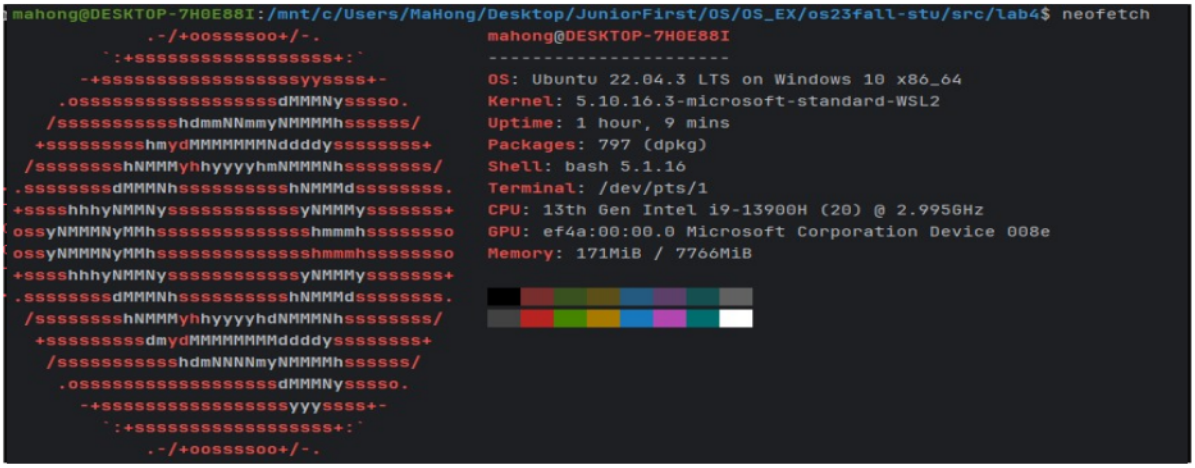
Lab 4: RV64 用户态程序

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. 背景知识

3.1 用户模式和内核模式

处理器存在两种不同的模式：**用户模式**（U-Mode）和**内核模式**（S-Mode）。

- 在用户模式下，执行代码无法直接访问硬件，必须委托给系统提供的接口才能访问硬件或内存。
- 在内核模式下，执行代码对底层硬件具有完整且不受限制的访问权限，它可以执行任何 CPU 指令并引用任何内存地址。

处理器根据处理器上运行的代码类型在这两种模式之间切换。应用程序以用户模式运行，而核心操作系统组件以内核模式运行。

3.2 目标

在 lab3 中，我们启用了**虚拟内存**，这为进程间地址空间相互隔离打下了基础。然而，我们当时只创建了内核线程，它们共用了地址空间 （共用一个内核页表 `swapper_pg_dir` ）。在本次实验中，我们将引入**用户态进程**。

- 当启动用户态应用程序时，内核将为该应用程序创建一个进程，并提供了专用虚拟地址空间等资源。
    - 每个应用程序的虚拟地址空间是私有的，一个应用程序无法更改属于另一个应用程序的数据。

    - 每个应用程序都是独立运行的，如果一个应用程序崩溃，其他应用程序和操作系统将不会受到影响。
- 用户态应用程序可访问的虚拟地址空间是受限的。
    - 在用户态下，应用程序无法访问内核的虚拟地址，防止其修改关键操作系统数据。
    - 当用户态程序需要访问关键资源的时候，可以通过**系统调用**来完成用户态程序与操作系统之间的互动。

3.3 系统调用约定

系统调用是用户态应用程序请求内核服务的一种方式。在 RISC-V 中，我们使用 `ecall` 指令进行系统调用。当执行这条指令时，处理器会提升特权模式，跳转到异常处理函数，处理这条系统调用。

Linux 中 RISC-V 相关的系统调用可以在 `include/uapi/asm-generic/unistd.h` 中找到， syscall(2) 手册页上对RISC-V架构上的调用说明进行了总结，系统调用参数使用 a0 - a5 ， 系统调用号使用 a7 ， 系统调用的返回值会被保存到 a0, a1 中。

3.4 sstatus[SUM] PTE[U]

当页表项 PTE[U] 置 0 时，该页表项对应的内存页为内核页，运行在 U-Mode 下的代码*无法*访问。当页表项 PTE[U] 置 1 时，该页表项对应的内存页为用户页，运行在 S-Mode 下的代码*无法*访问。如果想让 S 特权级下的程序能够访问用户页，需要对 sstatus[SUM] 位置 1 。但是无论什么样的情况下，用户页中的指令对于 S-Mode 而言都是**无法执行**的。

3.5 用户态栈与内核态栈

当用户态程序在用户态运行时，其使用的栈为**用户态栈**。当进行系统调用时，陷入内核处理时使用的栈为**内核态栈**。因此，需要区分用户态栈和内核态栈，且需要在异常处理的过程中对栈进行切换。

3.6 ELF 程序

ELF, short for Executable and Linkable Format. 是当今被广泛使用的应用程序格式

将程序封装成 ELF 格式的意义包括以下几点：

- ELF 文件可以包含将程序正确加载入内存的元数据（metadata）。
- ELF 文件在运行时可以由加载器（loader）将动态链接在程序上的动态链接库（shared library）正确地从硬盘或内存中加载。
- ELF 文件包含的重定位信息可以让该程序继续和其他可重定位文件和库再次链接，构成新的可执行文件。

4. 实验步骤

4.1 准备工程

- 修改 `vmlinux.lds`，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);
        _sramdisk = .;
        *(.uapp .uapp*)
        _eramdisk = .;
        . = ALIGN(0x1000);

    } >ramv AT>ram
...
```

- 修改 `defs.h`，在 `defs.h` 中**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 `mm.h` 中的函数和定义

```
// 分配 page_cnt 个页的地址空间，返回分配内存的地址。保证分配的内存在虚拟地址和物理地址上都是连续的
uint64_t alloc_pages(uint64_t page_cnt);
// 相当于 alloc_pages(1);
uint64_t alloc_page();
// 释放从 addr 开始的之前按分配的内存
void free_pages(uint64_t addr);
```

- 从 `repo` 同步以下文件和文件夹。并按照下面的位置来放置这些新文件。

```
.
├── arch
|   └── riscv
|       └── Makefile
|       └── include
|           └── mm.h
|           └── stdint.h
|       └── kernel
|           └── mm.c
├── include
|   └── elf.h (this is copied from newlib)
└── user
    ├── Makefile
    ├── getpid.c
    ├── link.lds
    ├── printf.c
    ├── start.S
    ├── stddef.h
    ├── stdio.h
    ├── syscall.h
    └── uapp.S
```

- 修改**根目录**下的 Makefile, 将 `user` 纳入工程管理。

```
    ${MAKE} -C user all
    ......
```

- 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 `_sramdisk` 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。

4.2 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;
};
```

4.2.1 修改 task_init

- 将 `sepc` 设置为 `USER_START`。

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

- 配置 `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） 。

经过查阅手册，得到 `sstatus` 寄存器的分布

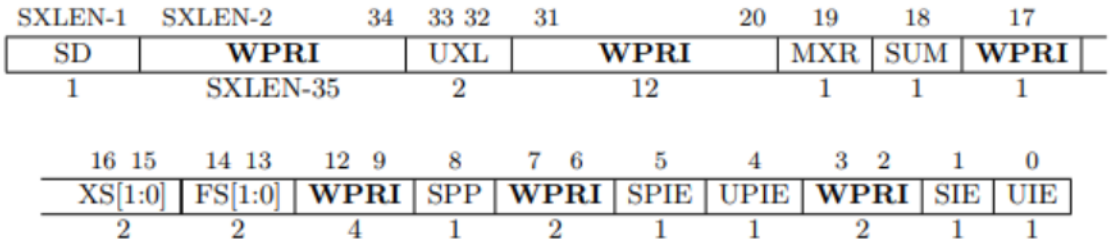


Figure 4.2: Supervisor-mode status register (**sstatus**) for RV64.

```
uint64 sstatus = csr_read(sstatus);
sstatus = sstatus & ~(1<<8);
sstatus = sstatus | 0x40020;
// 将 sstatus 存入 task
task[i]->thread.sstatus = sstatus;
```

- 将 `sscratch` 设置为 `U-Mode` 的 sp，其值为 `USER_END` （即，用户态栈被放置在 `user space` 的最后一个页面）

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

- 对于每个进程，创建属于它自己的页表。

  为了避免 `U-Mode` 和 `S-Mode` 切换的时候切换页表，我们将内核页表 （ `swapper_pg_dir` ） 复制到每个进程的页表中。

  为了实现复制，定义函数 `memcpy`，在 `string.c` 中：

```
void *memcpy(void *dst, const void *src, uint64 n) {
    char *cdst = (char *)dst;
    char *csrc = (char *)src;
    for (uint64 i = 0; i < n; ++i)
        cdst[i] = csrc[i];

    return dst;
}
```

使用该函数实现页表的复制

```
// 页表的分配和映射
pagetable_t tbl = (pagetable_t)kalloc();
// 内核页表的复制
memcpy(tbl, swapper_pg_dir, PGSIZE);
```

- 将 `uapp` 所在的页面映射到每个进行的页表中。注意，在程序运行过程中，有部分数据不在栈上，而在初始化的过程中就已经被分配了空间（比如我们的 `uapp` 中的 `counter` 变量）。所以，二进制文件需要先被 **拷贝** 到一块某个进程专用的内存之后再进行映射，防止所有的进程共享数据，造成预期外的进程间相互影响。

```
// 拷贝二进制文件，以避免进程之间的互相影响
uint64 size_uapp = (uint64)_eramdisk - (uint64)_sramdisk;
uint64 num_pages = size_uapp / PGSIZE + 1;
uint64 tar_ = alloc_pages(num_pages);
uint64 src_ = (uint64)(_sramdisk);
memcpy(tar_, src_, size_uapp);
```

- 将 `uapp` 进行映射

```
// 映射 uapp，权限为 U|X|W|R|V
uint64 va = USER_START;
create_mapping(tbl, va, tar_ - PA2VA_OFFSET, size_uapp, 0b11111);
```

- 设置用户态栈。对每个用户态进程，其拥有两个栈：用户态栈和内核态栈；其中，内核态栈在 `lab3` 中我们已经设置好了。我们可以通过 `alloc_page` 接口申请一个空的页面来作为用户态栈，并映射到进程的页表中。

```
// 申请空页表作为用户态栈
task[i]->thread_info->user_sp = kalloc();
// 映射 U-Mode Stack，权限为 U|-|W|R|V
pa = task[i]->thread_info->user_sp - PA2VA_OFFSET;
va = USER_END - PGSIZE;
create_mapping(tbl, va, pa, PGSIZE, 0b10111);
```

4.2.2 修改 __switch_to

- 加入 `sepc, sstatus, sscratch` 保存和恢复的逻辑

```
# 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
ld t1, 112(t0)
csrw sepc, t1
ld t1, 120(t0)
csrw sstatus, t1
ld t1, 128(t0)
csrw sscratch, t1
```

- 加入页表切换逻辑

```
    # 切换页表
    csrr t1, satp
    sd t1, 136(t0)
    ......

    # 切换页表
    ld t1, 136(t0)
    csrw satp, t1
    # 刷新缓冲区
    sfence.vma zero, zero
```

- 在切换了页表之后，需要通过 `fence.i` 和 `vma.fence` 来刷新 TLB 和 ICache。

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

4.3.1 修改 _trap

- RISC-V 中只有一个栈指针寄存器(sp)，需要完成用户栈与内核栈的切换。

```
    # 用户栈和内核栈的切换
    .macro switch_U_S
        csrr t1, sscratch
        csrw sscratch, sp
        add sp, t1, x0
    .endm
```

- 修改 `_trap` 。同理 在 `_trap` 的首尾我们都需要做类似的操作。若内核线程(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。

```
    # 根据 sscratch 的值判断触发异常进程的类型
    csrr t1, sscratch
    bne t1, x0, _utraps
```

- 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，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` ）。

```
_utraps:

    # 切换 U/S-Mode Stack
    switch_U_S
    ...
    switch_U_S

    sret
```

4.3.2 修改 `__dummy`

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。

```
__dummy:

    # U-S-Mode Stack
    switch_U_S

    sret
```

4.3.3 修改 trap_handler

- 添加 `struct pt_regs` 结构， 在 _trap 中我们将寄存器的内容连续的保存在 `S-Mode Stack` 上，因此我们可以将这一段看做一个叫做 `pt_regs` 的结构体。我们可以从这个结构体中取到相应的寄存器的值

```
struct pt_regs {
    uint64 x[32];
    uint64 sepc;
    uint64 sstatus;
};
```

- `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:

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    ...void trap_handler(uint64 scause, uint64 sepc, struct pt_regs *regs) {
    if (scause & 0x8000000000000000 == 0x8000000000000000) {
        if (scause & 0x10 == 0x10) {

            // printk("scause : %d\n", scause);
            // printk("kernel is running!\n");
            // printk("[S] Supervisor Mode Timer Interrupt\n");

            clock_set_next_event();
            do_timer();

        }
    } else {
        // interrupt == 0 && exception code == 8
            if(scause == 8) { // 进行系统调用
                sys_call(regs);
        }
    }

    return;
    // interrupt / exception 不做处理
}
```

4.4 添加系统调用

4.4.1 64 和 172 号系统调用

- 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)

```
void sys_call(uint64 x[]) {
    uint64 func = x[17]; // 获取id

    if (func == SYS_GETPID) { // 如果是172号系统调用

        x[10] = current->pid;

    } else if (func == SYS_WRITE){ // 如果是64号系统调用

        if (x[10] == 1) {
            ((char*)(x[11]))[x[12]] = '\0';  // 使 char 以 '\0' 结尾
            printk((char*)(x[11]));
            x[10] = x[12]; // 存储字符串长度
        }
    }
    return;
}
```

4.4.2 修改 entry.S

- 针对系统调用这一类异常， 我们需要手动将 `sepc + 4` （`sepc` 记录的是触发异常的指令地址，由于系统调用这类异常处理完成之后， 我们应该继续执行后续的指令，因此需要我们手动修改 `sepc` 的地址，使得 `sret` 之后 程序继续执行）。

```
    ld t0, 32*8(sp)
    addi t0, t0, 4 # spec += 4, 系统调用之后继续执行
    csrw sepc, t0
```

4.5 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。

```c
int start_kernel() {

    printk("[S-Mode] Hello RISC-V\n");

    schedule();

    test(); // DO NOT DELETE !!!

    return 0;
}
```

5. 测试纯二进制文件

- `uapp.S` 文件

```
.section .uapp

.incbin "uapp.bin"
```

测试结果：

```
...buddy_init done!
...proc_init done!
[S-Mode] Hello RISC-V

[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
[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: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, 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
```

6. 添加 ELF 支持

6.1 ELF Header

ELF 文件中包含了将程序加载到内存所需的信息。当我们通过 `readelf` 来查看一个 ELF 可执行文件的时候，我们可以读到被包含在 ELF Header 中的信息。

其中包含了两种将程序分块的粒度，Segment（段）和 Section（节），我们以段为粒度将程序加载进内存中。可以看到，给出的样例程序包含了三个段，我们需要将 LOAD 标志的段装入内存。

6.2 修改 `uapp.S`

将 `uapp.S` 中的 payload 换成 ELF 文件。

```
/* user/uapp.S */
.section .uapp

.incbin "uapp"
```

6.3 完成装载函数

```
static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)_sramdisk;   // ELF 文件的起始地址

    uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;   // 获得 phdr 的起始地址
    int phdr_cnt = ehdr->e_phnum;   // 获得段的个数

    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) {  // 如果段的 type 是 PT_LOAD, 则装入内存

            // 先将 uapp 拷贝，再做地址映射
            uint64 offset = (uint64)(phdr->p_vaddr) - PGROUNDDOWN(phdr-
>p_vaddr);
            uint64 num_pages = (phdr->p_memsz + offset) / PGSIZE + 1;

            uint64 tar_ = alloc_pages(num_pages);
            uint64 src_ = (uint64)(_sramdisk) + phdr->p_offset;
            memcpy(tar_ + offset, src_, phdr->p_memsz);

            // 映射 U-Mode Stack, 权限为 U|X|W|R|V
            create_mapping(task->pgd, PGROUNDDOWN(phdr->p_vaddr), tar_ -
PA2VA_OFFSET, num_pages * PGSIZE, 0b11111);
        }
    }

    // 映射 U-Mode Stack, 权限为 U|-|W|R|V
    create_mapping(task->pgd, (USER_END)-PGSIZE, task->thread_info->user_sp -
PA2VA_OFFSET, PGSIZE, 0b10111);

    // 将 sepc 设置为 ELF 文件中可执行段的起始地址
    task->thread.sepc = ehdr->e_entry;
    // sstatus
    uint64 sstatus = csr_read(sstatus);
    sstatus = sstatus & ~(1<<8);
    sstatus = sstatus | 0x40020;

    task->thread.sstatus = sstatus;

    // 将 sscratch 设置为 U-mode 的 sp
    task->thread.sscratch = USER_END;

    // 将分配到的页表地址转换为实际页表，装入 satp
    uint64 satp = csr_read(satp);
    satp = ((satp >> 44) << 44);
    satp |= (((uint64)task->pgd - PA2VA_OFFSET) >> 12);
    task->pgd = satp;

}
```

6.4 实验结果

```
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
...proc_init done!
[S-Mode] Hello RISC-V

[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
[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: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, 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: 4, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.4
```

7. 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）

   一对一的。

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

   `p_filesz` 表示 `segment` 在文件中的字节大小，`p_memsz` 表示 `segment` 段装载到内存后的大小。由于 `p_filesz` 中含有没有初始化的 `.bss` 段等内容，将这些内容在硬盘中存储会浪费空间，所以在装入内存之后才会开辟空间，故大小不同。

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？

   因为不同的用户进程，虽然虚拟地址相同，但是映射的物理地址不同。

   用户没有常规的方法知道自己栈所在的物理地址，这是虚拟内存的保护和隔离作用。

8. 实验心得

通过本实验，我进一步理解了操作系统用户空间和内核空间这两个概念，理解了 csr 寄存器的功能，并理解了用户态和内核态的切换逻辑。

我理解了二进制可执行文件的结构，明白了操作系统对可执行文件的装载过程。