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 背景知识

3.0 前言

在 lab4 中，我们开启虚拟内存，这为进程间地址空间相互隔离打下了基础。之前的实验中我们只创建了内核线程，他们共用了地址空间（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。当启动用户模式应用程序时，内核将为该应用程序创建一个进程，为应用程序提供了专用虚拟地址空间等资源。因为应用程序的虚拟地址空间是私有的，所以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不会受到影响。同时，用户模式应用程序可访问的虚拟地址空间也受到限制，在用户模式下无法访问内核的虚拟地址，防止应用程序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。

3.1 User 模式基础介绍

处理器具有两种不同的模式：用户模式和内核模式。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。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。处理器根据处理器上运行的代码类型在两种模式之间切换。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 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。

3.3 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。

3.4 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**，当调用 SYSCALL时候，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，并在异常处理的过程中需要对栈进行切换。

3.5 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。例如当我们运行 `gcc <some-name>.c` 后产生的 `a.out` 输出文件的格式就是 ELF。

```
$ cat hello.c
#include <stdio.h>

int main() {
    printf("hello, world\n");
    return 0;
}
$ gcc hello.c
$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=dd33139196142abd22542134c20d85c571a78b0c,
for GNU/Linux 3.2.0, not stripped
```

将程序封装成 ELF 格式的重要意义是，在其中可以包含如何将程序正确地加载入内存的 metadata，并且在运行时可以由 loader 来将动态链接在程序上的动态链接库(shared library)正确地从磁盘或内存中加载(load)，以及，ELF 文件中包含的重定位信息可以让该程序继续和别的可重定位文件和库再次链接，构成新的可执行文件。

为了简化实验步骤，我们使用的是静态链接的程序，所以不会涉及链接动态链接库的内容。

4 实验步骤

4.1 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。
- 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
...
```

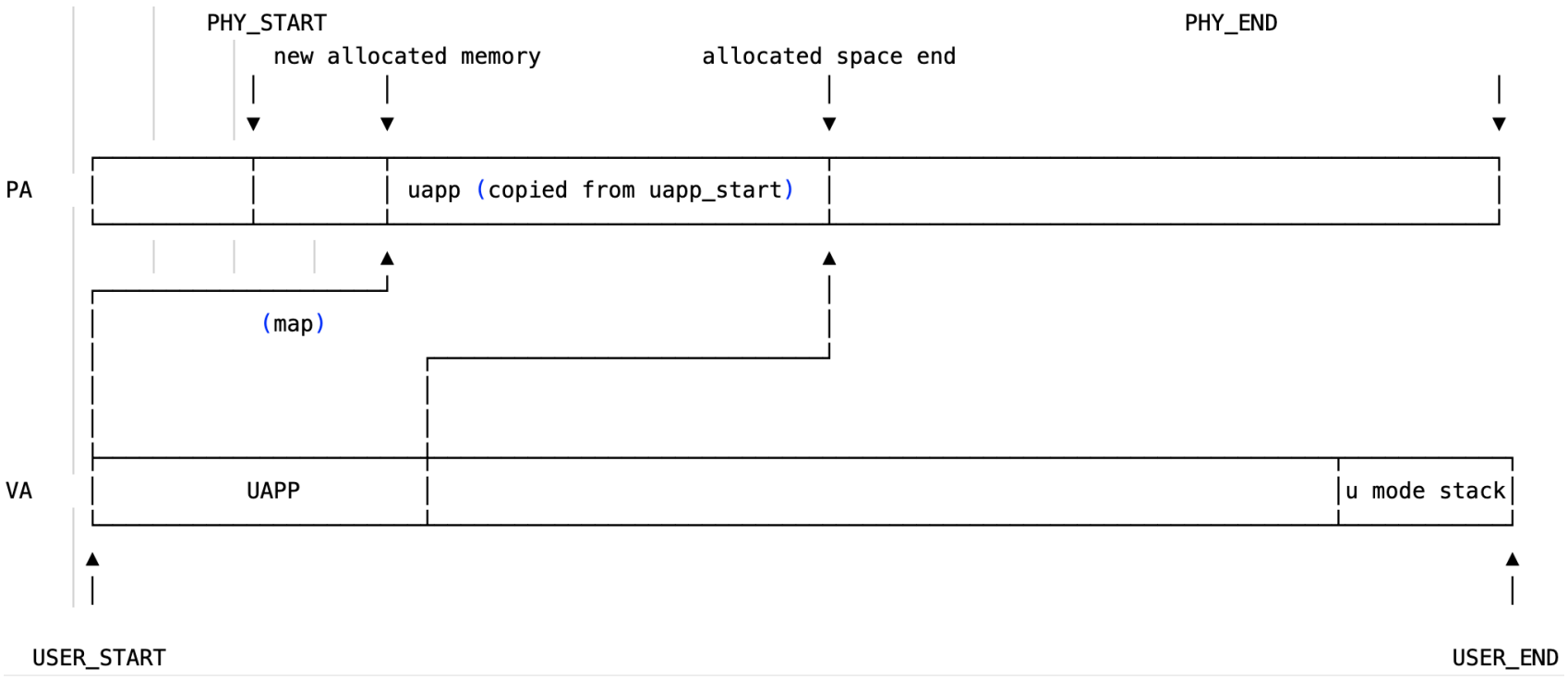
4.2 创建用户态进程

- 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;
};
```

> **Warning**： 经测试实现中并不需要 `thread_info` 这个成员，可以考虑不使用这个成员，或者将其删除，并更改 `switch_to` 中各个变量的偏移量，让我们的 OS 保持原来的行为。

- 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。
  - 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）。
  - 修改 __switch_to， 需要加入 保存/恢复 `sepc` `sstatus` `sscratch` 以及 切换页表的逻辑。
  - 在切换了页表之后，需要通过 `fence.i` 和 `vma.fence` 来刷新 TLB 和 ICache。

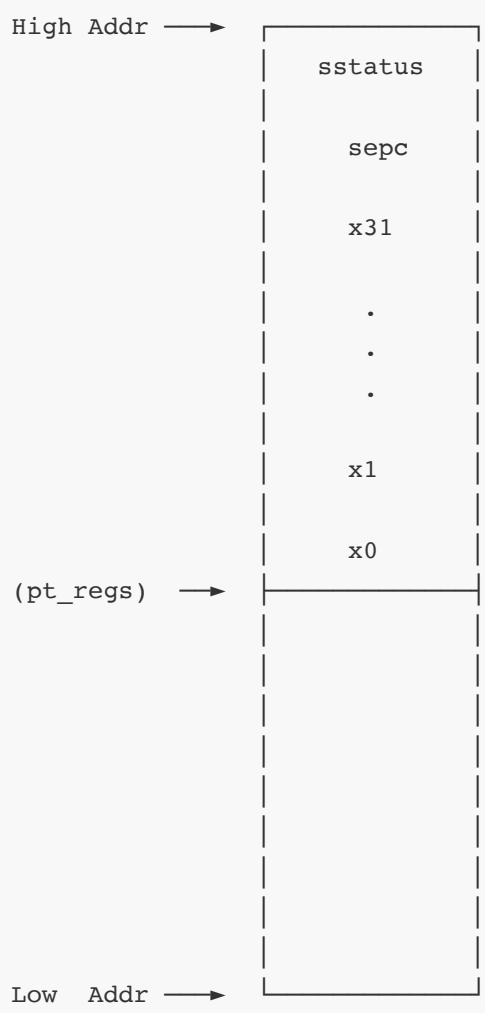


4.3 修改中断入口/返回逻辑 (_trap) 以及中断处理函数 （ trap_handler ）

- 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` ） 。
- 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。
- 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）**
- `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) {
    ...
}
```

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：



请同学自己补充 `struct pt_regs` 的定义， 以及在 `trap_handler` 中补充处理 SYSCALL 的逻辑。

4.4 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)。
- 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）。

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

4.6 测试纯二进制文件

- 由于加入了一些新的 .c 文件，可能需要修改一些Makefile文件，请同学自己尝试修改，使项目可以编译并运行。
- 输出示例

```
OpenSBI v0.9

      ____                   _____ ____ _____
     / __ \                 / ____|  _ \_   _|
    | |  | |_ __   ___ _ __ | (___ | |_) || |
    | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
    | |__| | |_) |  __/ | | |____) | |_) || |_
     \____/| .__/ \___|_| |_|_____/|____/_____|
           | |
           |_|

    ...

    Boot HART MIDELEG         : 0x0000000000000222
    Boot HART MEDELEG         : 0x000000000000b109

    ...mm_init done!
    ...proc_init done!
```

```
    [S-MODE] Hello RISC-V

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

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

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

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

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）
```

其中相对文件偏移Offset指出相应segment的内容从ELF文件的第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.

剩下的域的用法我们希望同学们通过阅读 `man elf` 命令和使用 Google（注意不是百度）来获取。大家可以参考以下的代码来将程序 load 进入内存。

```
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) {
            // copy the program section to another space
           // mapping the program section with corresponding size and flag
        }
    }
    // pc for the user program
    task->thread.sepc = ehdr->e_entry;
    // other task setting keep same
}
```

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

作业提交

同学需要提交实验报告以及整个工程代码。在提交前请使用 `make clean` 清除所有构建产物。