件的字节大小，p_memsz 是 segment 在文件的字节大小。

p_memsz 通常要大于 p_filesz，因为其可能包含存储没有初始化数据的.bss 段，将该段数据存储在硬盘中会浪费空间，所以在装入内存后开辟这段空间，导致内存大小和文件大小不同

3. 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？

因为不同的用户虽然虚拟地址相同但是其所映射的物理地址不同。

用户没有常规的方法知道自己的栈所在的物理地址，这是虚拟内存的作用之一。

```
...mm_init done!
...proc_init done!

[S-MODE] Hello RISC-V

[U-MODE] pid: 4, sp is 0000003ffffffe0, this is print No.1

[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.2

[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.2
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.3

[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.3
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.4

[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.3

[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.4
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.5
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.6

[U-MODE] pid: 4, sp is 0000003ffffffe0, this is print No.2
```

图 1: 结果