

Lab 4: RV64 用户模式

实验步骤

1 准备工程

- 修改 `vmlinux.lds.S`

```
1      .data : ALIGN(0x1000){
2          _sdata = .;
3
4          *(.sdata .sdata*)
5          *(.data .data.*)
6
7          _edata = .;
8
9          . = ALIGN(0x1000);
10         uapp_start = .;
11         *(.uapp .uapp*)
12         uapp_end = .;
13         . = ALIGN(0x1000);
14
15     } >ramv AT>ram
```

- 修改 `defs.h`

```
1 #define USER_START (0x0000000000000000) // user space start virtual address
2 #define USER_END   (0x0000004000000000) // user space end virtual address
```

- 修改根目录下的 Makefile, 将 `user` 纳入工程管理

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

2 创建用户态进程

- 修改 `task_init`

```
1 /* 线程状态段数据结构 */
2 struct thread_struct {
3     uint64_t ra;
4     uint64_t sp;
5     uint64_t s[12];
6     uint64_t sepc, sstatus, sscratch;
7 };
8
```

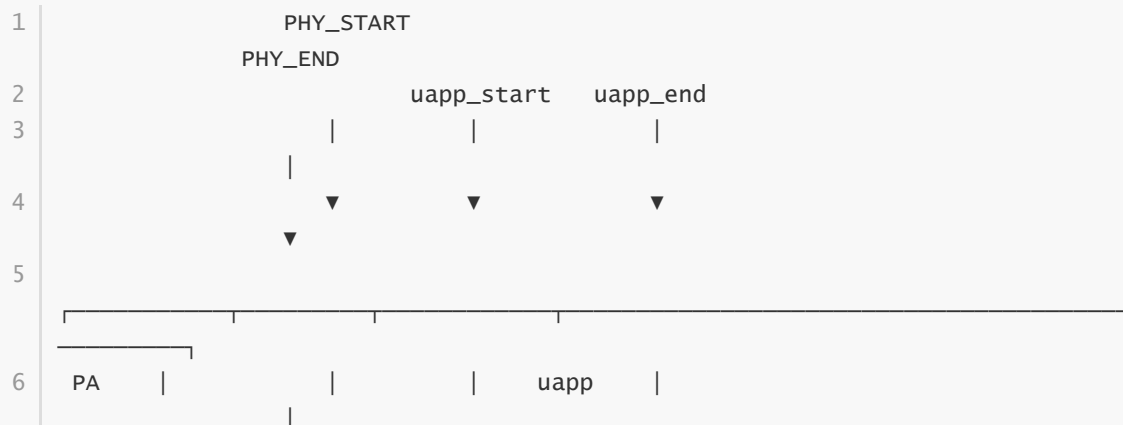
```

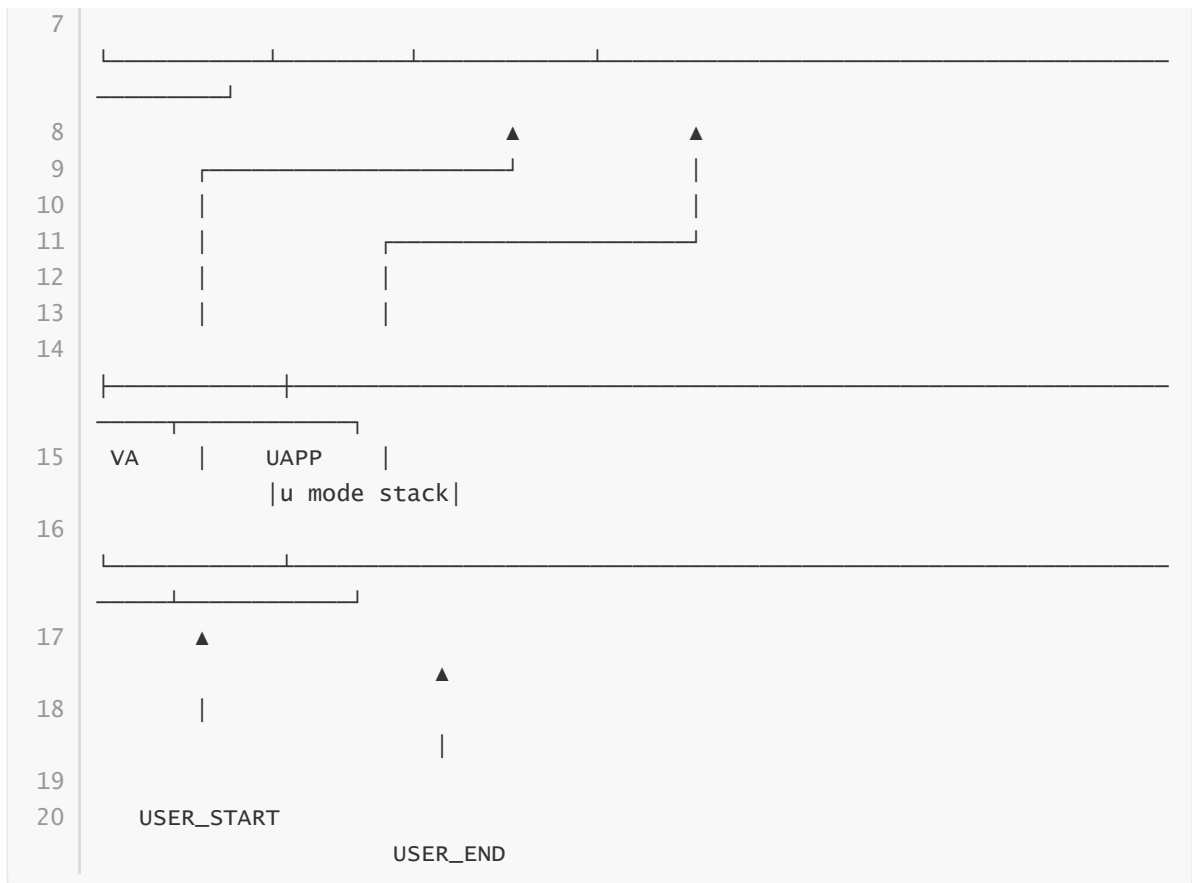
9  /* 线程数据结构 */
10 struct task_struct {
11     struct thread_info* thread_info;
12     uint64_t state;
13     uint64_t counter;
14     uint64_t priority;
15     uint64_t pid;
16     struct thread_struct thread;
17     pagetable_t pgd;
18 };
19 for(int i = 1; i < NR_TASKS; i++){
20     task[i] = (struct task_struct *)kalloc();
21     task[i] -> state = TASK_RUNNING;
22     task[i] -> counter = 0;
23     task[i] -> priority = rand();
24     task[i] -> pid = i;
25     task[i] -> thread.ra = (uint64_t)0;
26     task[i] -> thread.sp = (uint64_t)task[i] + PGSIZE;
27     task[i] -> thread.sepc = USER_START;
28     // SPP=0 SPIE=1 SUM=1
29     task[i] -> thread.sstatus = (csr_read(sstatus)) | (1<<5) | (1<<18) &
0xFFFFFFFFFFFFFFFF;
30     task[i] -> thread.sscratch = USER_END;
31     // 页表
32     pagetable_t pgtbl = (pagetable_t)kalloc();
33     memcpy(pgtbl, swapper_pg_dir, sizeof(swapper_pg_dir)/sizeof(char));
34     // 映射uapp的内容
35     create_mapping(pgtbl, 0, U64(uapp_start) - PA2VA_OFFSET, U64(uapp_end) -
U64(uapp_start), 7, 1);
36     // 映射用户栈
37     create_mapping(pgtbl, USER_END - PGSIZE, kalloc() - PA2VA_OFFSET,
PGSIZE, 3, 1);
38     // 为了简单, 这里我们直接把page number存入pgd
39     task[i] -> pgd = (pagetable_t) ((U64(pgtbl) - PA2VA_OFFSET) >> 12);
40 }

```

加入用户线程后 `task_struct` 有以下几处新增的域:

- `sepc`: `sret` 的返回位置, 初始设置为 `USER_START`
- `sstatus`: 设置 `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`: 用于在内核态和用户态切换的过程中保存 `sp`, 初始设置为 U-Mode 的 `sp`
- `pgd`: 每一个进程需要有自己独立的页表, 因为 `uapp` 段映射到物理内存中的位置不同





- 修改 `__switch_to`, 需要加入 保存/恢复 `sepc` `sstatus` `sscratch` 以及 切换页表的逻辑

```

1  .globl __switch_to
2  __switch_to:
3      # a0: prev
4      # a1: next
5      # 保存当前的状态到thread_struct
6
7      sd ra, 40(a0)
8      sd sp, 48(a0)
9      sd s0, 56(a0)
10     sd s1, 64(a0)
11     sd s2, 72(a0)
12     sd s3, 80(a0)
13     sd s4, 88(a0)
14     sd s5, 96(a0)
15     sd s6, 104(a0)
16     sd s7, 112(a0)
17     sd s8, 120(a0)
18     sd s9, 128(a0)
19     sd s10, 136(a0)
20     sd s11, 144(a0)
21     csrr t0, sepc
22     sd t0, 152(a0)
23     csrr t0, sstatus
24     sd t0, 160(a0)
25     csrr t0, sscratch
26     sd t0, 168(a0)
27
28     # 根据thread_struct恢复之前的状态
29     ld ra, 40(a1)
30     ld sp, 48(a1)

```

```

31     ld s0, 56(a1)
32     ld s1, 64(a1)
33     ld s2, 72(a1)
34     ld s3, 80(a1)
35     ld s4, 88(a1)
36     ld s5, 96(a1)
37     ld s6, 104(a1)
38     ld s7, 112(a1)
39     ld s8, 120(a1)
40     ld s9, 128(a1)
41     ld s10, 136(a1)
42     ld s11, 144(a1)
43     ld t0, 152(a1)
44     csrwrw sepc, t0
45     ld t0, 160(a1)
46     csrwrw sstatus, t0
47     ld t0, 168(a1)
48     csrwrw sscratch, t0
49
50     # 切换页表
51     ld t0, 176(a1)
52     li t1, 0x8000000000000000
53     or t0, t0, t1
54     csrwrw satp, t0
55     sfence.vma zero, zero
56     ret

```

3 修改中断入口/返回逻辑 (_trap) 以及中断处理函数 (trap_handler)

- 由于我们的用户态进程运行在 U-Mode 下，使用的运行栈也是 U-Mode Stack，因此当触发异常时，我们首先要对栈进行切换（U-Mode Stack -> S-Mode Stack）。同理 让我们完成了异常处理，从 S-Mode 返回至 U-Mode，也需要进行栈切换（S-Mode Stack -> U-Mode Stack）

```

1  _traps:
2      # 保存U-mode的sp
3      csrwrw sp, sscratch, sp
4      .....
5      # 返回U-mode
6      csrwrw sp, sscratch, sp
7      sret

```

- 修改 __dummy。在初始化时，thread_struct.sp 保存了 S-Mode sp，thread_struct.sscratch 保存了 U-Mode sp，因此在 S-Mode -> U->Mode 的时候，我们只需要交换对应的寄存器的值即可。

```

1  __dummy:
2      // sret回去之后进入了用户态，需要使用用户态的栈
3      csrwrw sp, sscratch, sp
4      sret

```

- uapp 使用 ecall 会产生 ECALL_FROM_U_MODE exception。因此我们需要在 trap_handler 里面进行捕获。修改 trap_handler 如下：

```

1  #include "printk.h"
2  #include "clock.h"
3  #include "proc.h"
4  #include "string.h"
5  struct pt_regs
6  {
7      unsigned long zero;
8      .....
9      unsigned long t6;
10     unsigned long sepc;
11 };
12 extern struct task_struct* current;
13
14 void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *
regs) {
15     // 通过 `scause` 判断trap类型
16     // 如果是interrupt 判断是否是timer interrupt
17     // 如果是timer interrupt 则打印输出相关信息, 并通过 `clock_set_next_event()`
设置下一次时钟中断
18     // `clock_set_next_event()` 见 4.5 节
19     // 其他interrupt / exception 可以直接忽略
20     unsigned long int_bit = 0x8000000000000000;
21     if(scause & int_bit){
22         switch (scause & ~int_bit){
23             case 0x5:
24                 do_timer();
25                 break;
26             default:
27                 printk("Int scause:%d\n", scause);
28                 break;
29         }
30     }else{
31         switch (scause){
32             case 0x8:
33                 ECALL_FROM_U_MODE
34                 .....
35                 break;
36             default:
37                 printk("Exception scause:%d\n", scause);
38                 regs->sepc+=4;
39                 break;
40         }
41     }
42 }

```

4 添加系统调用

- 系统调用规范
 - 64 号系统调用 `sys_write(unsigned int fd, const char* buf, size_t count)` 该调用将用户态传递的字符串打印到屏幕上, 此处fd为标准输出 (1), buf为用户需要打印的起始地址, count为字符串长度, 返回打印的字符数。(具体见 user/printf.c)
 - 172 号系统调用 `sys_getpid()` 该调用从current中获取当前的pid放入a0中返回, 无参数。(具体见 user/getpid.c)

```

1 case 0x8:
2     switch (regs->a7){
3         case 64:
4             if(regs -> a0 == 1){
5                 char buf[1000] = {0};
6                 memcpy(buf, (void *)regs -> a1, regs -> a2);
7                 regs -> a0 = printk(buf);
8             }
9             break;
10        case 172:
11            regs -> a0 = current -> pid;
12            break;
13    }
14    regs->sepc+=4;
15    break;

```

- 增加 `syscall.c` `syscall.h` 文件，并在其中实现 `getpid` 以及 `write` 逻辑

```

1 #define SYS_WRITE    64
2 #define SYS_GETPID  172

```

5 修改 head.S 以及 start_kernel

- 之前 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 运行。
- 在 `start_kernel` 中调用 `schedule()` 注意放置在 `test()` 之前。
- 将 `head.S` 中 `enable interrupt sstatus.SIE` 逻辑注释，确保 `schedule` 过程不受中断影响。

```

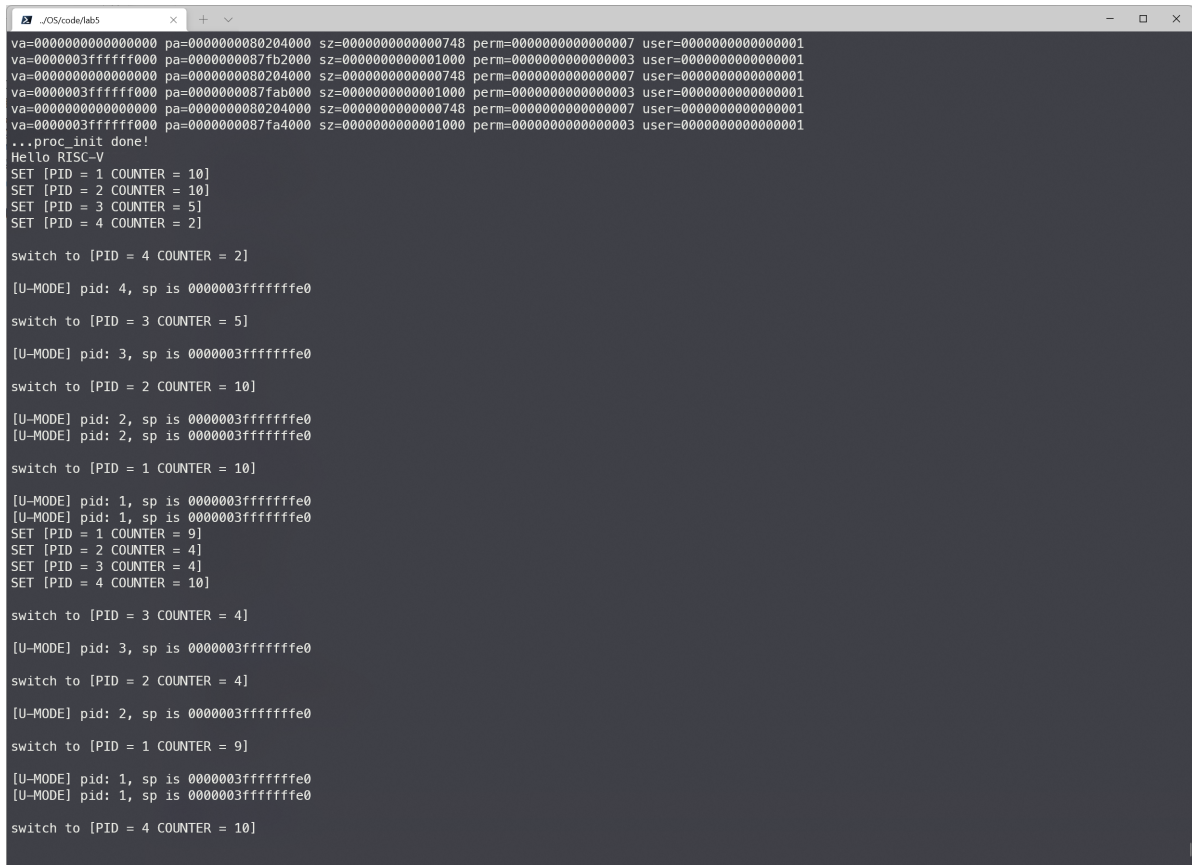
1 _start:
2     la sp, boot_stack_top
3     call setup_vm
4     call relocate
5     call mm_init
6     call setup_vm_final
7
8     # set stvec = _traps
9     la t0, _traps
10    csrw stvec, t0
11
12    # set sie[STIE] = 1
13    csrr t0, sie
14    ori t0, t0, 0x20
15    csrw sie, t0
16
17    # set first time interrupt
18    rdttime t0
19    li a0, 10000000
20    add a0, a0, t0
21    li a1, 0
22    li a2, 0
23    li a3, 0
24    li a4, 0
25    li a5, 0
26    li a6, 0
27    li a7, 0
28    call sbi_ecall

```

```
29 | call task_init
30 | call start_kernel
```

```
1 | #include "printk.h"
2 | #include "defs.h"
3 | #include "sbi.h"
4 | #include "mm.h"
5 | #include "proc.h"
6 |
7 | extern void test();
8 |
9 | int start_kernel(){
10 |     printk(" Hello RISC-V\n");
11 |     schedule();
12 |     test(); // DO NOT DELETE !!!
13 |     return 0;
14 | }
```

编译及测试



```
va=0000000000000000 pa=0000000080204000 sz=0000000000000748 perm=0000000000000007 user=0000000000000001
va=0000003fffffff00 pa=0000000087fb2000 sz=0000000000001000 perm=0000000000000003 user=0000000000000001
va=0000000000000000 pa=0000000080204000 sz=0000000000000748 perm=0000000000000007 user=0000000000000001
va=0000003fffffff00 pa=0000000087fab000 sz=0000000000001000 perm=0000000000000003 user=0000000000000001
va=0000000000000000 pa=0000000080204000 sz=0000000000000748 perm=0000000000000007 user=0000000000000001
va=0000003fffffff00 pa=0000000087fa4000 sz=0000000000001000 perm=0000000000000003 user=0000000000000001
...proc_init done!
Hello RISC-V
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 5]
SET [PID = 4 COUNTER = 2]

switch to [PID = 4 COUNTER = 2]

[U-MODE] pid: 4, sp is 0000003fffffff00

switch to [PID = 3 COUNTER = 5]

[U-MODE] pid: 3, sp is 0000003fffffff00

switch to [PID = 2 COUNTER = 10]

[U-MODE] pid: 2, sp is 0000003fffffff00
[U-MODE] pid: 2, sp is 0000003fffffff00

switch to [PID = 1 COUNTER = 10]

[U-MODE] pid: 1, sp is 0000003fffffff00
[U-MODE] pid: 1, sp is 0000003fffffff00
SET [PID = 1 COUNTER = 9]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 4]
SET [PID = 4 COUNTER = 10]

switch to [PID = 3 COUNTER = 4]

[U-MODE] pid: 3, sp is 0000003fffffff00

switch to [PID = 2 COUNTER = 4]

[U-MODE] pid: 2, sp is 0000003fffffff00

switch to [PID = 1 COUNTER = 9]

[U-MODE] pid: 1, sp is 0000003fffffff00
[U-MODE] pid: 1, sp is 0000003fffffff00

switch to [PID = 4 COUNTER = 10]
```