lab5 RV64 用户态程序

4.2 创建用户态进程

首先是按要求将对应的task_struct改掉，我将thread_info删去了。

```
typedef unsigned long* pagetable_t;

/* 用于记录 `线程` 的 `内核栈与用户栈指针` */
// struct thread_info {
//     uint64_t kernel_sp;
//     uint64_t user_sp;
// };

/* 线程状态段数据结构 */

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

    uint64_t sepc, sstatus, sscratch;       You, 7 second
};

/* 线程数据结构 */

struct task_struct {
    // struct thread_info* thread_info;
    uint64_t state;    // 线程状态
    uint64_t counter;  // 运行剩余时间
    uint64_t priority; // 运行优先级 1最低 10最高
    uint64_t pid;      // 线程id

    struct thread_struct thread;

    pagetable_t pgd;
};
```

4.2.1 修改task_init

之后是改变task_init，其中我们新建一个page来放pgd，之后把根页表各项复制过来，之后是map user memory，我们将每一页的va放到新一页里，再把新页map到USER_START处。之后再新建一个Ustack map到USER_END处，最后是相应CSRreg的改变。

```
    task[i]->thread.ra = (uint64_t)__dummy;
    task[i]->thread.sp = pgNum + PGSIZE;

    task[i]->pgd = (pagetable_t)alloc_page();
    // task[i]->thread_info->kernel_sp = pgNum + PGSIZE;
    // task[i]->thread_info->user_sp = Ustack + USER_END;

    // map kernel pgdir
    // here just need to map root pg
    for (int j = 0; j < 512; j++){
        task[i]->pgd[j] = swapper_pg_dir[j];
    }

    // printk("uapp_start = %lx, uapp_start = %lx\n", uapp_start, uapp_end);
    // printk("Ustack = %lx\n", Ustack);

    uint64_t map_user;
    // mapping user memory U|-|W|R|V
    for(va = uapp_start; va < uapp_end; va += PGSIZE){
        map_user = alloc_page();
        char *dst = (char *)map_user;
        char *src = (char *)va;
        // copy memory
        for(uint64_t j = 0; j < MIN(0x1000, uapp_end - va); j++){
            dst[j] = src[j];
        }
        // here pretend each user space is started with 0x0
        create_mapping(task[i]->pgd, USER_START+va-(uint64_t)uapp_start, map_user - PA2VA_OFFSET, PGSIZE, 0b11111);
    }

    Ustack = alloc_page();
    create_mapping(task[i]->pgd, USER_END - PGSIZE, Ustack - PA2VA_OFFSET, PGSIZE, 0b10111);

    uint64_t satp = csr_read(satp);
    // Mode + Asid + ppn
    satp = ((satp >> 44) << 44) | ((unsigned long)task[i]->pgd - PA2VA_OFFSET) >> 12;
    task[i]->pgd = (pagetable_t)satp;

    task[i]->thread.sepc = USER_START;

    uint64_t sstatus = csr_read(sstatus);
    // SSP SPIE SUM
    sstatus = sstatus & ~(1<<8);
    sstatus = sstatus | 1<<5;
    sstatus = sstatus | 1<<18;
    task[i]->thread.sstatus = sstatus;
    task[i]->thread.sscratch = USER_END;
}
printk("...proc_init done!\n");
```

由于我们要支持虚拟地址寻址，所以找页表地址时得返回虚拟地址。

```
98        // here return the virtual address
99        return (uint64 *)(next_pgtbl + PA2VA_OFFSET);
00    }
01
```

4.2.2 修改__switch_to

对于__switch_to，由于我们的task_struct多了sepc, sscratch, sstatus，所以也要将对应的CSRreg取出或者存入。最后再刷新页表即可。

```
    csrr t1, sepc
    sd t1, 112(a0)
    csrr t1, sstatus
    sd t1, 120(a0)
    csrr t1, sscratch
    sd t1, 128(a0)
    csrr t1, satp
    sd t1, 136(a0)

    # restore state from next process
    # YOUR CODE HERE
    add a1, a1, t0

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ld t2,112(a1)
    csrw sepc,t2
    ld t2,120(a1)
    csrw sstatus,t2
    ld t2,128(a1)
    csrw sscratch,t2
    ld t2,136(a1)
    csrw satp,t2

    sfence.vma zero, zero
    fence.i
```

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

首先，修改_dummy，只要将sscratch读出来再和sp对换即可。

```
    .global __dummy
__dummy:
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0
    sret
```

之后是修改_trap。我们利用sscratch是否为0看是否要对换。

```
    .globl _traps
_traps:
    # YOUR CODE HERE
    csrr t0, sscratch
    beq t0, x0, _switch_reg
    csrw sscratch, sp
    mv sp, t0
_switch_reg:
    # 1. save 32 registers and sepc to stack
    add sp, sp, -33*8
    sd x0, 0(sp)
```

```
    add sp, sp, 33*8

    csrr t0, sscratch
    beq t0, x0, _switch_reg_end
    csrw sscratch, sp
    mv sp, t0
_switch_reg_end:
    # 4. return from trap
    sret
```

之后是补充 struct pt_regs的定义， 以及在 trap_handler 中补充处理 SYSCALL 的逻辑。我们直接将pt_reg放到syscall.h之中了。

```
#include "stdint.h"

#define SYS_WRITE   64
#define SYS_GETPID  172

struct pt_regs{
    uint64_t x[32];
    uint64_t sepc;
    // uint64_t sstatus;
};

void syscall(struct pt_regs *reg);
```

```
    // interrupt
    if (scause & (1 << 31)){
        // if (scause % 16 == 4){
        //     printk("[U] User Mode Timer Interrupt!\n")
        // }
        if (scause % 16 == 5){
            // printk("[S] Supervisor Mode Timer Interrupt!\n");
            clock_set_next_event();
            do_timer();
        }
    } else{
        if (scause % 16 == 8) {
            // printk("ECALL_FROM_U_MODE Interrupt!\n");
            syscall(regs);
        }
    }
}
```

4.4 添加系统调用

我们创建syscall.c用于系统调用。其中x[17]就是a6，如果是GETPID就返回pid，SYS_WRITE就调用printk解决。注意返回的值改到regs里。

```
5
6   extern struct task_struct* current;
7   void syscall(struct pt_regs *regs) {
8       //get pid
9       if(regs->x[17] == SYS_GETPID) {
10          // here a0 for x[10]
11          regs->x[10] = current->pid;
12      }
13      //sys_write
14      else if (regs->x[17] == SYS_WRITE) {
15          // fd
16          if (regs->x[10] == 1) {
17              // count
18              uint64_t end = regs->x[12];
19              char *out = regs->x[11];
20              out[end] = '\0';
21              regs->x[10] = printk(out);
22              // ((char*)(regs->x[11]))[end] = '\0';
23              // regs->x[10] = printk((char *)(regs->x[11]));
24          }
25      }
26  }
```

之后是对于syscall，要手动将sepc + 4，这是_traps的内容，只要读出scause即可。

```
    addi t1, x0, 8
    csrr t2, scause
    bne t1, t2, no_add4
    addi t0, t0, 4          // for syscall sepc += 4
no_add4:
    csrw sepc, t0
    ld x0, 0(sp)
    ld x1, 8(sp)
```

4.5 修改 head.S 以及 start_kernel

这里就比较简单了，只要把对应的部分注释掉即可。不赘述。

4.6 测试纯二进制文件

可以看到，我们的counter是按顺序增加而且每个pid都不同。

```
...proc_init done!
 Hello RISC-V
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.4
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.4
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.5
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.6
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
```

4.7 添加 ELF 支持

对于ELF的支持，我们只是将之前的user memory load的部分改成了load_program。

```
    // here just need to map root pg
    for (int j = 0; j < 512; j++){
        task[i]->pgd[j] = swapper_pg_dir[j];
    }

    // uint64_t map_user;
    // // mapping user memory U|-|W|R|V
    // for(va = uapp_start; va < uapp_end; va += PGSIZE){
    //     map_user = alloc_page();
    //     char *dst = (char *)map_user;
    //     char *src = (char *)va;
    //     // copy memory
    //     for(uint64_t j = 0; j < MIN(0x1000, uapp_end - va); j++){
    //         dst[j] = src[j];
    //     }
    //     // here pretend each user space is started with 0x0
    //     create_mapping(task[i]->pgd, USER_START+va-uapp_start, map_user - PA2VA_OFFSET, PGSIZE, 0b11111);
    // }
    load_program(task[i]);

    Ustack = alloc_page();
    create_mapping(task[i]->pgd, USER_END - PGSIZE, Ustack - PA2VA_OFFSET, PGSIZE, 0b10111);

    uint64_t satp = csr_read(satp);
    // Mode + Asid + ppn
    satp = ((satp >> 44) << 44) | ((unsigned long)task[i]->pgd - PA2VA_OFFSET) >> 12;
    task[i]->pgd = (pagetable_t)satp;
```

之后是load_program的部分，对于phdr_cnt的部分，我们alloc_page拿出对应p_memsz需要的page，注意映射不是从下标0开始，而是对应的vaddr开始，这点很重要。而且对于create_mapping来说，va需要为0x000结尾，这也挺重要的。

```
static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)uapp_start;

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

    Elf64_Phdr* phdr;
    for (int i = 0; i < phdr_cnt; i++) {
        phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
        if (phdr->p_type == PT_LOAD) {
            // alloc space and copy content
            uint64_t u_start = (uint64_t)ehdr + phdr->p_offset;

            uint64_t va = phdr->p_vaddr;
            uint64_t map_user = alloc_page(phdr->p_memsz/PGSIZE + 1);
            char *dst = (char *)map_user;
            char *src = (char *)u_start;
            // copy memory
            for(uint64_t j = va&0xfff; j < (va&0xfff) + phdr->p_filesz; j++){
                dst[j] = src[j-va&0xfff];
            }
            for( ; va < phdr->p_vaddr + phdr->p_memsz; va += PGSIZE){
                // pretend each user space is started with p_vaddr
                // the va has to be 000 tailed
                create_mapping(task->pgd, va >> 12 << 12, map_user - PA2VA_OFFSET, PGSIZE, 0b11111);
            }
        }
    }
```

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）

是多对一的。

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

p_filesz 可加载段会多包含一个 .bss 段，这些是未初始化的数据，所以没有必要放在磁盘上。因此它只会在 ELF 文件加载到内存中时才占用空间；所以p_memsz会大于等于p_filesz。

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？

因为每个进程的页表都不同，映射到的物理地址也不同的即使虚拟地址相同，也不会相互干扰。照理来说是没有常规方法的，用户既不知道映射规则，也不知道页表信息。

实验心得

这次实验对于细节有很多的要求，而且debug也没有之前实验的直观，所以之后很多都是只能靠猜bug的位置。