

**操作系统原理课程设计报告**

|  |  |
| --- | --- |
| 姓 名： | 邹雅 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | ACM1901 |
| 学 号： | U201915035 |
| 指导教师： | 张杰 |
|  |  |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2022年 3月 14日

目 录

[实验一 1](#_Toc100306717)

[1.1 lab1\_1 系统调用 1](#_Toc100306718)

[1.2 lab1\_2 异常处理 2](#_Toc100306719)

[1.3 lab1\_3 （外部）中断 3](#_Toc100306720)

[实验二 5](#_Toc100306721)

[2.1 lab2\_1 虚实地址转换 5](#_Toc100306722)

[2.2 lab2\_2 简单内存分配和回收 6](#_Toc100306723)

[2.3 lab2\_3 缺页异常 7](#_Toc100306724)

[1.4 lab2\_challenge1 复杂缺页异常 9](#_Toc100306725)

[实验三 11](#_Toc100306726)

[3.1 lab3\_1 进程创建 11](#_Toc100306727)

[3.2 lab3\_2 进程yield 12](#_Toc100306728)

[3.3 lab3\_3 循环轮转调度 13](#_Toc100306729)

[3.4 lab3\_challenge1 进程等待和数据段复制 14](#_Toc100306730)

[3.5 lab3\_challenge2 实现信号量 17](#_Toc100306731)

# 实验一

## lab1\_1 系统调用

1. 实验目的

实现应用app\_helloworld.c ，应用实现的功能非常简单，即在屏幕上打印"Hello world!\n"。

1. 实验内容

通过观察我们可以发现app\_helloworld.c文件中有两个函数调用printu和exit，而这两个函数都在user/user\_lib.c中进行了实现。继续查看可以发现这两个函数最后都转换成对do\_user\_call的调用。

跟踪一系列的函数调用，我们可以发现do\_user\_call函数最终处理的执行的函数在handle\_syscall() 函数中。

在handle\_syscall() 函数中，第一条语句是tf->epc += 4 ，因为在系统调用完成之后程序返回需要继续执行下一句语句。下一条 panic 语句，可以看到是对lab1\_1的提示信息，我们只需要在此完善即可。我们应该调用do\_syscall()函数，完成进一步系统调用功能。

do\_syscall()函数有8个参数，这些参数保存在a0-a7寄存器中，而我们可以看到函数中使用了前三个参数，而后5个寄存器内容中都为0。

6 **typedef** **struct** trapframe {

7 // space to store context (all common registers)

8 /\* offset:0 \*/ riscv\_regs regs;

9

10 // process's "user kernel" stack

11 /\* offset:248 \*/ uint64 kernel\_sp;

12 // pointer to smode\_trap\_handler

13 /\* offset:256 \*/ uint64 kernel\_trap;

14 // saved user process counter

15 /\* offset:264 \*/ uint64 epc;

16 }trapframe;

在handle\_syscall() 函数中，tf是一个trapframe结构，我们可以发现其中有一个属性是regs，寄存器的值保存在其中。

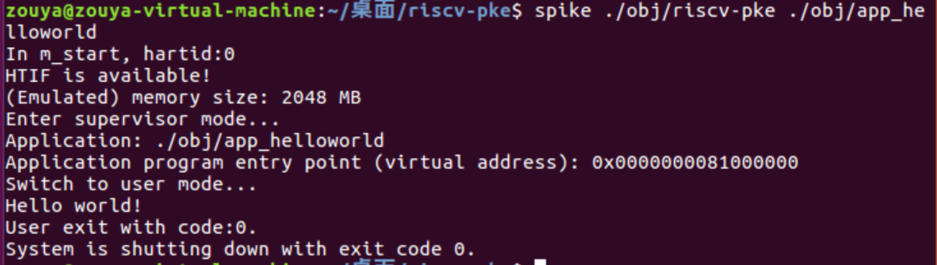
另外，我们还需将函数返回值赋给a0寄存器进行保存。所以需要修改添加的代码如下：

long a0\_return = do\_syscall(tf->regs.a0, tf->regs.a1, tf->regs.a2, 0, 0, 0, 0, 0);

tf->regs.a0 = a0\_return;

1. 实验调试及心得

完成结果如下所示：

****

**图1.1**

通过该实验可以了解和掌握操作系统中系统调用机制的实现原理。通过明确 do\_syscall 函数的参数以及返回值的使用，可以知道明确 trapframe 结构中通用寄存器的使用。

## lab1\_2 异常处理

1. 实验目的

在用户U模式下执行的应用企图执行RISC-V的特权指令csrw sscratch, 0。该指令会修改S模式的栈指针，如果允许该指令的执行，执行的结果可能会导致系统崩溃。在本实验中需要通过调用handle\_illegal\_instruction函数完成异常指令处理，阻止app\_illegal\_instruction的执行。

1. 实验内容

执行不能在用户模式运行的特权级指令这类异常属于非法指令异常，即CAUSE\_ILLEGAL\_INSTRUCTION，它对应的异常码是02。观察发现，在delegate\_traps()函数没有处理的CAUSE\_ILLEGAL\_INSTRUCTION异常，这说明该异常的处理还是交给M模式来处理。我们需要了解M模式的trap处理入口，以便继续跟踪其后的处理过程。

M模式的trap处理会调用handle\_mtrap()函数。可以看到，handle\_mtrap()函数对在M态处理的多项异常都进行了处理，处理的方式几乎全部是调用panic函数，让机器停机。对于CAUSE\_ILLEGAL\_INSTRUCTION尚未处理。

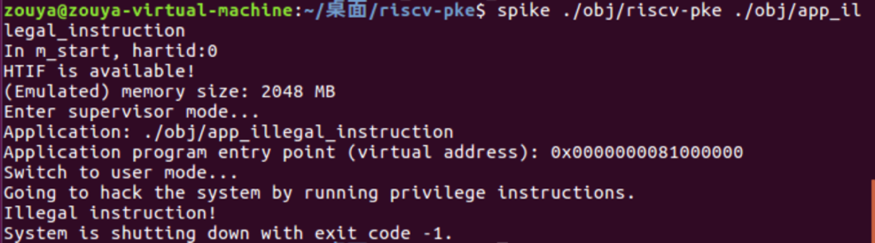
所以这里添加上对handle\_illegal\_instruction()函数的调用，来完成lab1\_2。

**case** CAUSE\_ILLEGAL\_INSTRUCTION:

handle\_illegal\_instruction()

1. 实验调试及心得

完成结果如下所示：



**图1.2**

通过完成本实验可以了解和掌握操作系统中异常（exception）的产生原理以及处理的原则。我们可以了解到异常可以在M模式和S模式处理，在M模式中处理异常通常是编写对应处理函数并调用。

## lab1\_3 （外部）中断

1. 实验目的

运行一个应用程序。该程序逻辑包含一个长度为100000000次的循环，循环每次将整型变量i加一，当i的值是5000000的整数倍时，输出"wait i的值\n"。程序并不是从开始运行到最终结束的，而是在运行过程中受到了系统的外部时钟中断（timer irq）的“干扰”。

而在这个实验中给出的PKE操作系统内核，在时钟中断部分并未完全做好，导致（模拟）RISC-V机器碰到第一个时钟中断后就会出现崩溃。我们需要完成PKE操作系统内核未完成的时钟中断处理过程，使得它能够完整地处理时钟中断。

1. 实验内容

在本实验中，我们接触的时钟中断也是在机器模式下触发的。为了处理时钟中断，我们的PKE代码在lab1\_2的基础上，在m\_start函数中新增了timerinit()函数。在该函数中，设置了下一次timer触发的时间，即当前时间的TIMER\_INTERVAL（即1000000周期）之后。另外，还设置了MIE寄存器中的MIE\_MTIE位，即允许我们的（模拟）RISC-V机器在M模式处理timer中断。

时钟中断触发后， mtrapvec函数将被调用。和lab1\_2一样，最终将进入handle\_mtrap函数继续处理。handle\_mtrap函数将通过对mcause寄存器的值进行判断，确认是时钟中断（CAUSE\_MTIMER）后，将调用handle\_timer()函数进行进一步处理。

而handle\_timer()函数会先设置下一次timer（再次）触发的时间为当前时间+TIMER\_INTERVAL，并对SIP（即S模式的中断等待寄存器）寄存器进行设置，将其中的SIP\_SSIP位进行设置，完成后返回。至此，时钟中断在M态的处理就结束了，剩下的动作交给S态继续处理。而handle\_timer()对SIP寄存器设置，会导致PKE操作系统内核在S模式收到一个来自M态的时钟中断请求（CAUSE\_MTIMER\_S\_TRAP）。

而在handle\_mtimer\_trap()函数需要完成对时钟中断对处理。首先，我们看到在该函数上面定义了一个全局变量g\_ticks，用它来对时钟中断的次数进行计数并输出。为了确保我们的系统持续正常运行，该计数应每次都会完成加一操作。所以handle\_mtimer\_trap()首先需要对g\_ticks进行加一。其次，由于处理完中断后，SIP寄存器中的SIP\_SSIP位仍然为1，如果该位持续为1的话会导致我们的模拟RISC-V机器始终处于中断状态。所以，handle\_mtimer\_trap()还需要对SIP的SIP\_SSIP位清零，以保证下次再发生时钟中断时，M态的函数将该位置一会导致S模式的下一次中断。

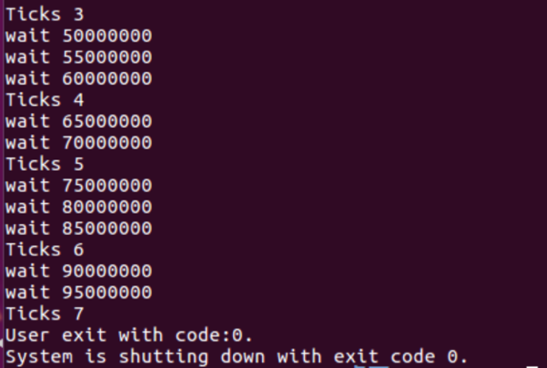
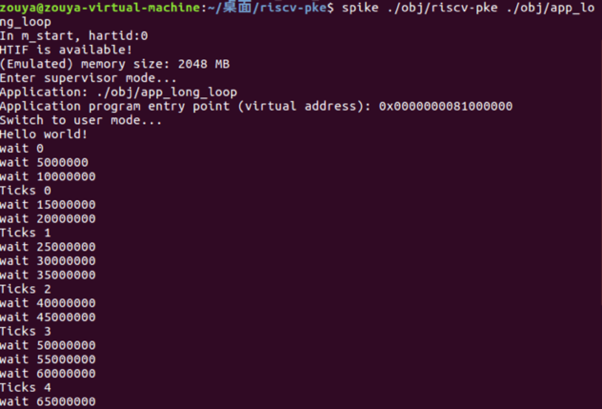
完成以下代码后，可以处理完成时钟中断。

g\_ticks++;

write\_csr(sip,0L);

1. 实验调试及心得

完成结果如下所示：



**图1.3**

通过本实验我们知道，一个进程运行固定时间后会发生时钟中断。时钟中断是通过特定的寄存器（MIE、SIP）来保存值进行的，而在时钟中断处理结束需要清空值。

# 实验二

## lab2\_1 虚实地址转换

1. 实验目的

本实验的应用app\_helloworld\_no\_lds代码跟lab1\_1一样，打印出“Hello world!\n”。但是，不同的地方在于，它的编译和链接并未指定程序中符号的逻辑地址。因此我们需要在本实验中完成给定逻辑地址到物理地址的转换。

1. 实验内容

我们的应用并未如愿地打印出“Hello world!\n”，这是因为printu("Hello world!\n");中的“Hello world!\n”字符串本质上是存储在.rodata段，它被和代码段（.text）一起被装入内存。从逻辑地址结构来看，它的逻辑地址就应该位于“用户代码段”，显然低于0x80000000。

而printu函数是一个典型的系统调用，它的执行逻辑是通过ecall指令，陷入到内核完成到屏幕的输出。对于内核而言，显然不能继续使用“Hello world!\n”的逻辑地址对它进行访问，而必须将其转换成物理地址。

因此我们需要实现user\_va\_to\_pa()函数，完成给定逻辑地址到物理地址的转换。

应用的printu函数调用发生之后，最终会来到sys\_user\_print()函数。该函数最终通过调用sprint将结果输出，但是在输出前，需要将buf地址转换为物理地址传递给sprint，这一转换是通过user\_va\_to\_pa()函数完成的。

转换逻辑地址到物理地址的过程如下所述。为了在page\_dir所指向的页表中查找逻辑地址va，就必须通过调用页表操作相关函数找到包含va的页表项（PTE），通过该PTE的内容得知va所在的物理页面的首地址，最后再通过计算va在页内的位移得到va最终对应的物理地址。

最终实现的代码如下所示：

uint64 va\_int = (uint64)va;

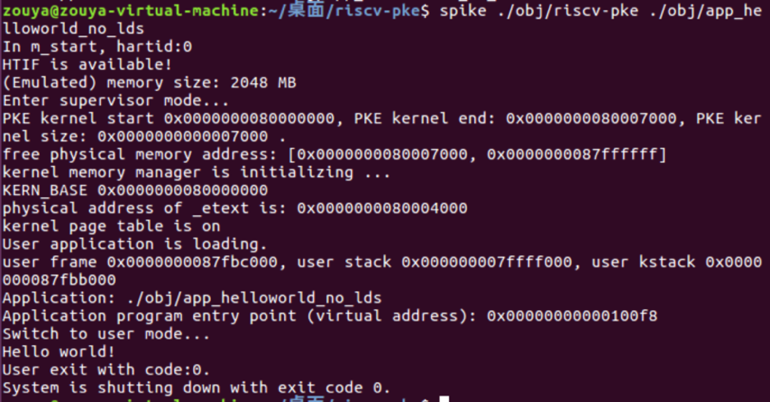
uint64 pa\_start = lookup\_pa(page\_dir,va\_int);

uint64 pa = pa\_start + (va\_int&((1<<PGSHIFT)-1));

return (void\*)pa;

1. 实验调试及心得

实验结果最终如下所示：



**图2.1**

本实验中我们完成了逻辑地址到物理地址到转换函数。可见转换逻辑其实不难，找到虚拟地址对应的起始物理地址，再把起始物理地址加上虚拟地址的偏移量即可得到最终的物理地址。

## lab2\_2 简单内存分配和回收

1. 实验目的

本实验应用的逻辑如下所述：首先分配一个内存页面来存放my\_structure结构，然后往my\_structure结构的实例中存储信息，打印信息，并最终将之前所分配的空间释放掉。

本实验中新定义了两个用户态函数naive\_malloc()和naive\_free()，它们最终会转换成系统调用，完成内存的分配和回收操作。

从输出结果来看分配内存已经做好，且打印出了我们预期的结果。但是，naive\_free对应的功能并未完全做好。因此在本实验中我们需要完成naive\_free对应的功能，并获得预期的结果输出。

1. 实验内容

一般来说，应用程序执行过程中的动态内存分配和回收，是操作系统中的堆管理的内容。在本实验中，我们是为PKE操作系统内核实现一个“堆”。

为实现naive\_free()的内存回收过程，我们需要了解其内存是如何分配给应用程序，并提供给应用程序使用的。我们需要阅读函数sys\_user\_allocate\_page()。

这个函数先分配了一个首地址为pa的物理页面，其次给出了pa对应的逻辑地址va = g\_ufree\_page，并对g\_ufree\_page进行了递增操作。最后将pa物理页面映射给了应用程序va地址进行使用。

以上了解了内存的分配过程后，我们就能够了解其反过程的回收应该怎么做了，大概分为以下步骤：

* 找到一个给定va所对应的页表项PTE；
* 如果找到，通过该PTE的内容得知va所对应物理页的首地址pa；
* 回收pa对应的物理页，并将PTE中的Valid位置为0。

通过以上分析，我们可以得到补充的代码如下所示：

if(free){

pte\_t\* pte\_free = page\_walk(page\_dir,va,0);

if(pte\_free){

free\_page((void\*)PTE2PA(\*pte\_free));

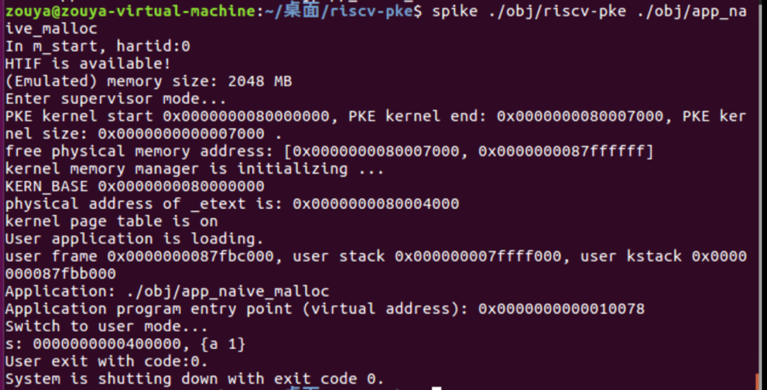
\*pte\_free = \*pte\_free & (~PTE\_V);

}

}

1. 实验调试及心得

实验结果如下所示：



**图2.2**

在本实验中，我们模拟实现了应用程序使用“堆”的过程，分配内存以及回收内存。

## lab2\_3 缺页异常

1. 实验目的

本实验中关注以下问题：给定一个递增的等差数列：0, 1, 2, ..., n，如何求该数列的和？

本实验应用给出了它的递归（recursive）解法。通过定义一个函数sum\_sequence(n)，将求和问题转换为sum\_sequence(n-1) + n的问题。问题中n依次递减，直至为0时令sum\_sequence(0)=0。

而此时未完成的结果直接运行回出现handle\_page\_fault错误。

因此本实验中我们需要在PKE操作系统内核中完善用户态栈空间的管理，使得它能够正确处理用户进程的“压栈”请求。

1. 实验内容

函数递归调用时，函数调用的路径会被完整地保存在栈中，也就是说函数的下一次调用会将上次一调用的现场压栈。显然，在以上计算等差数列的和的程序中，n值给得越大，就会导致越深的栈，而栈越深需要的内存空间也就越多。

通过中对用户进程逻辑地址空间的讨论，我们知道应用程序最开始被载入时，它的用户态栈空间仅有1个4KB的页面。显然，只要以上的程序给出的n值“足够”大，就一定会“压爆”用户态栈。而以上运行结果中，出问题的地方（即handle\_page\_fault后出现的地址，0x7fffdff8）也恰恰在用户态栈所对应的空间。

以上分析表明，之所以运行程序会出现错误（handle\_page\_fault），是因为给sum\_sequence()函数的n值太大，把用户态栈“压爆”了。

在此，我们处理的是缺页异常，不能也不应该将应用进程杀死。正确的做法是：发生缺页异常时，判断发生缺页的是不是用户栈空间，如果是则分配一个物理页空间，最后将该空间通过vm\_map映射到用户栈上以扩充用户栈空间。

我们知道产生缺页异常的本质还是应用往未被映射的内存空间“写”以及访问所导致的，所以CAUSE\_STORE\_PAGE\_FAULT是我们应该关注的异常。

通过阅读delegate\_traps()函数，我们看到该函数显然已将CAUSE\_STORE\_PAGE\_FAULT代理给了S模式，所以，接下来我们就应阅读kernel/strap.c文件中对于这类异常的处理。

在handle\_user\_page\_fault这个函数里，我们找到了之前运行程序出错的地方，我们只需要改正这一错误实现缺页处理，使得程序获得正确的输出就好。

实现缺页处理的思路如下：

* 输入的参数stval存放的是发生缺页异常时，程序想要访问的逻辑地址。通过该参数我们可以判断缺页的逻辑地址在用户进程逻辑地址空间中的位置。在本实验中不必实现此判断，默认逻辑地址合法。
* 分配一个物理页，将所分配的物理页面映射到stval所对应的虚拟地址上。

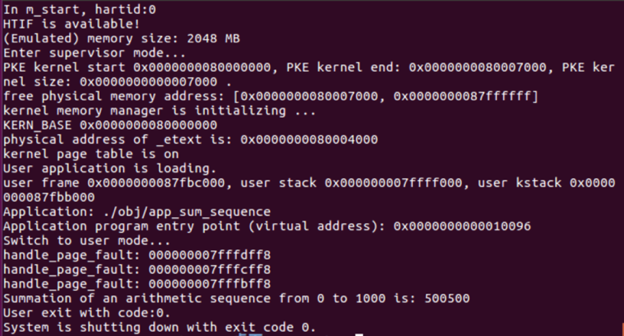
根据以上思路，我们可以得到补充的代码如下所示：

stval = (stval>>PGSHIFT)<<PGSHIFT;

user\_vm\_map((pagetable\_t)current->pagetable,stval,PGSIZE,(uint64)alloc\_page(),prot\_to\_type(PROT\_WRITE | PROT\_READ, 1));

1. 实验调试及心得

实验结果如下所示：



**图2.3**

在本实验中，我们了解到，由于栈空间的限制，如果不断地调用递归函数，可能会使栈中需要保存的内容过多而超过栈的空间。因此我们需要判断缺页异常是否是由于栈空间不足引起的，并进行继续分配栈空间。

## lab2\_challenge1 复杂缺页异常

1. 实验目的

该实验逻辑思路与 lab2\_3 一致，不一致的在于需要将每一步递归的结果保存在数组ans中。

创建数组时，我们使用了当前的malloc函数申请了一个页面（4KB）的大小，对应可以存储的个数上限为1024。在函数调用时，我们试图计算1025求和，首先由于n足够大，所以在函数递归执行时会触发用户栈的缺页；其次，1025在最后一次计算时会访问数组越界地址，由于该处虚拟地址尚未有对应的物理地址映射，因此属于非法地址的访问，这是不被允许的，对于这种缺页异常，应该提示用户并退出程序执行。

根据预期输出结果可以看出：前面的缺页异常是由于函数递归调用引起的，而最后一个缺页是对动态申请的数组进行越界访问造成的，访问非法地址，程序报错并退出。

在本实验中需要通过修改PKE内核，使得对于不同情况的缺页异常进行不同的处理。

1. 实验内容

在本实验中，我们需要修改kernel/strap.c中的异常处理函数。对于合理的缺页异常，扩大内核栈大小并为其映射物理块；对于非法地址缺页，报错并退出程序。

在 lab2\_3 中，我们已经处理了由于栈空间不足而产生的缺页异常。同样在 kernel/strap.c 中的handle\_user\_page\_fault函数中，在 lab2\_3 中我们默认了逻辑地址合法。而在本实验中则不可以默认，我们需要对逻辑地址进行判断。通过判断stval参数是不是比USER\_STACK\_TOP小，且比我们预设的可能的用户栈的最小栈底指针要大（这里，我们可以给用户栈空间一个上限，例如20个4KB的页面），若满足，则为合法的逻辑地址。当逻辑地址不合法时，程序报错且退出。

对 CAUSE\_STORE\_PAGE\_FAULT 类型异常处理代码如下所示：

if((stval < USER\_STACK\_TOP) && (stval > (USER\_STACK\_TOP - 20\*PGSIZE))){

stval = (stval>>PGSHIFT)<<PGSHIFT;

user\_vm\_map((pagetable\_t)current->pagetable,stval,PGSIZE,(uint64)alloc\_page(),prot\_to\_type(PROT\_WRITE | PROT\_READ, 1));

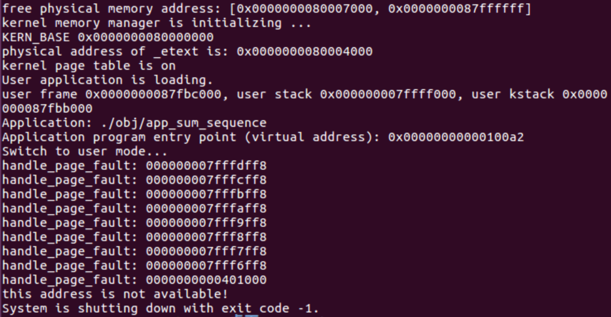
} else {

panic("this address is not available!");

}

1. 实验调试及心得

实验结果如下所示：



**图2.4**

在本实验中我们知道了如何处理数组访问越界。

# 实验三

## lab3\_1 进程创建

1. 实验目的

主进程调用fork()函数，后者产生一个系统调用，基于主进程这个模板创建它的子进程。我们需要在PKE操作系统内核中实现子进程到父进程代码段的映射，以最终完成fork函数。

1. 实验内容

从应用程序（user/app\_naive\_fork.c）开始，跟踪fork()函数的实现路径。直至来到kernel/process.c文件中的do\_fork()函数。

该函数使用循环来拷贝父进程的逻辑地址空间到其子进程。我们看到，对于trapframe段（case CONTEXT\_SEGMENT）以及堆栈段（case CODE\_SEGMENT），do\_fork()函数采用了简单复制的办法来拷贝父进程的这两个段到子进程中，这样做的目的是将父进程的执行现场传递给子进程。

然而，对于父进程的代码段，为了减少系统开销我们不应直接复制，而应通过映射的办法，将子进程中对应的逻辑地址空间映射到其父进程中装载代码段的物理页面。注意对页面的权限设置应为可读和可执行。

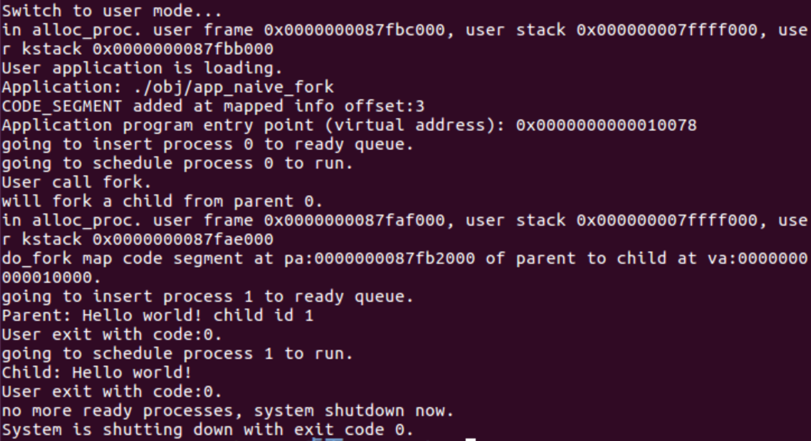
注意到在预期输出中，有一句预期输出，如下第一行代码所示。而pa、va采用了实际转换时的输出。之后，采用user\_vm\_map函数把子进程中对应的逻辑地址空间映射到其父进程中装载代码段的物理页面。需要补充的代码如下所示：

sprint("do\_fork map code segment at pa:%llx of parent to child at va:%llx.\n",lookup\_pa(parent->pagetable,parent->mapped\_info[i].va),parent->mapped\_info[i].va);

user\_vm\_map(child->pagetable,parent->mapped\_info[i].va,PGSIZE,lookup\_pa(parent->pagetable,parent->mapped\_info[i].va),prot\_to\_type(PROT\_WRITE | PROT\_READ | PROT\_EXEC,1));

1. 实验调试及心得

完成结果如下所示：



**图3.1**

在本实验中完成了对子进程的fork。我们可以知道fork子进程需要把父进程对应段的内容拷贝或映射到新的地址空间。

## lab3\_2 进程yield

1. 实验目的

3-2的应用程序通过fork系统调用创建了一个子进程，且父进程和子进程都进入了一个很长的循环。在循环中，无论是父进程还是子进程，在循环的次数是10000的整数倍时，除了打印信息外都调用了yield()函数，来释放自己的执行权。PKE操作系统中的yield()功能未完善，导致应用无法正常执行下去。因此本实验中需要完善yield()函数。

1. 实验内容

找到kernel/syscall.c文件中的do\_syscall()函数中，系统调用类型为SYS\_user\_yield时回调用同文件的sys\_user\_yield() 函数。在该函数中我们进行完善。

进程释放CPU的动作应该是：

* 将当前进程置为就绪状态（READY）；
* 将当前进程加入到就绪队列的队尾；
* 转进程调度。

由此可知应该补充的代码如下所示：

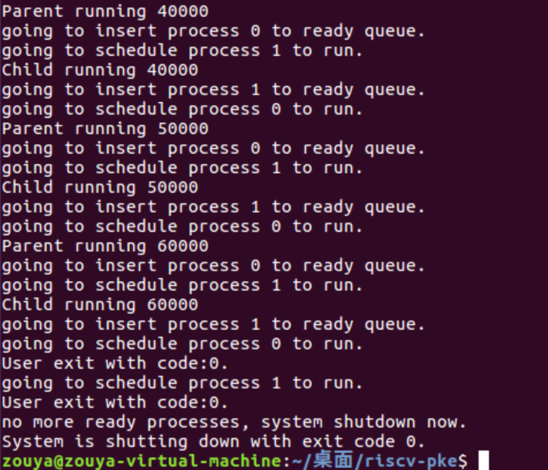
current->status = READY;

insert\_to\_ready\_queue(current);

schedule();

1. 实验调试及心得

完成结果如下所示：



**图3.2**

在本实验中，我们了解了使当前进程释放CPU是将当前进程状态转为就绪态，并且加入就绪队列之后再次进行进程调度。

## lab3\_3 循环轮转调度

1. 实验目的

lab3\_3给出的应用仍然是父子两个进程，同样执行两个大循环。但与lab3\_2不同的是，这两个进程在执行各自循环体时，没有主动释放CPU。显然，这样的设计会导致某个进程长期占据CPU，而另一个进程无法得到执行。

注意到进程的执行体很长时，执行过程中时钟中断被触发。由此我们可以通过利用时钟中断来实现进程的循环轮转调度，进而可以避免由于一个进程的执行体过长，导致系统中其他进程无法得到调度的问题。

在本实验中需要实现kernel/strap.c文件中的rrsched()函数。

1. 实验内容

如果单纯为了实现进程的轮转，避免单个进程长期霸占CPU的情况，只需要简单地在时钟中断被触发时做重新调度即可。然而，为了实现时间片的概念，以及控制进程在单时间片内获得的执行长度，我们在kernel/sched.h文件中定义了“时间片”的长度：

6 //length of a time slice, in number of ticks

7 #define TIME\_SLICE\_LEN 2

可以看到时间片的长度（TIME\_SLICE\_LEN）为2个ticks，这就意味着我们要每隔两个ticks触发一次进程重新调度动作。

为配合调度的实现，我们在进程结构中定义了整型成员tick\_count，以实现循环轮转调度。循环轮转调度时，应判断当前进程的tick\_count加1后是否大于等于TIME\_SLICE\_LEN。若答案为肯定的，则应将当前进程的tick\_count清零，并将当前进程加入就绪队列，转进程调度；若答案为否定的，则应将当前进程的tick\_count加1，并返回。

应该完善的代码如下所示：

current->tick\_count++;

if(current->tick\_count >= TIME\_SLICE\_LEN){

current->tick\_count = 0;

current->status = READY;

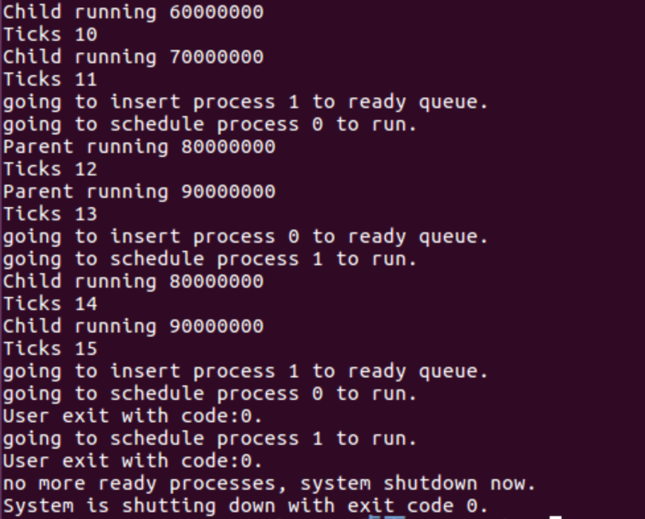
insert\_to\_ready\_queue(current);

schedule();

}

1. 实验调试及心得

完成结果如下所示：



**图3.3**

在本实验中，我们了解了如何根据时间片完成进程调度，这一动作需要借助在进程结构体中增加保存时间片的变量来完成。

## lab3\_challenge1 进程等待和数据段复制

1. 实验目的

父进程的有些操作需要子进程运行结束后获得结果才能继续执行，这时wait函数起到进程同步的作用。

在lab3\_challenge1程序中，父进程把flag变量赋值为0，然后fork生成一个子进程，接着通过wait函数等待子进程的退出。子进程把自己的变量flag赋值为1，然后fork生成孙子进程，接着通过wait函数等待孙子进程的退出。孙子进程给自己的变量flag赋值为2并在退出时输出信息，然后子进程退出时输出信息，最后父进程退出时输出信息。由于fork之后父子进程的数据段相互独立（同一虚拟地址对应不同的物理地址），子进程对全局变量的赋值不影响父进程全局变量的值。

在本实验中，我们需要实现wait函数完成进程等待和流转。

1. 实验内容

本实验的具体要求为：

* 补充do\_fork函数，lab3\_1实现了代码段的复制，你需要继续实现数据段的复制并保证fork后父子进程的数据段相互独立。

在do\_fork函数中增加如下语句：

case DATA\_SEGMENT:

{

uint64 data\_pa = (uint64)alloc\_page();

user\_vm\_map(child->pagetable,parent->mapped\_info[i].va,PGSIZE,data\_pa, prot\_to\_type(PROT\_WRITE | PROT\_READ,1));

memcpy((void\*)data\_pa, (void\*)lookup\_pa(parent->pagetable,parent->mapped\_info[i].va) , PGSIZE );

child->mapped\_info[child->total\_mapped\_region].va = parent->mapped\_info[i].va;

child->mapped\_info[child->total\_mapped\_region].npages =

parent->mapped\_info[i].npages;

child->mapped\_info[child->total\_mapped\_region].seg\_type = DATA\_SEGMENT;

child->total\_mapped\_region++;

}

* 通过修改PKE内核和系统调用，为用户程序提供wait函数的功能，wait函数接受一个参数pid：
* 当pid为-1时，父进程等待任意一个子进程退出即返回子进程的pid；
* 当pid大于0时，父进程等待进程号为pid的子进程退出即返回子进程的pid；
* 如果pid不合法或pid大于0且pid对应的进程不是当前进程的子进程，返回-1。

因此我们需要在 kernel/syscall.h 中，添加一个syscall类型SYS\_user\_wait。

#define SYS\_user\_wait (SYS\_user\_base + 6)

接着在 kernel/syscall.c 中，添加处理该系统调用的函数 sys\_user\_wait。

ssize\_t sys\_user\_wait(uint64 pid){

if(pid == -1){

current->status = BLOCKED;

current->child\_wait\_pid = current->pid + 1;

schedule();

return current->pid + 1;

}else if(pid > 0 && pid > current->pid){

current->status = BLOCKED;

current->child\_wait\_pid = pid;

schedule();

return pid;

}else{

return -1;

}

}

可以看到为了标注该进程等待的子进程，在进程结构体中增加了child\_wait\_pid 变量，用于保存正在等待子进程的pid。

修改函数 sys\_user\_exit ，用于在一个进程退出时唤醒等待它的进程。

ssize\_t sys\_user\_exit(uint64 code) {

sprint("User exit with code:%d.\n", code);

// in lab3 now, we should reclaim the current process, and reschedule.

process \*parent = current->parent;

uint64 pid\_current = current->pid;

free\_process( current );

if(parent != NULL && parent->status == BLOCKED && parent->child\_wait\_pid == pid\_current){

parent->status = READY;

insert\_to\_ready\_queue(parent);

}

schedule();

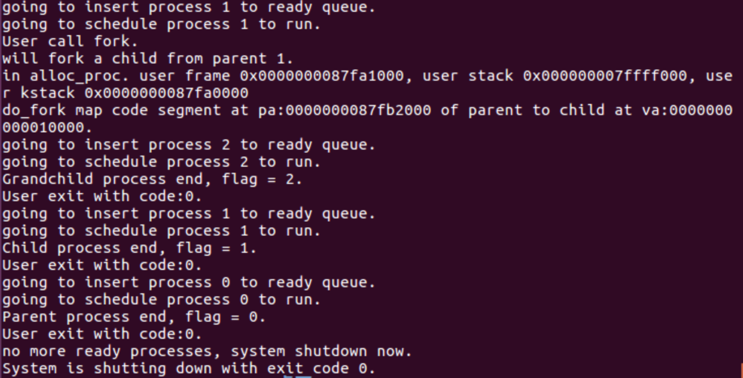
return 0;

}

通过以上修改，可以完成本实验内容。

1. 实验调试及心得

实验结果如下所示：



**图3.4**

在本实验中，我们完成了wait函数，唤醒等待进程。为了在一个进程退出时可以唤醒等待它的进程，我们还修改了系统调用中处理exit的函数。

## lab3\_challenge2 实现信号量

1. 实验目的

lab3\_challenge2 是lab3\_challenge1的进阶版。程序通过信号量的增减，控制主进程和两个子进程的输出按主进程，第一个子进程，第二个子进程，主进程，第一个子进程，第二个子进程……这样的顺序轮流输出。

1. 实验内容

对内核代码的修改可能包含以下内容：

* 添加系统调用，使得用户对信号量的操作可以在内核态处理。

我们需要在 kernel/syscall.h 中，添加syscall类型SYS\_user\_sem\_new、SYS\_user\_sem\_P和SYS\_user\_sem\_V。

* 在内核中实现信号量的分配、释放和PV操作，当P操作处于等待状态时能够触发进程调度。

为了保存信号量，我在 kernel/process.c 中添加了一个全局变量信号量数组semaphore sems[NPROC] 。

结构体 semaphore 定义如下所述：

typedef struct semaphore{

int sem\_id;

int val;

int status;

}semaphore;

sem\_id 用于标注信号量，val用于保存信号量的值。而status是模仿进程分配为了找到一个空余信号量而设置的变量。

在 sys\_user\_sem\_new 函数中，首先根据status的内容，找到一个空余的信号量，接着将传入的信号量初始值保存在 val 变量中。

为了简单实现本实验内容，我将信号量和进程相互对应，由此可以用信号量控制相应进程的流转。

在 sys\_user\_sem\_P 函数中，现将当前的信号量值减1，如果此时 val 已经小于0，那么该进程进入等待状态，代码实现如下所示：

current->status = BLOCKED;

schedule();

在 sys\_user\_sem\_V 函数中，操作的信号量就不是当前进程对应的信号量了，而是当前进程的下一个需要流转的进程对应的信号量。把需要修改的信号量取出，其值加1。如果对应需要流转的进程已经为ZOMBIE状态也就是回收状态，那么则直接返回不处理。而如果此时val值大于等于0，且对应需要流转的进程状态也不是READY时，需要将该进程设为就绪态并加入就绪队列。

实现代码如下所示：

if(procs[next].status == ZOMBIE) return 0;

if(sems[next].val >= 0 && procs[next].status != READY){

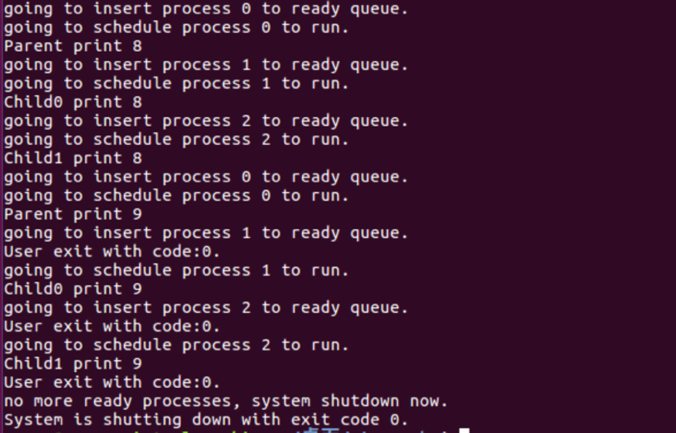
procs[next].status = READY;

insert\_to\_ready\_queue(&procs[next]);

}

1. 实验调试及心得

实验结果运行如下所示：



**图3.5**

本实验中，实现了信号量。为了简单实现，在这里我简化使用了进程和信号量对应的思路。但我相信在实际使用中会有更好的实现思路，在接下来的学习过程中我会继续学习探索。