Lab5 用户态程序

赵伊蕾 学号：3200104866

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. 实验过程

2.1 用户态思想

在本实验中，我们将开启用户态模式，为许多应用程序提供专用的虚拟地址空间等资源。这些应用程序之间的空间是互相隔离独立的，不能互相影响。且用户模式下，应用程序不可访问内核的虚拟地址，用户态程序只能通过接口的系统调用来完成与操作系统之间的互动。

2.2 创建用户态进程

2.2.1 修改进程结构体

因为用户态进程的属性牵涉到寄存器具体某一位的值，需要对 `sepc` `sstatus` `sscratch` 做设置，所以我们需要将这几个变量加入我们之前用到的`thread_struct`中保存起来，以便于在进程之间切换的时候能够用来判断。同时，由于用户态之间的隔离性，彼此使用的页表也不相同，所以我们还需要在`task_struct`中保存指向页表的指针。

```
struct thread_struct {
    // ==== add ====
    uint64_t sepc, sstatus, sscratch;
};

struct task_struct {
    // ===== add ====
    pagetable_t pgd;
    uint64_t kernel_sp, user_sp;
};
```

创建用户态进程的第一步就是为进程**设置用户态Stack**。

这一步骤在`task_init`函数中完成。之前**lab3**已经在这函数中完成了S-Mode Stack的分配，所以现在还需要划分用户栈。具体思路是先分配一个空的页作为S-Mode Stack，将其放置在user space的最后，然后通过设置 `sepc` `sstatus` `sscratch` 这几个寄存器来完成切换页表的逻辑。为了后续程序方便起见，我还在`task_struct`函数中多添加了两个变量`kernel_sp`、`user_sp`来分别指示kernel stack和user stack的地址。

2.2.2 task_init函数新增设置用户态stack

1. **分配空页面作为S-Mode的Stack**

   这里用到了`alloc_page`函数接口来进行页面分配，并将返回的页面起始地址存入user_sp中。

2. **为每一进程创建自己的页表 完成映射**

   之前我们新设置的`pagetable_t pgd`变量需要用来存页表的地址，所以这里也分配一个空白页作为页表给这一用户态进程。

   对这一新的页表，我们需要做两步映射和一步拷贝。`uapp`所在的页面和我们刚才创建的U-Mode Stack需要映射到表中，还有内核页表(`swapper_pg_dir`)要复制到表里，避免两个模式之间转换的时候切换页表。

   这里特别需要注意到一个细节问题，因为有些数据可能不在栈上，但是初始化的时候已经被分配了空间(也就是一些全局变量，比如user/getpid.c 中的counter变量)。为了防止所有的进程共享数据，对隔离性造成影响，所以我们要先进行拷贝，再进行映射。

memcpy这一语句是将内核页表`swapper_pg_dir`复制到每一进程的页表中，然后在对用户进程的内存段进行映射，对用户程序的stack进行映射。

特别需要注意!!为了保证几个用户进程之间的相互隔离性，我们在对用户进程内存段进行映射的时候，必须对其先拷贝到物理内存中，然后再映射；否则一些全局变量(例如getpid.c中的counter)会被不同的用户进程所共享，这是不被允许的。

```
pagetable_t pgtbl = (pagetable_t)alloc_page();
memcpy(pgtbl, swapper_pg_dir, PGSIZE); // 同时为了避免 U-Mode 和 S-Mode 切换的时候切换页表

// 这里为了不同用户台程序之间可以互相隔离，所以会先对用户态内存进行复制，然后再进行映射。
uint64 va = USER_START;
uint64 pa = alloc_pages((uint64)(uapp_end - uapp_start)/PGSIZE) - PA2VA_OFFSET;
memcpy(pa + PA2VA_OFFSET, (uint64*)uapp_start, (uint64)(uapp_end - uapp_start));
create_mapping(pgtbl, va, pa, (uint64)(uapp_end - uapp_start), 0b11111 );

va = USER_END - PGSIZE;
pa = task[i]-> user_sp - PA2VA_OFFSET;
create_mapping(pgtbl, va, pa, PGSIZE, 0b10111);
```

3. **设置用户态进程的寄存器，实现Mode间切换**

   为了保证我们的程序进入用户态之后能再返回S-Mode，我们需要对一些状态寄存器进行设置。

- sepc寄存器

  对每个用户态进程我们需要将`sepc`修改为`USER_START`，这样在切换mode的时候，保证进程可以从头开始访问用户虚拟内存空间，进行用户态下的程序。

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

- sstatus寄存器

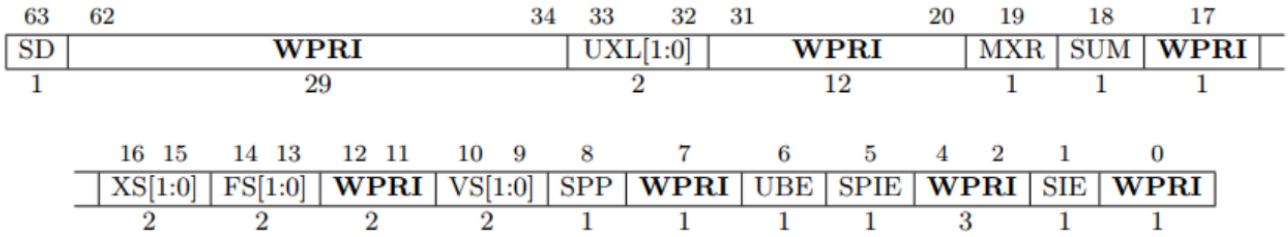


Figure 4.2: Supervisor-mode status register (sstatus) when SXLEN=64

字段	含义	功能
SSP	表示当前在哪种MODE下	0表示之前的mode为U-mode，1表示之前的mode是S-mode
SPIE	表示在trap中是否禁止S-mode的中断	当trap在s-mode时，SPIE设置到SIE中，并且把SIE设置为0。当执行SRET指令后，SIE设置到SPIE中，然后将SPIE设置为0.
SUM	S-mode下访问U-mode内存的权限。	0：对U-mode可访问的页面，S-mode访问将出错。1：S-mode访问U-mode的虚拟内存被允许

- `SSP`置0，使得程序可以通过sret返回之后进入U-Mode

```
sstatus &= ~(1<<8);
```

- `SPIE` 置1，使得sret 之后开启中断

```
sstatus |= 1<<5;
```

- `SUM` 置1，使得S-Mode 可以访问 User 页面

```
sstatus |= 1<<18;
```

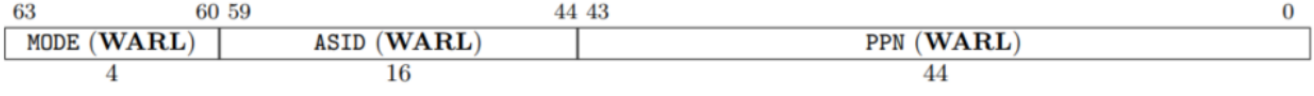
- sscratch寄存器

`sscratch` 设置为 `U-Mode` 的 sp，其值为 `USER_END` 。这是因为用户态下的**stack**是被放在了用户态虚拟内存空间的最后一个空白页，也就是地址`0x4000000000 - 4KB = 0x3ffffffe0`。

```
task[i]->thread.sscratch = USER_END;
```

4. **完成切换页表的逻辑**

页表的根地址是存储在寄存器satp中的，所以在切换到用户态的时候，我们还需要将用户态进程的页表也替换掉。所以我们需要去修改`satp`寄存器的PPN那部分内容，将用户态下页表的PPN存进去。



```
 55                30 29        21 20        12 11                          0
 ------------------------------------------------------------------------------
|       PPN[2]       |   PPN[1]   |   PPN[0]   |          page offset          |
 ------------------------------------------------------------------------------
                            Sv39 physical address
```

思路如下：1. 先取出目前的`satp`寄存器，保留[44,63]位的内容。2. 计算得用户态的`pagetable`的物理地址PPN[12:55]，然后存入satp的低44位中。

```
uint64 satp = csr_read(satp);
uint64 PPN = ((uint64)pgtbl - PA2VA_OFFSET) >> 12;
satp = (satp >> 44) << 44;
satp |= PPN;
task[i]->pgd = satp;
```

2.2.3 __switch_to函数加入修改寄存器和页表切换逻辑

原本`__switch_to`函数的逻辑就是将寄存器的值通通存到栈中，使得切换进程之后可以load进新的进程的这些寄存器的值，以及存储之前进程的寄存器。由于我们开启了用户态虚拟内存用到了一些特权寄存器，所以在`__switch_to`函数中我们也要新添加对于这几个寄存器的store 、 load逻辑。

因为在`thread_task`这个结构体中，我们将这几个变量定义在了原来那部分存储内容之后，所以我们在栈上也是接着前面的14个寄存器继续存下去。这里还需要特别注意的是由于我们还涉及到切换页表的操作，所以这里顺便把satp也存了。

```
/* 线程状态段数据结构 */
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];

    uint64 sepc, sstatus, sscratch;
};
```

```
# store sepc, sstatus, sscratch, satp
csrr t1, sepc
sd t1, 19*8(a0)
... do the same to sstatus, sscratch, sepc...
...
# load sepc, sstatus, sscratch, satp
ld t1, 19*8(a1)
csrw sepc, t1
... do the same to sstatus, sscratch, sepc...
```

2.2.4 修改中断入口/返回逻辑 (_trap) 以及中断处理函数

RISC-V由于只有一个栈寄存器，所以当涉及到栈切换的时候，sp寄存器的值也需要相应进行替换，从而实现用户栈和内核栈的切换。当异常产生，进程需要进行S-Mode进行处理，栈也需要从U-Mode切换到S-Mode，这就需要我们对sp指针的值进行替换；如果异常处理结束，还需要替换回来。

1. 修改`__dummy`

在`task_init`函数进行初始化的时候，我们先让sp指针记录的是S-Mode下的`sp`，`sscratch`中保存的事U-Mode下的`sp`。所以当异常产生，需要从S-Mode切换到U-Mode时，我们需要**交换这两个sp指针的值**。

```
__dummy:
# YOUR CODE HERE
csrr t0, sscratch
csrw sscratch, sp
add sp, t0, zero
sret
```

2. 修改`_trap`

在_trap中我们也要处理异常所产生的状态切换，这里特别需要考虑到kernel thread是没有用户态的，更加没有user stack，所以对于内核线程我们不需要做切换操作。因此，在_trap的首尾我们都需要先判断是否是内核线程，如果不是的话再进行stack的切换。

判断思路是查看sscratch的值，因为如果是内核线程，并没有user stack，则其也不存在sp。如果sscratch是0，也就是说当前是内核线程，则直接跳转到我们原来的__switch_to函数；否则进行stack切换。

```
csrr t0, sscratch
beq t0, zero, _normal_switch
# switch to U-Mode stack
csrr t0, sscratch
csrw sscratch, sp
add sp, t0, zero
```

2.2.5 捕获处理异常

用户态程序使用`ecall`会产生`ECALL_FROM_U_MODE` **exception**。以前的异常是由`trap_handler`先判断类型，然后进行处理；所以用户态ECALL产生的异常我们也交由`trap_handler`来处理。不过，我们在这里为函数新增一个参数`struct pt_regs *regs`，向其返回一个寄存器结构体，也就是我们之前在__switch_to函数中存在栈上的那一段寄存器的值。

对于中断的类型和原因，我们需要用到`scause`寄存器。我们之前在lab2中完成的时钟中断是trap，也就是scause的**Interrupt**位置1，**Exception Code**置5的情况。而此时我们需要处理的用户态调用ECALL，是**Interrupt**位置0，**Exception Code**置8的情况。所以我们在`trap_handler`这一函数中修改产生中断的分类处理方法，调用`syscall`函数进行处理。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs){
  if(scause >> 63 == 1){ // trap
        if((scause & 0b11111) == 5){
            clock_set_next_event();
            do_timer();
        }
    }
    else if(scause >> 63 == 0){
        if((scause & 0b11111) == 8)
            syscall(regs);
    }
}
```

2.2.6 完成系统调用

首先，我们需要对`struct pt_regs`进行定义。由于`trap_handler`函数增加了这一参数，且我们需要在`entry.S`中交换了进程之后马上调用这一函数，所以根据`__switch_to`中完成的内容可知，我们需要传入的参数就是在`__switch_to`中我们存下来的32个寄存器和特权状态寄存器。当然，我们这也可以只保存返回a0～a7寄存器还有sepc。我们这里全部返回是因为`__switch_to`函数中用sp直接定位比较方便，否则还需要再计算a0是多少偏移量等等。

定义如下：

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

为了保证`trap_handler`函数的正确调用，我们还需要在调用之前为其增加一个参数。

修改**__switch_to**如下：

将sp值计入a2寄存器中作为第三个参数。因为sp刚好指向存储了寄存器的那段stack空间，所以我们可以通过sp偏移量来取出相应reg的值。

```
# call trap_handler
csrr a0, scause
csrr a1, sepc
# sp是函数—trap_handler的第三个参数，表示pt_regs的起始地址
add a2, sp, zero
call trap_handler
```

由于系统调用这一异常，原来程序会返回sepc的位置，也就是异常发生处，但是为了程序能够继续执行下去，这里我们需要手动对`sepc + 4`，使其指向下一指令。所以，我们在处理完一场之后，将sepc+4之后再存。

```
ld t0, 256(sp)
addi t0, t0, 4
csrw sepc, t0
```

在本实验中需要完成两个系统调用函数`SYS_WRITE`和`SYS_GETPID`。我们可以通过x[17]也就是a7寄存器来判断系统调用的类型，然后分别进行判断处理。

```
sys_write(unsigned int fd, const char* buf, size_t count);
```

`syscall函数`如下：

```
void syscall(struct pt_regs* regs) {
    uint64_t ecall = regs->x[17];
    if (ecall == SYS_WRITE) {
        sys_write((unsigned int)regs->x[10], (const char *)regs->x[11], (size_t)regs-
>x[12]);
    }
    else if (ecall == SYS_GETPID) {
        regs->x[10] = current->pid;
    }
}

void sys_write(unsigned int fd, const char* buf, size_t count){
    if (fd == 1) {
        fd = printk((char *)buf);
    }
}
```

2.2.7 修改head.S 和 start_kernel

这里我们需要OS boot完成之后立即调度uapp的运行，即减少一个时间片的运行时间。

将 head.S 中 `enable interrupt sstatus.SIE` 逻辑注释，确保 schedule() 过程不受中断影响。

`main.c`中，在**start_kernel**之后立马进行调度

```
int start_kernel() {
    printk("[S-MODE] Hello RISC-V\n");
    schedule();
    test();
    return 0;
}
```

3. 实验现象

这里我们将输出每一用户程序sp的位置，以及其全局变量counter的值，以此来检验不同用户态程序之间的内存是否完全隔离开来了。我们可以发现，每一进程代表的用户态程序的counter都是从1开始从头计数，并不会相互影响，所以该程序满足了用户态内存隔离性。

```
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
...proc_init done!
[S-MODE] Hello RIS -V

switch to [PID = 4  OUNTER = 2 PRIORITY = 4]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2

switch to [PID = 3  OUNTER = 5 PRIORITY = 10]
[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: 3, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.4
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.5

switch to [PID = 1  OUNTER = 10 PRIORITY = 1]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2
QEMU: Terminated
```

4. 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**

我们所设计的操作系统比较简单，用户态线程和内核态线程的对应关系是一对一的。这是因为我们用户态空间的一个线程，对应的就是内核空间一个完整的运行时，包括task_struct和kernel stack。而且我们在分配内存空间的时候，一个线程分配一个kernel stack，所以就是一对一的关系。

如果是多对一情况的话，则内核态线程进入sleeping状态之后，它只能看到一个实体，无法调度内核态的线程，因为不知道要调哪个。

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

答：`p_filesz`字段对应于文件中段的字节大小，而`p_memsz`是段的内存大小。`p_memsz`大于（或等于）`p_filesz`的原因是可加载的细分可能包含`.bss`部分，其中包含未初始化的数据。将这些数据存储在磁盘上会很浪费，因此只有在ELF文件加载到内存后才会占用空间。此事实由`SHT_NOBITS`部分的`.bss`类型表示。

根据ELF规范，在`p_memsz`大于`p_filesz`的情况下，段的初始化区域后面的额外字节被定义为保持值0。

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？**

因为虽然不同进程的栈虚拟地址是相同的，但其映射的物理地址却是不同的。

由于用户的权限问题，用户态权限受限，不可能通过`walk pagetable`来获得物理地址。不过内核提供了`pagemap`接口，它会对每个`page`生成了一个64`bit`的描述符，来描述虚拟地址这一页对应的物理页帧号或者SWAP里面的偏移。

在Linux中，文件目录/proc记录了当前进程的信息，也就是虚拟文件系统。`/proc/pid/pagemap`这一目录下的`pagemap`文件记录着所链接进程的物理地址信息，允许用户态进程查看每个虚拟页映射到的物理页地址。`pagemap`每一项64bit的值组成如下：

```
* Bits 0-54  page frame number (PFN) if present//present为1时，bit0-54表示物理页号
* Bits 0-4   swap type if swapped
* Bits 5-54  swap offset if swapped
* Bit  55    pte is soft-dirty (see Documentation/vm/soft-dirty.txt)
* Bit  56    page exclusively mapped (since 4.2)
* Bits 57-60 zero
* Bit  61    page is file-page or shared-anon (since 3.5)
* Bit  62    page swapped
* Bit  63    page present//如果为1，表示当前物理页在内存中；为0，表示当前物理页不在内存中
```

所以，每一项的映射方式不同于真正的虚拟地址映射，其文件中遵循独立的对应关系，即虚拟地址相对于0x0经过的页面数是对应项在文件中的偏移量。于是如果用户态程序想要得到物理地址的话，可以先将栈虚拟地址在`pagemap`s中的对应项(`entry`)取出来，判断其bit[`63`]位是否为`1`，如果是`1`的话，说明当前物理页在内存中，则对应项中的物理页号加上偏移地址就能得到物理地址。