**浙江大学实验报告**

课程名称： 计操作系统 实验类型： 综合

实验项目名称： RV64 用户态程序

学生姓名： 专业： 计算机科学与技术 学号：

实验日期： 2023 年 11 月 17 日

1. **实验内容**

**记录实验过程并截图，对每一步的命令及结果进行必要的解释**

1. 实验目的

1.1 创建用户态进程，并设置 sstatus 来完成内核态转换至用户态。

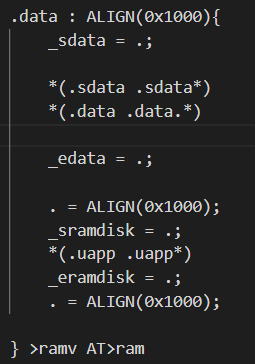
1.2 正确设置用户进程的用户态栈和内核态栈， 并在异常处理时正确切换。

1.3 补充异常处理逻辑，完成指定的系统调用（SYS\_WRITE, SYS\_GETPID）功能。

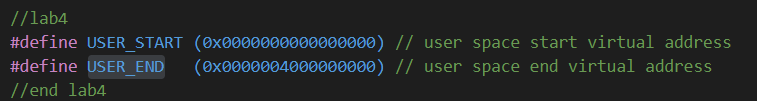
1. 准备工程

此次实验基于lab3所实现的代码进行。

因为我们需要实现用户态程序的执行，所以我们需要在vmlinux.lds文件中将用户态程序uapp加载至.data程序段。



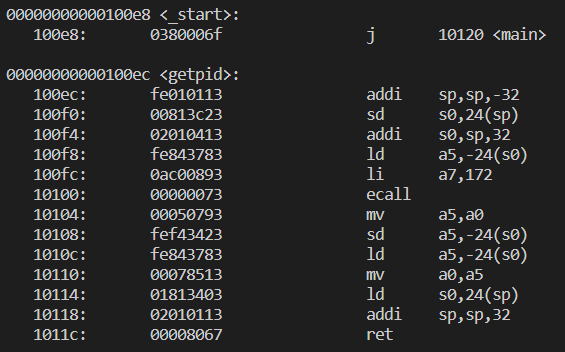
另外，我们还需要修改defs.h文件，向其中添加用户态程序调用的宏定义信息。



接着，我们从仓库中获取文件，并按一定的次序排布。另外，在本实验中，mm.c文件被修改，虽然kalloc()页面分配函数与free()页面释放函数依然兼容，但我们最好还是使用alloc\_pages(uint64\_t page\_cnt)多页面分配函数、allo\_page()单页面分配函数以及free\_pages(uint64\_t addr)已分配页面释放函数来介导内存的分配。

本次实验中，我们将用户态的程序与实现均放于user文件夹中管理。并依赖已有文件，生成uapp.o文件、uapp.elf文件、uapp.bin文件以及uapp文件 。 通过 objdump 我们可以看到 uapp 使用 ecall 来调用 SYSCALL 。从而将控制权交给处在 S-Mode 的操作系统，并由内核来处理相关异常。

Uapp文件的相关代码信息如下：

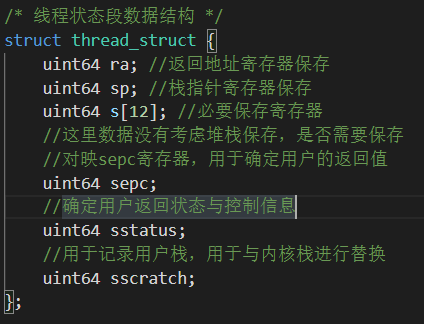


在本次实验中，我们首先需要将用户态程序 strip 成纯二进制文件来运行。这种情况下，用户程序运行的第一条指令应位于二进制文件的开始位置, 也就是uapp\_start 处的指令就是我们要执行的第一条指令。

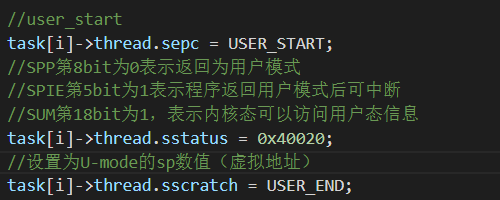
1. 创建用户态进程

本次实验要求建立4个用户态进程（不包含idle进程），因而我们只需要将 proc.h 文件中的NR\_TASKS 常量改为（1 + 4）即可。

创建用户态进程时需要要对 sepc、sstatus、sscratch 这三个CSR寄存器作设置，因而我们需要将其加入 thread\_struct 结构体中。

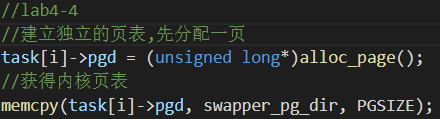
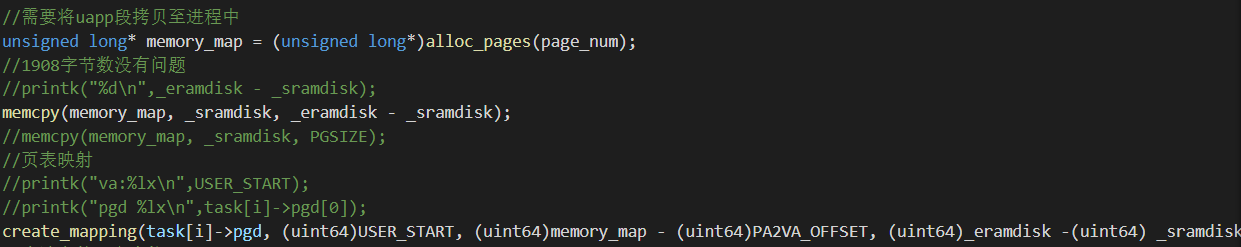


首先，我们需要修改task\_init函数。在task\_init函数中，我们需要为刚添加的三个成员属性进行赋值。其中，sepc用于记录用户程序开始的虚拟地址，sstatus记录一些控制信息，sscratch用于记录用户态程序的用户栈指定sp（这里均指向用户态虚拟内存的end地址）。

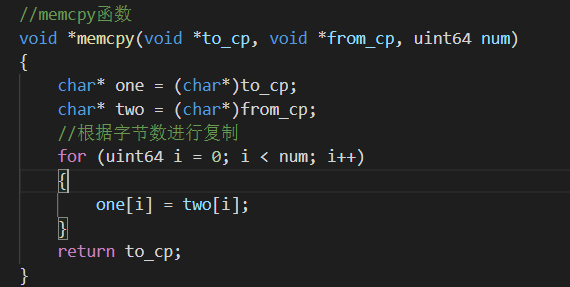
****

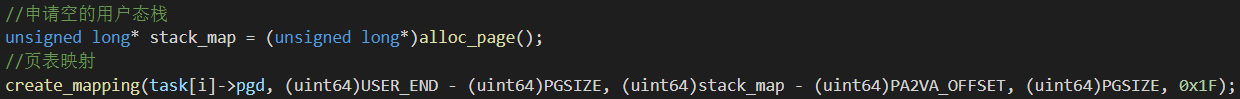
此外，每个用户态程序都需要创建属于其的页表。为了避免 U-Mode 和 S-Mode 切换时切换页表，我们将内核页表复制到每个进程的页表中。

另外，我们需要将uapp这一用户态程序所在的页面映射到每个进行的页表中。因为在程序运行过程中，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。所以，二进制文件需要先被拷贝至某个进程专用的一块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。

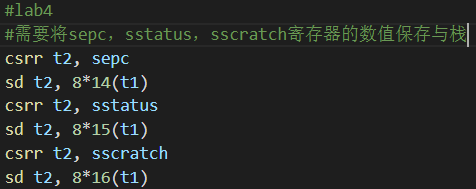
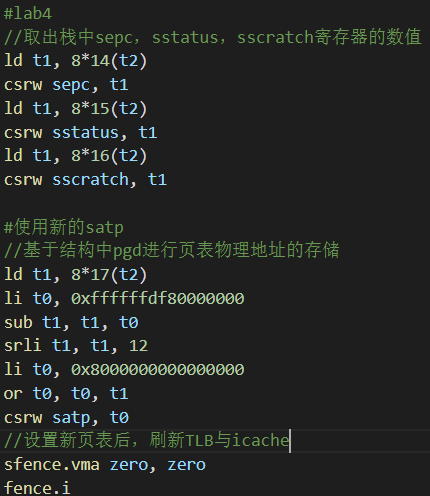


Create\_mapping函数已在lab3的代码中实现，在本实验中，为了实现页面信息的复制，我们还需要实现memcpy函数。Memcpy函数的实现如下，在此我们根据字节进行相关的复制操作。



在完成用户态程序的复制与映射操作后，我们需要为每个用户态进程分配一个用户栈。我们可以通过 alloc\_page() 函数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。

另外，我们还需要修改\_\_switch\_\_to函数，需要在该函数中加入保存与恢复 sepc、sstatus、sscratch这三个CSR寄存器的数值与信息，以及完成切换页表的逻辑操作（对satp寄存器进行更新）。

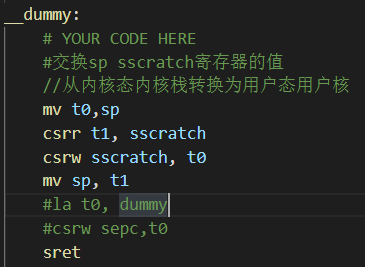


1. 修改中断入口、返回逻辑与中断处理函数

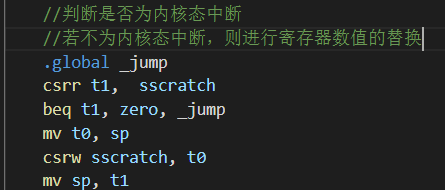
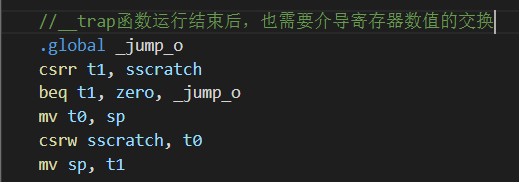
与ARM架构不同的是，本实验使用RISC-V架构只含有一个栈指针寄存器，因此操作系统在介导用户态与内核态的切换时，需要完成用户栈与内核栈的切换任务。

由于我们的用户态进程运行在U-Mode下， 使用的运行栈也是用户态用户栈， 因此当触发异常时，我们首先要对栈进行切换，切换至内核态内核栈。同理，我们在异常处理操作之后，从内核态返回用户态时，也需要进行栈切换的操作。

因而，我们需要修改 \_\_dummy函数。在本实验中，我们在初始化时，使用 thread\_struct.sp 结构属性保存了内核栈，使用thread\_struct.sscratch 结构属性保存了用户栈， 因此我们只需要在用户栈与内核栈的切换操作中交换对应寄存器的数值即可。



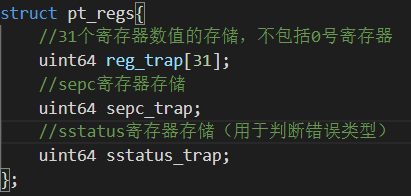
另外，我们也需要修改 \_trap中断处理引导函数。与前同理，在\_trap函数的首尾我们都需要做类似的寄存器交换数值操作。需要特比注意的是，如果在内核线程触发异常，我们则不需要进行栈的替换切换（一直使用内核栈）；另外，我们可以认为内核线程的栈指针永远指向的内核栈，而sscratch寄存器的数值为0。



因为uapp（用户态程序）使用ecall会产生ECALL\_FROM\_U\_MODE exception（数值表现为scause寄存器中数值为8）。因此我们需要在trap\_handler函数里面进行捕获。

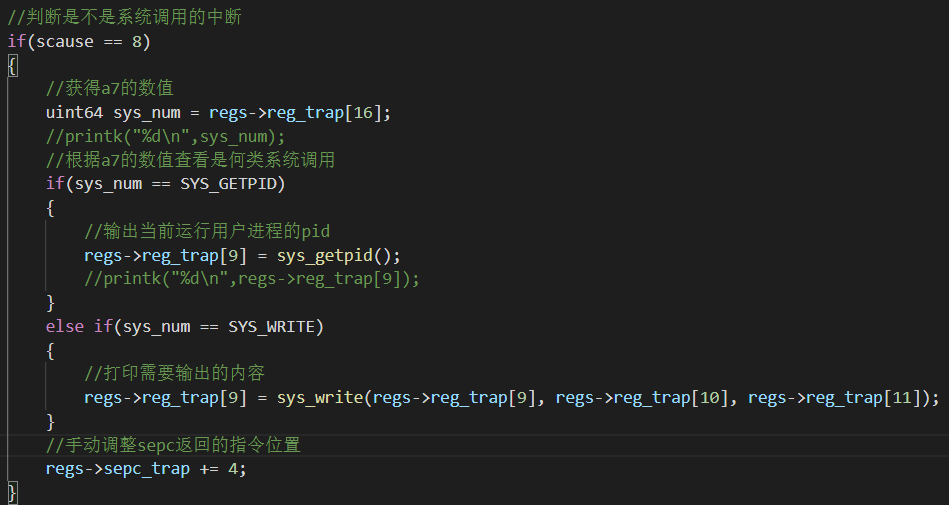
本实验中，我们对trap\_handler函数的参数进行了扩展，新增加了第三个参数regs。在\_trap函数中我们将寄存器的内容连续的保存在内核栈上， 因此我们可以将这一段看做一个pt\_regs结构体。进而我们可以通过参数传递的方式，在trap\_handler函数中使用这个结构体（栈段）。

结构体的定义如下：



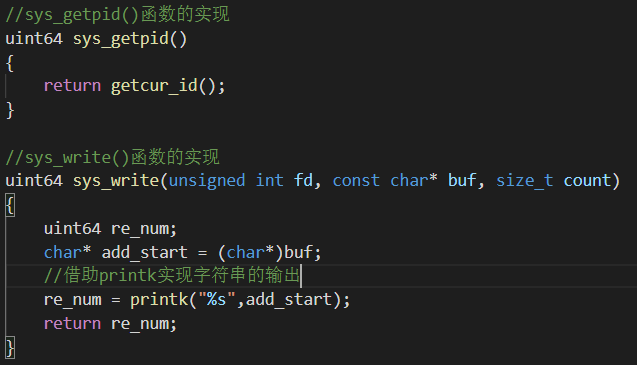
本次实验要求实现的系统调用函数仅有64号与172号系统调用。其中64号系统调用为sys\_write(unsigned int fd, const char\* buf, size\_t count)函数。该函数调用将用户态传递的字符串并打印到屏幕上，此处fd表示标准输出，buf为用户需要打印字符串的起始地址，count为字符串长度；该系统调用返回打印的字符数。而172号系统调用为sys\_getpid()函数。该调用从current中获取当前的pid并放入a0中返回，无输入参数。

在系统调用中，我们总是使用a0 - a5来传递参数，使用a7来记录系统调用号；而系统调用的返回值则往往被保存到 a0, a1寄存器中。



另外，针对系统调用这类异常，我们还需要手动将sepc + 4，sepc记录的是触发异常的指令地址，由于系统调用处理完成后，我们应该继续执行后续的指令，因此我们需要手动的修改sepc的地址，使sret指令进行合理的跳转。具体的代码实现见上图。

另外，我们需要添加syscall.c文件，来实现这两个系统调用。对于sys\_getpid()函数，我们直接获得当前运行进程的pid即可；对于sys\_write(unsigned int fd, const char\* buf, size\_t count)函数而言，我们可以借助printk的实现，来输出相应的字符串。



最后我们还需要修改head.S以及start\_kernel函数。在先前的实验中， 在 OS boot完成之后，我们还需要等待一个时间片，才会进行进程的调度。而在本实验中，我们需要更改为在OS boot完成之后立即调度uapp运行。这样在设计的过程中，就可以不用考虑idle进程的参数设计与赋值。

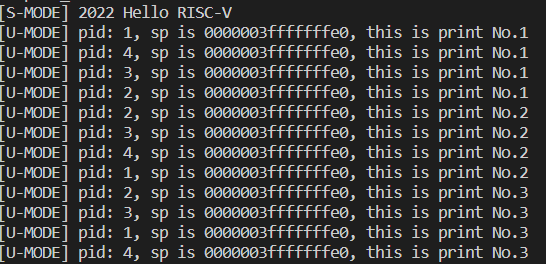
具体而言，我们只需要在start\_kernel中将调用schedule()的位置放置于 test()之前即可。

另外，我们还需要将head.S中enable interrupt sstatus.SIE逻辑注释，确保schedule()函数的执行过程不受（时钟）中断影响。

完成以上操作后，我们的实验也就设计成功。

使用make run语句运行该程序，我们可以得到一下结果。

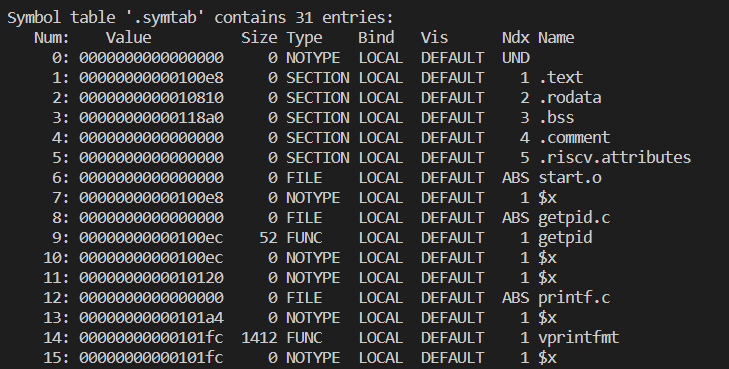
首先在内核态进行操作系统的启动，接着进入用户态进行用户态程序的运行。其中，我们从输出的语句中可以看出，四个用户进程根据SJF的时间片长短进行着CPU使用的调度。且最后的Number数说明了设置第几次时间片重新复制后的调度循环。另外，pid的正确输出，也证明了syscall()函数的正确实现。我们还可以发现，所有用户态进程的指针，都指向了同一个地址位置。因为虽然每个进程具体使用的内存地址不同，但其在虚拟地址处对映的用户栈位置却保持着一致。



5.ELF格式程序装入

ELF是当今被广泛使用的应用程序格式。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。在本实验完成了上述的设计后，我们可以尝试将二进制文件的载入更改为ELF文件的使用与载入。

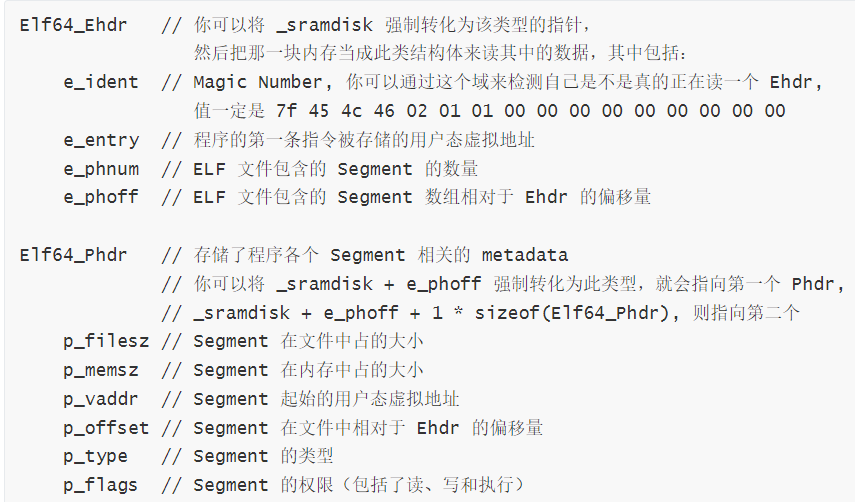
ELF 文件中包含了将程序加载到内存所需的信息。当我们通过 readelf 来查看一个 ELF 可执行文件的时候，我们可以读到被包含在ELF Header中的信息。Uapp文件的部分信息如下：



可见，ELF header中包含了两种将程序分块的粒度，Segment和 Section，我们将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。

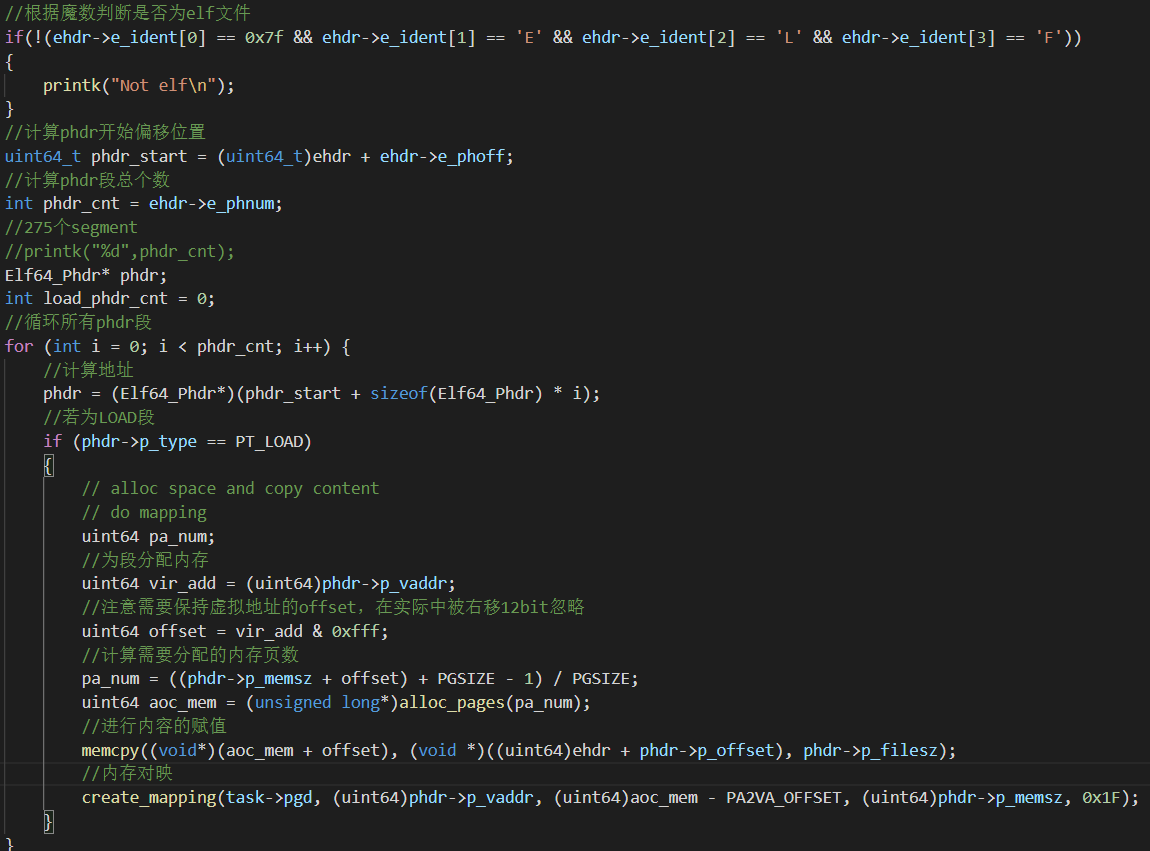
因此，我们可以设计load\_program(struct task\_struct\* task)函数来介导ELF文件初始化载入的完成。

首先，我们需要介绍ELF文件的文件结构，这便于我们对有效信息的提取。

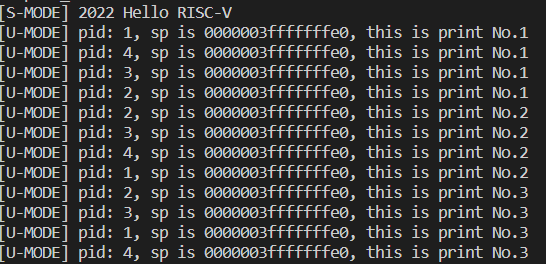


其中相对文件偏移p\_offset指出相应segment的内容从ELF文件的第p\_offset字节开始,在文件中的大小为p\_filesz,它需要被分配到以p\_vaddr为首地址的虚拟内存位置,且在该内存中它占用大小为p\_memsz。也就是说,这个 segment使用的内存就是[p\_vaddr, p\_vaddr + p\_memsz)这一连续区间,然后我们需要将segment的内容从ELF文件中读入到这一内存区间,并将[p\_vaddr + p\_filesz, p\_vaddr + p\_memsz)对应的物理区间清零。清零操作已在初始化分配页面是完成。

另外，Elf64\_Ehdr\* ehdr = (Elf64\_Ehdr\*)\_sramdisk（可以使用强行的结构体转化操作），从地址\_sramdisk开始，便是我们要找的Ehdr.Elf64\_Phdr\* phdrs = (Elf64\_Phdr\*)(\_sramdisk + ehdr->phoff)，这是一个 Phdr 数组，其中的每个元素都是一个 Elf64\_Phdr。而phdrs[ehdr->phnum - 1]是表示最后一个 Phdr。phdrs[0].p\_type == PT\_LOAD，可以用来判断这个 Segment 的类型是 不是LOAD，进而决定是否需要在初始化时将其加载进内存。(void\*)(\_sramdisk + phdrs[1].p\_offset)将会指向第二个段中的内容的开头。函数的具体实现主题部分如下：



最后，我们make run运行该程序，也可以得到如下的结果。说明我们对ELF格式文件的载入成功。



1. **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）**

在本实验中，我们使用的用户态线程和内核态线程的对应关系应该是一对一的。因为我们首先建立了内核态线程，并分别为其申请了页表、用户栈以及装载用户程序的页面，接着一一将sepc设置为用户程序的起始地址，这可以说明对应关系时一对一的。

另外，只有在用户程序中我们才输出“[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1”语句，而输出语句时对应的不同pid进程（线程）号，也说明对应关系是一对一的。

**2.为什么 Phdr 中，p\_filesz 和 p\_memsz 是不一样大的？**

p\_filesz 和 p\_memsz不相等，可能是因为在可执行文件中，某些段可能包含了一些在运行时生成或计算的数据。即这些数据本身在文件中是不存在的，但在程序运行的过程中却需要为其分配内存空间来进行存储。

例如，对于一个包含临时变量的段而言，这些变量在编译时是未被初始化的，因此在可执行文件中该段的大小可能只包含这些变量的定义和以及一些初始化数据，而在程序运行时，操作系统会为这些变量在内存中分配实际的空间，所以 p\_memsz往往会大于p\_filesz。

1. **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？**

因为使用了虚拟地址的技术，页表不同，虚拟地址对映的物理地址不同，因而多个进程可以使用同一个栈虚拟地址，而其数据却不会相互干扰。虽然进程看到的虚拟地址是一样的，但因为他们的页表不同，他们获得的物理地址是不同的，而实际操作是对物理地址进行的，因而操作的数据不会互相干扰。

一般而言，用户可以通过自己的页表，将虚拟地址转化为物理地址，从而得知自己栈所在的物理地址。

1. **讨论、心得**

写完后复盘本实验，发现实验思路其实十分明了，设计也很合理，但自己还是因为这样那样的错误，导致写本lab花费了大量的时间。这里也便列举一些需要注意的point。

1. 框架并没有实现memcpy函数，memcpy函数需要自己进行正确的设计。在设计memcpy的过程中，我不聪明的使用unsigned long作为复制最小单位，进而引发了很多问题。一方面，第三个参数不能输入为字节数；另一方面，若不使用字节数，必然会带来除以8以统一unsigned long位数的操作，此时便会存在部分内容复制时丢失的情况（非8倍数，整除丢失）。
2. 在satp设置的过程中，因为我们进入的是用户模式下的用户态程序，因而对于flags而言，我们需要把相应的U位也置位为1，即第4位也要置位为1，这样才能正常的进行页表的映射与开启。

历经波折的实验，完结撒花。