项目说明文档

操作系统课程设计

——xv6 及 Labs 课程项目

作 者 姓 名： 袁泓博

学 号： 2152208

指 导 教 师： 王冬青

学 院 专 业： 软件学院 软件工程

同济大学

Tongji University

目录

[**1.** **Lab1: Xv6 and Unix utilities** 6](#_Toc143177768)

[1.1 Boot xv6 (easy) 6](#_Toc143177769)

[1.1.1 实验目的 6](#_Toc143177770)

[1.1.2 实验步骤 6](#_Toc143177771)

[1.1.3 实验中遇到的问题和解决办法 6](#_Toc143177772)

[1.1.4 实验心得 6](#_Toc143177773)

[1.2 sleep (easy) 6](#_Toc143177774)

[1.2.1 实验目的 6](#_Toc143177775)

[1.2.2 实验步骤 6](#_Toc143177776)

[1.2.3 实验中遇到的问题和解决办法 8](#_Toc143177777)

[1.2.4 实验心得 8](#_Toc143177778)

[1.3 pingpong (easy) 8](#_Toc143177779)

[1.3.1 实验目的 8](#_Toc143177780)

[1.3.2 实验步骤 8](#_Toc143177781)

[1.3.3 实验中遇到的问题和解决办法 9](#_Toc143177782)

[1.3.4 实验心得 10](#_Toc143177783)

[1.4 primes (moderate)/(hard) 10](#_Toc143177784)

[1.4.1 实验目的 10](#_Toc143177785)

[1.4.2 实验步骤 10](#_Toc143177786)

[1.4.3 实验中遇到的问题和解决办法 11](#_Toc143177787)

[1.4.4 实验心得 11](#_Toc143177788)

[1.5 find (moderate) 12](#_Toc143177789)

[1.5.1 实验目的 12](#_Toc143177790)

[1.5.2 实验步骤 12](#_Toc143177791)

[1.5.3 实验中遇到的问题和解决办法 13](#_Toc143177792)

[1.5.4 实验心得 14](#_Toc143177793)

[1.6 xargs (moderate) 14](#_Toc143177794)

[1.6.1 实验目的 14](#_Toc143177795)

[1.6.2 实验步骤 14](#_Toc143177796)

[1.6.3 实验中遇到的问题和解决办法 16](#_Toc143177797)

[1.6.4 实验心得 17](#_Toc143177798)

[**2.** **Lab2: system calls** 17](#_Toc143177799)

[2.1 System call tracing (moderate) 17](#_Toc143177800)

[2.1.1 实验目的 17](#_Toc143177801)

[2.1.2 实验步骤 17](#_Toc143177802)

[2.1.3 实验中遇到的问题和解决办法 20](#_Toc143177803)

[2.1.4 实验心得 20](#_Toc143177804)

[2.2 Sysinfo (moderate) 20](#_Toc143177805)

[2.2.1 实验目的 20](#_Toc143177806)

[2.2.2 实验步骤 20](#_Toc143177807)

[2.2.3 实验中遇到的问题和解决办法 23](#_Toc143177808)

[2.2.4 实验心得 23](#_Toc143177809)

[**3.** **Lab3: page tables** 23](#_Toc143177810)

[3.1 Speed up system calls (easy) 23](#_Toc143177811)

[3.1.1 实验目的 23](#_Toc143177812)

[3.1.2 实验步骤 24](#_Toc143177813)

[3.1.3 实验中遇到的问题和解决办法 26](#_Toc143177814)

[3.1.4 实验心得 26](#_Toc143177815)

[3.2 Print a page table (easy) 26](#_Toc143177816)

[3.2.1 实验目的 26](#_Toc143177817)

[3.2.2 实验步骤 26](#_Toc143177818)

[3.2.3 实验中遇到的问题和解决办法 27](#_Toc143177819)

[3.2.4 实验心得 28](#_Toc143177820)

[3.3 Detecting which pages have been accessed (hard) 28](#_Toc143177821)

[3.3.1 实验目的 28](#_Toc143177822)

[3.3.2 实验步骤 28](#_Toc143177823)

[3.3.3 实验中遇到的问题和解决办法 30](#_Toc143177824)

[3.3.4 实验心得 30](#_Toc143177825)

[**4.** **Lab4: traps** 31](#_Toc143177826)

[4.1 RISC-V assembly (easy) 31](#_Toc143177827)

[4.1.1 实验目的 31](#_Toc143177828)

[4.1.2 实验步骤 31](#_Toc143177829)

[4.1.3 实验中遇到的问题和解决办法 33](#_Toc143177830)

[4.1.4 实验心得 33](#_Toc143177831)

[4.2 Backtrace (moderate) 34](#_Toc143177832)

[4.2.1 实验目的 34](#_Toc143177833)

[4.2.2 实验步骤 34](#_Toc143177834)

[4.2.3 实验中遇到的问题和解决办法 35](#_Toc143177835)

[4.2.4 实验心得 35](#_Toc143177836)

[4.3 Alarm (hard) 35](#_Toc143177837)

[4.3.1 实验目的 35](#_Toc143177838)

[4.3.2 实验步骤 35](#_Toc143177839)

[4.3.3 实验中遇到的问题和解决办法 38](#_Toc143177840)

[4.3.4 实验心得 38](#_Toc143177841)

[**5.** **Lab5: Copy-on-Write Fork for xv6** 38](#_Toc143177842)

[5.1 Implement copy-on write(hard) 38](#_Toc143177843)

[5.1.1 实验目的 38](#_Toc143177844)

[5.1.2 实验步骤 39](#_Toc143177845)

[5.1.3 实验中遇到的问题和解决办法 43](#_Toc143177846)

[5.1.4 实验心得 43](#_Toc143177847)

[**6.** **Lab6: Multithreading** 44](#_Toc143177848)

[6.1 Uthread: switching between threads (moderate) 44](#_Toc143177849)

[6.1.1 实验目的 44](#_Toc143177850)

[6.1.2 实验步骤 44](#_Toc143177851)

[6.1.3 实验中遇到的问题和解决办法 45](#_Toc143177852)

[6.1.4 实验心得 45](#_Toc143177853)

[6.2 Using threads (moderate) 45](#_Toc143177854)

[6.2.1 实验目的 45](#_Toc143177855)

[6.2.2 实验步骤 45](#_Toc143177856)

[6.2.3 实验中遇到的问题和解决办法 48](#_Toc143177857)

[6.2.4 实验心得 48](#_Toc143177858)

[6.3 Barrier(moderate) 48](#_Toc143177859)

[6.3.1 实验目的 48](#_Toc143177860)

[6.3.2 实验步骤 48](#_Toc143177861)

[6.3.3 实验中遇到的问题和解决办法 49](#_Toc143177862)

[6.3.4 实验心得 50](#_Toc143177863)

[**7.** **Lab7: networking** 50](#_Toc143177864)

[7.1 Your Job (hard) 50](#_Toc143177865)

[7.1.1 实验目的 50](#_Toc143177866)

[7.1.2 实验步骤 50](#_Toc143177867)

[7.1.3 实验中遇到的问题和解决办法 53](#_Toc143177868)

[7.1.4 实验心得 53](#_Toc143177869)

[**8.** **Lab8: locks** 53](#_Toc143177870)

[8.1 Memory allocator (moderate) 53](#_Toc143177871)

[8.1.1 实验目的 53](#_Toc143177872)

[8.1.2 实验步骤 53](#_Toc143177873)

[8.1.3 实验中遇到的问题和解决办法 54](#_Toc143177874)

[8.1.4 实验心得 54](#_Toc143177875)

[8.2 Buffer cache (hard) 55](#_Toc143177876)

[8.2.1 实验目的 55](#_Toc143177877)

[8.2.2 实验步骤 55](#_Toc143177878)

[8.2.3 实验中遇到的问题和解决办法 57](#_Toc143177879)

[8.2.4 实验心得 58](#_Toc143177880)

[**9.** **Lab9: file system** 58](#_Toc143177881)

[9.1 Large files (moderate) 58](#_Toc143177882)

[9.1.1 实验目的 58](#_Toc143177883)

[9.1.2 实验步骤 58](#_Toc143177884)

[9.1.3 实验中遇到的问题和解决办法 61](#_Toc143177885)

[9.1.4 实验心得 62](#_Toc143177886)

[9.2 Symbolic links (moderate) 62](#_Toc143177887)

[9.2.1 实验目的 62](#_Toc143177888)

[9.2.2 实验步骤 62](#_Toc143177889)

[9.2.3 实验中遇到的问题和解决办法 65](#_Toc143177890)

[9.2.4 实验心得 65](#_Toc143177891)

[**10.** **Lab10: mmap** 65](#_Toc143177892)

[10.1 mmap (hard) 65](#_Toc143177893)

[10.1.1 实验目的 65](#_Toc143177894)

[10.1.2 实验步骤 65](#_Toc143177895)

[10.1.3 实验中遇到的问题和解决办法 70](#_Toc143177896)

[10.1.4 实验心得 70](#_Toc143177897)

# **Lab1: Xv6 and Unix utilities**

## Boot xv6 (easy)

## 实验目的

启动xv6，初步掌握xv6

## 实验步骤

获取实验室的xv6源代码并切换到util分支：

$ **git clone git://g.csail.mit.edu/xv6-labs-2021**

$ **cd xv6-labs-2021**

$ **git checkout util**

利用make qemu指令运行xv6：

$ **make qemu**

此时输入ls指令能看到内容输出

## 实验中遇到的问题和解决办法

在完成实验的过程中遇到了比较多困难，包括不知道如何装模拟机，如何启动项目等，在和同学交流后顺利解决。

## 实验心得

通过本实验，我初步了解了 xv6 这一操作系统内核，同时也了解了 qemu 模拟器的使用方法。

## sleep (easy)

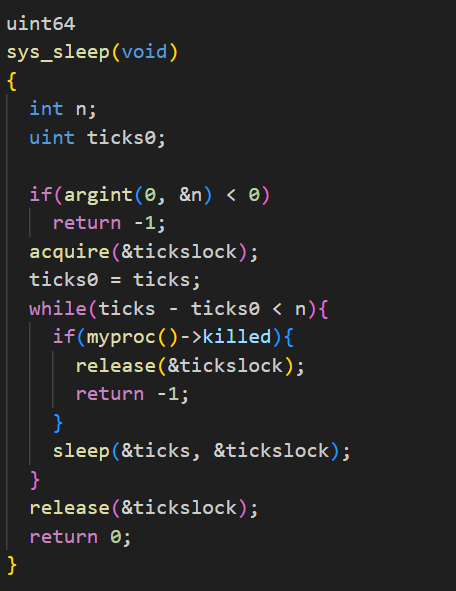
## 实验目的

实现xv6的UNIX程序sleep：您的sleep应该暂停到用户指定的计时数。

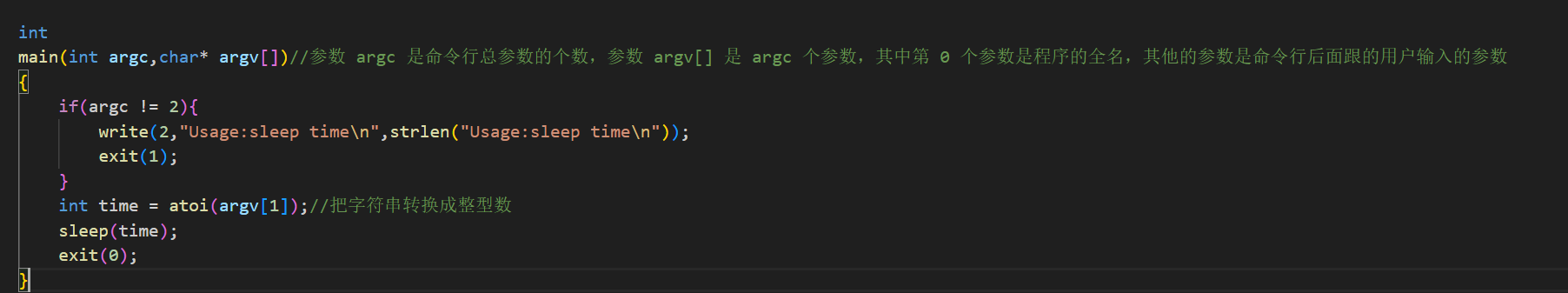
## 实验步骤

使用系统调用sleep

参阅kernel/sysproc.c以获取实现sleep系统调用的xv6内核代码：



user/user.h提供了sleep的声明以便其他程序调用，用汇编程序编写的user/usys.S可以帮助sleep从用户区跳转到内核区



确保main函数调用exit()以退出程序

将sleep程序添加到Makefile中的UPROGS中即可运行



## 实验中遇到的问题和解决办法

在实验前我并不知道如何创建系统调用，通过网络查询得到了解决。

## 实验心得

通过此实验我了解了什么是系统调用

## pingpong (easy)

## 实验目的

编写一个使用UNIX系统调用的程序来在两个进程之间“ping-pong”一个字节，使用两个管道，每个方向一个。父进程应该向子进程发送一个字节;子进程应该打印“<pid>: received ping”，其中<pid>是进程ID，并在管道中写入字节发送给父进程，然后退出;父级应该从读取从子进程而来的字节，打印“<pid>: received pong”，然后退出。

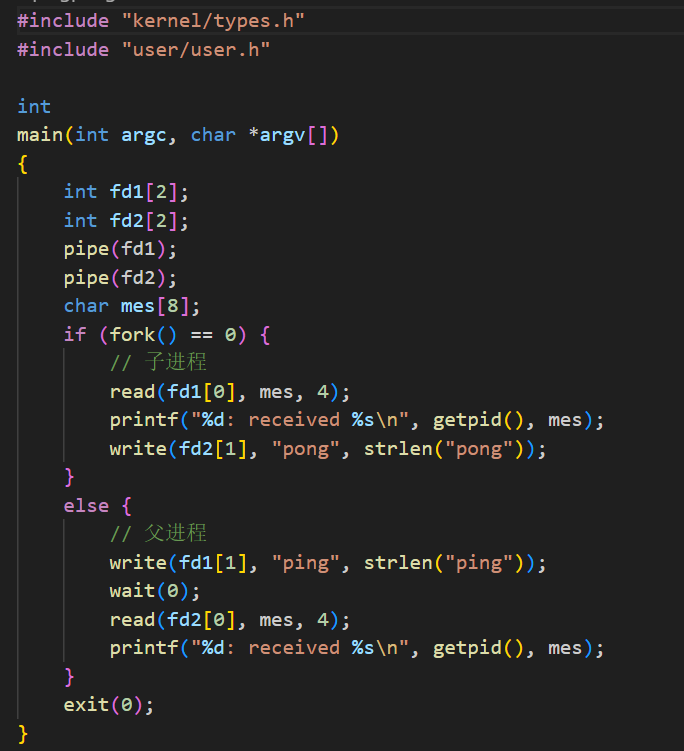
## 实验步骤

使用pipe来创造管道

使用fork创建子进程

使用read从管道中读取数据，并且使用write向管道中写入数据

使用getpid获取调用进程的pid：



将程序加入到Makefile的UPROGS



## 实验中遇到的问题和解决办法

本次实验中我不知道管道如何创建、创建完成后如何关闭，后续通过网络查找资料得到解答。

## 实验心得

了解了父子进程，以及 fork()函数创建父子进程的方法

## primes (moderate)/(hard)

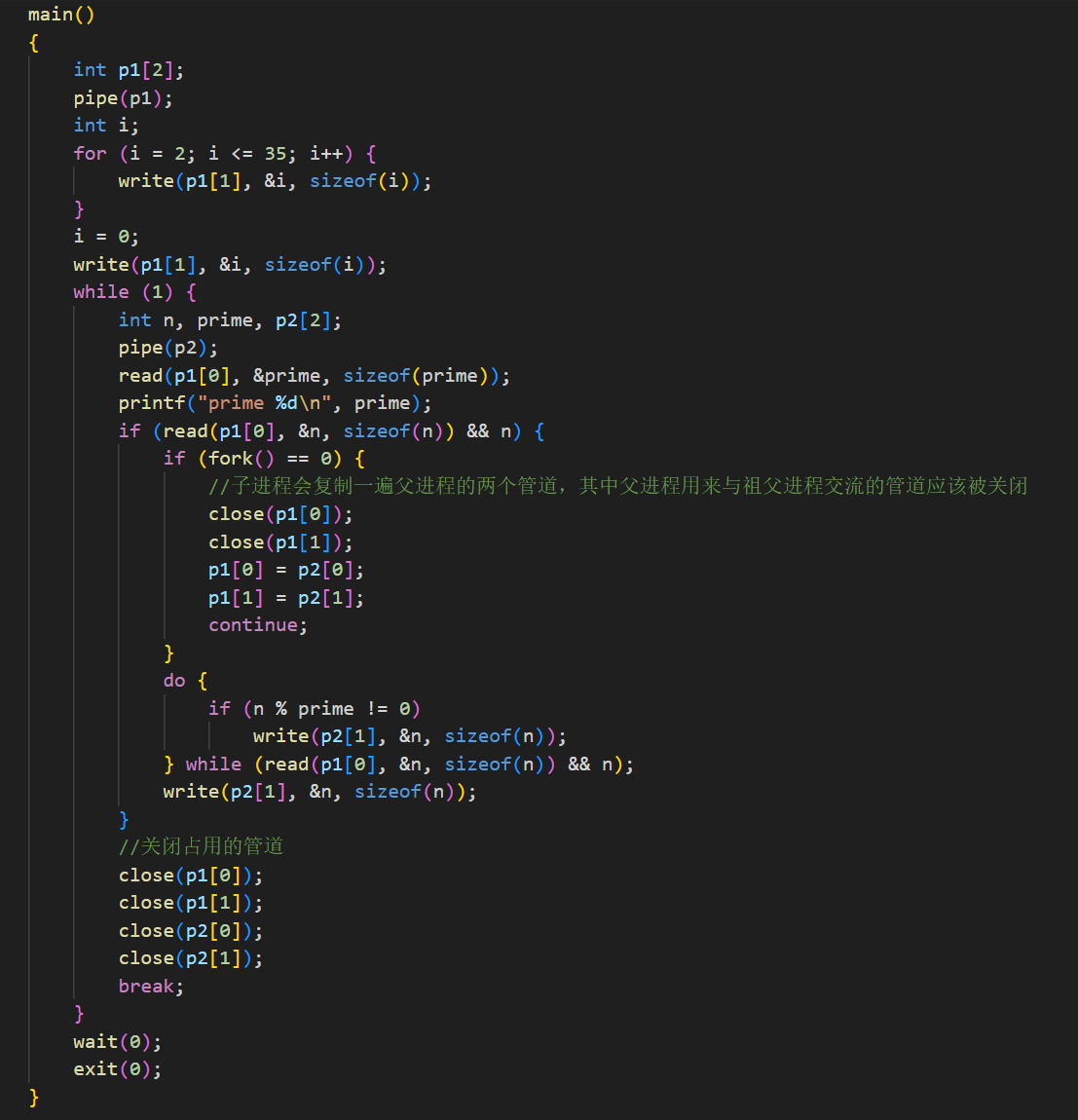
## 实验目的

使用管道编写prime sieve(筛选素数)的并发版本。

## 实验步骤

关闭进程不需要的文件描述符

一旦第一个进程达到35，它应该使用wait等待整个管道终止，包括所有子孙进程等等。因此，主primes进程应该只在打印完所有输出之后，并且在所有其他primes进程退出之后退出，当管道的write端关闭时，read返回零



将程序添加到Makefile中的UPROGS



## 实验中遇到的问题和解决办法

在此实验前我不了解素数筛到底是什么，通过翻阅相关资料得以解决。

## 实验心得

掌握了素数筛，以及利用管道编写素数筛的方法。

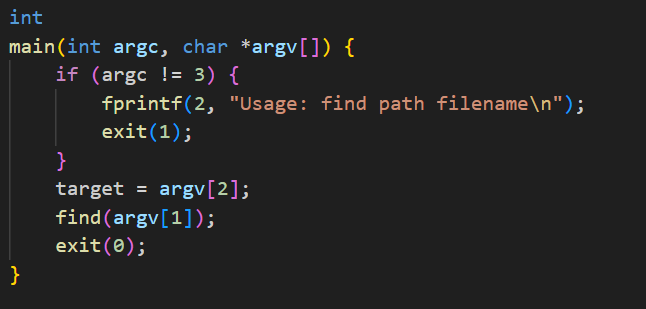
## find (moderate)

## 实验目的

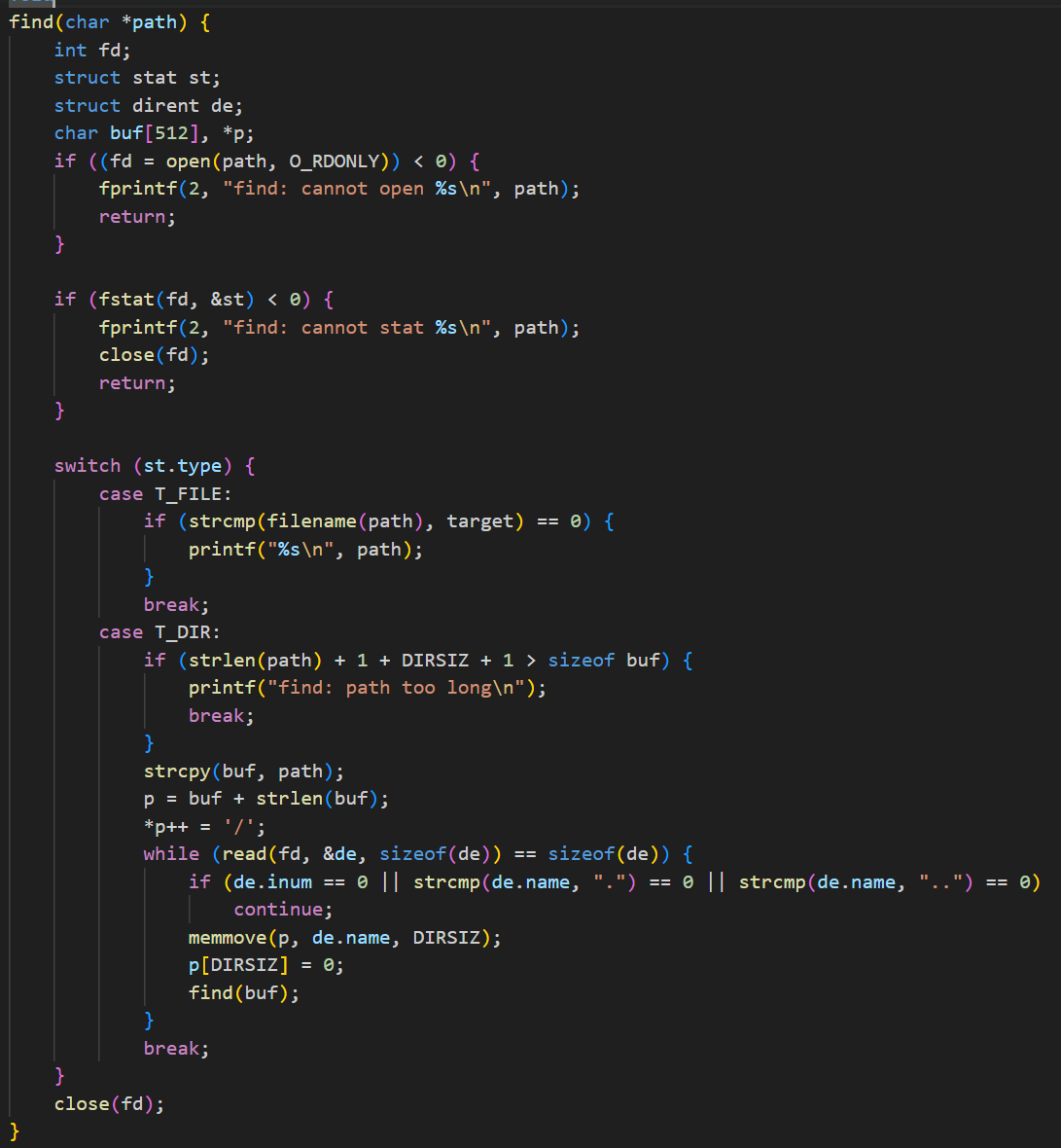
写一个简化版本的UNIX的find程序：查找目录树中具有特定名称的所有文件

## 实验步骤

参考 ls 方法，我可以发现 find 实现方法的整体思路应当为深度优先搜索。因此，在 main 函数中我只需要检查参数。如果参数合法，只需要直接调用深度优先搜索函数 find 即可：



对于 find 函数，它的作用是在每一级中检索是否有满足要求的文件：



在 MakeFile 文件中添加 find 方法即可



## 实验中遇到的问题和解决办法

编写代码的过程遇到了许多问题，通过调试等方法解决了。

## 实验心得

完成了find 程序的编写，更加了解了xv6 这一操作系统内核。

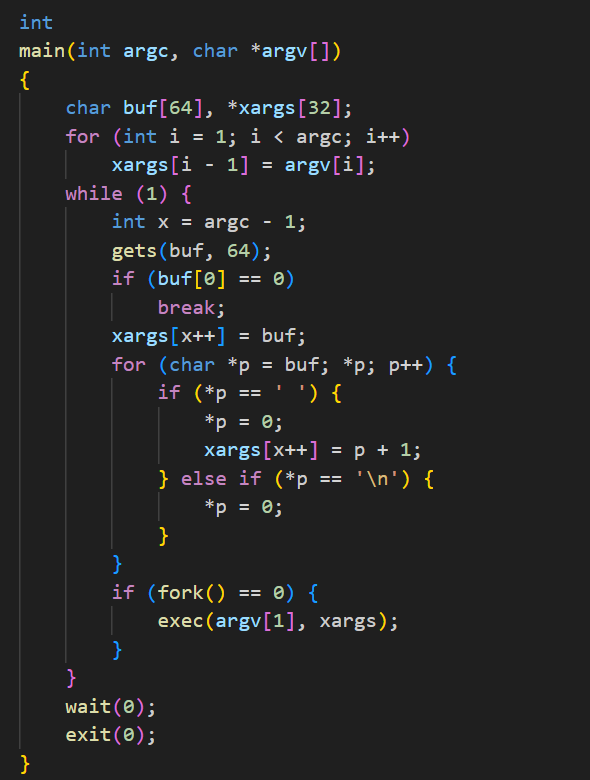
## xargs (moderate)

## 实验目的

编写一个简化版UNIX的xargs程序：它从标准输入中按行读取，并且为每一行执行一个命令，将行作为参数提供给命令

## 实验步骤

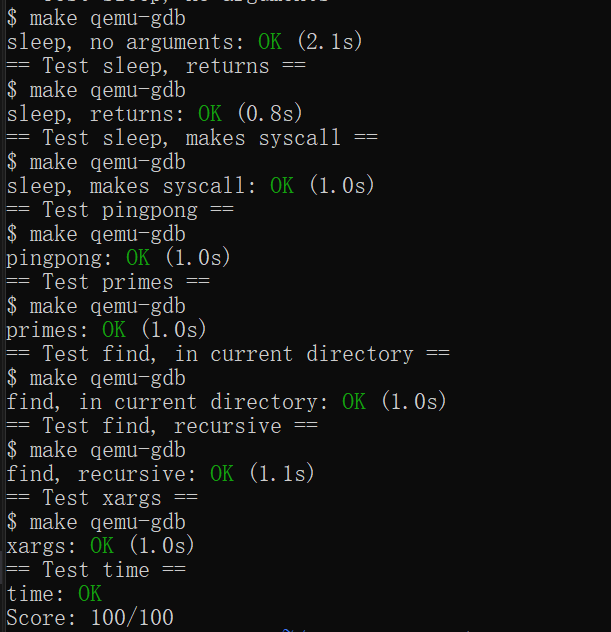
使用fork和exec对每行输入调用命令，在父进程中使用wait等待子进程完成命令。要读取单个输入行，一次读取一个字符，直到出现换行符（'\n'）



将程序添加到Makefile中的UPROGS：



最后所有实验检验得分：



## 实验中遇到的问题和解决办法

在完成实验的过程中，我遇到了许多由于不熟悉操作和编写的困难，通过和同学交流和查阅网上资料解决了这些问题。

## 实验心得

完成 了xargs 程序的编写 ，熟悉了xv6 这一操作系统内核。此外，我还初步了解到了系统调用接口，并且知道了操作系统内核和用户的区别。

# **Lab2: system calls**

## System call tracing (moderate)

## 实验目的

通过向xv6添加一些新的系统调用，帮助了解系统调用是如何工作的，并了解xv6内核的一些内部结构。

## 实验步骤

将代码切换到syscall分支：

$ **git fetch**

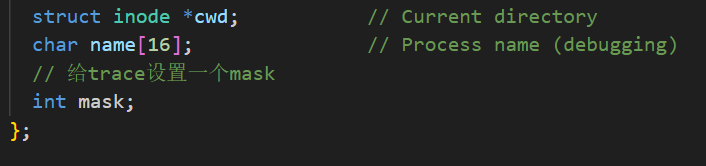
$ **git checkout syscall**

$ **make clean**

把Makefile加上：



我们要在/kernel/proc.h文件的proc结构体中加入一个新的变量，让每个进程都有一个自己的mask：



主要的实现就是在/usr/syscall.c文件的syscall函数，通过观察该函数可以发现：

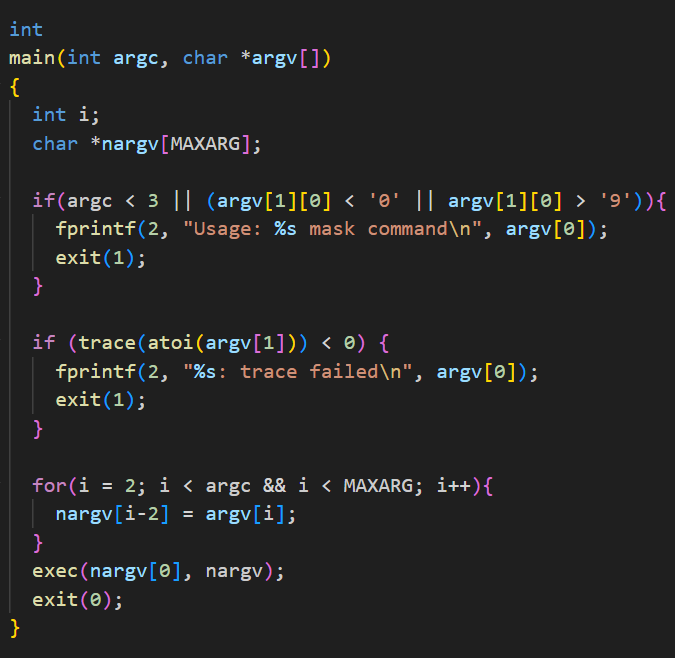
p->trapframe->a0 = syscalls[num]();

调用了系统调用命令，并且把返回值保存在了a0寄存器中（RISCV的C规范是把返回值放在a0中)，所以我们只要在调用系统调用时判断是不是mask规定的输出函数，如果是就输出。

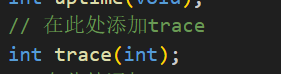
由于mask是按位判断的，且proc结构体里的name是整个线程的名字，不是函数调用的函数名称，所以我们不能用p->name，而要自己定义一个数组，因此最终函数写为：



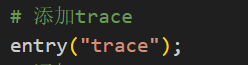
为了把mask参数传进来需要用到trace.c 文件，而这里已经给了trace的用户态函数：



可以看到trace函数传入的是一个数字，那我们只要在系统调用trace里把这个数字给到现在的线程，把trace这个系统调用加入到内核中声明，首先是/usre/user.h文件加入，这里声明了用户态可以调用的系统调用：



/user/usys.pl文件加入：



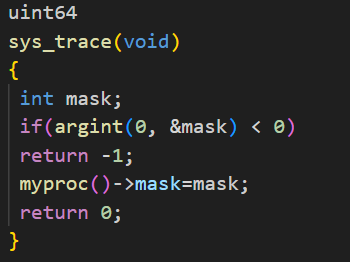
接下来还要给内核态的系统调用trace加上声明和定义，在kernel/syscall.c加上：



在下面的函数指针数组\*syscalls[]加上：



然后在kerlnel/sysproc.c加上sys\_trace的定义实现：



即可成功运行

## 实验中遇到的问题和解决办法

在做这次实验的过程中我遇到的问题相对比较多，该实验需要阅读比较多的内核 kernel 源代码，包括 syscall()等函数，只有熟悉了这些函数才能够比较好的完成实验内容。

## 实验心得

本次实验我进一步了解了系统调用，掌握了添加系统调用的方法。

## Sysinfo (moderate)

## 实验目的

添加系统调用 sysinfo，用于收集正在运行的系统的信息。

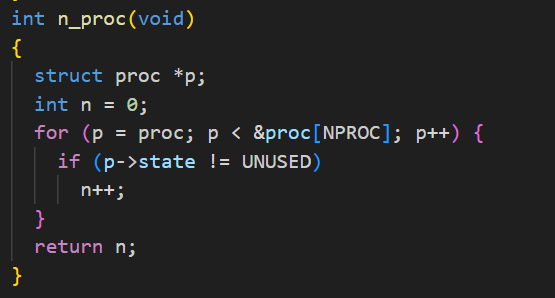
## 实验步骤

通过阅读发现在/kernel/proc.c文件中定义了一个数组：

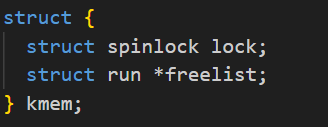


这个数组就保存着所有的进程，所以只要遍历这个数组判断状态就行

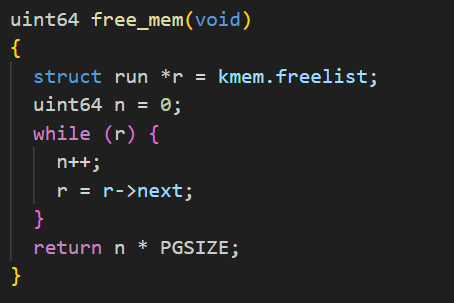
所以实现在kernel/proc.c中加入：



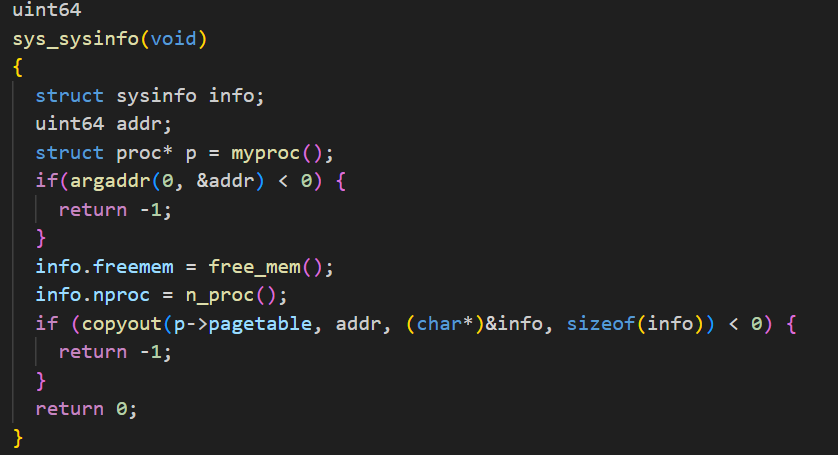
针对于当前可以使用的空间，可以参考 kalloc.c 文件。其中定义了一个链表 kemem，每个链表都指向了一个可用空间，而 kmem 则保存了最后一个链表



所以kmem.freelist永远指向最后一个可用页，那我们只要顺着这个链表往前走，直到NULL为止：



然后在kernel/sysproc.c文件里加入sysinfo实现，同时加入sysinfo结构体对应头文件：



即完成本次实验。

通过make grade测试：



## 实验中遇到的问题和解决办法

在完成此实验的时候，由于对进程的定义仅仅停留在课本上，因此刚开始的时候难以下手。但是通过查阅相关资料，我了解到 xv6 中对进程的定义， 以及存储所有进程的变量和进程状态变量，仿照着已经存在的内核函数，终于完成了该部分。

## 实验心得

这次实验帮我理解系统调用的工作原理和 xv6 内核的工作过程。总体来说，这次实验让我在实验一的基础上更加了解系统调用，同时也为未来的实验打下了比较扎实的基础。

# **Lab3: page tables**

## Speed up system calls (easy)

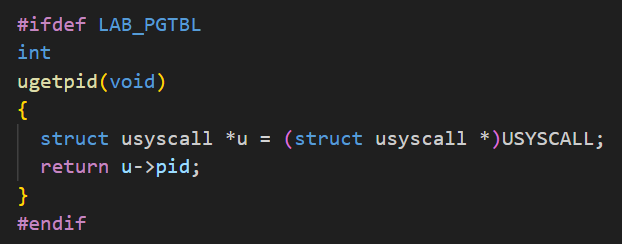
## 实验目的

通过加速系统调用实验学习如何将映射插入页表，在内核和用户程序之间创建一个共享的只读页，这样内核往这个页里写入数据的时候，用户程序就可以不经复杂的系统调用直接读取它了。

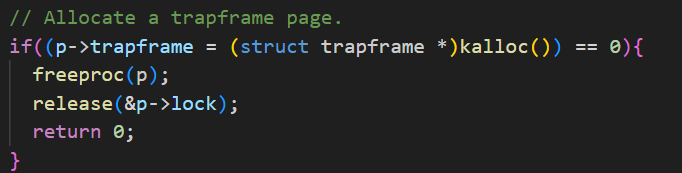
## 实验步骤

对于本实验，ugetpid()已在用户空间端提供，并将自动使用 USYSCALL 映射。

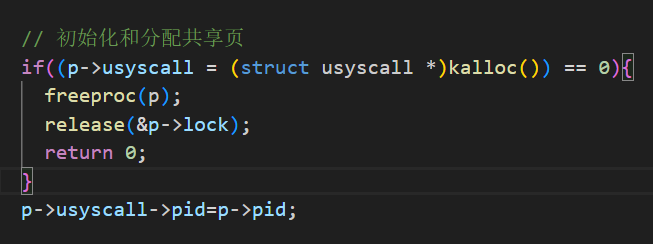
看一下ugetpid()：



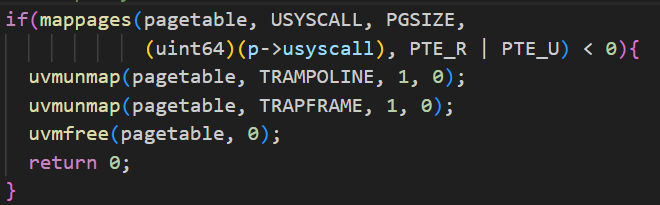
Makefile里面已经设置了相应的CFLAGS，编译的时候会自动加上这个LAB\_PGTBL宏，发现ugetpid()直接从USYSCALL这个地址读数据，因此我们需要把usyscall结构写到此页表的开头。这个USYSCALL是紧邻trapframe下端的一页，然后提示说在proc\_pagetable里面设置映射,对USYSMAP的映射就在这里进行。xv6是先给trapframe分配一块内存再把TRAPFRAME映射到它上面。看一下allocproc()，这个程序首先循环搜索进程表，搜索到UNUSED进程就为其分配内存，然后给进程表p赋各种值：



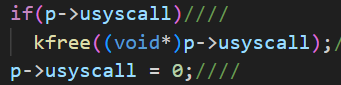
很显然p->trapframe就是在这里初始化的，在allocproc()初始化之后在proc\_pagetable()之中映射。我们可以仿照trapframe的操作，在进程控制块struct proc中添加struct usyscall \*usyscall，作为共享页的物理地址。然后在kernel/proc.c中的allocproc()中完成共享页的分配和初始化：



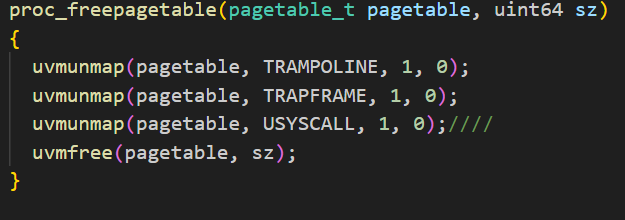
接着在用户页表中添加共享页映射。在kernel/proc.c的proc\_pagetable()中：



然后做好释放，仿照freeproc里对trapframe里的操作来释放usyspage：



修改下面这个proc\_freepagetable函数，加上对USYSCALL的操作：



最后加上对usyscall的声明：



即可完成该实验

## 实验中遇到的问题和解决办法

在实验前需要理解学习如何将映射插入页表，通过查阅相关资料解决。

## 实验心得

通过本次实验我学习了如何将映射插入页表，掌握了如何加速系统调用。

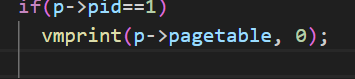
## Print a page table (easy)

## 实验目的

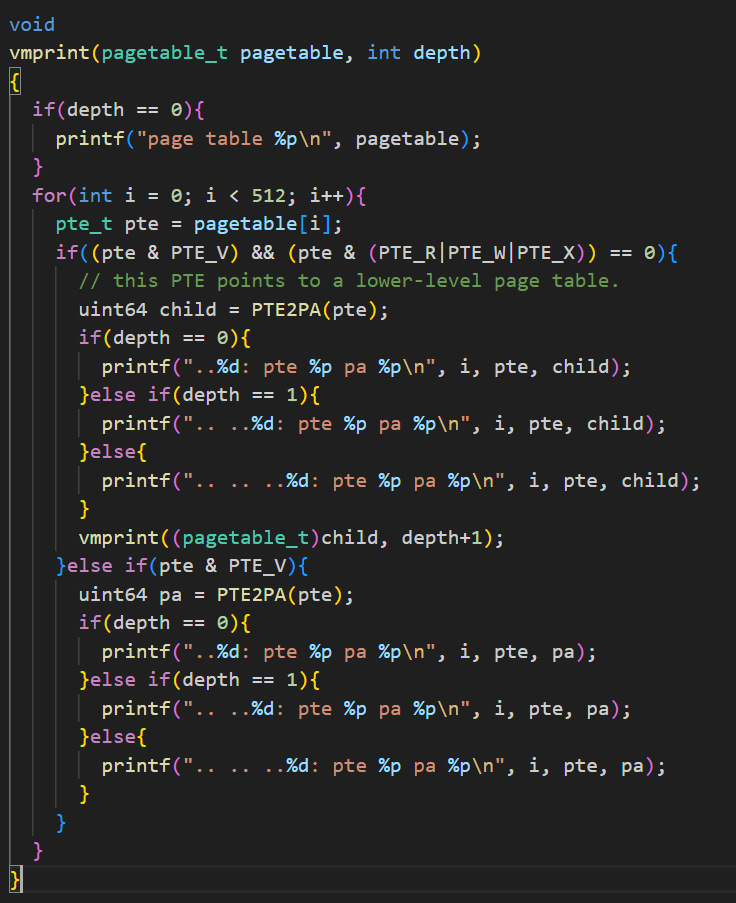
为了可视化 RISC-V 页表，并可能有助于将来的调试,编写一个打印页表内容的函数。定义一个名为vmprint()的函数。它应该接受一个pagetable\_t参数，并以下面描述的格式打印该页表。在 exec.c 中的return argc之前插入if(p->pid==1) vmprint(p->pagetable)，以打印第一个进程(init)的页表。

## 实验步骤

在exec.c中插入



在返回argc之前，以打印第一个进程的页表。在创建 init 进程时，调用这个函数打印页表。提示中说到: The function freewalk may be inspirational.仿照vm.c里面的freewalk()写代码：



## 实验中遇到的问题和解决办法

在实验前需要理解逻辑地址到物理地址的映射关系，通过查阅相关资料解决。

## 实验心得

通过本次实验我了解 RISC-V 中的页表机制，掌握了如何实现一个页表映射。

## Detecting which pages have been accessed (hard)

## 实验目的

实现pgaccess()，报告哪些页面被系统调用过，从一个用户页表地址开始，搜索所有被访问过的页并返回一个bitmap来显示这些页是否被访问过。

## 实验步骤

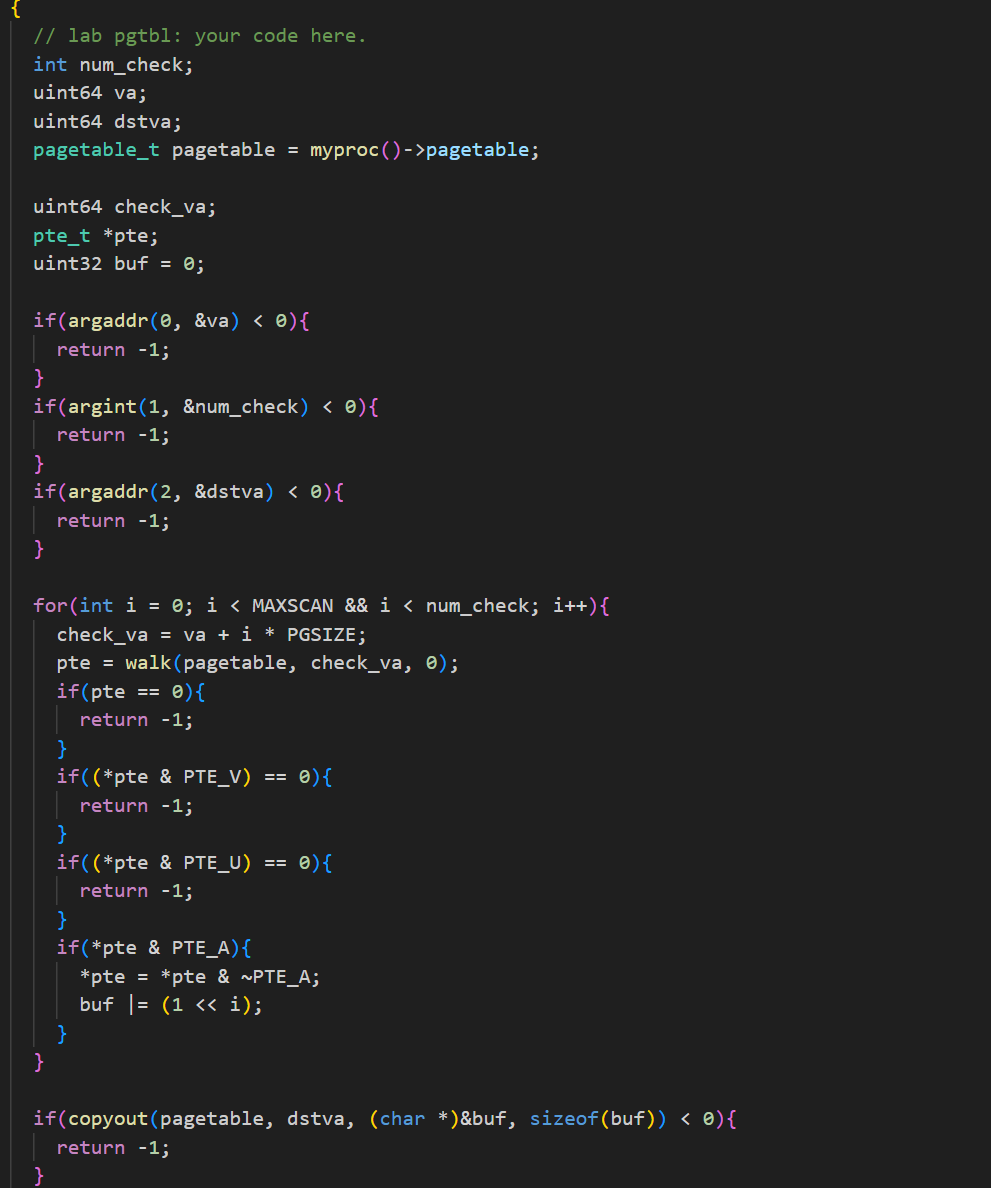
先定义在kernel/riscv.h中PTE\_A



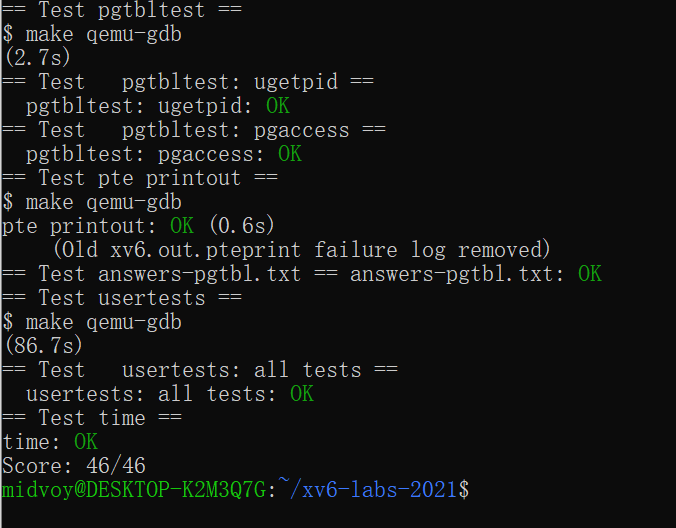
在defs.h中声明walk



然后需要先在kernel/sysproc.c中实现系统调用的外壳，即sys\_pgaccess()。它要完成的工作就是获取系统调用参数。另外，由于实际实现系统调用功能的函数pgaccess()需要在vm.c中实现，而vm.c并为引用proc.h，因此我们还需要在sysproc.c中把进程的页表以函数参数的形式附加给pgaccess()：



再创建answers-pgtbl.txt填入1,2的测试结果，即可完成实验：



## 实验中遇到的问题和解决办法

本次实验还涉及到了操作系统中地址的定位，在完成的过程中要注意越界判断等。

## 实验心得

这个实验花费时间较多。因为涉及到了操作系统中页表结构，同时还涉及到了逻辑地址到物理地址之间的转换，只有把这些概念牢牢掌握才能够比较好的完成这一次的实验。

# **Lab4: traps**

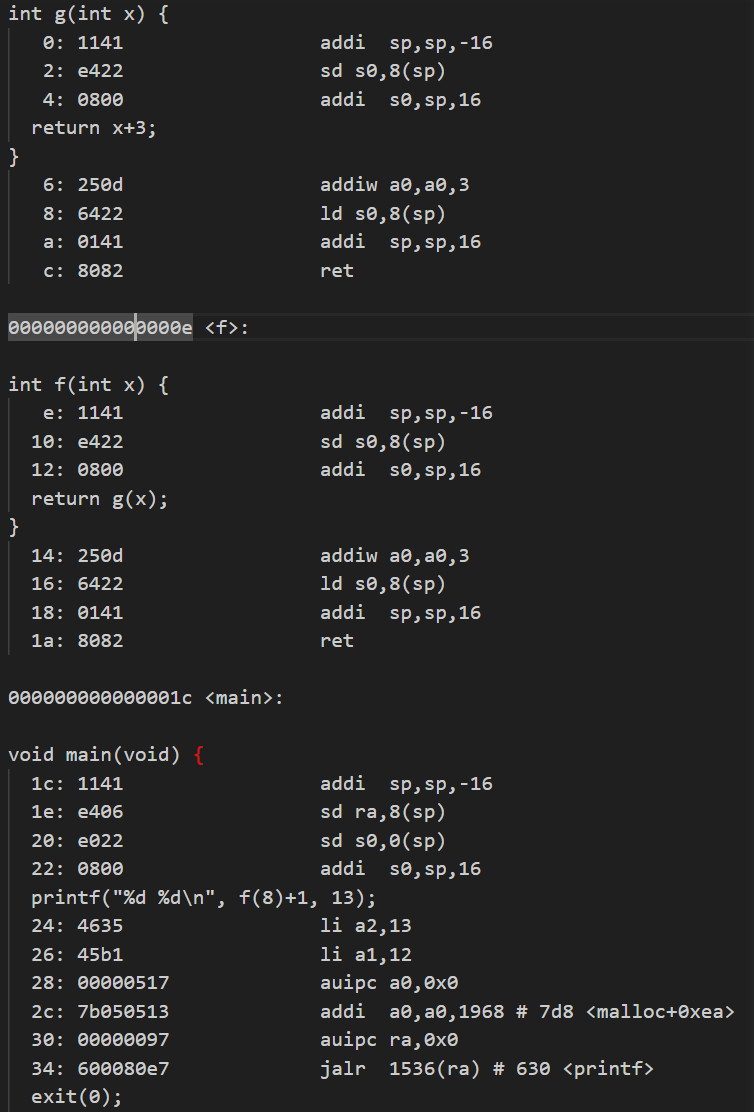
## RISC-V assembly (easy)

## 实验目的

了解 RISC-V 程序集，阅读 call.asm 中的函数 g、f 和 main 的代码，并回答问题。

## 实验步骤

xv6仓库中有一个文件user/call.c。执行make fs.img编译它，并在user/call.asm中生成可读的汇编版本,阅读call.asm中函数g、f和main的代码:



对于以下问题：

1. Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

函数各参数显然是在a0-a7中传递的，再看一下printf附近的汇编代码，根据"li a2,13"显然可知是a2。

1. Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)

没有调用，编译器直接优化了，把f(8)+1的值直接计算出来传入printf中。

1. At what address is the function printf located?

这个auipc是把0x0左移12位，再加上PC值0x30存到ra里。jalr是先根据ra算出跳转地址，再把现在的PC+4存入ra，作为稍后printf的返回地址。所以printf的位置是0x30+1536=0x630。

1. What value is in the register ra just after the jalr to printf in main?

把运行到jalr处的PC+4存入ra，也就是0x38。

1. Run the following code.

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

What is the output? Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

结果为：He110 World; 57616不需要进行改变，编译器会进行转换

1. In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

printf("x=%d y=%d", 3);

打印出寄存器a2的值，因为printf会从a2寄存器中读取第三个参数作为y的值。

## 实验中遇到的问题和解决办法

回答这些问题需要查阅大量资料，耗费了许多时间。

## 实验心得

本次实验我进一步了解系统调用所发挥的重要作用，通过回答这些问题我更清楚的认识到了系统调用的作用。

## Backtrace (moderate)

## 实验目的

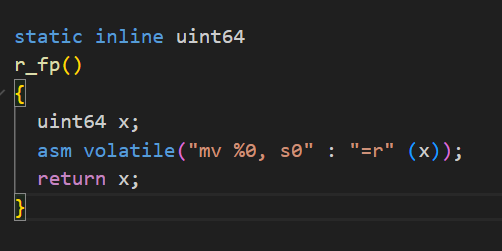
完成 Backtrace 函数，当发生错误的时候查看当前堆栈中的系统调用。

## 实验步骤

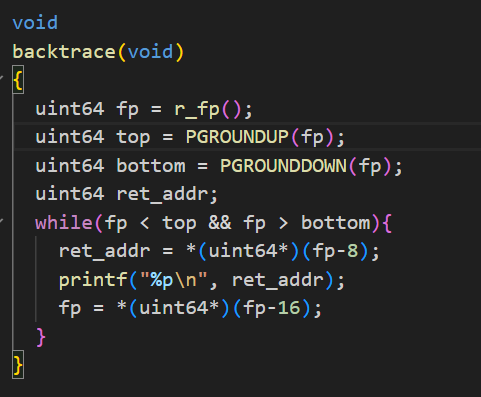
首先在 defs.h 中声明 backtrace()：



构造一个定义在 ricsv.h中的函数来读取当前的帧指针



实现backtrace，递归打印函数调用栈。使用r\_fp获取当前栈帧地址，由于栈是由高地址向低地址增长的，因此使用PGROUNDUP获得栈底地址，之后循环打印栈帧的函数的返回地址：



## 实验中遇到的问题和解决办法

通过查阅资料完成了Backtrace 函数。

## 实验心得

本次实验我完成了 Backtrace 函数，理解 xv6 中的堆栈，同时试着实现一个用户级中断处理。

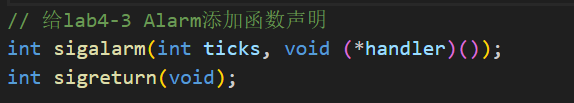
## Alarm (hard)

## 实验目的

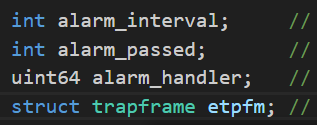
添加系统调用 sigalarm，当用户程序运行了 n 个 ticks 后，触发回调函数。

## 实验步骤

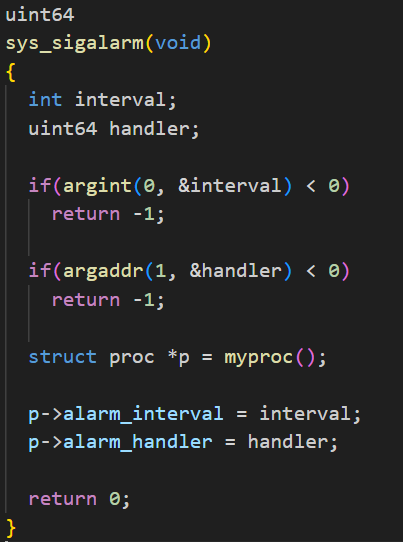
首先需要参照实验4.2将 sigalarm()和 sigreturn()声明为系统调用：



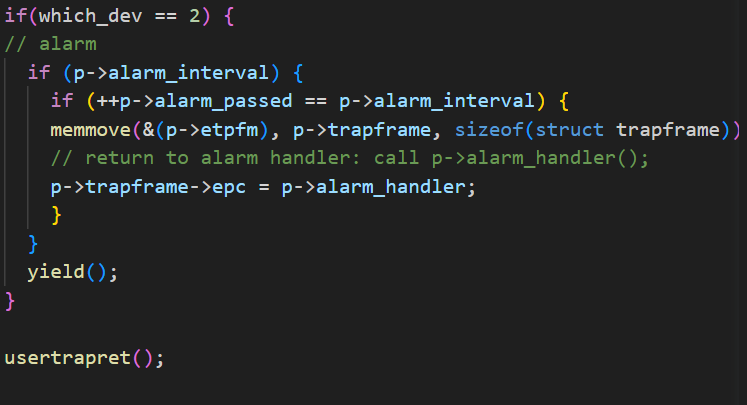
在 proc 结构体里添加需要的字段（间隔时长、已运行时长计数、处理函数地址(vm)、保存 alarm 前的寄存器）：



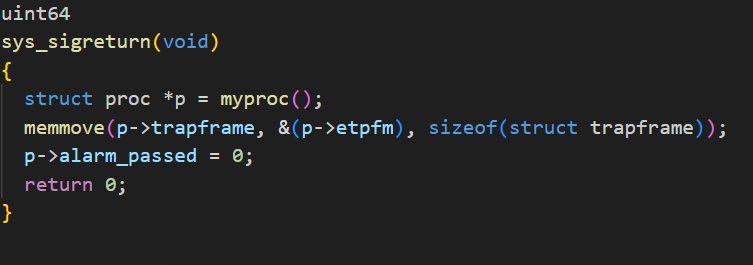
添加 sys\_sigalarm 系统调用：接收参数，为进程设置 alarm 的间隔时长和处理函数：



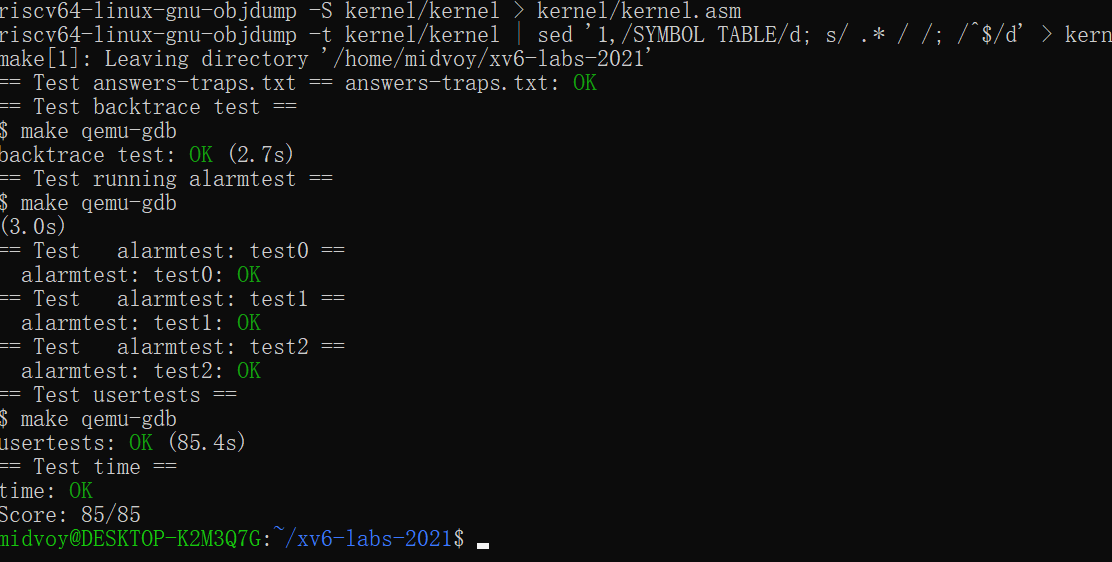
在 usertrap中处理时钟中断时，如果进程需要 alarm 就保存当前 trapframe 到 etpfm，调用 handler （把 handler 地址放到 trapframe->epc，回到用户空间之后就会运行该函数）：



实现 sys\_sigalarm 系统调用，恢复 alarm 前的 trapframe（回到用户空间就会接着 alarm 之前的 PC 开始运行），把 alarm\_passed 计数器置为零（允许下一次 alarm）：



即可完成实验，测试：



## 实验中遇到的问题和解决办法

在这个实验中，我遇到的问题包括不知道哪些状态需要保存下来、保存到什么地方去、保存之后怎么办等等。但是真正把这一个实验做下来之后，我对中断也有了更加深刻的理解，对操作系统中断机制也有了更好的认识。

## 实验心得

本次实验我掌握了通过中断进行系统调用的过程及其所发挥的作用。了解了简单的汇编语言，知道汇编语言是如何发挥作用的。

# **Lab5: Copy-on-Write Fork for xv6**

## Implement copy-on write(hard)

## 实验目的

理解什么是 Copy On Write（COW），其思想和作用是什么。熟悉简单 fork() 系统调用的工作机制，提出一种 COW 的实现手段。理解 xv6 中的 fork 函数，并修改内核程序，实现 COW。

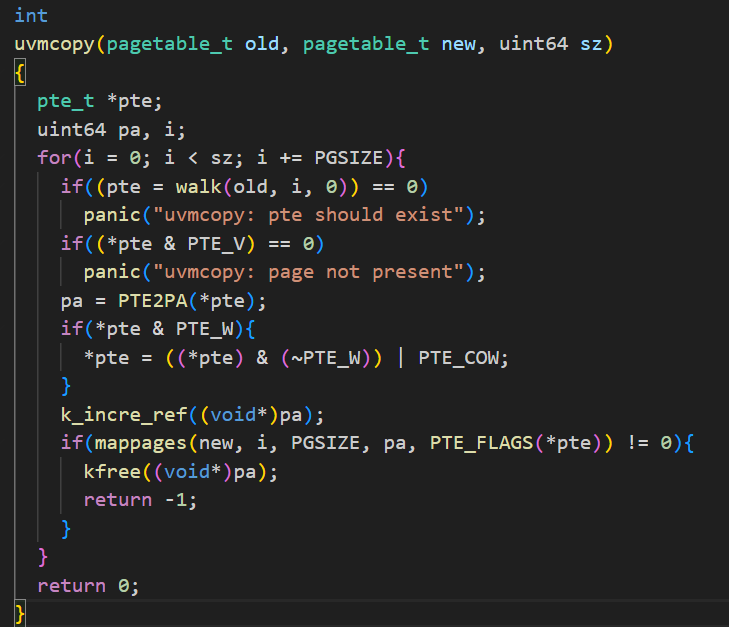
## 实验步骤

fork()与内存有关的是uvmcopy(p->pagetable, np->pagetable, p->sz)语句。uvmcopy同时复制了页表和物理内存。那么我们修改fork使用的uvmcopy()，使其只复制页表，而不复制内存，并清除PTE\_W，设置PTE\_C。

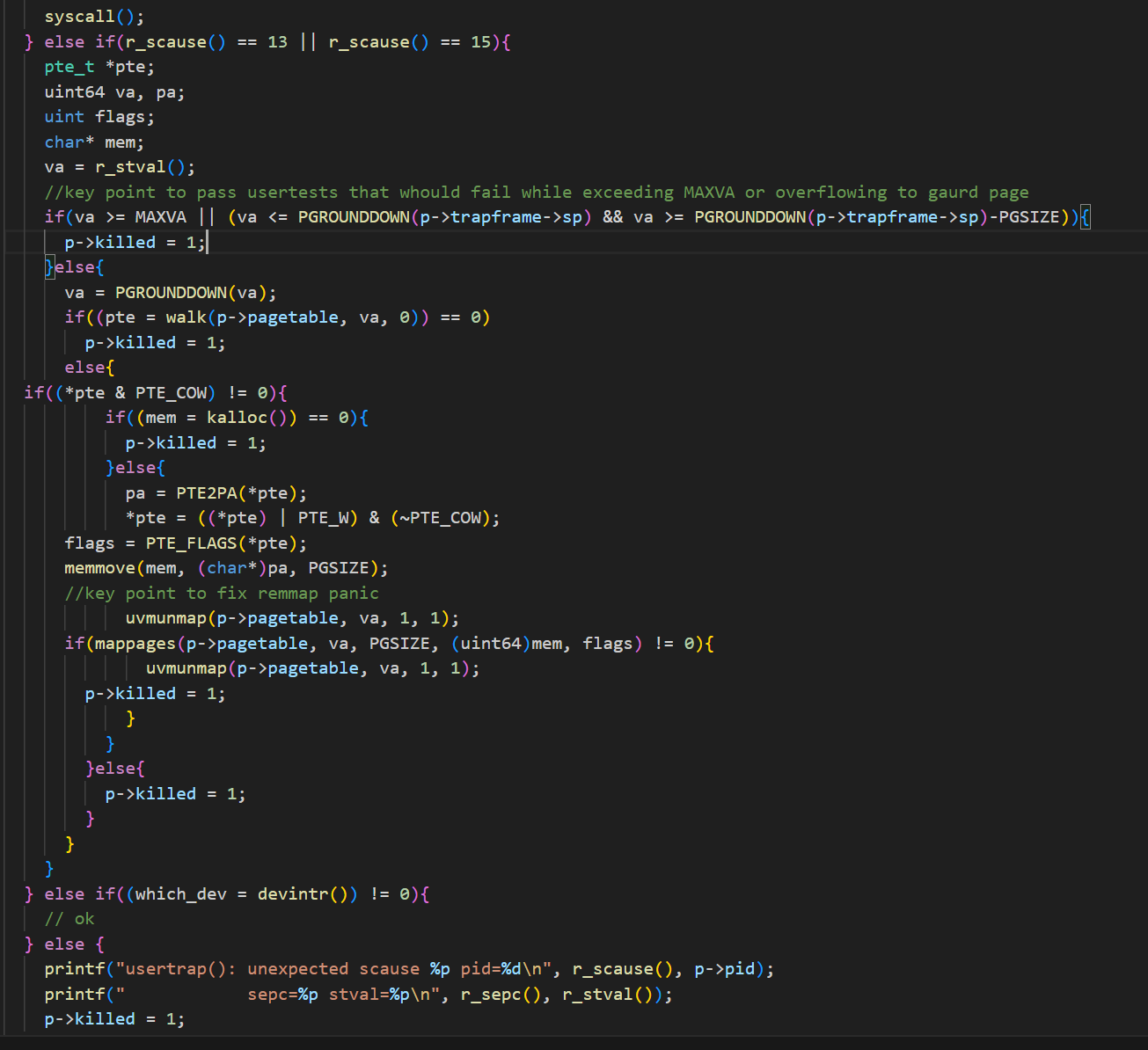
修改uvmcopy()将父进程的物理页映射到子进程，而不是分配新页。在子进程和父进程的PTE中清除PTE\_W标志。，但是得加一个标准位，使用RISC-V PTE中的RSW位来实现此目的：



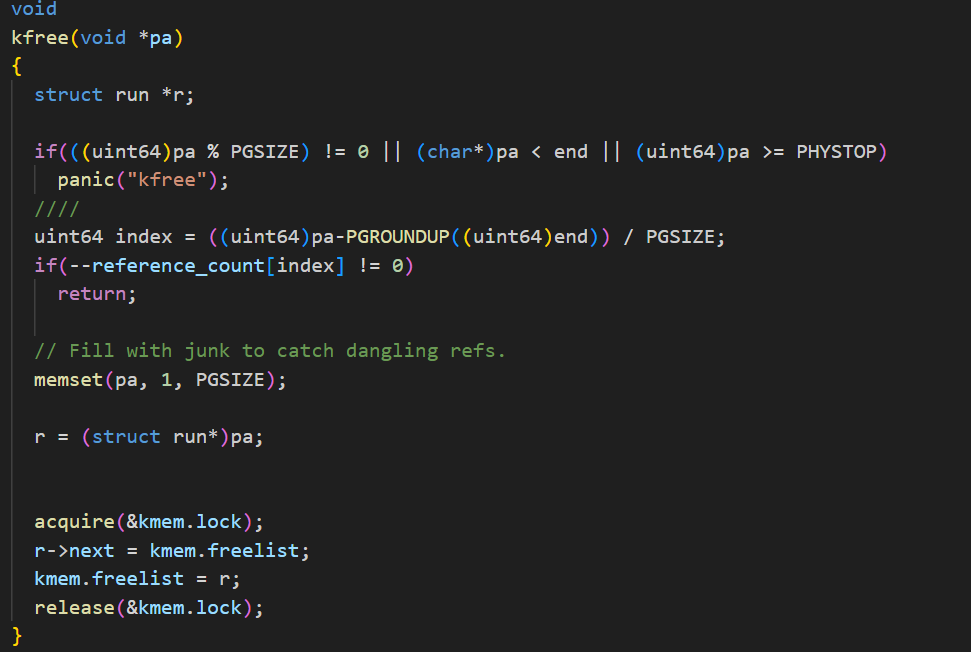
修改uvmcopy()：

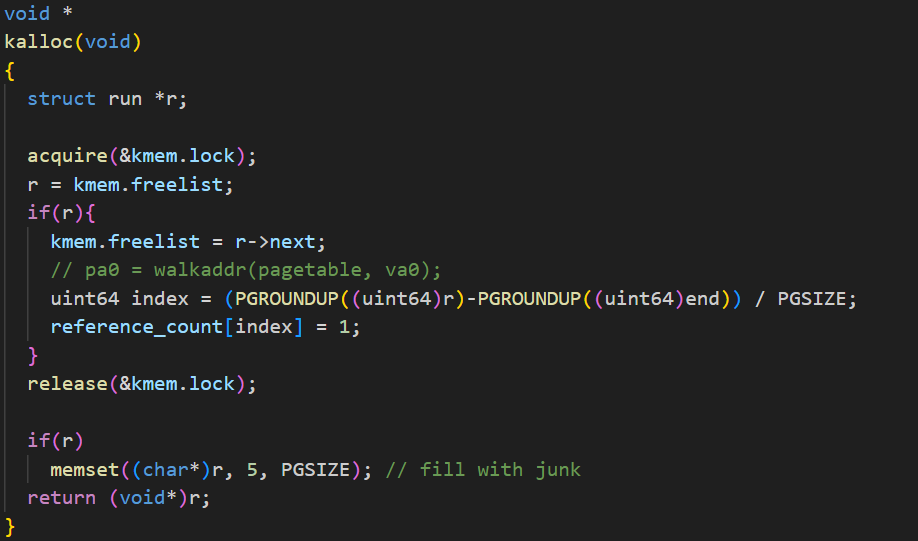


修改usertrap，处理页面错误,COW页面出现页面错误时，使用kalloc()分配一个新页面，并将旧页面复制到新页面，然后将新页面添加到PTE中并设置PTE\_W：

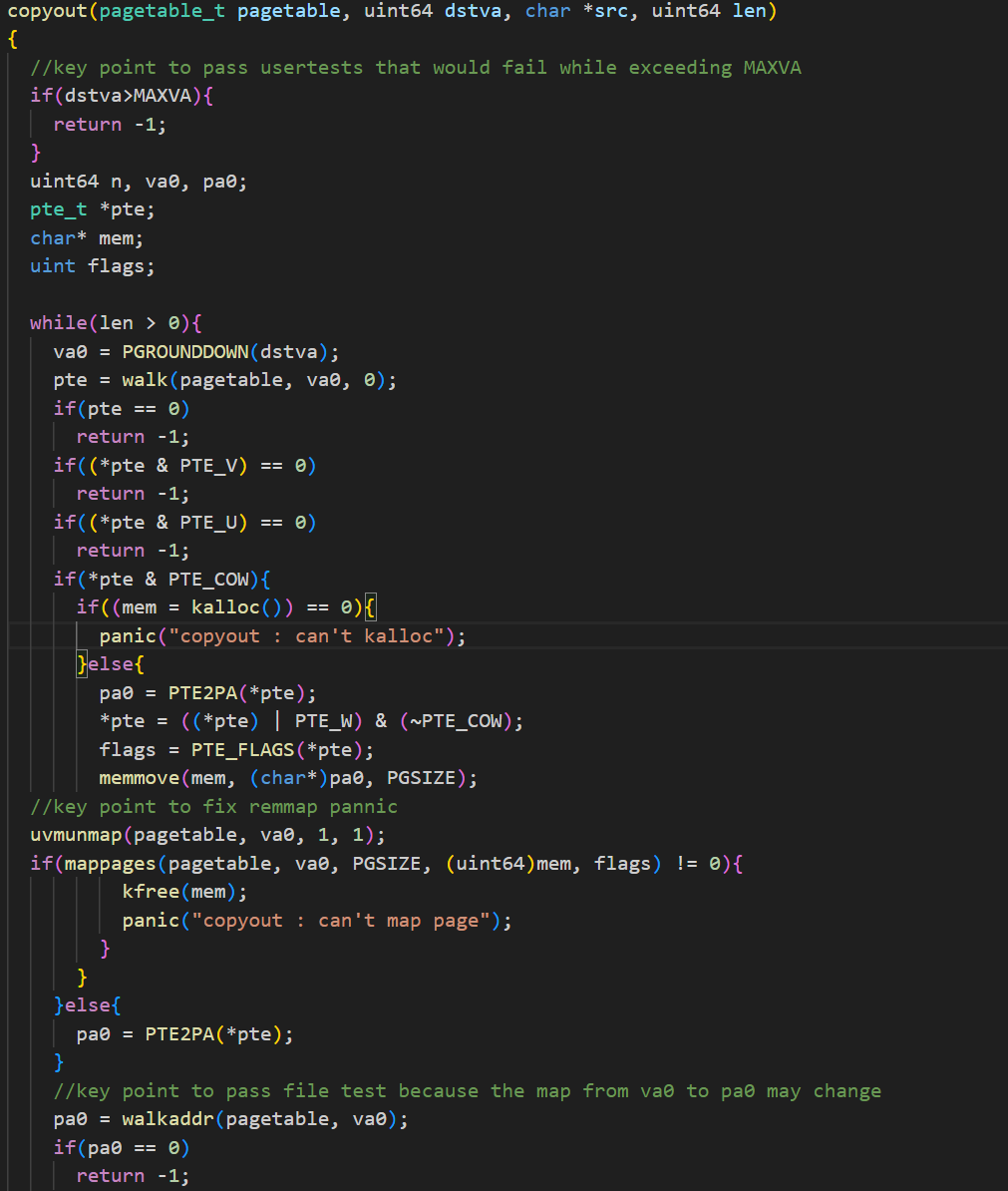


修改kalloc（赋内存函数）和kfree（销毁内存函数）函数，在kalloc中初始化内存引用计数为1，在kfree函数中对内存引用计数减1，如果引用计数为0时才真正删除：

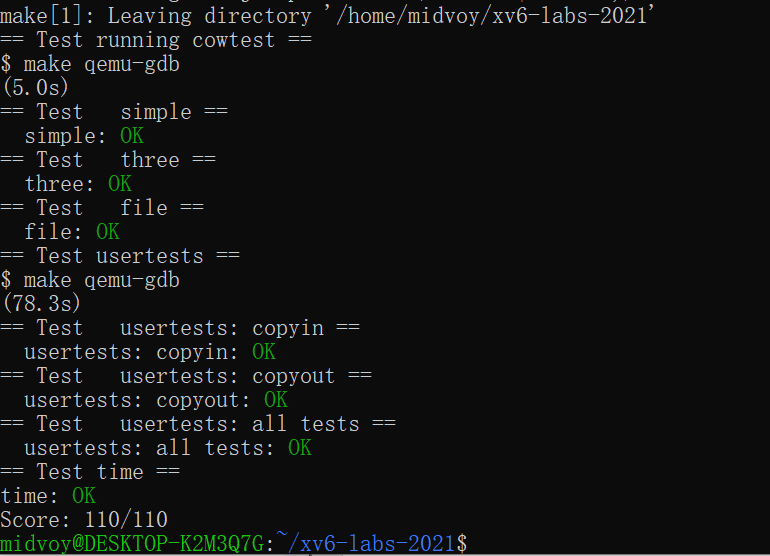




修改copyout(),每次都对其中的 va0 尝试 cow\_alloc，不用管是否成功，后面的逻辑会处理异常。这里只需要改动 copyout 而不需要改 copyin 是因为前者是内核拷贝到用户，是会对一个用户页产生写的操作，而后者是用户拷到内核，只是去读这个用户页的内容，COW页允许读。



即可完成实验：



## 实验中遇到的问题和解决办法

该实验前我并不理解fork()系统调用，首先需要理解系统中的 fork 函数原型——也就是 Unix 系统中 fork 所完成的工作，为此我查阅了一些资料。

## 实验心得

本次实验是关于内存分配策略的设计。本次实验实现了按需复制的 fork 策略，这也是实现了 fork 系统调用的操作系统中一般都会实现的优化方式，通过这次实验，我对操作系统中内存分配的方式和机制有了更深刻的体会，完成 COW 功能的过程也帮助我回顾了一块内存的分配、使用、销毁回收这个完整的生命周期。

# **Lab6: Multithreading**

## Uthread: switching between threads (moderate)

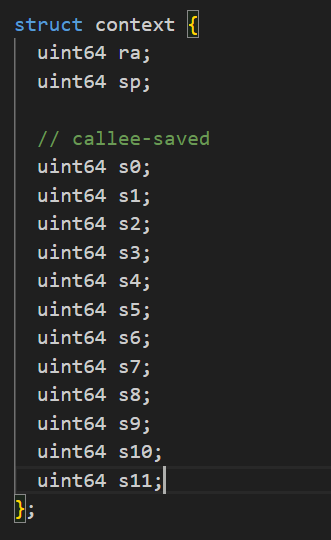
## 实验目的

为用户级线程系统设计上下文切换机制，然后实现用户级进程的创建和切换。

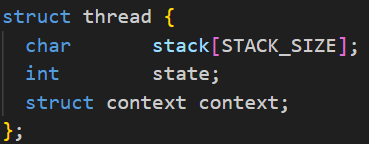
## 实验步骤

实验要求修改 user/uthread.c 和 user/uthread\_switch.S 程序，从而实现进程间的切换需要保存被替换进程的上下文和恢复进入运行态进程的上下文。

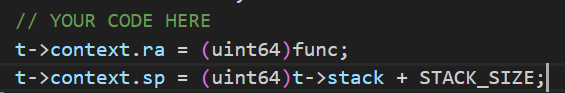
因此需要有一个结构体来存储需要保存的内容，即进程运行时的寄存器：



这个结构体应该由进程本身保存和维护，因此要添加到进程结构体中：



在进程切换时，只需要交换调用方保存的进程上下文。ra 寄存器代表的是返回地址（return address），sp 寄存器代表的是栈底指针（stack pointer），这两个寄存器并不需要在切换进程时交换，而是由系统更新：



## 实验中遇到的问题和解决办法

之前并没有编写过多线程的程序，只知道多线程程序可以运用多个处理器，可以达到提高速度的效果。通过查阅相关资料解决了问题。

## 实验心得

本次实验使我熟悉了多线程编程，理解了进程切换方式和执行顺序。

## Using threads (moderate)

## 实验目的

探索使用哈希表的线程和锁的并行编程，使用 UNIX pthread 线程库实现一个线程安全的哈希表。

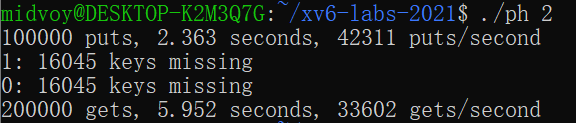
## 实验步骤

本实验是一个使用 Unix 环境 pthread 库来学习多线程编程的练习。

按照说明，先运行指令 make ph，之后运行./ph 1，结果如下：



接下来运行./ph 2，发现有丢失的键，说明实现有误：



解决该问题需要考虑为何丢失了一些键。因为同时有多个线程工作，所以当一个进程往哈希表中填入一个键时，可能有其他进程也希望填入一个键。如果两个键的哈希值不相等，那么插入是没有问题的，两个键都可以成功在不同的哈希筒中出现。但是如果两个键哈希值相同，那么其插入的位置一样，就会导致其中一个（相对后写入的）将另一个新插入的键覆盖，从而导致了键的丢失。

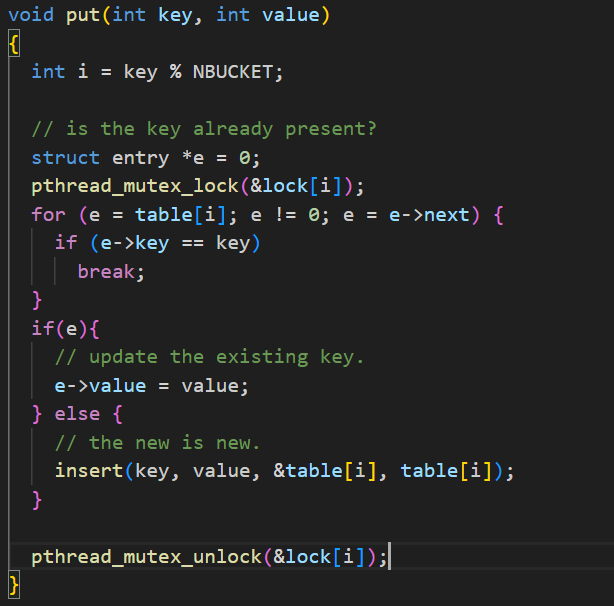
正确的做法是分别加锁：对于每个哈希筒，每次只允许一个进程访问。以下是实现步骤：

声明锁数组，并在 main 函数中初始化。

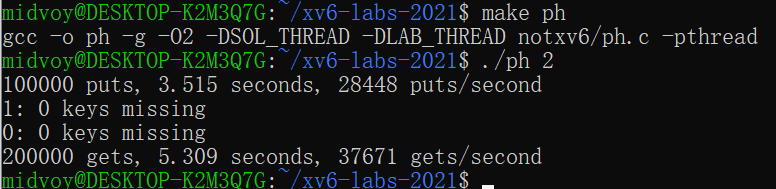




在 put 函数中使用锁：



完成修改后运行./ph 2，没有丢失的键，结果如下：



## 实验中遇到的问题和解决办法

实验前并不知道如何添加锁，通过查阅相关资料对其了解更深了。

## 实验心得

在实验中，实现了进程切换、运用锁来完成线程间的互斥，理解了锁的含义，能够判断何时、何处应该加锁。

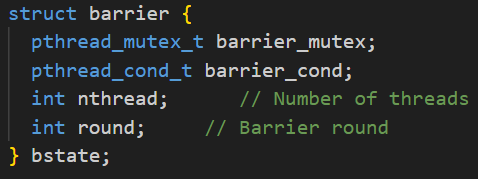
## Barrier(moderate)

## 实验目的

实现 barrier 函数。其作用是：当进程到达 barrier 函数调用时，会开始等待其他进程，当所有进程都到达 barrier 时，才停止等待。

## 实验步骤

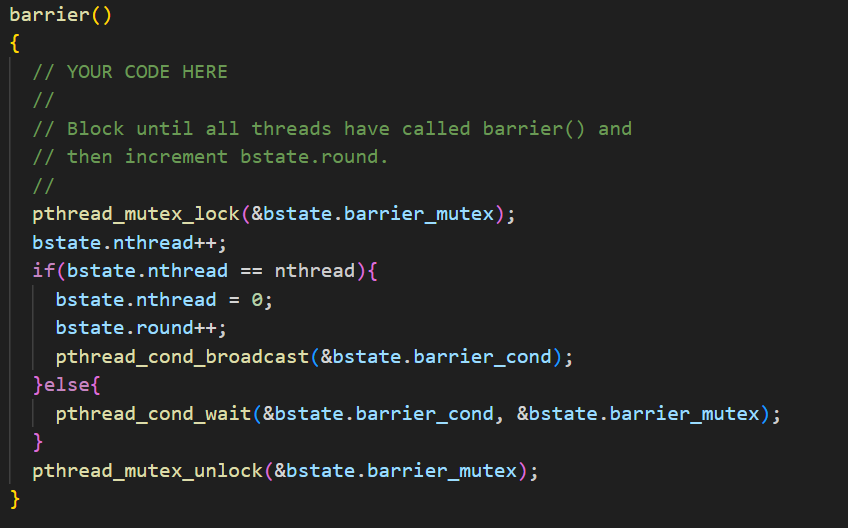
barrier 函数会在所有进程都调用 barrier 时才解锁所有进程。在实现开始前，发现已有定义的数据结构 bstate：



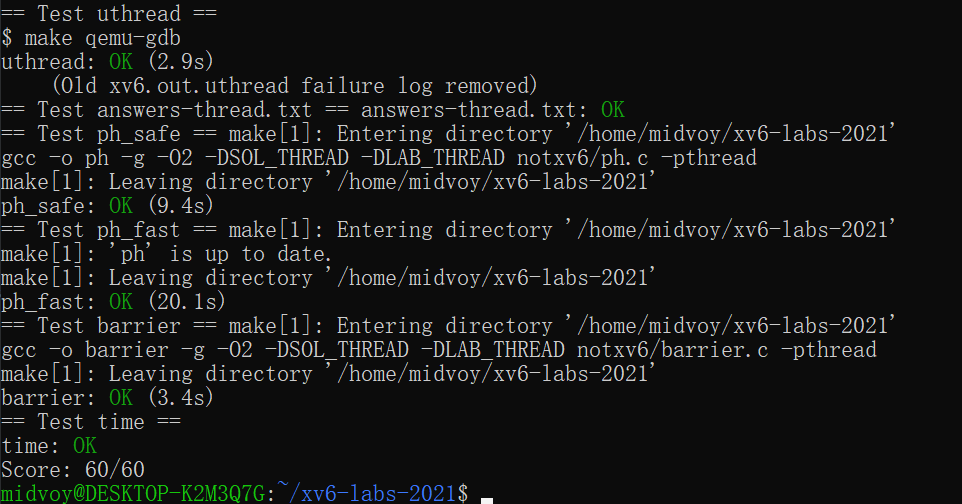
通过比较 bstate.nthread 与全局变量 nthread，可以分别判断是否使进程等待或者唤醒其他进程。

库函数 pthread\_cond\_wait(&cond, &mutex); 使进程释放mutex 锁并进入睡眠，等待 cond 将其唤醒。pthread\_cond\_broadcast(&cond); 将所有等待 cond 的进程唤醒。

因此，barrier 函数的实现如下：



测试实验成绩：



## 实验中遇到的问题和解决办法

实验前对系统的使用方法不熟悉，通过上手慢慢了解了锁和条件信号量的使用方法。

## 实验心得

在实验中，我运用锁和条件信号量来完成多线程之间的同步。实验加深了我对于多线程的理解，提供了练习多线程编程的机会。

# **Lab7: networking**

## Your Job (hard)

## 实验目的

在本实验室中将为网络接口卡（NIC）编写一个xv6设备驱动程序。

## 实验步骤

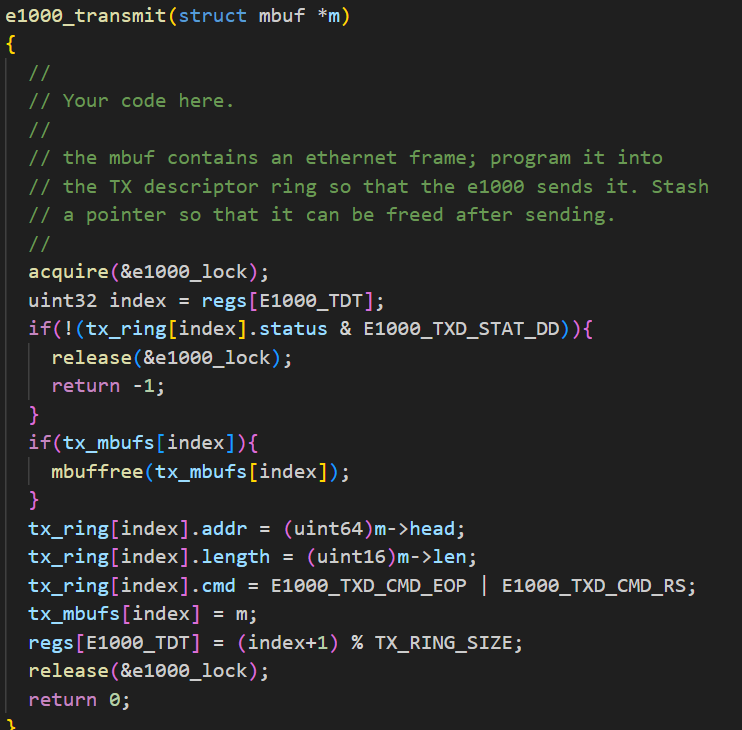
lab 要求简单来说就是实现网卡驱动的 transmit 和 recv 功能，

首先是 transmit 功能，这里比较麻烦的是确定 tx\_desc.cmd 的值，查阅下文档即可。文档中标注：

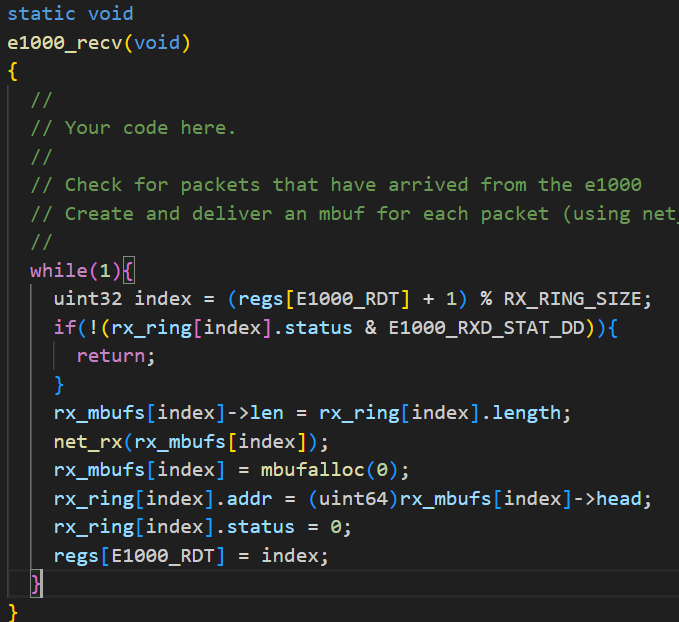
VLE, IFCS, and IC are qualified by EOP. That is, hardware interprets these bits ONLY when EOP is set.

Hardware only sets the DD bit for descriptors with RS set.

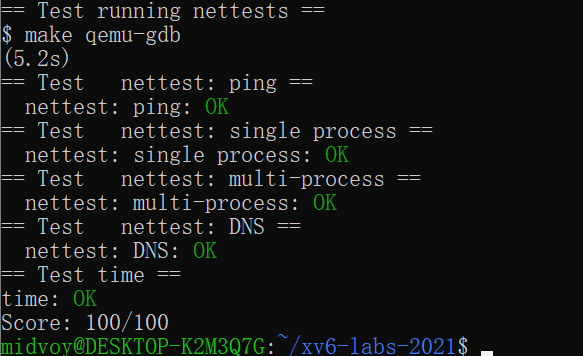
因此设置 DD bit 和 EOP bit 即可。其实头文件中关于 cmd 也只定义了这 2 个 bit。



接着实现 recv 功能，这里需要注意的是，recv 在一次中断可能收到多个 packet。因此我们需要从 tail 处开始，依次遍历，检查 DD bit 是否为 1，为 1 表示该描述符可以进行 net\_rx 操作。其次需要注意要在 net\_rx 前释放锁，否则会引起 panic。



即可完成实验：



## 实验中遇到的问题和解决办法

e1000\_recv的实现过程中有一句：重新分配一个新的mbuf，我理解出错，去重新创建了一个新的mbuf，根本上是因为我没理解到rx\_ring是被循环使用的，这个循环数组一直是用来写入，传递packet的

## 实验心得

本次实验看起来文档、字都很多，但实际的代码并不难，本次的提示就能作为伪代码，完全可以按着来写。难点应该在于理解整个收发包（recevice/transmit packet）的过程，cpu、网卡（ethernet）、RAM（buffer的存放处）这三者是如何通过xv6操作系统进行交互的。xv6得依靠network stack（网络栈）实现收发数据，即通过network stack收发packet。

# **Lab8: locks**

## Memory allocator (moderate)

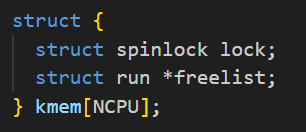
## 实验目的

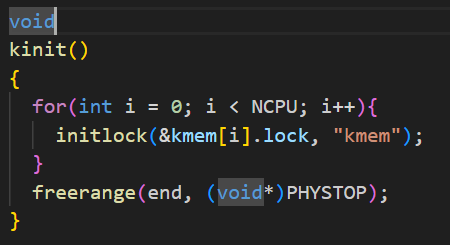
理解锁竞争，包括锁竞争产生的原因以及解决方法。重新设计系统管理内存的方式，实现每个 CPU 管理一个空闲链表和一个锁

## 实验步骤

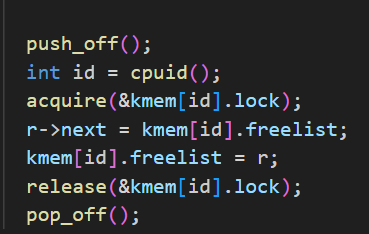
实现内存分配器，减少竞争，主要实现思路是原先只有一个freelist，一个lock给八个cpu竞争使用，而我们需要重新设计为八个freelist、八个lock给八个cpu使用。按照要求，一个 CPU 的空闲链表为空时，需要向别的 CPU“借”一些空间以保证所有空间都能正常分配。

首先声明符合 CPU 个数的锁-链表结构，并在函数 kinit 中正确初始化：

其中 NCPU 在 param.h 中声明，值为 8。  




在函数中，需要完成对 CPU 维护的锁-链表结构申请空间。在 CPU 空间不足时，需要从其他 CPU 维护的结构中“借”一些空间来完成这次的分配。 首先要获取 CPU 的 ID，在这一个过程中不能允许进程被其他进程抢占，所以要调用 push\_off 和 pop\_off 函数来关闭/打开中断：



## 实验中遇到的问题和解决办法

实验前并不熟悉如何实现内存分配器，减少竞争，通过查阅资料解决。

## 实验心得

本实验我理解锁了竞争，包括锁竞争产生的原因以及解决方法。理解通过增加锁的数量来降低锁竞争的原理，并应用到程序中。

## Buffer cache (hard)

## 实验目的

修改系统的 IO 缓冲区，通过散列的方式完成对缓冲区和锁的分割。

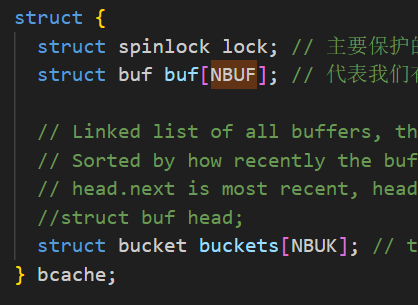
## 实验步骤

为了把一个大的cache池划分成几个小的cache池来减少锁争用，可以使用哈希的方法。把所有cache块按块号散列到不同的哈希桶中。查找cache块时就去相应的哈希桶中查找。

首先要决定哈希桶的个数，一个合适大小的质数是一个理想的选择，因此依照提示选择13:



接下来需要修改原先 buffer cache 的定义，以适应使用散列表的方式。用数组装填了缓冲区的头部、筒对应的磁盘块以及锁：



此外，这些值需要被正确地初始化：

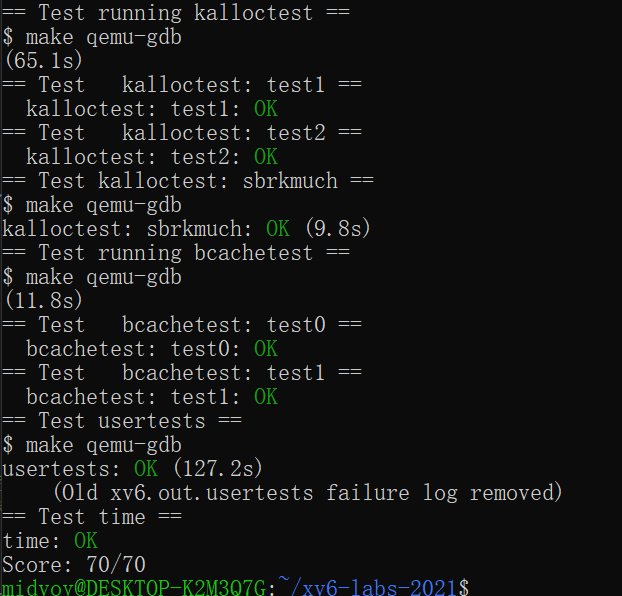


每次访问的缓冲区可以用 blockno 算出：





在bget和brelse 函数中，只需要将所有bcache.lock 改为bcache.buckets[buk\_id].lock，把 bcache.head 改为bcache.buckets[buk\_id].head，即可完成该实验:



## 实验中遇到的问题和解决办法

真正实现之前，一定要修改 kernel/param.h 里的宏 FSSIZE，改大成 10000,由于实验要求没有指出这一点，导致我在 usertests 的 writebig 测试点卡在 PANIC: balloc: out of blocks 很久

## 实验心得

本实验是关于并行编程中问题的性能优化。在处理多线程或多进程中资源抢占而导致的锁竞争现象，常用的做法是通过分割资源并且分别加锁。锁的数量越多，单个锁上的冲突就越少（但是也要考虑锁的数量增多的负面影响）。我在实验中体会了分割资源来使锁竞争得到缓解、提高资源并行利用率。

# **Lab9: file system**

## Large files (moderate)

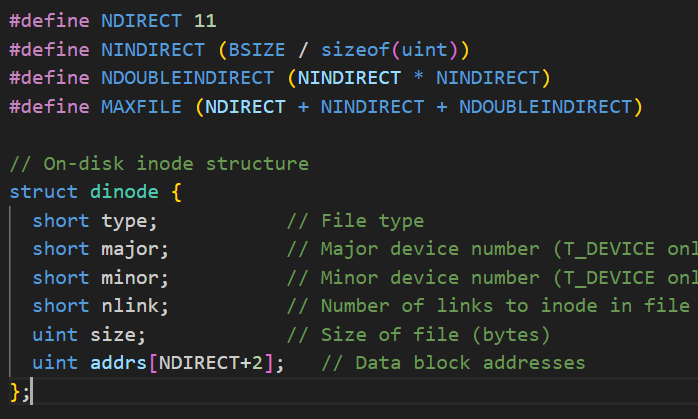
## 实验目的

理解 UNIX 系统以及 xv6 系统组织文件的方式，包括 inode 数据结构以及地址的映射关系。修改文件系统使其支持更大的文件存储。

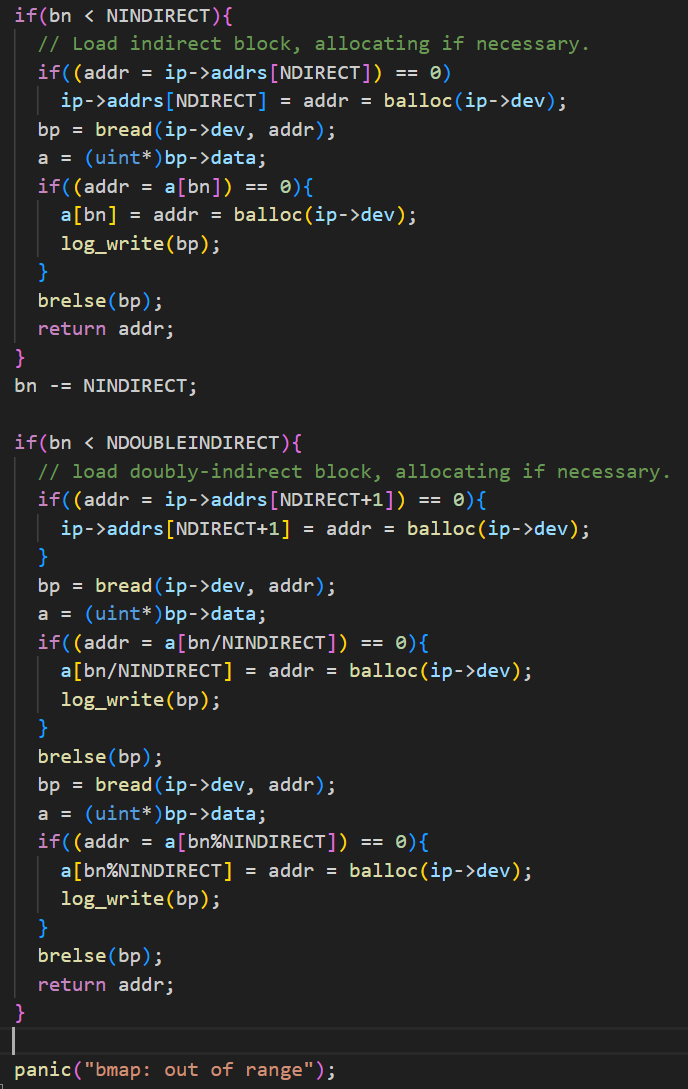
## 实验步骤

这里我们主要关心的是指向数据块的指针。原来的xv6系统里，有12个direct pointer和1个single indirect pointer，一共能容纳268个硬盘块，现在要求改成：前11个指针是direct pointer，第12个指针改成single indirect pointer，第13个指针改成double indirect pointer，这样就扩大文件容量到65803块。

首先修改inode的相关定义：



接下来要修改bmap()函数。bmap()负责的是：Return the disk block address of the nth block in inode ip.所以bmap会检查参数bn位于哪个区间，如果位于direct pointer之内就直接访问，如果没有对应数据块的话就调用balloc()分配一个新的硬盘块。如果超出了direct的范围，就把bn减去NDIRECT的偏移量，先访问addrs[11]指向的一级指针块，使用一级指针块里面的指针访问数据块。同理可以实现二级指针块：



然后修改itrunc()，释放所有数据块。itrunc()里面也给我们举了释放single indirect pointer的例子。直接模仿：



## 实验中遇到的问题和解决办法

第一部分修改宏的时候，没有把所有引用宏的地方都对应上，因此报了奇怪的错误

panic : virtio\_disk\_intr status

以后得注意，在修改已有代码段的时候，需要注意所有引用在修改后还是正确的

## 实验心得

本次实验理解了 UNIX 系统以及 xv6 系统组织文件的方式，包括 inode 数据结构以及地址的映射关系。

## Symbolic links (moderate)

## 实验目的

增加一个通过符号链接的系统调用。将目标符号与系统路径链接，使对符号文件的操作都同步到对相应路径文件的操作。

## 实验步骤

这一个实验是要实现符号链接，符号链接就是在文件中保存指向文件的路径名，在打开文件的时候根据保存的路径名再去查找实际文件。symlink的系统调用就是创建一个inode，设置类型为T\_SYMLINK，然后向这个inode中写入目标文件的路径就行了。

首先创建一个新的系统调用：









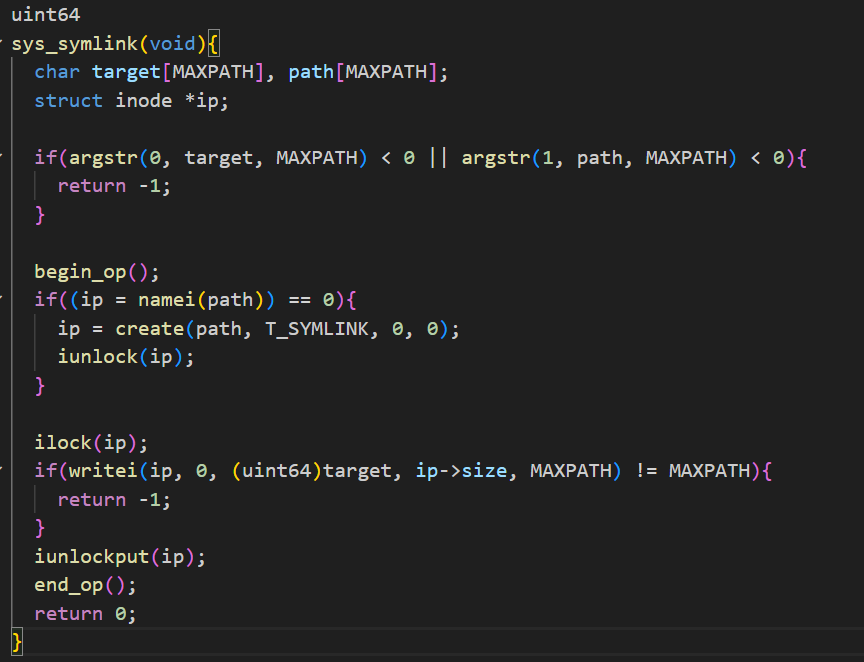


添加对应的宏，支持符号链接文件类型和不跟随打开模式：

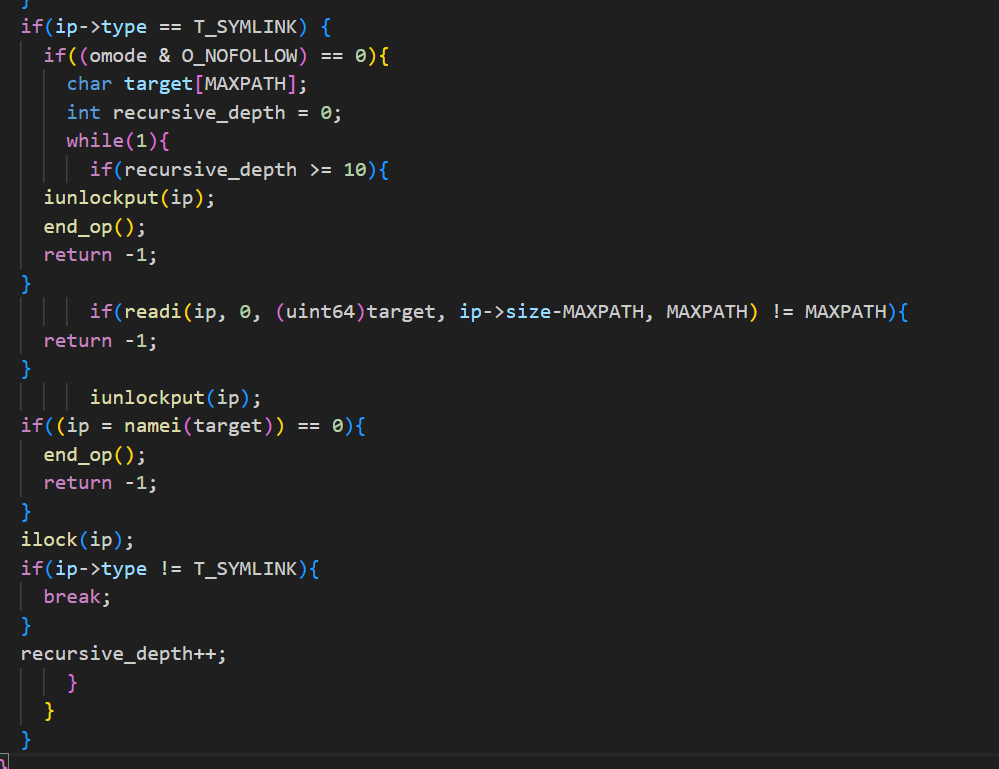




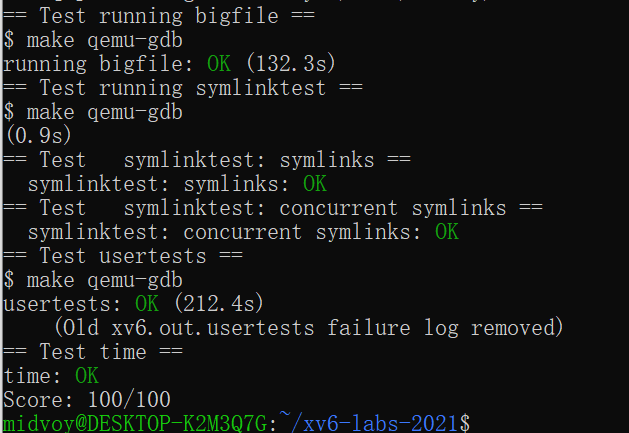
添加并实现系统调用 sys\_symlink 函数。其逻辑是：读取两个字符串作为参数，一个是链接到的目标文件路径 target，另一个是新建链接的路径 path。在路径下创建一个新文件并且将目标文件名填入新文件中：



修改 open 系统调用，以处理打开符号链接文件。打开符号链接文件时递归打开其指向的最终目标文件。以下代码位于 sys\_open 函数中：



即可完成实验：



## 实验中遇到的问题和解决办法

在实现的过程中最大的困难在于理解“软链接也是一种文件”这一概念。这个实 验让我加深了对电脑上那些左下角有箭头符号的文件本质是什么这一问题理解。

## 实验心得

本次试验是关于 xv6 系统文件系统的，拓展了可以索引的文件大小，也创建了符号链接这种软链接方式。

# **Lab10: mmap**

## mmap (hard)

## 实验目的

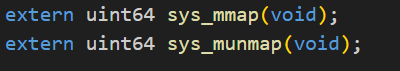
添加 mmap 与 mumap 两个系统调用。前者将一个文件内存映射到进程的地址空间，后者取消已有地址空间的映射。

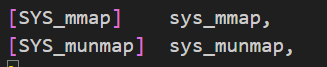
## 实验步骤

1. 首先实现 mmap 系统调用

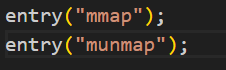
添加系统调用：



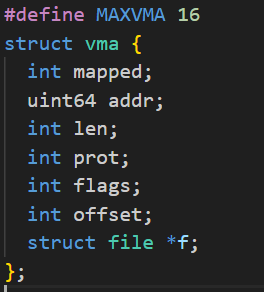






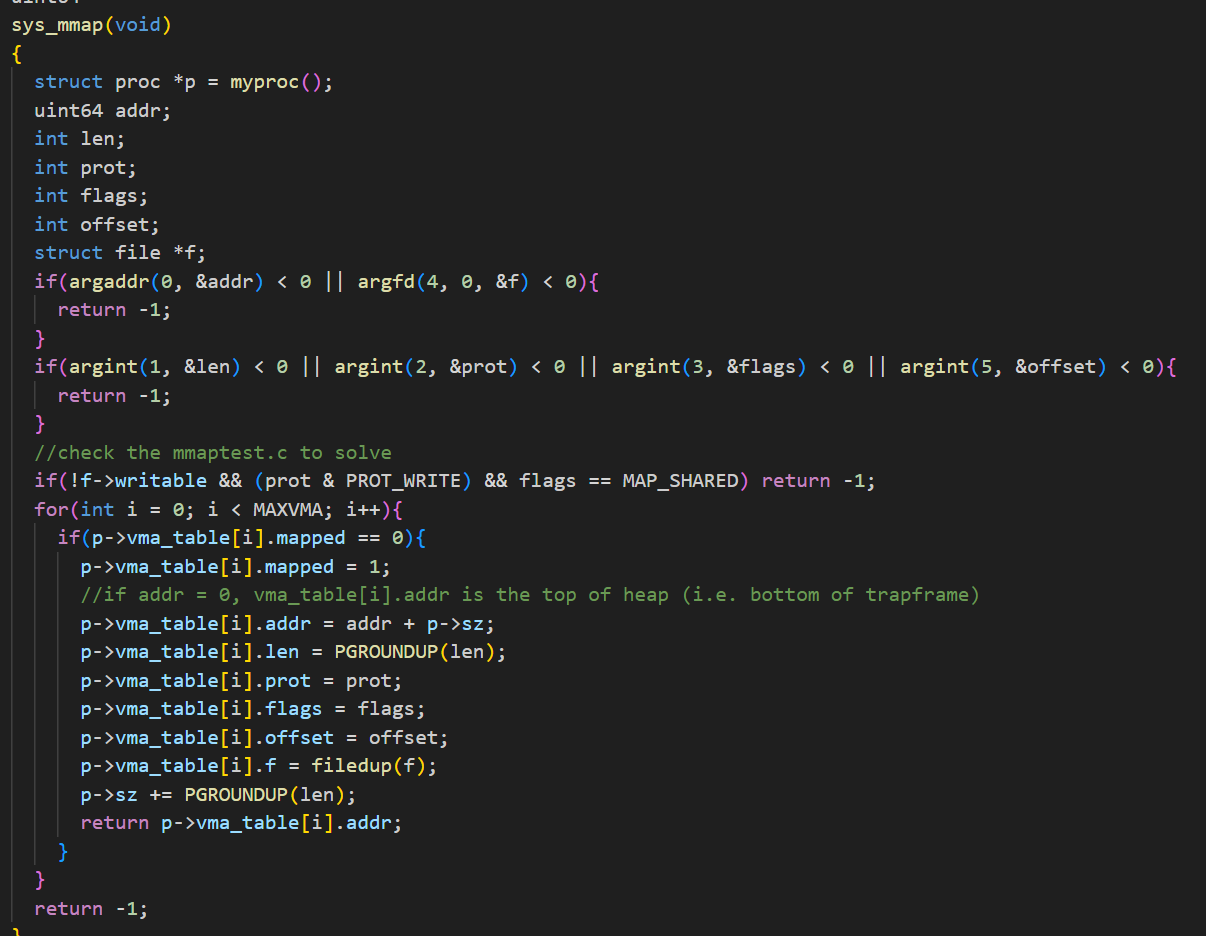


添加 vma 数据结构的定义，该结构记录地址，长度，权限，文件等由 mmap 创建的虚拟内存范围。定义后，给进程的数据结构添加一个 vmas 数组，并在allocproc 中初始化：

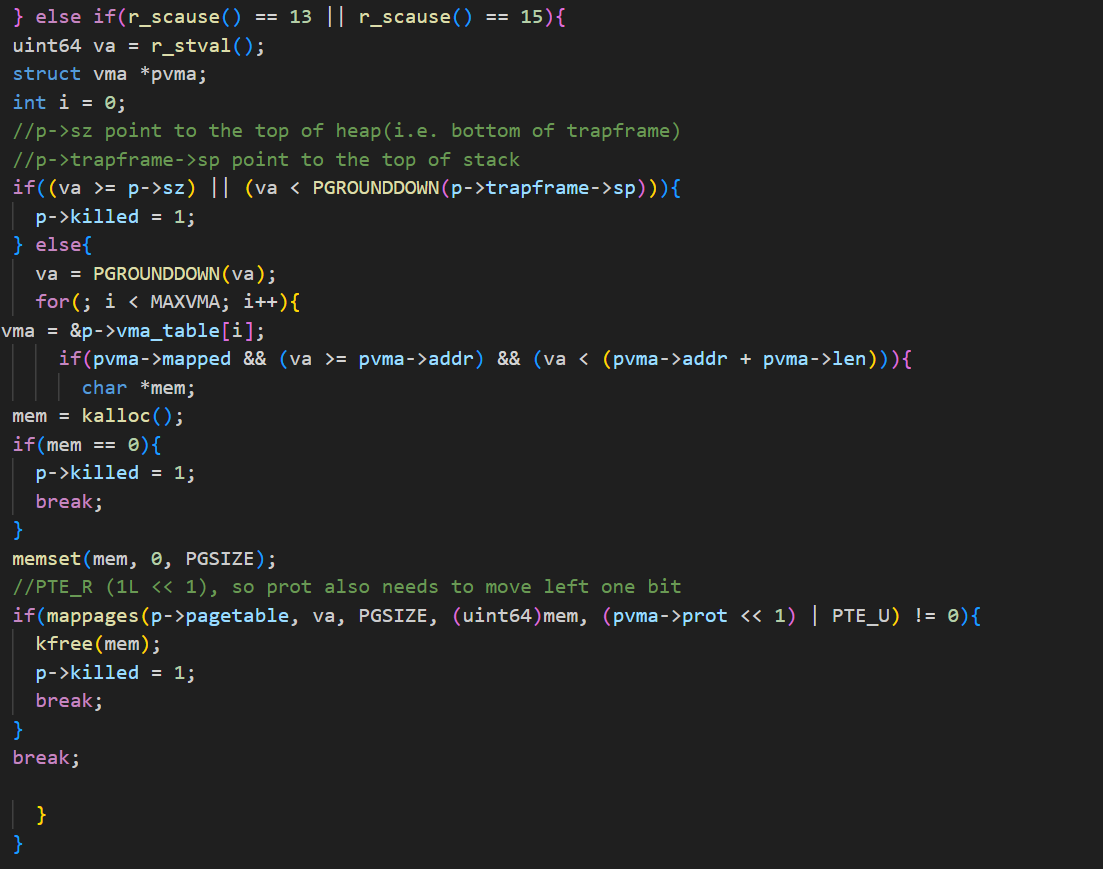




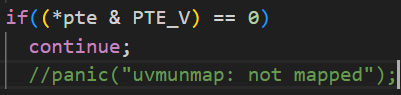
添加 mmap 系统调用的函数体。首先需要读取参数，之后在当前进程的 vmas字段中寻找一个空位，填充其对应的字段，就完成了记录该映射文件的工作：

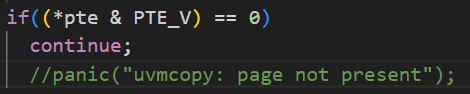


由于使用了懒加载方式，在访问未加载界面时，会产生一个缺页中断。我需要在 usertrap 函数中处理该中断来正确地加载页面。首先，缺页中断的地址会作为 r\_scause 的返回值；获取地址后，查找进程 vmas 字段中包含该地址的虚拟地址块，并对该地址所在的页面分配一个页大小的空间：



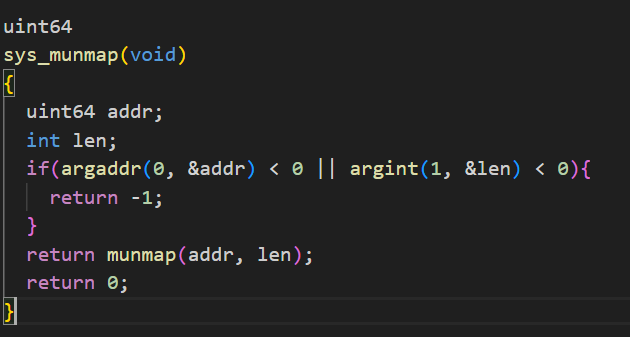
中断处理的懒加载方式允许了缺页错误的产生，因此在 uvmcopy 和uvmunmap 中需要删去对没有映射页面的报错：

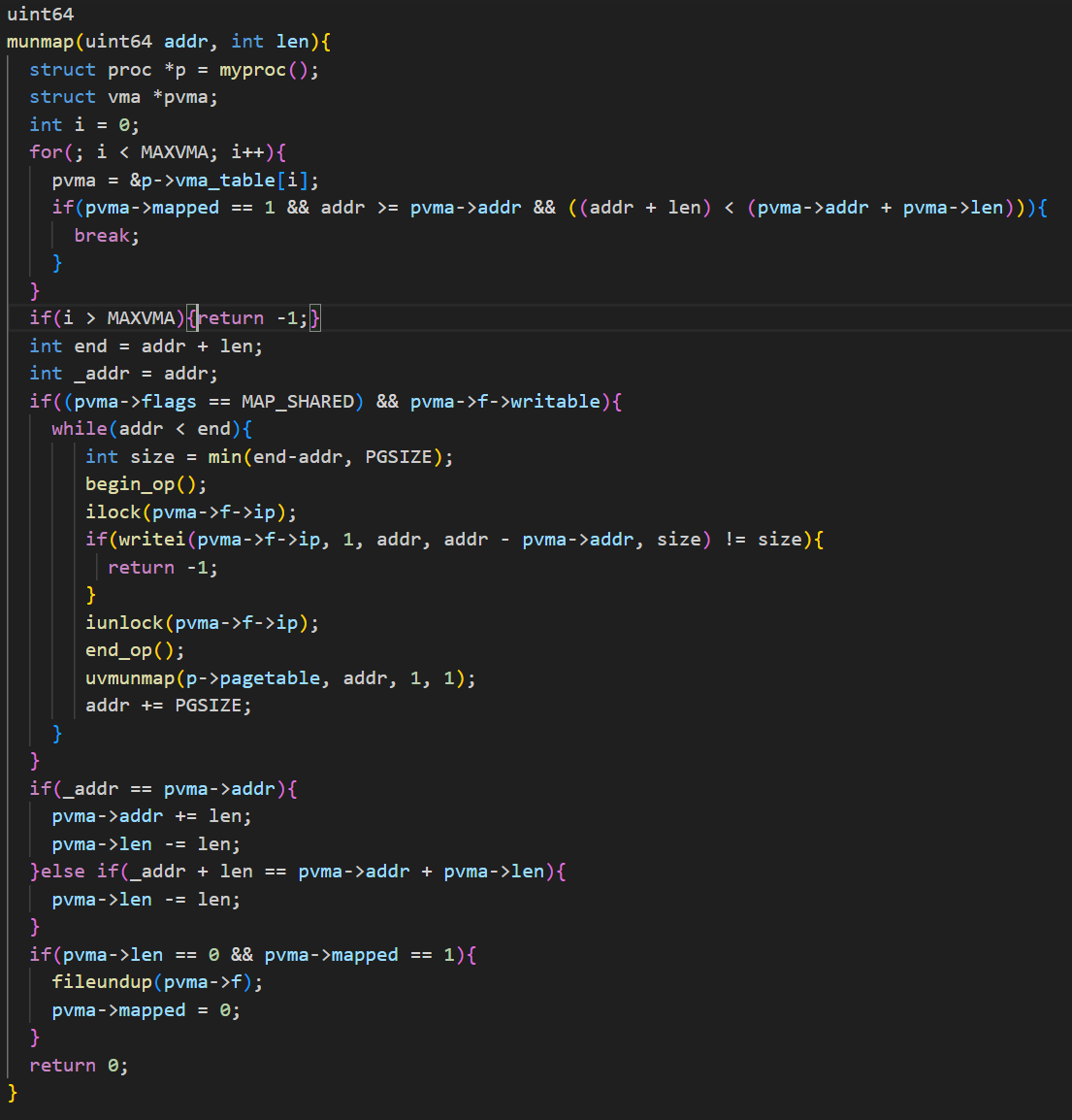




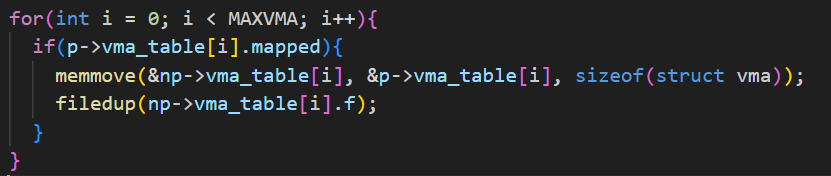
2.然后实现munmap系统调用

实现 munmap 系统调用。首先要获取系统调用的参数：取消映射的地址及长度。然后检查当前进程的 vmas 中有没有包含该地址的记录。对于找到的 vmas 记录，若标记为 MAP\_SHARED，则需要将文件块直接写回到磁盘。之后对于整个映射释放和释放一部分映射的操作分别完成释放：

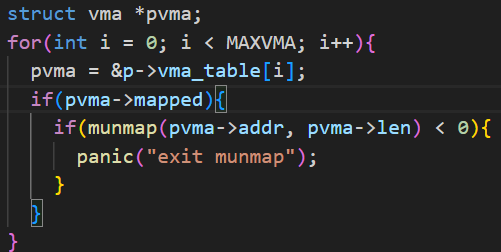




3. 修改 fork 函数，使子进程复制父进程的文件映射



4.修改 exit 函数，当退出进程时写回并释放相应的文件映射



运行即可完成实验：



## 实验中遇到的问题和解决办法

在实验的提示中详细地列举了需要完成的对源代码的修改和添加，但是数目很多，稍有遗漏就会产生难以排查的 bug。经过不断排查错误和查阅网上相关资料解决。

## 实验心得

本次实验也是关于系统与文件交互的内容，通过实验，实现了一个磁盘文件到虚拟内存地址的映射方式。这次实验让我对操作系统中的内存设计和组织理解更加深刻，让我了解了使用虚拟内存的原因和好处，理解创建和释放文件映射的方式，梳理了文件内存映射的生命周期。