

浙江大学

操作系统实验四

课程名称:	操作系统
作业名称:	RV64 用户态程序
姓 名:	仇国智
学 号:	3220102181
电子邮箱:	3220102181@zju.edu.cn
联系电话:	13714176104
指导教师:	申文博

2024 年 11 月 15 日

目录

1	实验内容	2
2	实验原理	2
2.1	用户模式和内核模式	2
2.2	目标	2
2.3	系统调用约定	3
2.4	sstatus[SUM] 与 PTE[U]	3
2.5	用户态栈与内核态栈	3
2.6	ELF 程序	3
3	实验具体过程与代码实现	4
3.1	创建用户态进程	4
3.2	修改 task_init()	5
3.3	修改 __switch_to	7
3.4	更新中断跳转和恢复逻辑	8
3.5	修改 trap_handler	12
3.6	添加系统调用	13
3.7	调整时钟中断	15
3.8	加载用户态文件	16
4	实验结果与分析	17
5	遇到的问题及解决方法	23
6	总结与心得	24
7	思考题	24
7.1	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)	24
7.2	系统调用返回为什么不能直接修改寄存器?	24
7.3	针对系统调用, 为什么要手动将 sepc + 4?	24
7.4	为什么 Phdr 中, p_filesz 和 p_memsz 是不一样大的, 它们分别 表示什么?	24
7.5	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?	24

8 实验指导建议	25
----------------	----

1 实验内容

- 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 实验原理

2.1 用户模式和内核模式

处理器存在两种不同的模式: 用户模式 (U-Mode) 和内核模式 (S-Mode).

- 在用户模式下, 执行代码无法直接访问硬件, 必须委托给系统提供的接口才能访问硬件或内存;
- 在内核模式下, 执行代码对底层硬件具有完整且不受限制的访问权限, 它可以执行任何 CPU 指令并引用任何内存地址.

处理器根据处理器上运行的代码类型在这两种模式之间切换. 应用程序以用户模式运行, 而核心操作系统组件以内核模式运行.

2.2 目标

在 Lab3 中, 我们启用了虚拟内存, 这为进程间地址空间相互隔离打下了基础. 然而, 我们当时只创建了内核线程, 它们共用了地址空间 (共用一个内核页表 `swapper_pg_dir`). 在本次实验中, 我们将引入用户态进程:

- 当启动用户态应用程序时, 内核将为该应用程序创建一个进程, 并提供了专用虚拟地址空间等资源.
 - 每个应用程序的虚拟地址空间是私有的, 一个应用程序无法更改属于另一个应用程序的数据.
 - 每个应用程序都是独立运行的, 如果一个应用程序崩溃, 其他应用程序和操作系统将不会受到影响.

- 用户态应用程序可访问的虚拟地址空间是受限的。
 - 在用户态下, 应用程序无法访问内核的虚拟地址, 防止其修改关键操作系统数据.
 - 当用户态程序需要访问关键资源的时候, 可以通过系统调用来完成用户态程序与操作系统之间的互动.

2.3 系统调用约定

系统调用是用户态应用程序请求内核服务的一种方式. 在 RISC-V 中, 我们使用 `ecall` 指令进行系统调用, 当执行这条指令时, 处理器会提升特权模式, 跳转到异常处理函数, 处理这条系统调用.

Linux 中 RISC-V 相关的系统调用可以在 `include/uapi/asm-generic/unistd.h` 中找到, `syscall(2)` 手册页上对 RISC-V 架构上的调用说明进行了总结, 系统调用参数使用 `a0 - a5`, 系统调用号使用 `a7`, 系统调用的返回值会被保存到 `a0`, `a1` 中.

2.4 `sstatus[SUM]` 与 `PTE[U]`

- 当页表项 `PTE[U]` 置 0 时, 该页表项对应的内存页为内核页, U-Mode 下的代码无法访问
- 当页表项 `PTE[U]` 置 1 时, 该页表项对应的内存页为用户页, S-Mode 下的代码无法访问

如果想让 S 特权级下的程序能够访问用户页, 需要对 `sstatus[SUM]` 位置 1.

2.5 用户态栈与内核态栈

当用户态程序进行系统调用陷入内核处理时, 内核态程序也需要使用栈空间, 而这肯定不能和用户态使用同一个栈, 所以我们需要为用户态程序和内核态程序分别分配栈空间, 并在异常处理的过程中对栈进行切换.

2.6 ELF 程序

ELF(Executable and Linkable Format) 是当今被广泛使用的应用程序格式. 将程序封装成 ELF 格式的意义包括以下几点:

- ELF 文件可以包含将程序正确加载入内存的元数据 (metadata)

- ELF 文件在运行时可以由加载器 (loader) 将动态链接在程序上的动态链接库 (shared library) 正确地 从硬盘或内存中加载
- ELF 文件包含的重定位信息可以让该程序继续和其他可重定位文件和库再次链接, 构成新的可执行文件

为了简化实验步骤, 我们使用的是静态链接的程序, 不会涉及链接动态链接库的内容.

3 实验具体过程与代码实现

3.1 创建用户态进程

结构体更新

```

1  /* 线程状态段数据结构 */
2  struct thread_struct {
3      ...
4      uint64_t sepc, sstatus, sscratch;
5  };
6
7  /* 线程数据结构 */
8  struct task_struct {
9      ...
10
11     struct thread_struct thread;
12
13     pagetable_ptr_t pagetable;
14 };

```

在本节中, 我们更新了两个结构体, 用于表示线程状态和任务信息. 这些更新涉及到线程的状态保存以及用户态进程的内存映射.

首先, 我们定义了一个新的线程状态数据结构 ‘thread_struct’, 该结构体包含了三个寄存器 ‘sepc’, ‘sstatus’ 和 ‘sscratch’. 这些寄存器分别用于存储程序计数器, 状态寄存器和临时保存值, 用于在用户态到内核态上下文切换时保存和恢复寄存器状态.

其次, 我们定义了一个 ‘pagetable’ 成员, 用于表示任务的页表信息, 确保任务能够正确地访问其虚拟内存.

3.2 修改 task_init()

```
1  void task_init()
2  {
3
4      ...
5
6      /* YOUR CODE HERE */
7      for (int i = 0; i < NR_TASKS; i++)
8      {
9          if (task[i] == NULL)
10         {
11             ...
12             task[i]->thread.sstatus = csr_read(sstatus) | SPIE |
                SUM;
13             task[i]->thread.sstatus &= (~SPP) & (~SIE);
14             task[i]->thread.sscratch = USER_END;
15             task[i]->pagetable = (pagetable_ptr_t)kalloc();
16
17             copy_pgtbl(task[i]->pagetable,
                (pagetable_ptr_t)swapper_pg_dir);
18             uint64_t user_statck = alloc_pages(1);
19             create_mapping(task[i]->pagetable, USER_END - PGSIZE,
                VA2PA(user_statck), PGSIZE, PTE_R | PTE_W | PTE_X |
                PTE_U | PTE_V);
20             // load_binary(task[i]->pagetable, _sramdisk,
                _eramdisk);
21             task[i]->thread.sepc = load_elf(task[i]->pagetable,
                _sramdisk);
22         }
23     }
24     ...
25 }
```

在此代码段中, 我们为每个进程初始化了必要的系统资源和配置. 具体来说:

1. 设置进程的 `sstatus` 寄存器, 清除特权位 (SPP), 设置 SPIE 和 SUM, 这样在退出内核态时, 能够正确恢复到用户态并开启中断, 同时在内核态时, 能够访问用户态的地址空间.
2. 我们还清除了 SIE 位, 避免在用户态初始化时被中断打断.
3. 为每个进程分配一个独立的页表, 并将其从 `swapper` 页目录中复制, 确保任务的虚拟地址空间正确设置.
4. 分配用户栈空间, 并将其映射到进程的页表中, 以便进程在用户态执行时使用.
5. 使用 `load_elf` (`load_binary`) 函数加载 ELF 文件 (可执行文件) 到内存中, 并配置页表相关的映射.
6. 将进程的入口点地址保存在 `sepc` 寄存器中, 以便在进程切换时能够正确恢复执行.

```
1  void copy_pgtbl(pagetable_ptr_t dst, pagetable_ptr_t src)
2  {
3      memcpy((void *)dst, (void *)src, PGSIZE);
4      for (int i = 0; i < 512; i++)
5      {
6          if (src[i] & PTE_V)
7          {
8              if (!(src[i] & PTE_R) && !(src[i] & PTE_W) && !(src[i]
9                  & PTE_X))
10             {
11                 pagetable_ptr_t current_pagetable1 =
12                     (pagetable_ptr_t)PA2VA(PTE2ADDR(src[i], 3));
13                 pagetable_ptr_t new_pagetable1 =
14                     (pagetable_ptr_t)kalloc();
15                 dst[i] = ADDR2PTE((uint64_t)VA2PA(
16                     (uint64_t)new_pagetable1 ), PTE_V, 3);
17                 memcpy((void *)new_pagetable1, (void
18                     *)current_pagetable1, PGSIZE);
19                 for (int j = 0; j < 512; j++)
20                 {
21                     if (current_pagetable1[j] & PTE_V)
```



```

17         {
18             if (!(current_pagetable1[j] & PTE_R) &&
                !(current_pagetable1[j] & PTE_W) &&
                !(current_pagetable1[j] & PTE_X))
19             {
20                 pagetable_ptr_t current_pagetable2 =
                    (pagetable_ptr_t)PA2VA(
                        PTE2ADDR(current_pagetable1[j], 3) );
21                 pagetable_ptr_t new_pagetable2 =
                    (pagetable_ptr_t)kalloc();
22                 new_pagetable1[j] = ADDR2PTE(
                    (uint64_t)VA2PA(
                        (uint64_t)new_pagetable2 ), PTE_V, 3
                    );
23                 memcpy((void *)new_pagetable2, (void
                    *)current_pagetable2, PGSIZE);
24             }
25         }
26     }
27 }
28 }
29 }
30 // printk("copy_pgtbl done!\n");
31 }

```

我们还编写了一个函数 `copy_pgtbl`, 用于遍历页表, 并复制页表项, 以实现页表的层次复制, 确保每个进程的页表独立, 并且能够正确访问虚拟地址空间.

3.3 修改 `__switch_to`

```

1     .extern set_pgtbl
2     .globl __switch_to
3 __switch_to:
4     ...
5     csrr s1, sepc
6     csrr s2, sstatus

```

```

7    csrr s3, sscratch
8    sd s1, 112(a0)
9    sd s2, 120(a0)
10   sd s3, 128(a0)
11
12   addi s1, a1, 0x0
13   ld a0, 136(s1)
14   call set_pgtbl
15   addi a1, s1, 0x0
16
17   ld s1, 112(a1)
18   ld s2, 120(a1)
19   ld s3, 128(a1)
20   csrw sepc, s1
21   csrw sstatus, s2
22   csrw sscratch, s3
23   ...
24   ret

```

我们在 `__switch_to` 函数中, 需要添加新加入的 `sstatus`, `sepc`, `sscratch` 寄存器的保存和恢复逻辑. 同时, 我们还需要添加页表切换的逻辑, 注意我们这里只需要配置参数为新进程的页表地址, 随后调用上一个实验中实现的 `set_pgtbl` 函数即可.

3.4 更新中断跳转和恢复逻辑

```

1 _traps:
2
3     csrrw sp, sscratch, sp
4     bnez sp, _usertrap
5
6 _kerneltrap:
7     csrrw sp, sscratch, sp
8     ...
9
10 _usertrap:
11     # sd x0, -272(sp)

```

```

12     sd x1, -264(sp)
13     # sd x2, -256(sp)
14     sd x3, -248(sp)
15     sd x4, -240(sp)
16     sd x5, -232(sp)
17     sd x6, -224(sp)
18     sd x7, -216(sp)
19     sd x8, -208(sp)
20     sd x9, -200(sp)
21     sd x10, -192(sp)
22     sd x11, -184(sp)
23     sd x12, -176(sp)
24     sd x13, -168(sp)
25     sd x14, -160(sp)
26     sd x15, -152(sp)
27     sd x16, -144(sp)
28     sd x17, -136(sp)
29     sd x18, -128(sp)
30     sd x19, -120(sp)
31     sd x20, -112(sp)
32     sd x21, -104(sp)
33     sd x22, -96(sp)
34     sd x23, -88(sp)
35     sd x24, -80(sp)
36     sd x25, -72(sp)
37     sd x26, -64(sp)
38     sd x27, -56(sp)
39     sd x28, -48(sp)
40     sd x29, -40(sp)
41     sd x30, -32(sp)
42     sd x31, -24(sp)
43     csrr t1, sepc
44     sd t1, -16(sp)
45     csrr t2, sstatus
46     sd t2, -8(sp)

```

```

47     csrr t3, sscratch
48     sd t3, -256(sp)
49
50     addi sp, sp, -272
51     csrr a0, scause
52     csrr a1, sepc
53     addi a2, sp, 0
54     call trap_handler
55
56     addi sp, sp, 272
57     ld t1, -16(sp)
58     csrwrw sepc, t1
59     ld t2, -8(sp)
60     csrwrw sstatus, t2
61     csrwrw sscratch, sp
62     # ld x0, -272(sp)
63     ld x1, -264(sp)
64     # ld x2, -256(sp)
65     ld x3, -248(sp)
66     ld x4, -240(sp)
67     ld x5, -232(sp)
68     ld x6, -224(sp)
69     ld x7, -216(sp)
70     ld x8, -208(sp)
71     ld x9, -200(sp)
72     ld x10, -192(sp)
73     ld x11, -184(sp)
74     ld x12, -176(sp)
75     ld x13, -168(sp)
76     ld x14, -160(sp)
77     ld x15, -152(sp)
78     ld x16, -144(sp)
79     ld x17, -136(sp)
80     ld x18, -128(sp)
81     ld x19, -120(sp)

```

```

82    ld x20, -112(sp)
83    ld x21, -104(sp)
84    ld x22, -96(sp)
85    ld x23, -88(sp)
86    ld x24, -80(sp)
87    ld x25, -72(sp)
88    ld x26, -64(sp)
89    ld x27, -56(sp)
90    ld x28, -48(sp)
91    ld x29, -40(sp)
92    ld x30, -32(sp)
93    ld x31, -24(sp)
94    ld x2, -256(sp)
95    sret

```

sscratch 和 sp 的状态如下, 我们可以利用 sscratch 寄存器来判断进入中断前是否是用户态. 我们首先将 sp 和 sscratch 寄存器交换, 如果 sp 为 0, 则说明进入中断前是内核态, 否则为用户态.

- 用户态:
 - sscratch = 内核栈地址
 - sp 指向用户栈
- 内核态:
 - sscratch = 0
 - sp 指向内核栈

在判断进入中断前是否是用户态后, 假如是内核态, 则需要交换回去, 继续执行上一个实验实现的内核态中断处理逻辑, 否则则按以下用户态中断处理逻辑执行.

1. 保存非 sp 的通用寄存器到内核栈中的 pt_regs 结构体中.
2. 保存 sepc 和 sstatus 寄存器到 pt_regs 结构体中.
3. 从 sscratch 寄存器中读取原 sp 值, 并将其保存到 pt_regs 结构体中.
4. 清零 sscratch 寄存器.

5. 配置 `trap_handler` 函数的栈空间
6. 配置 `trap_handler` 函数参数: `scause`, `sepc`, `pt_regs` 结构体.
7. 调用 `trap_handler` 函数.
8. 恢复 `sp` 至原来内核栈顶.
9. 恢复 `sepc` 和 `sstatus` 寄存器.
10. 将 `sp`(内核栈) 值写入 `sscratch` 寄存器.
11. 恢复通用寄存器 (非 `sp`)
12. 恢复 `sp` 寄存器.
13. 执行 `sret` 指令, 从内核态返回.

```
1 __dummy:
2     csrrw sp, sscratch, sp
3     sret
```

我们定义了一个空函数 `__dummy`, 用于在用户态初始化时, 提供从内核态退出的接口. 只需交换 `sp` 和 `sscratch` 寄存器, 配置 `sp` 至用户栈顶, 然后执行 `sret` 指令, 即可从内核态返回.

3.5 修改 `trap_handler`

我们首先定义一个数据结构用于储存用户态切换到内核态保存的寄存器.

```
1 struct pt_regs {
2     uint64_t x[32];
3     uint64_t sepc;
4     uint64_t sstatus;
5 };
```

w

```
1 void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs
    *regs)
2 {
3
```

```

4     if ((scause >> 63) == 0)
5     {
6         // Handle exceptions
7         switch (scause)
8         {
9             ...
10            case 0x8:
11                call_syscall(regs);
12                return;
13            ...
14        }
15    }
16    ...
17 }

```

当 `scause` 表明是系统调用时, 我们调用 `call_syscall` 函数, 传递 `pt_regs` 结构体作为参数, 处理系统调用.

3.6 添加系统调用

```

1     static uint64_t sys_write(struct pt_regs *regs);
2     static uint64_t sys_getpid(struct pt_regs *regs);
3     static uint64_t (*syscall_table[])(struct pt_regs *regs) = {
4         [SYS_WRITE] = sys_write,
5         [SYS_GETPID] = sys_getpid,
6     };
7
8     void call_syscall(struct pt_regs *regs)
9     {
10         int callnum = regs->x[17];
11         // printk("syscall %d\n", callnum);
12         if (callnum >= 0 && callnum < sizeof(syscall_table) /
13             sizeof(syscall_table[0]) && syscall_table[callnum])
14         {
15             regs->x[10] = syscall_table[callnum](regs);
16         }
17     }

```

```

16     else
17     {
18         regs->x[10] = -1;
19     }
20     regs->sepc += 4;
21 }
22
23 static uint64_t sys_write(struct pt_regs *regs)
24 {
25     int fd = regs->x[10];
26     char *buf = (char *)regs->x[11];
27     int count = regs->x[12];
28     if (fd == 1)
29     {
30         for (int i = 0; i < count; i++)
31         {
32             printk("%c", buf[i]);
33         }
34         return count;
35     }
36     else
37     {
38         return -1;
39     }
40 }
41
42 static uint64_t sys_getpid(struct pt_regs *regs)
43 {
44     return getpid();
45 }

```

我们采用一个函数指针数组 `syscall_table` 来储存系统调用函数的地址, 通过系统调用号来索引对应的系统调用函数, 并调用.

3.7 调整时钟中断

在初始化内核时关闭中断, 中断在内核态返回到用户态时开启, 避免时钟中断干扰内核初始化.

```
1  _start:
2  la sp, boot_stack_top
3
4  la a0, _traps
5  csrw stvec, a0
6
7  li a0, STIE
8  csrs sie, a0
9
10
11 # li a0, SIE
12 # csrs sstatus, a0
13
14 call setup_vm
15 call relocate
16 call mm_init
17 call setup_vm_final
18 call task_init
19 call clock_init
20
21 call start_kernel
```

在初始化完成后, 主动调用 `schedule`, 切换到用户态进程, 因为此时不会发生中断, 所以需要手动进行第一次进程切换.

```
1  extern void test();
2  int start_kernel() {
3      schedule();
4      test();
5      return 0;
6  }
```

3.8 加载用户态文件

加载可执行文件

```
1 void load_binary(pagetable_ptr_t pagetable, char *start, char
   *end)
2 {
3     uint64_t ramdisk_end = PGROUNDUP((uint64_t)end);
4     uint64_t ramdisk_start = PGROUNDDOWN((uint64_t)start);
5     uint64_t ramdisk_size = ramdisk_end - ramdisk_start;
6     uint64_t ret = (uint64_t)alloc_pages(ramdisk_size);
7     memcpy((void *)ret, start, ramdisk_size);
8     create_mapping(pagetable, USER_START, VA2PA(ret),
        ramdisk_size, PTE_R | PTE_W | PTE_X | PTE_U | PTE_V);
9 }
```

我们定义了一个 `load_binary` 函数, 直接将二进制文件加载到 `0x0` 的用户空间, 并建立页表映射.

加载 ELF 文件

```
1 uint64_t load_elf(pagetable_ptr_t pagetable, char *start)
2 {
3     Elf64_Ehdr *elf = (Elf64_Ehdr *)start;
4     for (int i = 0; i < elf->e_phnum; i++)
5     {
6         Elf64_Phdr *phdr = (Elf64_Phdr *)((uint64_t)start +
            elf->e_phoff + i * sizeof(Elf64_Phdr));
7         if (phdr->p_type != PT_LOAD)
8             continue;
9         uint64_t va = phdr->p_vaddr;
10        uint64_t va_start = PGROUNDDOWN(va);
11        // printk("va = %lx\n", va);
12        uint64_t memsz = phdr->p_memsz;
13        uint64_t va_end = PGROUNDUP(va + memsz);
14        uint64_t filesz = phdr->p_filesz;
```

```

15         uint64_t offset = phdr->p_offset;
16         uint64_t flags = phdr->p_flags;
17         uint64_t alloc_addr = (uint64_t)alloc_pages((va_end -
            va_start) / PGSIZE);
18         uint64_t alloc_offset = va % PGSIZE;
19         memset((void *)alloc_addr + alloc_offset + filesz, 0,
            memsz - filesz);
20         memcpy((void *)alloc_addr + alloc_offset, (void
            *) (uint64_t)start + offset, filesz);
21         uint64_t perm = PTE_V | PTE_U;
22         if (flags & PF_X)
23             perm |= PTE_X;
24         if (flags & PF_W)
25             perm |= PTE_W;
26         if (flags & PF_R)
27             perm |= PTE_R;
28         create_mapping(pagetable, va_start, VA2PA(alloc_addr),
            va_end - va_start, perm);
29     }
30     return elf->e_entry;
31 }

```

我们要解析 ELF 文件的程序头表, 并根据程序头表中的信息, 将 ELF 文件加载到内存中, 并建立页表映射设置权限. 其中相对文件偏移 `p_offset` 指出相应 segment 的内容从 ELF 文件的第 `p_offset` 字节开始, 在文件中的大小为 `p_filesz`, 它需要被分配到以 `p_vaddr` 为首地址的虚拟内存位置, 在内存中它占用大小为 `p_memsz`. 也就是说, 这个 segment 使用的内存就是 `[p_vaddr, p_vaddr + p_memsz)` 这一连续区间, 然后将 segment 的内容从 ELF 文件中读入到这一内存区间, 并将 `[p_vaddr + p_filesz, p_vaddr + p_memsz)` 对应的物理区间清零. 最后返回 ELF 文件的入口地址.

4 实验结果与分析

由于实验给出的验证代码过于简陋, 我编写了两种验证函数.

- 第一种是使用两种排序算法, 快速排序和归并排序, 对一个数组进行排序, 然后比较排序结果是否相同.

- 第二种是先将寄存器保存到栈中, 然后恢复寄存器, 比较恢复后的寄存器值是否与保存的寄存器值相同.

随后循环往复调用这两种验证函数, 中断就会发生在这两种验证函数中, 以验证中断处理是否正确.

排序验证代码如下:

```

1      /* gpt 生成的快排函数 */
2      ...
3
4      /* gpt 生成的归并函数 */
5      ...
6
7      bool compareSortResults(int arr[], int size) {
8          int arrQuick[size], arrMerge[size];
9
10         for (int i = 0; i < size; i++) {
11             arrQuick[i] = arr[i];
12             arrMerge[i] = arr[i];
13         }
14
15         quickSort(arrQuick, 0, size - 1);
16
17         mergeSort(arrMerge, 0, size - 1);
18
19         for (int i = 0; i < size; i++) {
20             if (arrQuick[i] != arrMerge[i]) {
21                 return false;
22             }
23         }
24
25         return true;
26     }

```

寄存器验证代码如下:

```

1      // compare regs except t1
2      bool compare_regs_saved(void) {

```

```

3      bool result = true;
4      __asm__ volatile (
5
6          "addi sp, sp, -248\n"
7
8
9          "sd x1, 0(sp)\n"
10         "sd x2, 8(sp)\n"
11         "sd x3, 16(sp)\n"
12         "sd x4, 24(sp)\n"
13         "sd x5, 32(sp)\n"
14         "sd x6, 40(sp)\n"
15         "sd x7, 48(sp)\n"
16         "sd x8, 56(sp)\n"
17         "sd x9, 64(sp)\n"
18         "sd x10, 72(sp)\n"
19         "sd x11, 80(sp)\n"
20         "sd x12, 88(sp)\n"
21         "sd x13, 96(sp)\n"
22         "sd x14, 104(sp)\n"
23         "sd x15, 112(sp)\n"
24         "sd x16, 120(sp)\n"
25         "sd x17, 128(sp)\n"
26         "sd x18, 136(sp)\n"
27         "sd x19, 144(sp)\n"
28         "sd x20, 152(sp)\n"
29         "sd x21, 160(sp)\n"
30         "sd x22, 168(sp)\n"
31         "sd x23, 176(sp)\n"
32         "sd x24, 184(sp)\n"
33         "sd x25, 192(sp)\n"
34         "sd x26, 200(sp)\n"
35         "sd x27, 208(sp)\n"
36         "sd x28, 216(sp)\n"
37         "sd x29, 224(sp)\n"

```

```

38      "sd x30, 232(sp)\n"
39      "sd x31, 240(sp)\n"
40
41
42      "ld t1, 0(sp)\n"
43      "bne t1, x1, not_equal\n"
44      "ld t1, 8(sp)\n"
45      "bne t1, x2, not_equal\n"
46      "ld t1, 16(sp)\n"
47      "bne t1, x3, not_equal\n"
48      "ld t1, 24(sp)\n"
49      "bne t1, x4, not_equal\n"
50      "ld t1, 32(sp)\n"
51      "bne t1, x5, not_equal\n"
52      "ld t1, 40(sp)\n"
53      "bne t1, x6, not_equal\n"
54      "ld t1, 48(sp)\n"
55      "bne t1, x7, not_equal\n"
56      "ld t1, 56(sp)\n"
57      "bne t1, x8, not_equal\n"
58      "ld t1, 64(sp)\n"
59      "bne t1, x9, not_equal\n"
60      "ld t1, 72(sp)\n"
61      "bne t1, x10, not_equal\n"
62      "ld t1, 80(sp)\n"
63      "bne t1, x11, not_equal\n"
64      "ld t1, 88(sp)\n"
65      "bne t1, x12, not_equal\n"
66      "ld t1, 96(sp)\n"
67      "bne t1, x13, not_equal\n"
68      "ld t1, 104(sp)\n"
69      "bne t1, x14, not_equal\n"
70      "ld t1, 112(sp)\n"
71      "bne t1, x15, not_equal\n"
72      "ld t1, 120(sp)\n"

```

```

73      "bne t1, x16, not_equal\n"
74      "ld t1, 128(sp)\n"
75      "bne t1, x17, not_equal\n"
76      "ld t1, 136(sp)\n"
77      "bne t1, x18, not_equal\n"
78      "ld t1, 144(sp)\n"
79      "bne t1, x19, not_equal\n"
80      "ld t1, 152(sp)\n"
81      "bne t1, x20, not_equal\n"
82      "ld t1, 160(sp)\n"
83      "bne t1, x21, not_equal\n"
84      "ld t1, 168(sp)\n"
85      "bne t1, x22, not_equal\n"
86      "ld t1, 176(sp)\n"
87      "bne t1, x23, not_equal\n"
88      "ld t1, 184(sp)\n"
89      "bne t1, x24, not_equal\n"
90      "ld t1, 192(sp)\n"
91      "bne t1, x25, not_equal\n"
92      "ld t1, 200(sp)\n"
93      "bne t1, x26, not_equal\n"
94      "ld t1, 208(sp)\n"
95      "bne t1, x27, not_equal\n"
96      "ld t1, 216(sp)\n"
97      "bne t1, x28, not_equal\n"
98      "ld t1, 224(sp)\n"
99      "bne t1, x29, not_equal\n"
100     "ld t1, 232(sp)\n"
101     "bne t1, x30, not_equal\n"
102     "ld t1, 240(sp)\n"
103     "bne t1, x31, not_equal\n"
104
105     "li t1, 1\n"
106
107     "j done\n"

```

```

108
109     "not_equal:\n"
110     "li t1, 0\n"
111
112     "done:\n"
113
114     "addi sp, sp, 248\n"
115     "mv %[result], t1\n"
116     : [result] "=r" (result)
117     :
118     : "t1"
119 );
120 return result;
121 }

```

在主程序中, 调用两个验证方法.

```

1  int main() {
2      register void *current_sp __asm__("sp");
3      unsigned long long seed=0;
4      while (1) {
5          printf("[U-MODE] pid: %ld, sp is %p, this is print
6              No. %d\n", getpid(), current_sp, ++counter);
7          for (unsigned int i = 0; i < 0x4FFF; i++)
8          {
9              int test[100];
10             for(int i=0;i<100;i++)
11             {
12                 test[i]=rand(&seed);
13             }
14             if(!compareSortResults(test,100))
15             {
16                 printf("false\n");
17             }
18             if(!compare_regs_saved())
19             {

```



```

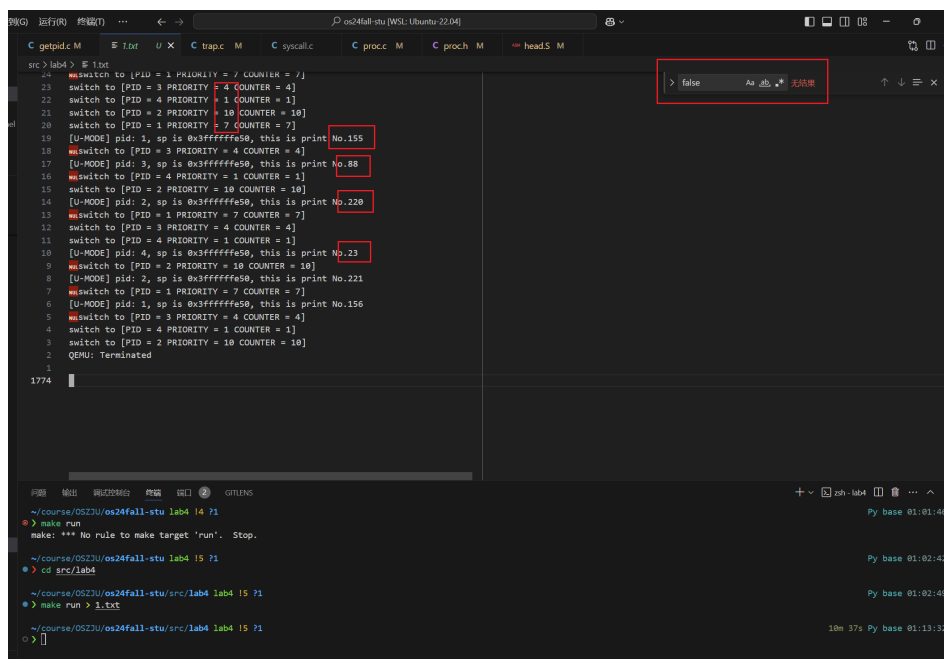
19     printf("false\n");
20 }
21 }
22 }
23 return 0;
24 }

```

我们使用命令

```
1 $ make run > 1.txt
```

验证结果如下:



```

src> lab4 > 1.txt
24 switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
23 switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
22 switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
21 switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
20 switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
19 [U-MODE] pid: 1, sp is 0x3ffffffe50, this is print No.155
18 switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
17 [U-MODE] pid: 3, sp is 0x3ffffffe50, this is print No.88
16 switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
15 switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
14 [U-MODE] pid: 2, sp is 0x3ffffffe50, this is print No.220
13 switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
12 switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
11 switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
10 [U-MODE] pid: 4, sp is 0x3ffffffe50, this is print No.23
9 switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
8 [U-MODE] pid: 2, sp is 0x3ffffffe50, this is print No.221
7 switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
6 [U-MODE] pid: 1, sp is 0x3ffffffe50, this is print No.156
5 switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
4 switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
3 switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
2 QEMU: Terminated
1
1774

```

图 1: 验证结果

我们可以看到, 无 false 输出, 说明排序结果和寄存器值均正确; 同时 getpid 的循环次数为 220:155:88:23, 和优先级比值 10:7:4:1 一致, 说明中断处理正确.

5 遇到的问题及解决方法

问题: 处理完中断后, 恢复寄存器的值随机出错.

解决方式: 在储存完 pt_regs 后, 需要更新 sp, 避免后面调用 trap_handler 写入 pt_regs.

6 总结与心得

在本次实验中, 我学习了用户态进程的创建和切换. 通过实验, 我掌握了用户态进程的创建和切换的原理, 深刻的理解了虚拟内存的映射机制, 以及 ELF 文件的加载过程.

7 思考题

7.1 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)

我们采用的是一对一的线程模型, 即每个用户态线程对应一个内核态线程.

7.2 系统调用返回为什么不能直接修改寄存器?

因为在退出中断的时候, 会复写所有寄存器的值, 恢复到 `pt_regs` 中的值, 所以直接修改寄存器是没有意义的, 必须要修改 `pt_regs` 中的值.

7.3 针对系统调用, 为什么要手动将 `sepc + 4`?

因为 `sepc` 存储的是触发系统调用的指令地址, 而不是系统调用的下一条指令地址, 所以需要手动将 `sepc + 4`. 否则在返回的时候会再次触发系统调用, 造成死循环.

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

在 `Phdr` 中, `p_filesz` 表示段在文件中的大小, `p_memsz` 表示段在内存中的大小. 通常情况下, `p_filesz <= p_memsz`. 这是因为比如 `bss` 段, 因为它是未初始化的数据, 所以不需要在文件储存以节省空间, 但是在内存中需要分配空间.

7.5 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?

这是因为不同的进程的页表是不一样的, 一般来说不同进程同一个虚拟地址对应的物理地址是不一样的. 用户没有常规的方法知道自己栈所在的物理地址. 除非用户

可以访问页表, 但是这会破坏隔离性, 造成安全问题.

8 实验指导建议

我建议增大程序正确性检查的粒度, 比如可以采用我在上面实验结果分析中采用的验证方法. 这样可以避免之后的实验中再爆出错误, 那样会大大增加 debug 的难度. 在目前的测试代码下, 我原先的错误代码会出现刚开始运行是完全正常的, 但是 1000+ 次时钟中断后会突然报错的情况.