

# OS lab5 Report

## 实验过程

### 4.1 准备工程

按照实验手册完成以下操作：

- 修改 `vmlinux.lds.S`
- 修改 `defs.h`
- 同步 `ropo`
- 修改根目录下的 Makefile, 将 `user` 纳入工程管理。注意此处 `user` 部分需要放在 `arch/riscv` 部分前面, 因为该部分中的 `vmlinux.lds.S` 用到了 `user` 部分的文件 (如 `uapp.o`)

```
1 all:
2     ${MAKE} -C user all
3     ${MAKE} -C lib all
4     ${MAKE} -C init all
5     ${MAKE} -C arch/riscv all
6
7     @echo -e '\n'Build Finished OK
8
```

### 4.2 创建用户态进程

1. 修改 `proc.h` 中的 `NR_TASKS` 为

```
1 #define NR_TASKS (1 + 4) // 用于控制 最大线程数量 (idle 线程 + 31 内核线程)
```

2. 更改 `thread_struct`, 添加 `sepc` `sstatus` `sscratch`。

```
1 /* 线程状态段数据结构 */
2 struct thread_struct {
3     uint64 ra;
4     uint64 sp;
5     uint64 s[12];
6     uint64_t sepc, sstatus, sscratch;
7 };
```

3. 按照实验手册修改 `task_struct`。
4. 修改 `task_init`

有几个修改的重点 (坑)：

- 由于我们的 `kernel_sp` 和 `user_sp` 是存在 `thread_info` 里面的, 而我们可以在 `proc.h` 中发现, 在 `task_struct` 里面的 `thread_info` 是一个指针, 这意味着我们并没有给它分配内存, 这时候如果我们直接给里面赋值就会报错。因此我们在赋值之前需要先分配一个内存地址给 `thread_info`。

- 这个在 `task_struct` 中出现的没有注释的 `pgd` 在经过分析之后应该是用于存储每个用户进程各自的 `satp`，其全名可能是 `page_dir`。该参数将会在我们切换进程的时候用到。

```

1      //set S-Mode and U-Mode stack
2      //一个坑,thread_info没有分配内存。
3      //解决方案:给一个page(不过有点浪费QAQ)
4      task[i]->thread_info = (struct thread_info*)alloc_page();
5      task[i]->thread_info->kernel_sp = task[i]->thread.sp;
6      // task[i]->thread_info->kernel_sp = (uint64)task[i] + PGSIZE;
7      task[i]->thread_info->user_sp = alloc_page();
8
9      //为每个用户态创建自己的页表
10     pagetable_t pageTable = alloc_page();
11     //复制内核页表,避免在陷入切换U/S-Mode时需要立即更换satp
12     for (uint64 i = 0; i < PGSIZE; i++)
13         ((char*)pageTable)[i] = ((char*)swapper_pg_dir)[i];
14
15     //映射uapp,给予XWR权限,UV也为1
16     uint64 virtualAddress = USER_START;
17     uint64 physicalAddress = (uint64)(uapp_start)-PA2VA_OFFSET;
18     create_mapping(pageTable, virtualAddress, physicalAddress, (uint64)
19     (uapp_end)-(uint64)(uapp_start), 31);
20
21     //映射U-Mode Stack,给予WR权限,UV也为1
22     virtualAddress = USER_END-PGSIZE;
23     physicalAddress = task[i]->thread_info->user_sp-PA2VA_OFFSET;
24     create_mapping(pageTable, virtualAddress, physicalAddress, PGSIZE,
25     23);
26
27     //set sepc
28     task[i]->thread.sepc = USER_START;
29
30     //set sstatus
31     uint64 pre_sstatus = csr_read(sstatus);
32     uint64 now_sstatus = pre_sstatus & 0xffffffffffffeff;
33     //sstatus[SPP] = 0
34     now_sstatus = now_sstatus | (1<<5); //sstatus[SPIE] = 1
35     now_sstatus = now_sstatus | (1<<18); //sstatus[SUM] = 1
36     task[i]->thread.sstatus = now_sstatus;
37
38     //set sscratch
39     task[i]->thread.sscratch = USER_END;
40
41     //设置pgd,即satp
42     uint64 pre_satp = csr_read(satp);
43     uint64 satp_prefix = (pre_satp >> 44) << 44;
44     uint64 now_satp = satp_prefix | (((uint64)pageTable-PA2VA_OFFSET) >>
45     12);
46     task[i]->pgd = now_satp;

```

## 5. 修改 `__switch_to`

```

1  # save state to prev process
2  # 略

```

```

3  # save csr
4      csrr t0, sepc
5      sd t0, 152(a0)
6      csrr t0, sstatus
7      sd t0, 160(a0)
8      csrr t0, sscratch
9      sd t0, 168(a0)
10     csrr t0, satp
11     sd t0, 176(a0)
12 # restore state from next process
13 # 略
14 # write csr
15     ld t0, 152(a1)
16     csrw sepc, t0
17     ld t0, 160(a1)
18     csrw sstatus, t0
19     ld t0, 168(a1)
20     csrw sscratch, t0
21     ld t0, 176(a1)
22     csrw satp, t0
23
24     # flush tlb
25     sfence.vma zero, zero
26
27     # flush icache
28     fence.i
29
30     ret

```

## 4.3 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`。该函数仅在某个线程第一次被调用时进入，我们不再需要在此处修改 `sepc`，因为在之前的 `__switch_to` 已经设置好了 `sepc`。在这里我们仅需要切换Stack，因为我们即将从内核态进入到用户态。

```

1  __dummy:
2      # la t0, dummy
3      # csrw sepc, t0;
4      # 不需要设置sepc,因为我们不再去dummy()了
5      # 而且__swich_to已经设置好了我们要去的sepc(USER_START)
6
7      # 交换sp和sscratch来切换U-stack和S-stack
8      csrr t0, sscratch
9      csrw sscratch, sp
10     mv sp, t0
11     sret

```

2. 修改 `_traps`

我们需要在 `_traps` 的首尾加上对 `sscratch` 的检查，如果 `sscratch` 不为0，说明触发trap的是用户进程，那我们我们就应该在进入trap前切换成 `S-stack`，在退出trap前切换成 `U-stack`。

3. 修改 `trap_handler`

我们为 `trap_handle` 函数增加 `struct pt_regs *regs` 定义

```
1 struct pt_regs
2 {
3     uint64 reg[32];
4     uint64 sepc;
5     uint64 sstatus;
6 };
```

## 4.4 添加系统调用

### 1. 添加系统调用模块

```
1 // 系统调用号使用a7,入参a0~a5,返回值a0/a1
2 void syscall(struct pt_regs *regs, uint64 sepc){
3     if(regs->reg[17] == 64){ //sys_write
4         sys_write(regs,regs->reg[10],regs->reg[11],regs->reg[12]);
5     }
6     else if(regs->reg[17] == 172){ //sys_getpid
7         sys_getpid(regs);
8     }
9     else{
10         printk("Other syscall occurred! a7=%d\n",regs->reg[17]);
11     }
12 }
13
14
15 void sys_write(struct pt_regs *regs, unsigned int fd, const char* buf,
16 size_t count){
17     if (fd == 1) {
18         uint64 return_value;
19         for(size_t i = 0; i < count; i++){
20             return_value = printk("%c", buf[i]);
21         }
22         regs->reg[10] = return_value;
23     }
24 }
25
26 void sys_getpid(struct pt_regs *regs){
27     regs->reg[10] = current->pid;
28 }
```

### 2. 手动 `sepc+4`

如果是用户态发出的系统调用，那我们需要手动在trap调用之后给 `sepc` 加上4来继续运行下一条指令。

```

1  # entry.S
2  # 首先使用t0取出sepc和sstatus
3      ld t0,264(sp)
4      csrw sstatus,t0
5      ld t0,256(sp)
6  # 如果不是系统调用，那就不用给sepc加上4了
7      csrr t1, scause
8      addi t2, zero, 0x8
9      bne t1, t2, _no_need_change_sepc
10     addi t0,t0,0x4
11 _no_need_change_sepc:
12     csrw sepc,t0

```

## 4.5 修改 head.S 以及 start\_kernel

1. 在 `start_kernel` 函数中加上 `schedule()`，这样我们就可以在内核启动后直接进入进程切换，切换到用户进程运行 `uapp`。但是这步并不会设置下一次时钟中断，所以我们不能在 `head.S` 中取消第一次时钟中断的设置。
2. 修改 `head.S`，关闭S-mode下的中断（即不设置 `sstatus[SIE]=1`），防止打断`start_kernel`中的`schedule()`

## 4.6 测试纯二进制文件

测试结果如下：



root@f40b9b9cbdfb: /have-fu



```
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x00000000000000222
Boot HART MEDELEG         : 0x0000000000000b109
setup_vm done!
...buddy_init done!
setup_vm_final begin!
pgdir memset done!
mapping begin!
mapping done!
satp set done!
...proc_init done!
2022 Hello RISC-V
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 5]
SET [PID = 4 COUNTER = 2]
using SJF, switch to [PID = 4 COUNTER = 2]
[U-MODE] pid: 4, sp is 0000003fffffffffe0
using SJF, switch to [PID = 3 COUNTER = 5]
[U-MODE] pid: 3, sp is 0000003fffffffffe0
using SJF, switch to [PID = 1 COUNTER = 10]
[U-MODE] pid: 1, sp is 0000003fffffffffe0
[U-MODE] pid: 1, sp is 0000003fffffffffe0
using SJF, switch to [PID = 2 COUNTER = 10]
[U-MODE] pid: 2, sp is 0000003fffffffffe0
[U-MODE] pid: 2, sp is 0000003fffffffffe0
SET [PID = 1 COUNTER = 9]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 4]
SET [PID = 4 COUNTER = 10]
using SJF, switch to [PID = 2 COUNTER = 4]
[U-MODE] pid: 2, sp is 0000003fffffffffe0
using SJF, switch to [PID = 3 COUNTER = 4]
[U-MODE] pid: 3, sp is 0000003fffffffffe0
using SJF, switch to [PID = 1 COUNTER = 9]
[U-MODE] pid: 1, sp is 0000003fffffffffe0
using SJF, switch to [PID = 4 COUNTER = 10]
|
```

## 思考题

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）

我们的每个用户态线程都有一个对应的内核态线程来进行调度，因此是一一对一的关系。

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？

我们的四个用户态都分别有两个栈

- S-Mode-stack：创建于 `task_init` 中，实际空间在存放 `task_struct` 数据的页的末尾
- U-Mode-stack：创建于 `task_init` 中，实际空间位于我们使用 `alloc_page` 开辟的空间

因此我们可以看出来，这4\*2个栈的物理位置都是不同的。然后我们在 `task_init` 中都将 U-Mode-stack 映射到了 `USER_END-PGSIZE`。而且每一个用户态进程都享有自己的页表（即之前提到的 `pgd`），因此虽说是同一个虚拟地址，但是通过各自的页表所找到的对应的物理地址还是不同的。这点也是由虚拟地址提供的强大功能之一。

因为用户栈的物理地址信息是储存在内核态的，所以正常情况下是没法拿到的。（但是或许我们可以写一个专门拿用户态栈物理地址的 `syscall`，就跟第172号系统调用拿 `pid` 一样，如果系统调用算常规方法的话）