Lab 4: RV64 用户态程序

3210105944 黄锦骏

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

...
```

> 如果你要使用 `uapp_start` 这个符号，可以在代码里这样来声明它：`extern char uapp_start[]`，这样就可以像一个字符数组一样来访问这块内存的内容。例如，程序的第一个字节就是 `uapp_start[0]`。

- 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
```

- 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。值得注意的是，我们在 `mm` 中添加了 `buddy system`，但是也保证了原来调用的 `kalloc` 和 `kfree` 的兼容。你应该无需修改原先使用了 `kalloc` 的相关代码，如果出现兼容性问题可以联系助教。为了减小大家的工作量，我们替大家实现了 Buddy System，大家可以直接使用这些函数来管理内存：

```
// 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);
```

```
.
├── 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。

```
.PHONY:all run debug clean
all: clean
    ...
    ${MAKE} -C user all
    ...

TEST:
    ...
    ${MAKE} -C user all
    ...
...

clean:
    ...
    ${MAKE} -C user clean
    ...
```

```
17
```

- 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。

```
0000000000000004 <getpid>:
   4:   fe010113            addi    sp,sp,-32
   8:   00813c23            sd      s0,24(sp)
   c:   02010413            addi    s0,sp,32
  10:   fe843783            ld      a5,-24(s0)
  14:   0ac00893            li      a7,172
  18:   00000073            ecall                                   <-
SYS_GETPID
...

00000000000000d8 <vprintfmt>:
...
60c:    00070513            mv  a0,a4
610:    00068593            mv  a1,a3
614:    00060613            mv  a2,a2
618:    00000073            ecall                                   <- SYS_WRITE
...
```

3.2 创建用户态进程

1. 修改`proc.h`
   - 将`NR_TASKS`修改为5
   - 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`pgd`。

```
#ifndef _PROC_H
#define _PROC_H
#include "types.h"

#define NR_TASKS (1+4) // 用于控制 最大线程数量 （idle 线程 + 31 内核线程）
```

```
6
7  typedef unsigned long* pagetable_t;
8
9  /* 线程状态段数据结构 */
10 struct thread_struct {
11     uint64 ra;
12     uint64 sp;
13     uint64 s[12];
14
15     uint64 sepc, sstatus, sscratch;
16 };
17
18 /* 线程数据结构 */
19 struct task_struct {
20     struct thread_info thread_info;
21     uint64 state;    // 线程状态
22     uint64 counter;  // 运行剩余时间
23     uint64 priority; // 运行优先级 1最低 10最高
24     uint64 pid;      // 线程id
25
26     struct thread_struct thread;
27
28     pagetable_t pgd;
29 };
30
```

2. 修改task_init

- 对每个用户态进程，其拥有两个 stack： `U-Mode Stack` 以及 `S-Mode Stack`， 其中 `S-Mode Stack` 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 `U-Mode Stack`。
- 为每个用户态进程创建自己的页表 并将 `uapp` 所在页面，以及 `U-Mode Stack` 做相应的映射，同时为了避免 `U-Mode` 和 `S-Mode` 切换的时候切换页表，我们也将内核页表 （`swapper_pg_dir`） 复制到每个进程的页表中。注意程序运行过程中，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，造成期望外的进程间相互影响。

```
1  // /arch/riscv/kernel/proc.c
2  //分配页表内存
3  task[i]->pgd = (pagetable_t)kalloc();
4  memset(task[i]->pgd , 0, PGSIZE);
5  // 拷贝内核页表到进程页表
6  for(int j = 0; j < PGSIZE / 8; j++){
7      task[i]->pgd[j]=swapper_pg_dir[j];
8  }
9  // 申请uapp在进程内的专用内存
10 uint64 uapp = alloc_pages((PGROUNDUP((uint64)uapp_end) -
   (uint64)uapp_start) / PGSIZE);
11 //将uapp.bin拷贝到进程内存
12 for(int t = 0; t < (PGROUNDUP((uint64)uapp_end) -
   (uint64)uapp_start) / PGSIZE; t++){
```

```
    uint64* src = (uint64*)(uapp_start + t * PGSIZE);
    uint64* dst = (uint64*)(uapp + t * PGSIZE);
    for (int k = 0; k < PGSIZE / 8; k++) {
        dst[k] = src[k];
    }
}
//创建uapp的映射
create_mapping(task[i]->pgd , (uint64)USER_START,  (uint64)uapp - (uint64)PA2VA_OFFSET, (uint64)uapp_end - (uint64)uapp_start, 31);
//创建用户栈的映射，映射到用户空间的最后一个页面
create_mapping(task[i]->pgd , (uint64)USER_END - (uint64)PGSIZE, (uint64)kalloc() - (uint64)PA2VA_OFFSET, (uint64)PGSIZE, 23);
```

- 对每个用户态进程我们需要将 `sepc` 修改为 `USER_START`，配置修改好 `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` 设置为 `U-Mode` 的 sp，其值为 `USER_END`（即 `U-Mode Stack` 被放置在 `user space` 的最后一个页面）。

```
// /arch/riscv/kernel/proc.c
task[i]->thread.sepc = USER_START;
task[i]->thread.sstatus = (1ULL << 5) + (1ULL << 18);
task[i]->thread.sscratch = USER_END;
```

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

```
# /arch/riscv/kernel/entry.S
__switch_to:
   ...

    # save state of sepc, sstatus, sscratch
    csrr t1, sepc
    sd t1, 160(a0)
    csrr t1, sstatus
    sd t1, 168(a0)
    csrr t1, sscratch
    sd t1, 176(a0)

   ...

    # load state of sepc, sstauts, sscratch
    ld t1, 160(a1)
    csrw sepc, t1
    ld t1, 168(a1)
    csrw sstatus, t1
    ld t1, 176(a1)
    csrw sscratch, t1

    #切换页表
    li t5, PA2VA_OFFSET
    li t0, 8
    slli t0, t0, 60 # Mode = 8
    ld t1, 184(a1)
    sub t1, t1, t5
    srli t1, t1, 12
```

```
    or t0, t0, t1

    csrw satp, t0
    sfence.vma zero, zero
    fence.i
    ret
```

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

- 与 ARM 架构不同的是，RISC-V 中只有一个栈指针寄存器(sp)，因此需要我们来完成用户栈与内核栈的切换。
- 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` ） 。

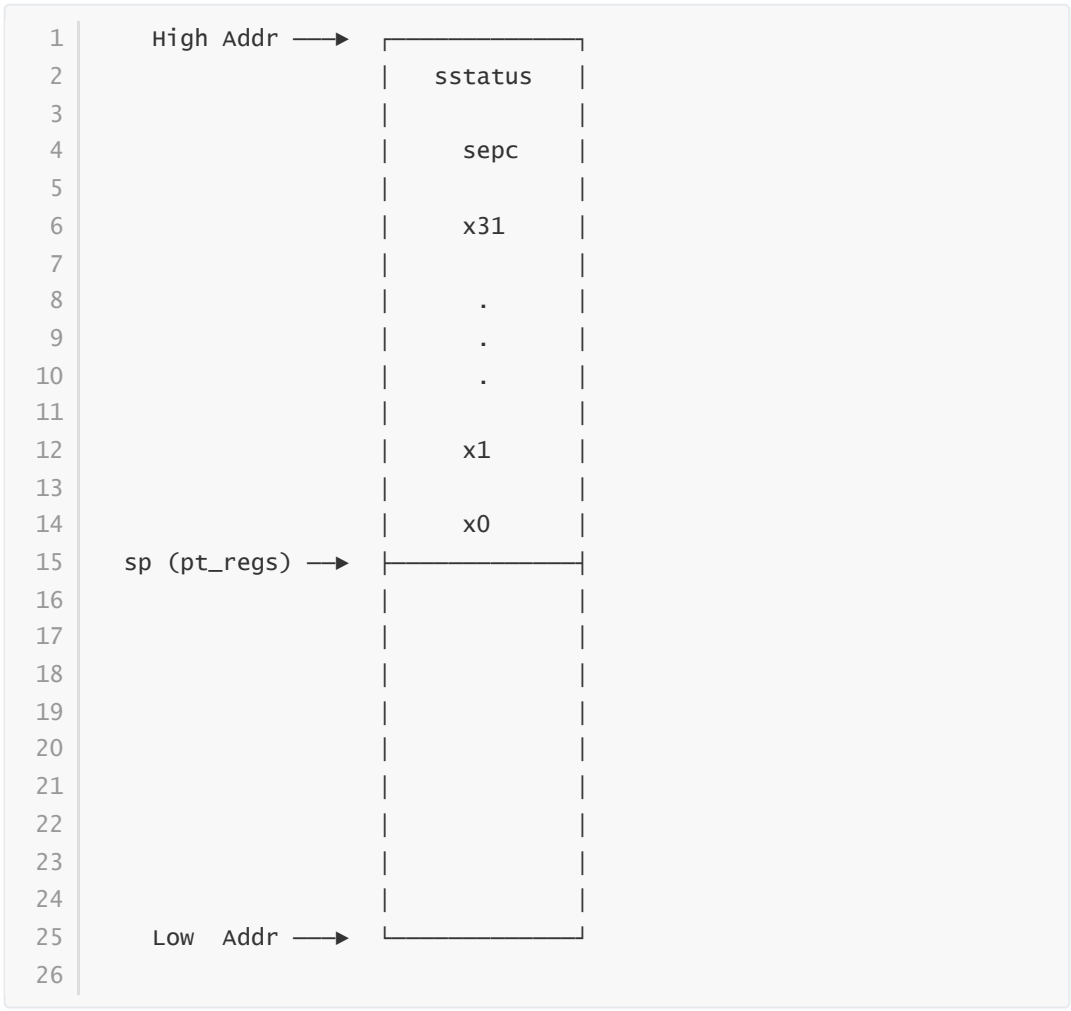
1. 修改 `__dummy` 。在 **4.2** 中 我们初始化时， `thread_struct.sp` 保存了 `S-Mode sp`，`thread_struct.sscratch` 保存了 `U-Mode sp`， 因此在 `S-Mode -> U->Mode` 的时候，我们只需要交换对应的寄存器的值即可。

```
__dummy:
    # la t0, dummy # 将 dummy() 的地址加载到寄存器 t0 中
    # csrw sepc, t0 # 将 dummy() 的地址写入寄存器 sepc

    # 交换 u-mode stack和 s-mode stack
    csrr t1, sscratch
    csrw sscratch, sp
    add sp, t1, x0

    sret
```

2. 修改 `_trap` 。同理 在 `_trap` 的首尾我们都需要做类似的操作。**注意如果是 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)**

```
_traps:
    # ----------
        #判断为内核线程 / 用户线程
        csrr t1, sscratch
        beq t1, x0, _trap_core

        csrw sscratch,sp
        add sp, t1, x0

        ...

        csrr t1, sscratch
        beq t1, x0, _trap_end
        csrw sscratch,sp
        add sp, t1, x0

        ...
```

3. 修改`trap_handler`，`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
    - 经查询得知，`` `ECALL_FROM_U_MODE`` exception对应的scause为`0x8`，捕获到就调用`sys_call`
    - 这里需要解释新增加的第三个参数`regs`， 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（比如 syscall 中我们需要从 a0 ~ a7 寄存器中取到参数）。这个结构体中的值也可以按需添加，同时需要在`_trap`中存入对应的寄存器值以供使用， 示例如下图：



    - `trap_handler`的实现以及`struct pt_regs`的定义如下：

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


void trap_handler(uint64 scause, uint64 sepc, struct pt_regs *regs) {

    if((scause&1UL << 63)&&((scause&0x5)==0x5))
    {
        // printk("[S] Supervisor Mode Timer Interrupt\n");
        clock_set_next_event();
        do_timer();
```

```
    }
    //Interrupt =1 && exception code == 8
    else if((scause&0x8) == 8)
    {
        sys_call(regs);
    }else{
        printk("scause=%ld__",scause);
    }
    return;
}
```

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)
- 增加 `syscall.c` `syscall.h` 文件， 并在其中实现 `getpid` 以及 `write` 逻辑。
- 系统调用的返回参数放置在 `a0` 中 (不可以直接修改寄存器， 应该修改 regs 中保存的内容)。
- 实现的`syscall`如下：

```
void sys_call(struct pt_regs *regs){
    uint64 func = regs->x[17];

    if(func == 172){
        regs->x[10] = current->pid;
    }
    else if(func == 64){
        //fd(a0) = 1
        if(regs->x[10] == 1){
            //a1 = buf, a2 = count
            ((char *)(regs->x[11]))[regs->x[12]] = '\0';
            //a0 = 输出长度
            printk((char *)(regs->x[11]));
            regs->x[10] = regs->x[12];
        }
    }
    return;
}
```

- 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）。

```
# /arch/riscv/kernel/entry.S
    ...

    ld t0, 8(x2)
    addi t0, t0, 4 # 手动修改 sepc 的地址，使得 sret 之后 程序继续执行
    csrw sepc, t0

    ...
```

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。

```
int start_kernel() {
    printk("2022");
    printk(" Hello RISC-V\n");

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

    return 0;
}

```

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

```
# /arch/riscv/kernel/head.S

    # 使schedule不受中断影响
    #li t6, 0x2
    #csrr t3, sstatus
    #or t3, t3, t6
    #csrw sstatus, t3
```

4.6 测试纯二进制文件

- 测试输出如下：

```
OpenSBI v0.9
   ____                   _____ ____ _____
  / __ \                 / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|
```

```
10
11  Platform Name             : riscv-virtio,qemu
12  Platform Features         : timer,mfdeleg
13  Platform HART Count       : 1
14  Firmware Base             : 0x80000000
15  Firmware Size             : 100 KB
16  Runtime SBI Version       : 0.2
17
18  Domain0 Name              : root
19  Domain0 Boot HART         : 0
20  Domain0 HARTs             : 0*
21  Domain0 Region00          : 0x0000000080000000-0x0000000008001ffff ()
22  Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff
    (R,W,X)
23  Domain0 Next Address      : 0x0000000080200000
24  Domain0 Next Arg1         : 0x0000000087000000
25  Domain0 Next Mode         : S-mode
26  Domain0 SysReset          : yes
27
28  Boot HART ID              : 0
29  Boot HART Domain          : root
30  Boot HART ISA             : rv64imafdcsu
31  Boot HART Features        : scounteren,mcounteren,time
32  Boot HART PMP Count       : 16
33  Boot HART PMP Granularity : 4
34  Boot HART PMP Address Bits: 54
35  Boot HART MHPM Count      : 0
36  Boot HART MHPM Count      : 0
37  Boot HART MIDELEG         : 0x0000000000000222
38  Boot HART MEDELEG         : 0x000000000000b109
39  ...buddy_init done!
40  ...proc_init done!
41  2022 Hello RISC-V
42  switch to [PID = 1 COUNTER = 4]
43  [U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
44  [U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2
45  [U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.3
46  [U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.4
47  switch to [PID = 4 COUNTER = 5]
48  [U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
49  [U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
50  [U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.3
51  [U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.4
52  [U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.5
53  switch to [PID = 3 COUNTER = 8]
54  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
55  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
56  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
57  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.4
58  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.5
59  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.6
60  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.7
61  [U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.8
62  switch to [PID = 2 COUNTER = 9]
63  [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: 2, sp is 0000003fffffffe0, this is print No.7
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.8
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.9
switch to [PID = 1 COUNTER = 1]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.5
switch to [PID = 2 COUNTER = 4]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.10
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.11
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.12
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.13
switch to [PID = 4 COUNTER = 4]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.6
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.7
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.8
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.9
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.10
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.11
switch to [PID = 3 COUNTER = 5]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.9
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.10
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
```

4.7 添加 ELF 支持

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：

```
$ readelf -a uapp
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x100e8
  Start of program headers:          64 (bytes into file)
  Start of section headers:          3200 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         9
  Section header string table index: 8
```

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         00000000000100e8  000000e8
       00000000000006fc  0000000000000000  AX       0     0     4
  [ 2] .rodata           PROGBITS         00000000000107e8  000007e8
       0000000000000078  0000000000000000   A       0     0     8
  [ 3] .bss              NOBITS           0000000000011860  00000860
       00000000000003f0  0000000000000000  WA       0     0     8
  [ 4] .comment          PROGBITS         0000000000000000  00000860
       000000000000002b  0000000000000001  MS       0     0     1
  [ 5] .riscv.attributes RISCV_ATTRIBUTE  0000000000000000  0000088b
       000000000000002e  0000000000000000           0     0     1
  [ 6] .symtab           SYMTAB           0000000000000000  000008c0
       00000000000002d0  0000000000000018           7    17     8
  [ 7] .strtab           STRTAB           0000000000000000  00000b90
       00000000000000a1  0000000000000000           0     0     1
  [ 8] .shstrtab         STRTAB           0000000000000000  00000c31
       0000000000000049  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  RISCV_ATTRIBUT 0x000000000000088b 0x0000000000000000 0x0000000000000000
                 0x000000000000002e 0x0000000000000000  R      0x1
  LOAD           0x00000000000000e8 0x00000000000100e8 0x00000000000100e8
                 0x0000000000000778 0x0000000000001b68  RWE    0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

 Section to Segment mapping:
  Segment Sections...
   00     .riscv.attributes
   01     .text .rodata .bss
   02

There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type RISC-V is not currently
supported.

Symbol table '.symtab' contains 30 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
```

```
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000000100e8     0 SECTION LOCAL  DEFAULT    1 .text
     2: 00000000000107e8     0 SECTION LOCAL  DEFAULT    2 .rodata
     3: 0000000000011860     0 SECTION LOCAL  DEFAULT    3 .bss
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .comment
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
.riscv.attributes
     6: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS start.o
     7: 00000000000100e8     0 NOTYPE  LOCAL  DEFAULT    1 $x
     8: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS getpid.c
     9: 00000000000100ec    52 FUNC    LOCAL  DEFAULT    1 getpid
    10: 00000000000100ec     0 NOTYPE  LOCAL  DEFAULT    1 $x
    11: 0000000000010120     0 NOTYPE  LOCAL  DEFAULT    1 $x
    12: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS printf.c
    13: 000000000001017c     0 NOTYPE  LOCAL  DEFAULT    1 $x
    14: 00000000000101d4  1412 FUNC    LOCAL  DEFAULT    1 vprintfmt
    15: 00000000000101d4     0 NOTYPE  LOCAL  DEFAULT    1 $x
    16: 0000000000010758     0 NOTYPE  LOCAL  DEFAULT    1 $x
    17: 0000000000010758   140 FUNC    GLOBAL DEFAULT    1 printf
    18: 0000000000012060     0 NOTYPE  GLOBAL DEFAULT  ABS
__global_pointer$
    19: 0000000000011860     0 NOTYPE  GLOBAL DEFAULT    2 __SDATA_BEGIN__
    20: 0000000000011860     4 OBJECT  GLOBAL DEFAULT    3 tail
    21: 00000000000100e8     0 NOTYPE  GLOBAL DEFAULT    1 _start
    22: 0000000000011868  1000 OBJECT  GLOBAL DEFAULT    3 buffer
    23: 0000000000011c50     0 NOTYPE  GLOBAL DEFAULT    3 __BSS_END__
    24: 0000000000011860     0 NOTYPE  GLOBAL DEFAULT    3 __bss_start
    25: 0000000000010120    92 FUNC    GLOBAL DEFAULT    1 main
    26: 000000000001017c    88 FUNC    GLOBAL DEFAULT    1 putc
    27: 0000000000011860     0 NOTYPE  GLOBAL DEFAULT    2 __DATA_BEGIN__
    28: 0000000000011860     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    29: 0000000000011c50     0 NOTYPE  GLOBAL DEFAULT    3 _end

No version information found in this file.
Attribute Section: riscv
File Attributes
  Tag_RISCV_arch: "rv64i2p0_m2p0_a2p0_f2p0_d2p0"
```

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，这里我们只关注 Type 为 LOAD 的段，LOAD 表示他们需要在开始运行前被加载进内存中，这是我们在初始化进程的时候需要执行的工作。

而“节”代表了更细分的语义，比如 `.text` 一般包含了程序的指令，`.rodata` 是只读的全局变量等，大家可以自行 Google 来学习更多相关内容。

所以代码怎么写?

首先我们需要将 `uapp.S` 中的 payload 给换成我们的 ELF 文件。

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

.incbin "uapp"

```

这时候从 `uapp_start` 开始的数据就变成了名为 `uapp` 的 ELF 文件，也就是说 `uapp_start` 处 32-bit 的数据不再是我们需要执行第一条指令了，而是 ELF Header 的开始。

这时候就需要你对 `task_init` 中的初始化步骤进行修改。我们给出了 ELF 相关的结构体定义（`elf.h`），大家可以直接使用。你可能会使用到的结构体或者域如下：

```
Elf64_Ehdr   // 你可以将 uapp_start 强制转化为改类型的指针，
               然后把那一块内存当成此类结构体来读其中的数据，其中包括：
    e_ident  // Magic Number，你可以通过这个域来检测自己是不是真的正在读一个 Ehdr，
               值一定是 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
    e_entry  // 程序的第一条指令被存储的用户态虚拟地址
    e_phnum  // ELF 文件包含的 Segment 的数量
    e_phoff  // ELF 文件包含的 Segment 数组相对于 Ehdr 的偏移量

Elf64_Phdr   // 存储了程序各个 Segment 相关的 metadata
             // 你可以将 uapp_start + e_phoff 强制转化为此类型，就会指向第一个
Phdr,
             // uapp_start + e_phoff + 1 * sizeof(Elf64_Phdr)，则指向第二个‘
    p_filesz // Segment 在文件中占的大小
    p_memsz  // Segment 在内存中占的大小
    p_vaddr  // Segment 起始的用户态虚拟地址
    p_offset // Segment 在文件中相对于 Ehdr 的偏移量
    p_type   // Segment 的类型
    p_flags  // Segment 的权限（包括了读、写和执行）

```

我们可以按照这些信息，在从 `uapp_start` - `uapp_end` 这个 ELF 文件中的内容 **拷贝** 到我们开辟的内存中。

其中相对文件偏移 `p_offset` 指出相应 segment 的内容从 ELF 文件的第 `p_offset` 字节开始, 在文件中的大小为 `p_filesz`, 它需要被分配到以 `p_vaddr` 为首地址的虚拟内存位置, 在内存中它占用大小为 `p_memsz`. 也就是说, 这个 segment 使用的内存就是 `[p_vaddr, p_vaddr + p_memsz)` 这一连续区间, 然后将 segment 的内容从ELF文件中读入到这一内存区间, 并将 `[p_vaddr + p_filesz, p_vaddr + p_memsz)` 对应的物理区间清零. （本段内容引用自南京大学PA)

你也可以参考这篇 blog 中关于 **静态** 链接程序的载入过程来进行你的载入。

这里有不少例子可以举，为了避免同学们在实验中花太多时间，我们告诉大家可以怎么找到实验中这些相关变量被存在了哪里：(注意以下的 `uapp_start` 类型使用的是 `char*`，如果你在使用其他类型，需要根据你使用的类型去调整针对指针的算数运算。)

- `Elf64_Ehdr* ehdr = (Elf64_Ehdr*)uapp_start`，从地址 uapp_start 开始，便是我们要找的 Ehdr.
- `Elf64_Phdr* phdrs = (Elf64_Phdr*)(uapp_start + ehdr->phoff)`，是一个 Phdr 数组，其中的每个元素都是一个 `Elf64_Phdr`.
- `phdrs[ehdr->phnum - 1]` 是最后一个 Phdr.
- `phdrs[0].p_type == PT_LOAD`，说明这个 Segment 的类型是 LOAD，需要在初始化时被加载进内存。
- `(void*)(uapp_start + phdrs[1].p_offset)` 将会指向第二个段中的内容的开头.

- `load_program` 具体实现如下：
    - 在实现过程中由于每个 `segment` 将要分配到的虚拟地址没有与页面对齐，所以需要进行对齐

- 在实现中发现，需要将对齐时多余的内存也要进行分配才能完成所有`segment`的对齐

```
// /arch/riscv/kernel/proc.c
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;
    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 page_num = (phdr->p_vaddr - PGROUNDDOWN((uint64)phdr->p_vaddr) + phdr->p_memsz + PGSIZE - 1) / PGSIZE;
            uint64 uapp = alloc_pages(page_num);
            memset(uapp, 0, page_num*PGSIZE);
            for(int t = 0; t < page_num; t++){
                uint64* src = (uint64*)(uapp_start + t * PGSIZE + phdr->p_offset);
                uint64* dst = (uint64*)(uapp + t * PGSIZE + phdr->p_vaddr - PGROUNDDOWN((uint64)phdr->p_vaddr));
                for (int k = 0; k < PGSIZE / 8; k++) {
                    dst[k] = src[k];
                }
            }
            // do mapping  X|W|R|V
            create_mapping(task->pgd, PGROUNDDOWN((uint64)phdr->p_vaddr), uapp - PA2VA_OFFSET, PGSIZE*page_num,
            ((phdr->p_flags & PF_X) << 3) | ((phdr->p_flags & PF_W) << 1) | ((phdr->p_flags & PF_R) >> 1) | 0b10001);
        }
    }

    // allocate user stack and do mapping
    // code...
    create_mapping(task->pgd, USER_END - PGSIZE, (uint64)kalloc - PA2VA_OFFSET, PGSIZE, 23);

    // following code has been written for you
    // set user stack
    // pc for the user program
    task->thread.sepc = ehdr->e_entry;
    // sstatus bits set
    task->thread.sstatus = (1ULL << 5) + (1ULL << 18);
    // user stack for user program
    task->thread.sscratch = USER_END;
}
```

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）

   用户态（一）对内核态（一）

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

   `p_filesz` 通常表示 段在文件中的大小，而 `p_memsz` 表示段在内存中的大小。大小不同的原因可能有如下三点：

   - 在执行加载程序时，OS可能要求段在内存中的大小是某个对齐值的倍数，就会导致 `p_memsz` 大于 `p_filesz`，多分配的空间会被填充为0
   - BSS段的全局变量在可执行文件中不占用实际空间，仅仅记录大小，而在内存中则需要分配空间并初始化。
   - 有些段可能存在冬天分配生成的数据，这些数据在文件中是不存在的

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？

   因为每个进程都有自己的专属页表，在不同的进程页表中，进程的栈的虚拟地址映射到的物理地址是不相同的。

   在Linux系统中，`/proc/pid/pagemap` 文件记录了对应进程的物理页号信息，我们可以通过该文件查看每个虚拟页映射到的物理页

```
# proc/pid/pagemap.  This file lets a userspace process find out which
   physical frame each virtual page is mapped to.  It contains one 64-
bit
   value for each virtual page, containing the following data (from
   fs/proc/task_mmu.c, above pagemap_read):

    * 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
```

5. 心得体会

在本次做实验的过程中，由于要修改的文件较多，所以才debug过程中十分艰难，在进trap的时候，一直报scause=12的错误，即Instruction Fetch Fault。于是去Debug初始化的部分和汇编文件，但是均发现没有什么奇怪的地方(表面上)，后来逐步Debug发现每次跳到用户空间第一个地址就寄了，于是就去查看页表的创建。发现在具体的创建过程中uapp.bin文件根本没有加载到内存中去..... 最后终于发现pull下来的文件和网站上不一样。

具体实践的过程中，发现自己对于步骤的理解并不十分清楚，在实现过程中不甚顺利，还得加强自己的编程能力。