Lab 5: RV64 用户态程序

1 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。

2 实验环境

- Same as previous labs.

3 实验步骤

3.1 准备工程

- 此次实验基于 lab4 同学所实现的代码进行。
- 需要修改 `vmlinux.lds.S`，将用户态程序 `uapp` 加载至 `.data` 段。按如下修改：

```
...

.data : ALIGN(0x1000){
        _sdata = .;

        *(.sdata .sdata*)
        *(.data .data.*)

        _edata = .;

        . = ALIGN(0x1000);
        uapp_start = .;
        *(.uapp .uapp*)
        uapp_end = .;
        . = ALIGN(0x1000);

    } >ramv AT>ram

...
```

需要修改 `defs.h`，在 `defs.h` **添加** 如下内容：

```
#define USER_START (0x0000000000000000) // user space start virtual address
#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。
- 在根目录下 `make` 会生成 `user/uapp.o` `user/uapp.elf` `user/uapp.bin`，以及我们最终测试使用的 ELF 可执行文件 `user/uapp` 。 通过 `objdump` 我们可以看到 uapp 使用 ecall 来调用 SYSCALL (在 U-Mode 下使用 ecall 会触发environment-call-from-U-mode异常)。从而将控制权交给处在 S-Mode 的 OS， 由内核来处理相关异常。

- 在本次实验中，我们首先会将用户态程序 strip 成纯二进制文件来运行。这种情况下，用户程序运行的第一条指令位于二进制文件的开始位置，也就是说 `uapp_start` 处的指令就是我们要执行的第一条指令。我们将运行纯二进制文件作为第一步，在确认用户态的纯二进制文件能够运行后，我们再将存储到内存中的用户程序文件换为 ELF 来进行执行。

3.2 创建用户态进程

- 本次实验只需要创建 4 个用户态进程，修改 `proc.h` 中的 `NR_TASKS` 即可。
- 由于创建用户态进程要对 `sepc` `sstatus` `sscratch` 做设置，我们将其加入 `thread_struct` 中。
- 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。我们为每个用户态进程都创建一个页表。修改 `task_struct` 如下。

```
// 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;

    struct thread_struct thread;

    pagetable_t pgd;
};
```

修改 task_init

对每个用户态进程，S-Mode Stack 在 Lab4 中已经设置好了。可以通过 `alloc_page` 接口申请一个空的页面来作为 U-Mode Stack

```
// allocate task pagetable
        task[i]->pgd = (pagetable_t)alloc_page();
// copy kernel pagetable to task pagetable
        memcpy((void *)(task[i]->pgd), (void *)((&swapper_pg_dir)), PGSIZE);
// change to physical address
        task[i]->pgd = (pagetable_t)VA2PA((uint64_t)task[i]->pgd);
```

修改 __switch_to

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

回复保存的逻辑与其他寄存器一致，切换页表先载入新页表的地址，设置好PPN、MODE后写入satp

```
__switch_to:
    # save `sepc` `sstatus` `sscratch`

    csrr t1, sepc
    sd t1, 112(t0)
    csrr t1, sstatus
    sd t1, 120(t0)
    csrr t1, sscratch
    sd t1, 128(t0)

    # restore `sepc` `sstatus` `sscratch`

    csrw sepc, t1
    ld t1, 120(t0)
    csrw sstatus, t1
    ld t1, 128(t0)
    csrw sscratch, t1

    # switch page table
    ld t1, 136(t0) # load new process' page table
    srli t1, t1, 12 # PA >> 12 = PPN
    li t2, 8 # MODE = 8
    slli t2, t2, 60 # MODE: satp[63:60]
    or t1, t1, t2
    csrw satp, t1
```

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

修改 `__dummy`

在 **4.2** 中 我们初始化时， `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。

交换sscratch和sp

```
csrr t1, sscratch
csrw sscratch, sp
mv sp, t1
sret
```

修改 `_trap`

如果是 内核线程(没有 U-Mode Stack) 触发了异常，则不需要进行切换。（内核线程的 sp 永远指向的 S-Mode Stack， sscratch 为 0）

trap开始时，如果sstatus的spp位不是0，那么处于S-Mode，不需要切换stack。否则切换stack

```
    # check mode
    csrr t0, sstatus
    andi t0, t0, 0x100 # SPP bit
    bne t0, zero, _trap_start  # SPP != 0: S-mode, don't need to switch stack
    csrr t1, sscratch  # switch stack pointers for U mode and S mode
    csrw sscratch, sp
    mv sp, t1
```

trap结束时，如果sstatus的spp位不是0，那么处于S-Mode，不需要切换stack。否则切换stack

```
    # resume original stack pointer (if trapped from U mode)
    csrr t0, sstatus
    andi t0, t0, 0x100 # SPP bit
    bne t0, zero, _trap_end  # SPP != 0: S-mode, don't need to switch stack
    csrr t1, sscratch  # switch stack pointers for U mode and S mode
    csrw sscratch, sp
    mv sp, t1
```

3.4 添加系统调用

- 本次实验要求的系统调用函数原型以及具体功能如下：
  - 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)

修改`trap_handler`

增加处理SYSCALL的逻辑

异常处理完成之后，手动 `regs->sepc += 4;` 继续执行后续的指令

```
switch (regs->reg[16])
        {
        case SYS_WRITE:
            sys_write(regs);
            break;
        case SYS_GETPID:
            sys_getpid(regs);
            break;
        }
        // 异常处理完成之后，继续执行后续的指令
        regs->sepc += 4;
```

write逻辑

返回值放置在a0，对应的是reg[9]

```
void sys_write(struct pt_regs *regs)
{
    unsigned int fd = regs->reg[9];
    const char *buf = regs->reg[10];
    int count = regs->reg[11];
    uint64_t ret = 0;
```

```
    for (int i = 0; i < count; i++)
    {
        if (fd == 1)
        {
            printk("%c", buf[i]);
            ret++;
        }
    }
    regs->reg[9] = ret;
}
```

getpid逻辑

返回值放置在a0，对应的是reg[9]

```
void sys_getpid(struct pt_regs *regs)
{
    regs->reg[9] = current->pid;
}
```

3.5 修改 head.S 以及 start_kernel

- 在之前的 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。

3.6 添加 ELF 支持

将 `uapp.S` 中的 payload 换成我们的 ELF 文件

```
.section .uapp

.incbin "uapp"
```

load elf文件

从地址 uapp_start 开始，便是我们要找的 `ehdr`

Phdr 数组的起始地址是 `ehdr + ehdr->e_phoff`

elf文件包含的segment数量为 `ehdr->e_phnum`

通过 `ehdr->e_phnum` 来控制循环结束

对于每个phdr，如果segment的类型是PT_LOAD，那么就将其拷贝到内存中

先开辟内存，将segment内容拷贝过来，然后建立虚拟映射

最后设置用户栈，pc，sstatus，sscratch

```
static uint64_t load_elf_program(struct task_struct *task)
{
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)(&uapp_start);// 从地址 uapp_start 开始，便是我们要找的 Ehdr
```

```
    uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;// Phdr 数组，其中的每个元
素都是一个 Elf64_Phdr
    int phdr_cnt = ehdr->e_phnum;// ELF 文件包含的 Segment 的数量

    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)
        {
            // alloc space and copy content
            uint64_t pg_num = (PGOFFSET(phdr->p_vaddr) + phdr->p_memsz - 1) /
PGSIZE + 1;// page amount of segment
            uint64_t seg_page = alloc_pages(pg_num);
            uint64_t seg_addr = ((uint64_t)(&uapp_start) + phdr->p_offset);
            memcpy((void *)(seg_page + PGOFFSET(phdr->p_vaddr)), (void *)
(seg_addr), phdr->p_memsz);
            // do mapping
            create_mapping((uint64 *)PA2VA((uint64_t)task->pgd),
PGROUNDDOWN(phdr->p_vaddr), VA2PA(seg_page), pg_num * PGSIZE, phdr->p_flags << 1
| 16);
        }
    }

    // allocate user stack and do mapping
    uint64_t u_stack_begin = alloc_page();
    create_mapping((uint64 *)PA2VA((uint64_t)task->pgd), USER_END - PGSIZE,
VA2PA(u_stack_begin), PGSIZE, 23);

    // set user stack
    task->thread_info.user_sp = USER_END;
    // pc for the user program
    task->thread.sepc = ehdr->e_entry; // 程序的第一条指令被存储的用户态虚拟地址
    // sstatus bits set
    // SPP = 0, SPIE = 1, SUM = 1
    task->thread.sstatus = (1 << 18) | (1 << 5);
    // user stack for user program
    task->thread.sscratch = USER_END;
}
```

```
SET [PID = 1 COUNTER = 1]
SET [PID = 2 COUNTER = 1]
SET [PID = 3 COUNTER = 6]
SET [PID = 4 COUNTER = 3]

switch to [PID = 1 COUNTER = 1]
[U-MODE] pid: 1, user main: 0000000000010120, sp is 0000003fffffffe0, this is print No.1

switch to [PID = 2 COUNTER = 1]
[U-MODE] pid: 2, user main: 0000000000010120, sp is 0000003fffffffe0, this is print No.1

switch to [PID = 4 COUNTER = 3]
[U-MODE] pid: 4, user main: 0000000000010120, sp is 0000003fffffffe0, this is print No.1

switch to [PID = 3 COUNTER = 6]
[U-MODE] pid: 3, user main: 0000000000010120, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, user main: 0000000000010120, sp is 0000003fffffffe0, this is print No.2

SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 5]
SET [PID = 3 COUNTER = 5]
```

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）

   答：一对一，一个内核态线程载入一个elf，一个elf对应一个用户态线程

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

   答：`p_memsz` 大于(或等于) `p_filesz` 的原因是，可加载的部分可能包含 `.bss` 节，该节包含未初始化的数据。将此数据存储在磁盘上会很浪费，因此，仅在ELF文件加载到内存后才占用空间。

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？

   答：因为虽然虚拟地址相同，但是不同进程的栈通过页表映射得到的物理地址不同。没有方法，只有内核能看到。