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](#_Toc143025353)

[1.1 Boot xv6 (easy) 6](#_Toc143025354)

[1.1.1 实验目的 6](#_Toc143025355)

[1.1.2 实验步骤 6](#_Toc143025356)

[1.1.3 实验中遇到的问题和解决办法 6](#_Toc143025357)

[1.1.4 实验心得 6](#_Toc143025358)

[1.2 sleep (easy) 6](#_Toc143025359)

[1.2.1 实验目的 6](#_Toc143025360)

[1.2.2 实验步骤 6](#_Toc143025361)

[1.2.3 实验中遇到的问题和解决办法 8](#_Toc143025362)

[1.2.4 实验心得 8](#_Toc143025363)

[1.3 pingpong (easy) 8](#_Toc143025364)

[1.3.1 实验目的 8](#_Toc143025365)

[1.3.2 实验步骤 8](#_Toc143025366)

[1.3.3 实验中遇到的问题和解决办法 9](#_Toc143025367)

[1.3.4 实验心得 10](#_Toc143025368)

[1.4 primes (moderate)/(hard) 10](#_Toc143025369)

[1.4.1 实验目的 10](#_Toc143025370)

[1.4.2 实验步骤 10](#_Toc143025371)

[1.4.3 实验中遇到的问题和解决办法 11](#_Toc143025372)

[1.4.4 实验心得 11](#_Toc143025373)

[1.5 find (moderate) 12](#_Toc143025374)

[1.5.1 实验目的 12](#_Toc143025375)

[1.5.2 实验步骤 12](#_Toc143025376)

[1.5.3 实验中遇到的问题和解决办法 13](#_Toc143025377)

[1.5.4 实验心得 13](#_Toc143025378)

[1.6 xargs (moderate) 14](#_Toc143025379)

[1.6.1 实验目的 14](#_Toc143025380)

[1.6.2 实验步骤 14](#_Toc143025381)

[1.6.3 实验中遇到的问题和解决办法 16](#_Toc143025382)

[1.6.4 实验心得 16](#_Toc143025383)

[**2.** **Lab2: system calls** 17](#_Toc143025384)

[2.1 System call tracing (moderate) 17](#_Toc143025385)

[2.1.1 实验目的 17](#_Toc143025386)

[2.1.2 实验步骤 17](#_Toc143025387)

[2.1.3 实验中遇到的问题和解决办法 20](#_Toc143025388)

[2.1.4 实验心得 20](#_Toc143025389)

[2.2 Sysinfo (moderate) 20](#_Toc143025390)

[2.2.1 实验目的 20](#_Toc143025391)

[2.2.2 实验步骤 20](#_Toc143025392)

[2.2.3 实验中遇到的问题和解决办法 23](#_Toc143025393)

[2.2.4 实验心得 23](#_Toc143025394)

[**3.** **Lab3: page tables** 23](#_Toc143025395)

[3.1 Speed up system calls (easy) 23](#_Toc143025396)

[3.1.1 实验目的 23](#_Toc143025397)

[3.1.2 实验步骤 23](#_Toc143025398)

[3.1.3 实验中遇到的问题和解决办法 26](#_Toc143025399)

[3.1.4 实验心得 26](#_Toc143025400)

[3.2 Print a page table (easy) 26](#_Toc143025401)

[3.2.1 实验目的 26](#_Toc143025402)

[3.2.2 实验步骤 26](#_Toc143025403)

[3.2.3 实验中遇到的问题和解决办法 27](#_Toc143025404)

[3.2.4 实验心得 28](#_Toc143025405)

[3.3 Detecting which pages have been accessed (hard) 28](#_Toc143025406)

[3.3.1 实验目的 28](#_Toc143025407)

[3.3.2 实验步骤 28](#_Toc143025408)

[3.3.3 实验中遇到的问题和解决办法 30](#_Toc143025409)

[3.3.4 实验心得 30](#_Toc143025410)

[**4.** **Lab4: traps** 31](#_Toc143025411)

[4.1 RISC-V assembly (easy) 31](#_Toc143025412)

[4.1.1 实验目的 31](#_Toc143025413)

[4.1.2 实验步骤 31](#_Toc143025414)

[4.1.3 实验中遇到的问题和解决办法 33](#_Toc143025415)

[4.1.4 实验心得 33](#_Toc143025416)

[4.2 Backtrace (moderate) 33](#_Toc143025417)

[4.2.1 实验目的 33](#_Toc143025418)

[4.2.2 实验步骤 33](#_Toc143025419)

[4.2.3 实验中遇到的问题和解决办法 33](#_Toc143025420)

[4.2.4 实验心得 33](#_Toc143025421)

[4.3 Alarm (hard) 33](#_Toc143025422)

[4.3.1 实验目的 33](#_Toc143025423)

[4.3.2 实验步骤 33](#_Toc143025424)

[4.3.3 实验中遇到的问题和解决办法 33](#_Toc143025425)

[4.3.4 实验心得 34](#_Toc143025426)

[**5.** **Lab5: Copy-on-Write Fork for xv6** 34](#_Toc143025427)

[5.1 Implement copy-on write(hard) 34](#_Toc143025428)

[5.1.1 实验目的 34](#_Toc143025429)

[5.1.2 实验步骤 34](#_Toc143025430)

[5.1.3 实验中遇到的问题和解决办法 34](#_Toc143025431)

[5.1.4 实验心得 34](#_Toc143025432)

[**6.** **Lab6: Multithreading** 34](#_Toc143025433)

[6.1 Uthread: switching between threads (moderate) 34](#_Toc143025434)

[6.1.1 实验目的 34](#_Toc143025435)

[6.1.2 实验步骤 34](#_Toc143025436)

[6.1.3 实验中遇到的问题和解决办法 34](#_Toc143025437)

[6.1.4 实验心得 34](#_Toc143025438)

[6.2 Using threads (moderate) 35](#_Toc143025439)

[6.2.1 实验目的 35](#_Toc143025440)

[6.2.2 实验步骤 35](#_Toc143025441)

[6.2.3 实验中遇到的问题和解决办法 35](#_Toc143025442)

[6.2.4 实验心得 35](#_Toc143025443)

[6.3 Barrier(moderate) 35](#_Toc143025444)

[6.3.1 实验目的 35](#_Toc143025445)

[6.3.2 实验步骤 35](#_Toc143025446)

[6.3.3 实验中遇到的问题和解决办法 35](#_Toc143025447)

[6.3.4 实验心得 35](#_Toc143025448)

[**7.** **Lab7: networking** 35](#_Toc143025449)

[7.1 Your Job (hard) 35](#_Toc143025450)

[7.1.1 实验目的 35](#_Toc143025451)

[7.1.2 实验步骤 35](#_Toc143025452)

[7.1.3 实验中遇到的问题和解决办法 35](#_Toc143025453)

[7.1.4 实验心得 36](#_Toc143025454)

[**8.** **Lab8: locks** 36](#_Toc143025455)

[8.1 Memory allocator (moderate) 36](#_Toc143025456)

[8.1.1 实验目的 36](#_Toc143025457)

[8.1.2 实验步骤 36](#_Toc143025458)

[8.1.3 实验中遇到的问题和解决办法 36](#_Toc143025459)

[8.1.4 实验心得 36](#_Toc143025460)

[8.2 Buffer cache (hard) 36](#_Toc143025461)

[8.2.1 实验目的 36](#_Toc143025462)

[8.2.2 实验步骤 36](#_Toc143025463)

[8.2.3 实验中遇到的问题和解决办法 36](#_Toc143025464)

[8.2.4 实验心得 36](#_Toc143025465)

[**9.** **Lab9: file system** 37](#_Toc143025466)

[9.1 Large files (moderate) 37](#_Toc143025467)

[9.1.1 实验目的 37](#_Toc143025468)

[9.1.2 实验步骤 37](#_Toc143025469)

[9.1.3 实验中遇到的问题和解决办法 37](#_Toc143025470)

[9.1.4 实验心得 37](#_Toc143025471)

[9.2 Symbolic links (moderate) 37](#_Toc143025472)

[9.2.1 实验目的 37](#_Toc143025473)

[9.2.2 实验步骤 37](#_Toc143025474)

[9.2.3 实验中遇到的问题和解决办法 37](#_Toc143025475)

[9.2.4 实验心得 37](#_Toc143025476)

[**10.** **Lab10: mmap** 38](#_Toc143025477)

[10.1 mmap (hard) 38](#_Toc143025478)

[10.1.1 实验目的 38](#_Toc143025479)

[10.1.2 实验步骤 38](#_Toc143025480)

[10.1.3 实验中遇到的问题和解决办法 38](#_Toc143025481)

[10.1.4 实验心得 38](#_Toc143025482)

# **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

待填

## 实验心得

待填

## sleep (easy)

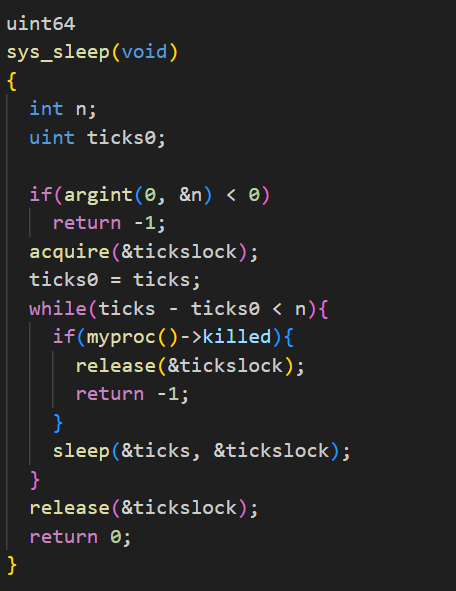
## 实验目的

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

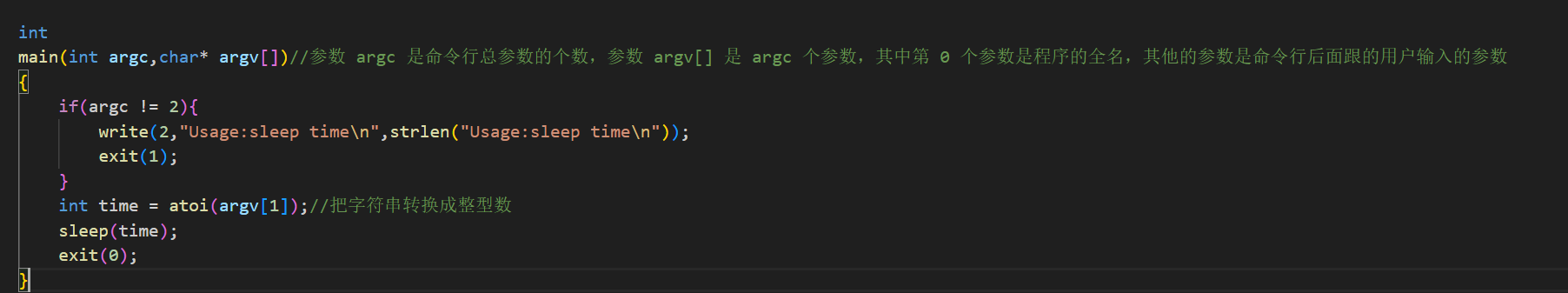
## 实验步骤

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

待填

## 实验心得

待填

## 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。

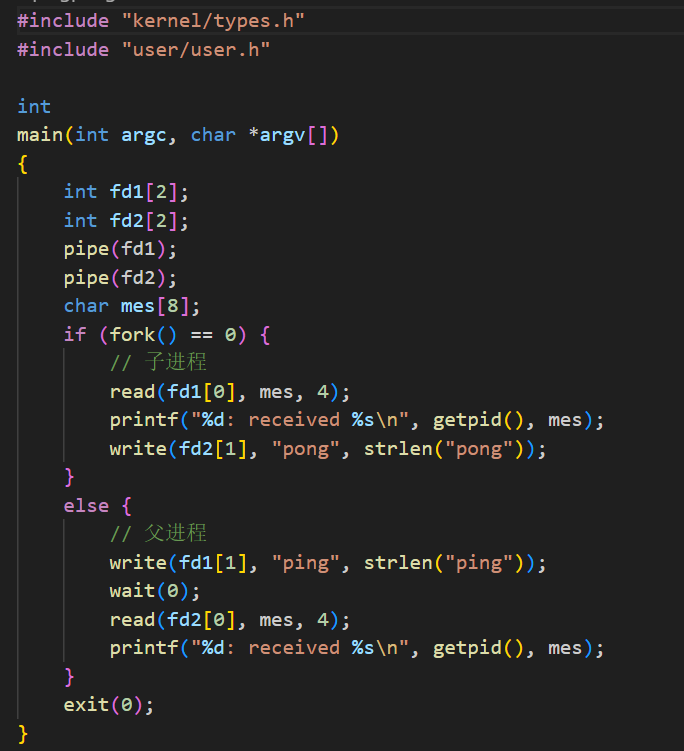
## 实验步骤

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

待填

## 实验心得

待填

## primes (moderate)/(hard)

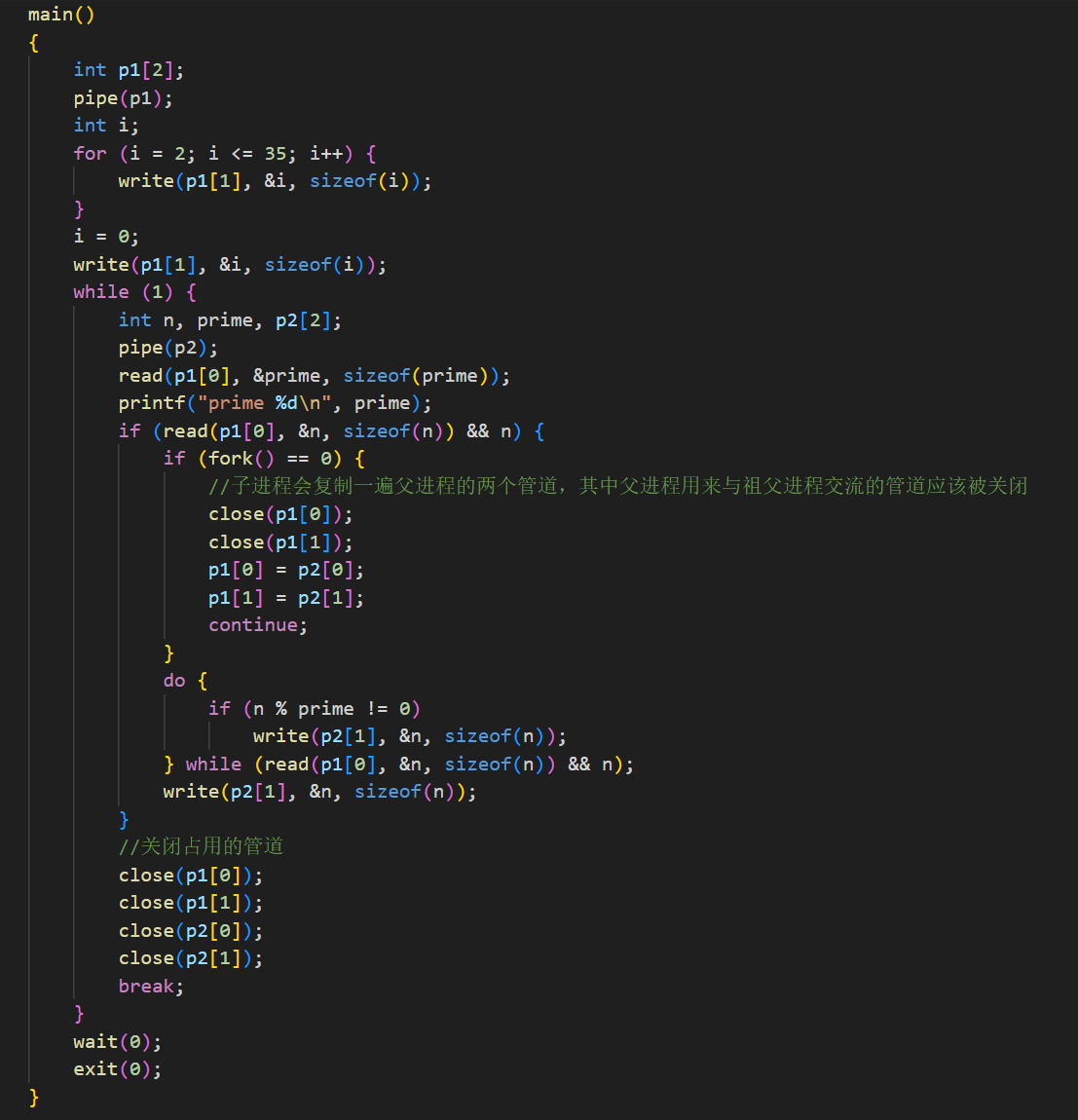
## 实验目的

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

待填

## 实验心得

待填

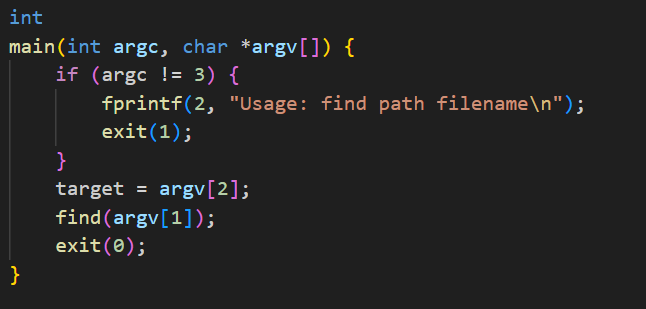
## find (moderate)

## 实验目的

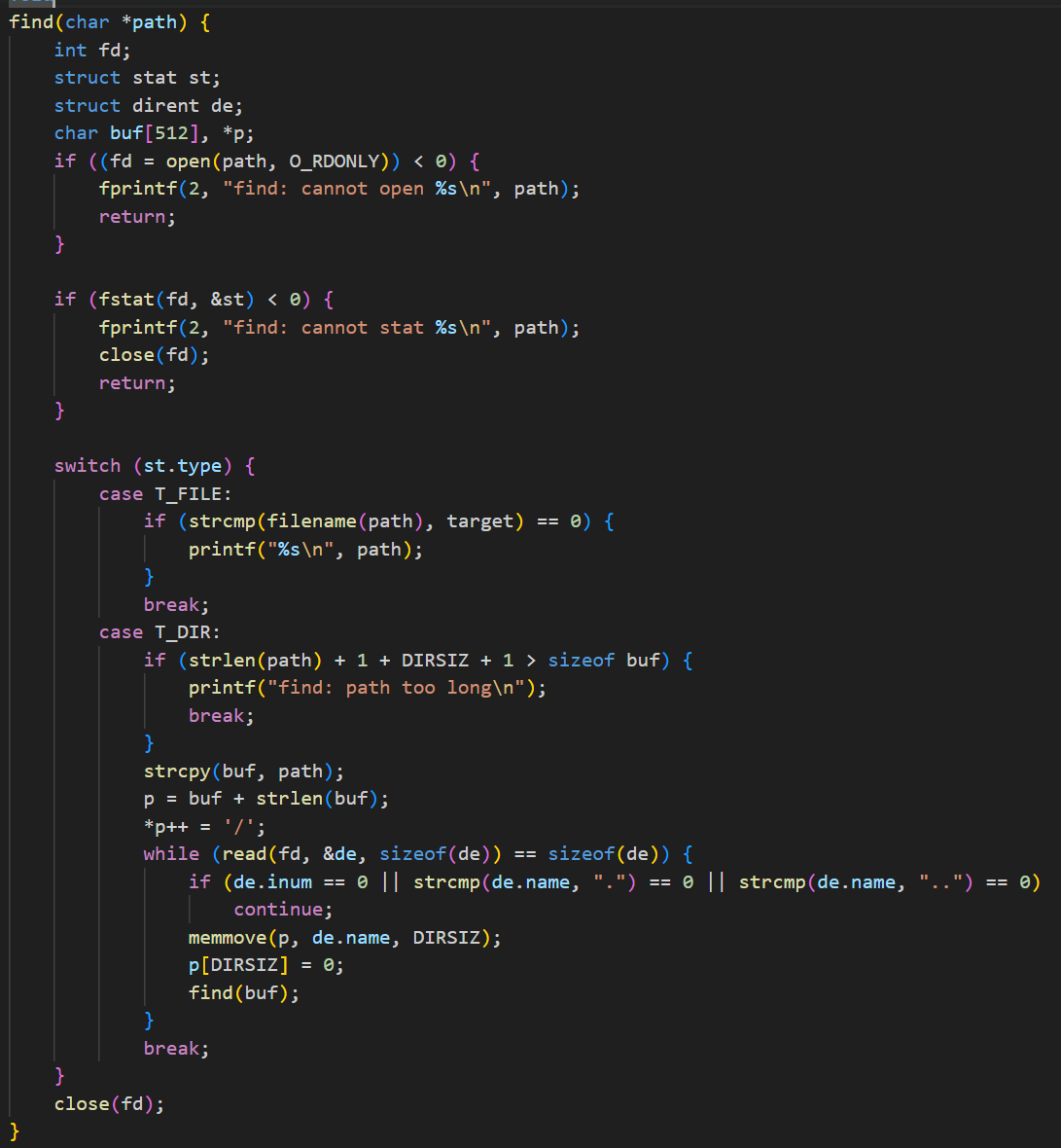
写一个简化版本的UNIX的find程序：查找目录树中具有特定名称的所有文件

## 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

待填

## 实验心得

待填

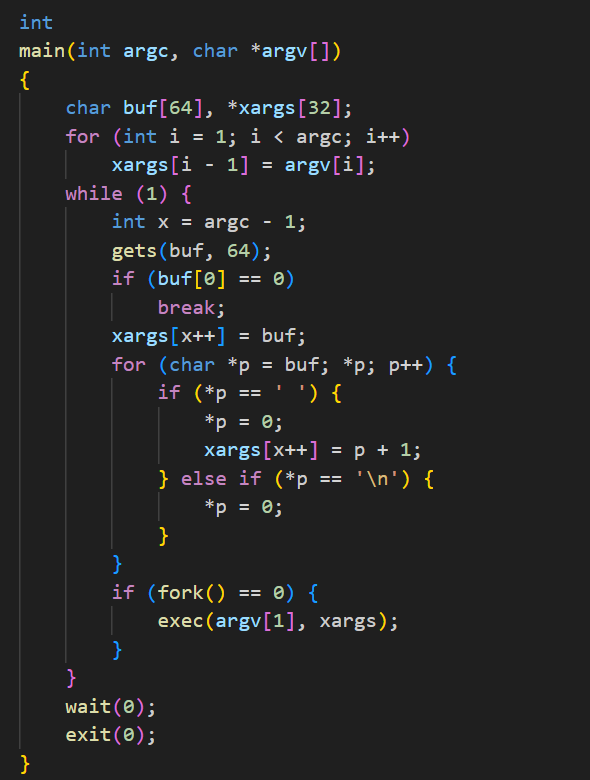
## xargs (moderate)

## 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

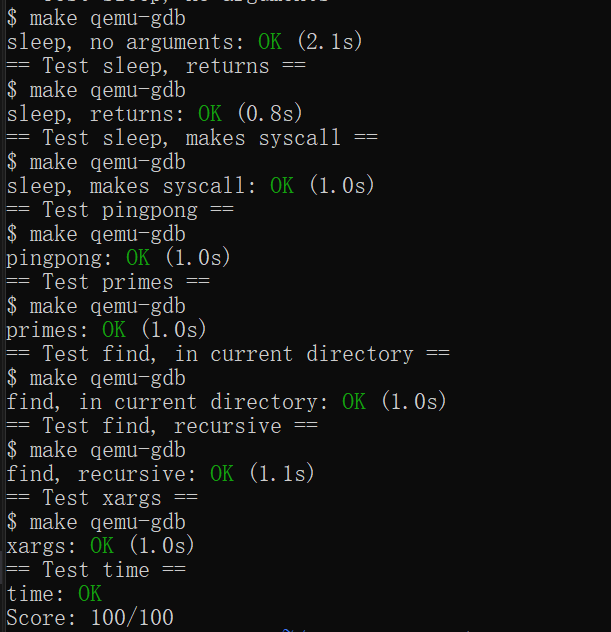
使用fork和exec对每行输入调用命令，在父进程中使用wait等待子进程完成命令。要读取单个输入行，一次读取一个字符，直到出现换行符（'\n'）



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

待填

## 实验心得

待填

# **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**

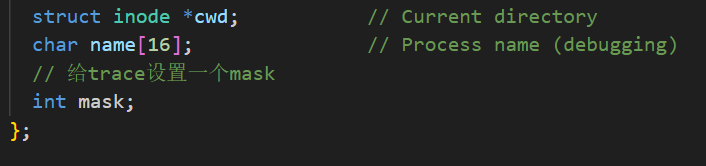
$ **git checkout syscall**

$ **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：

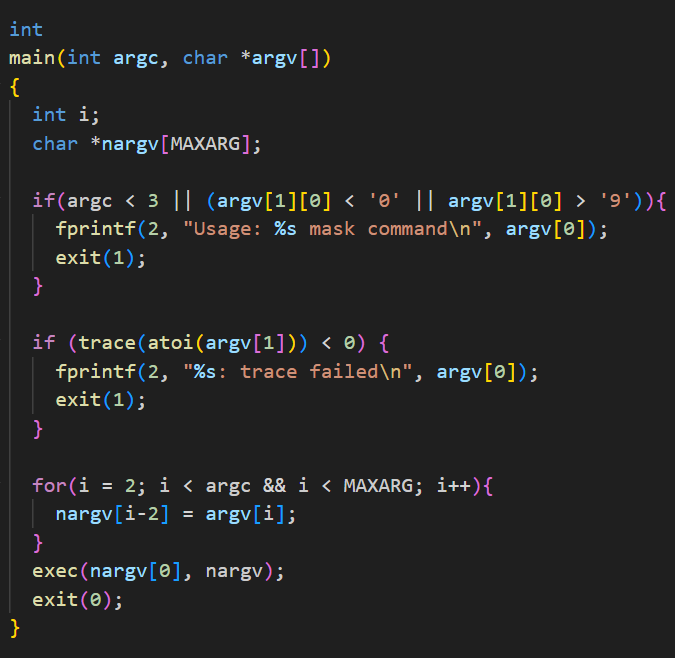
p->trapframe->a0 = syscalls[num]();

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。

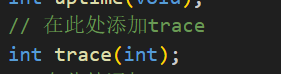
由于mask是按位判断的，且proc结构体里的name是整个线程的名字，不是函数调用的函数名称，所以我们不能用p->name，而要自己定义一个数组，因此最终函数写为：



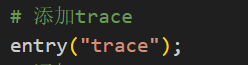
为了把mask参数传进来需要用到trace.c 文件，而这里已经给了trace的用户态函数：



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：



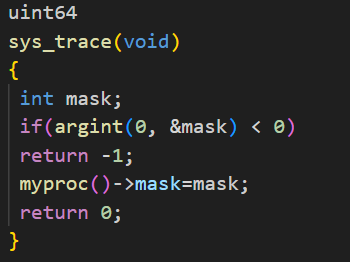
接下来还要给内核态的系统调用trace加上声明和定义，在kernel/syscall.c加上：



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

待填

## 实验心得

待填

## 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。

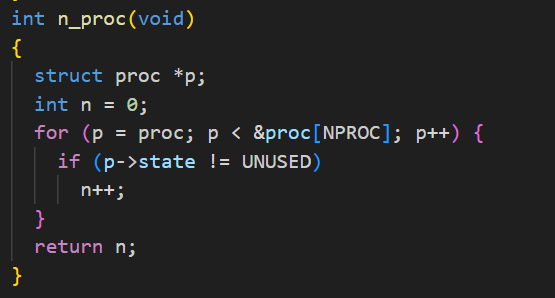
## 实验步骤

通过阅读发现在/kernel/proc.c文件中定义了一个数组：

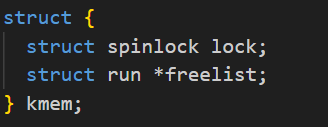


这个数组就保存着所有的进程，所以只要遍历这个数组判断状态就行

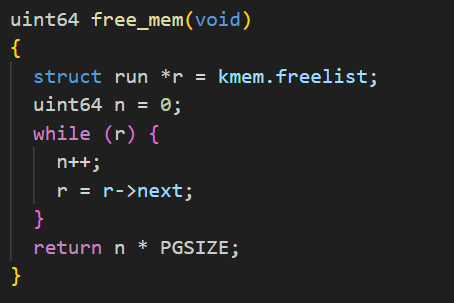
所以实现在kernel/proc.c中加入：



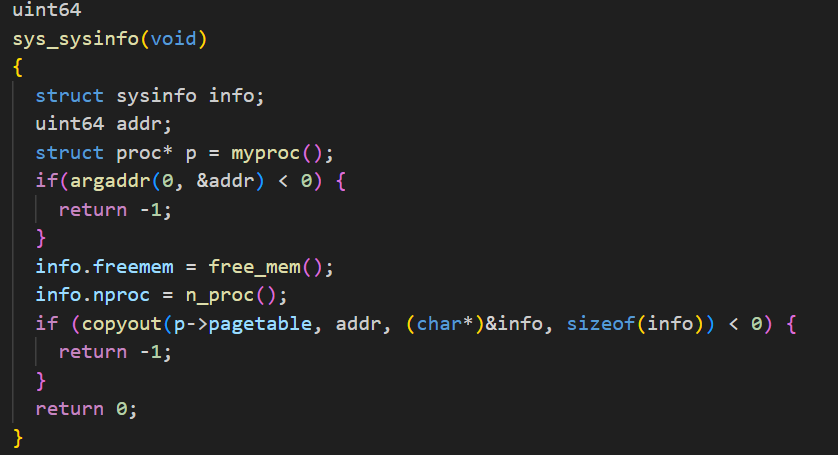
针对于当前可以使用的空间，可以参考 kalloc.c 文件。其中定义了一个链表 kemem，每个链表都指向了一个可用空间，而 kmem 则保存了最后一个链表



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：

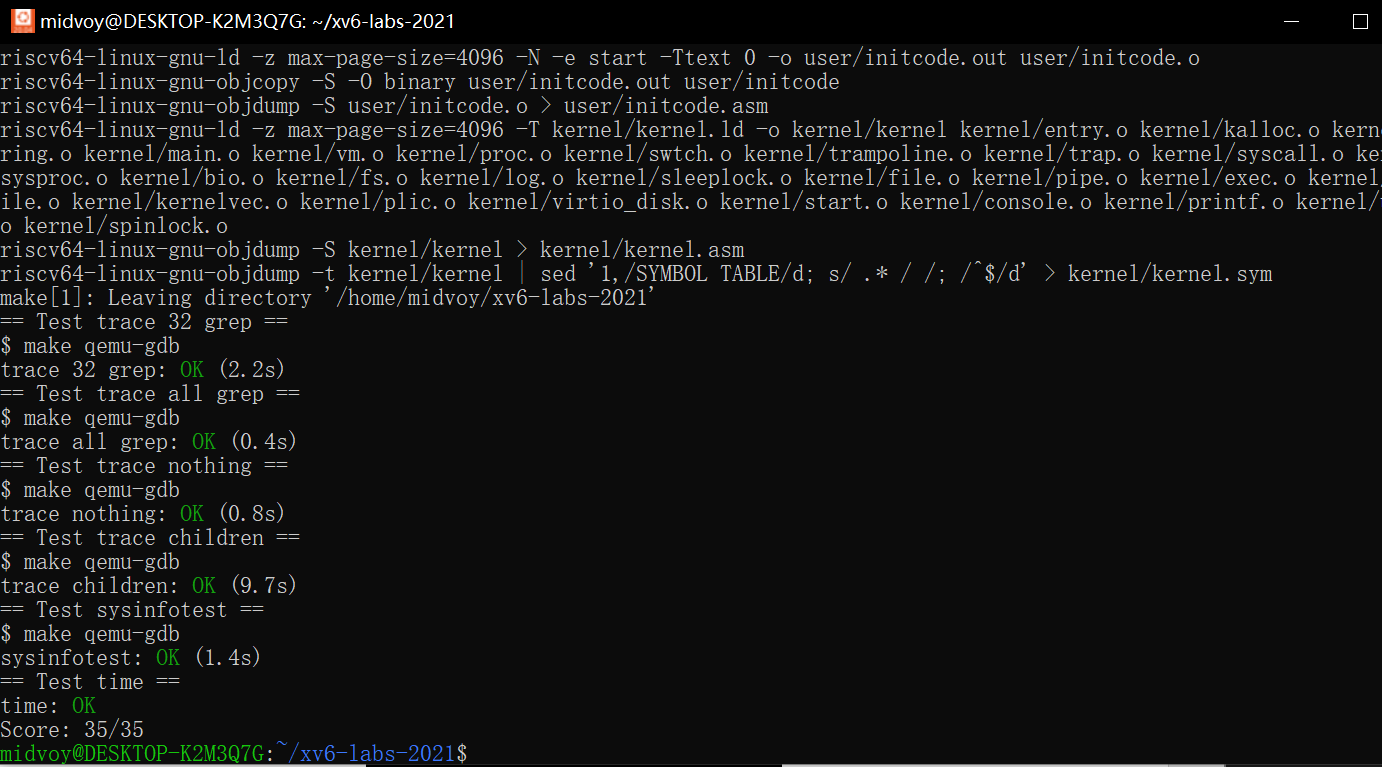


然后在kernel/sysproc.c文件里加入sysinfo实现，同时加入sysinfo结构体对应头文件：



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

[待填]

## 实验心得

待填

# **Lab3: page tables**

## Speed up system calls (easy)

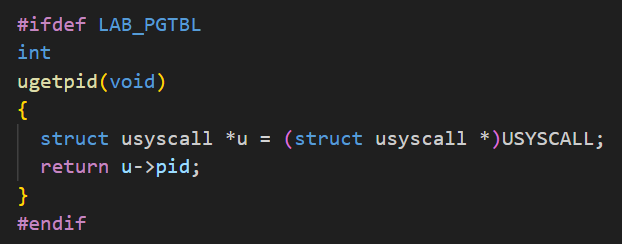
## 实验目的

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。

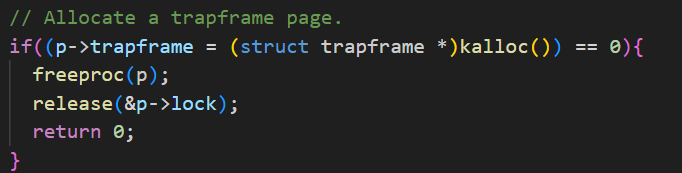
## 实验步骤

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

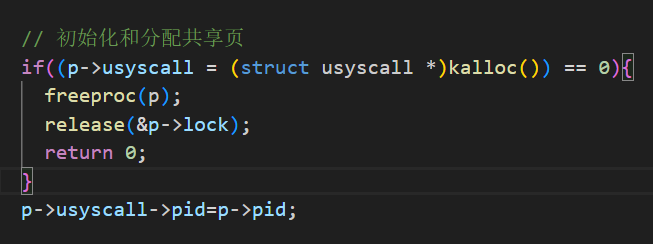
看一下ugetpid()：



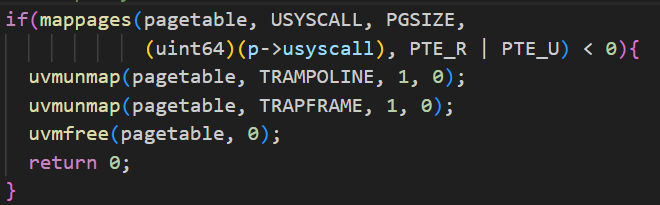
Makefile里面已经设置了相应的CFLAGS，编译的时候会自动加上这个LAB\_PGTBL宏，发现ugetpid()直接从USYSCALL这个地址读数据，因此我们需要把usyscall结构写到此页表的开头。这个USYSCALL是紧邻trapframe下端的一页，然后提示说在proc\_pagetable里面设置映射,对USYSMAP的映射就在这里进行。xv6是先给trapframe分配一块内存再把TRAPFRAME映射到它上面。看一下allocproc()，这个程序首先循环搜索进程表，搜索到UNUSED进程就为其分配内存，然后给进程表p赋各种值：



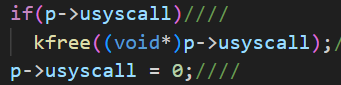
很显然p->trapframe就是在这里初始化的，在allocproc()初始化之后在proc\_pagetable()之中映射。我们可以仿照trapframe的操作，在进程控制块struct proc中添加struct usyscall \*usyscall，作为共享页的物理地址。然后在kernel/proc.c中的allocproc()中完成共享页的分配和初始化：



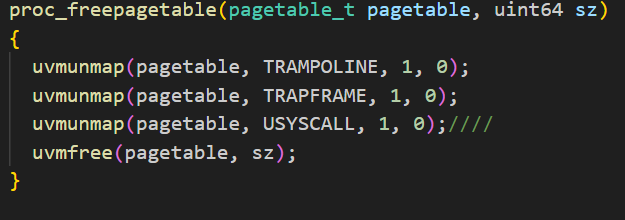
接着在用户页表中添加共享页映射。在kernel/proc.c的proc\_pagetable()中：



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

待填

## 实验心得

待填

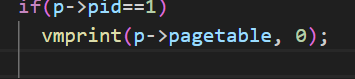
## Print a page table (easy)

## 实验目的

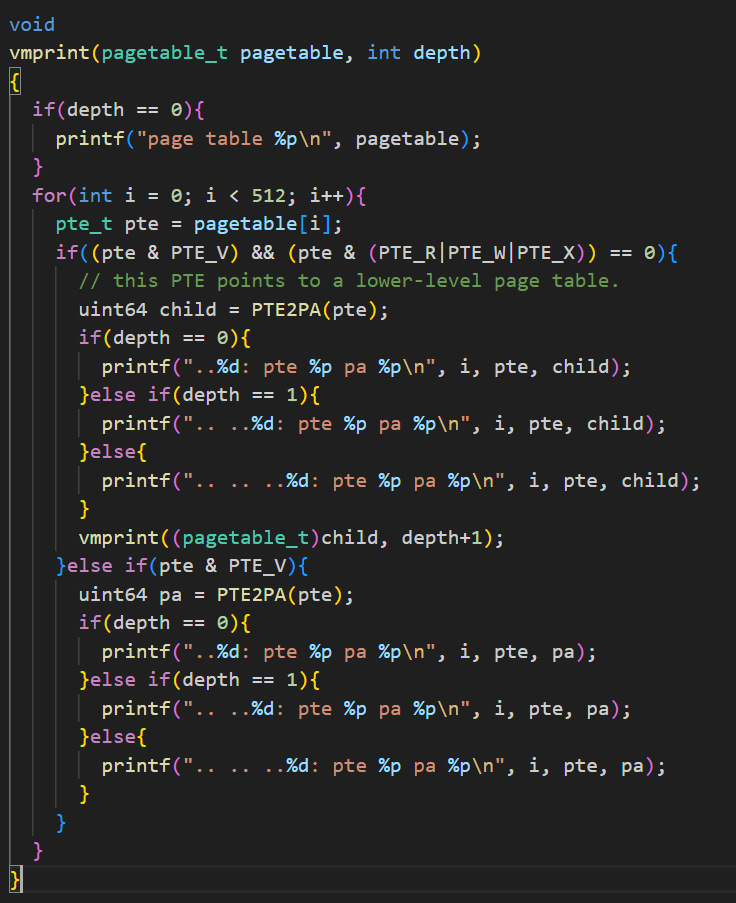
为了可视化 RISC-V 页表，并可能有助于将来的调试,编写一个打印页表内容的函数。定义一个名为vmprint()的函数。它应该接受一个pagetable\_t参数，并以下面描述的格式打印该页表。在 exec.c 中的return argc之前插入if(p->pid==1) vmprint(p->pagetable)，以打印第一个进程(init)的页表。

## 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

待填

## 实验心得

待填

## 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

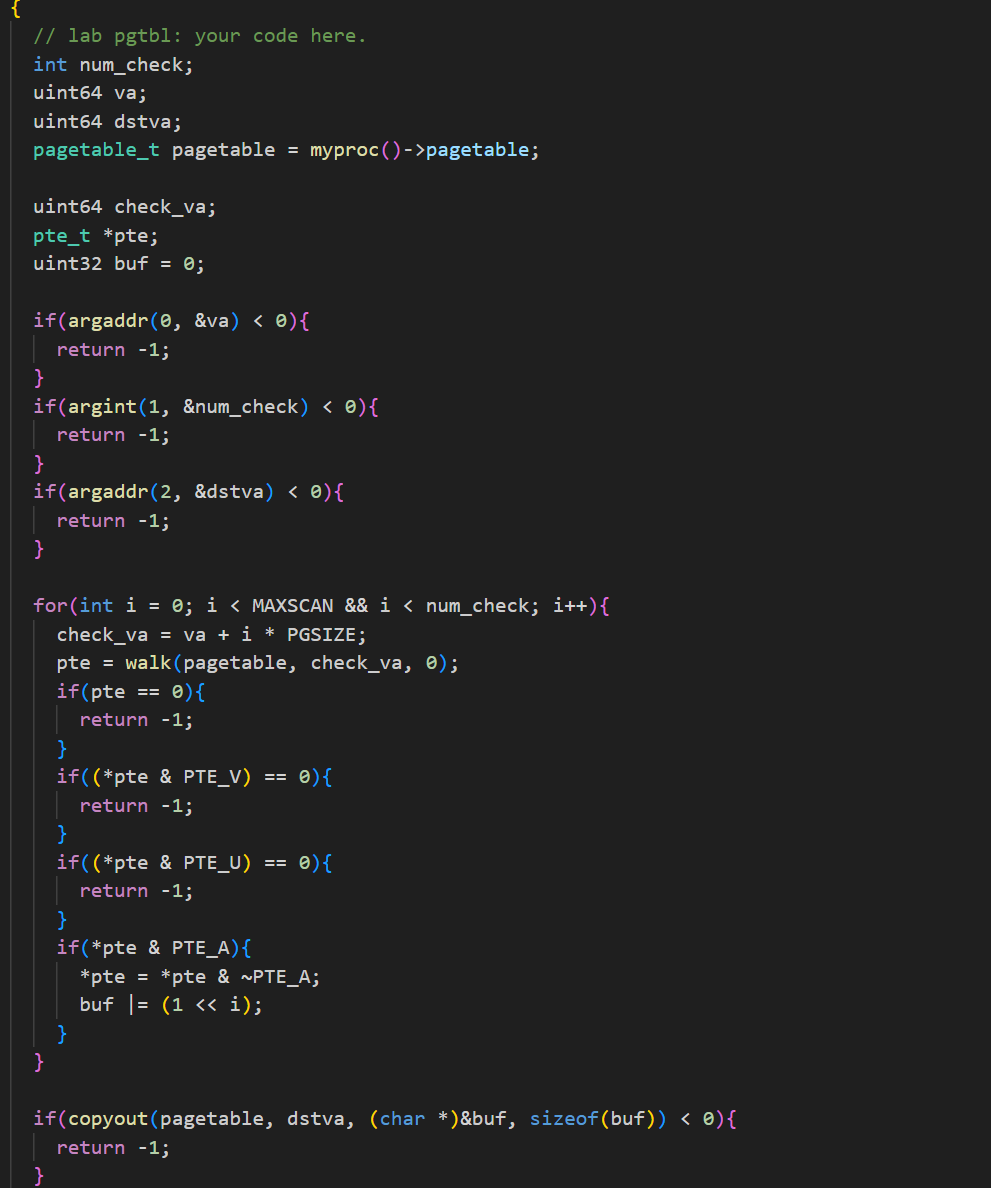
先定义在kernel/riscv.h中PTE\_A



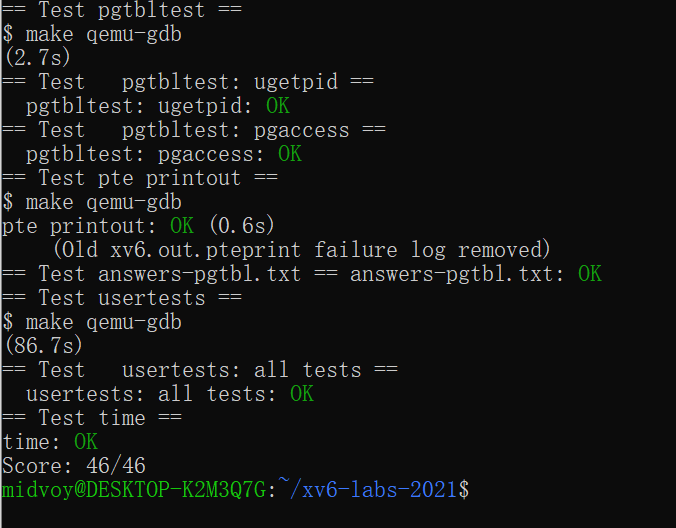
在defs.h中声明walk



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

待填

## 实验心得

待填

# **Lab4: traps**

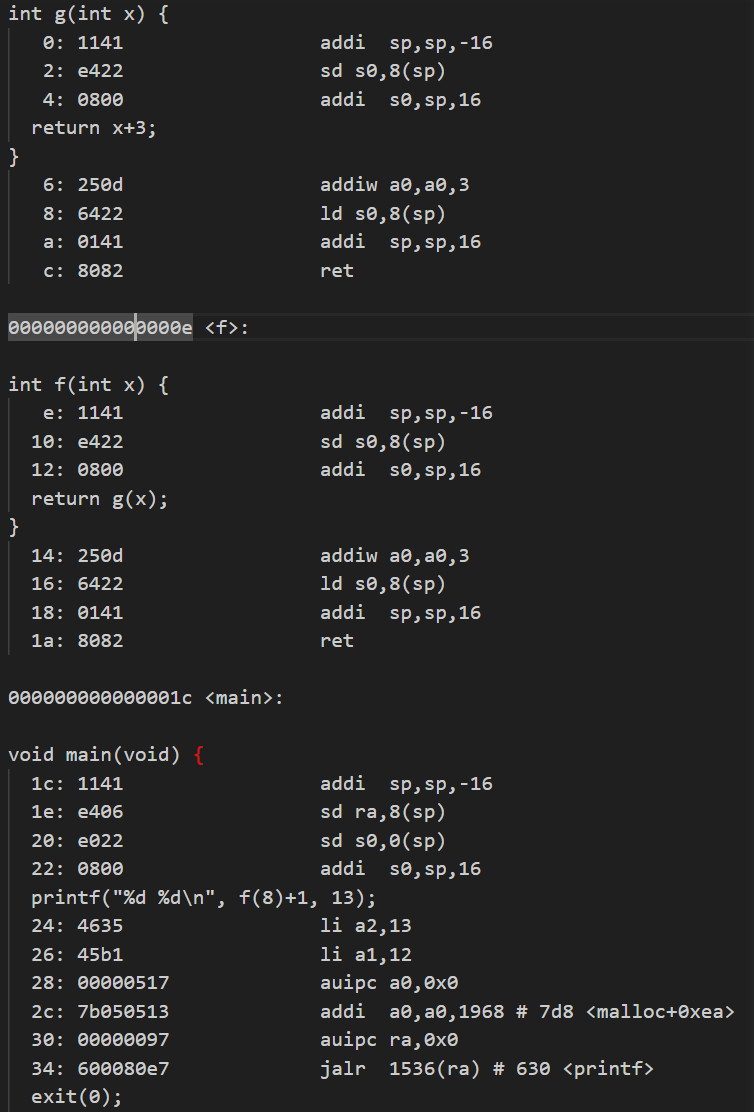
## RISC-V assembly (easy)

## 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

待填

## 实验心得

待填

## Backtrace (moderate)

## 实验目的

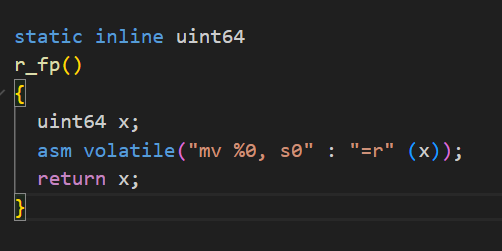
完成 Backtrace 函数，当发生错误的时候查看当前堆栈中的系统调用。

## 实验步骤

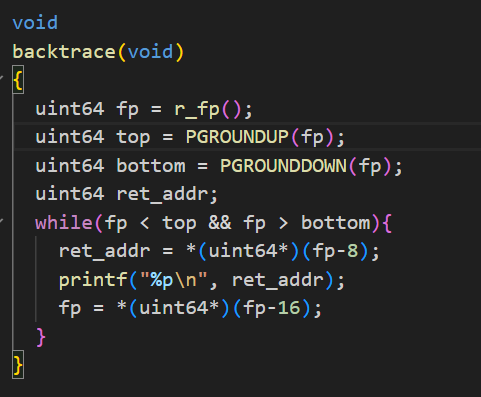
首先在 defs.h 中声明 backtrace()：



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

待填

## 实验心得

待填

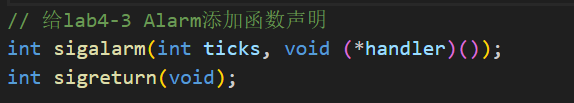
## Alarm (hard)

## 实验目的

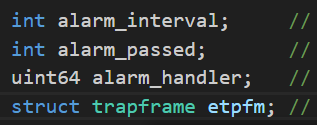
添加系统调用 sigalarm，当用户程序运行了 n 个 ticks 后，触发回调函数。

## 实验步骤

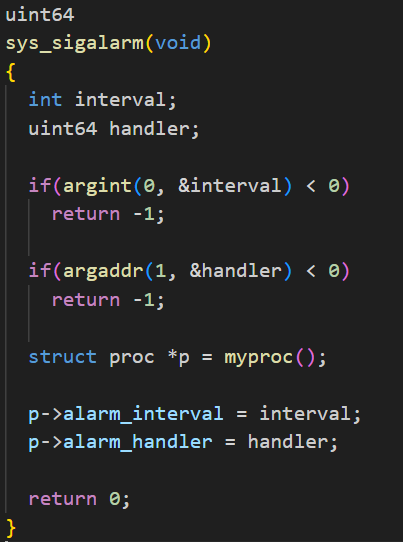
首先需要参照实验4.2将 sigalarm()和 sigreturn()声明为系统调用：



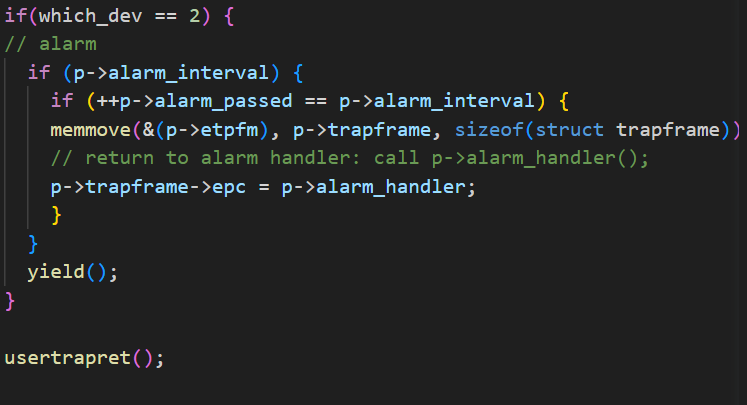
在 proc 结构体里添加需要的字段（间隔时长、已运行时长计数、处理函数地址(vm)、保存 alarm 前的寄存器）：



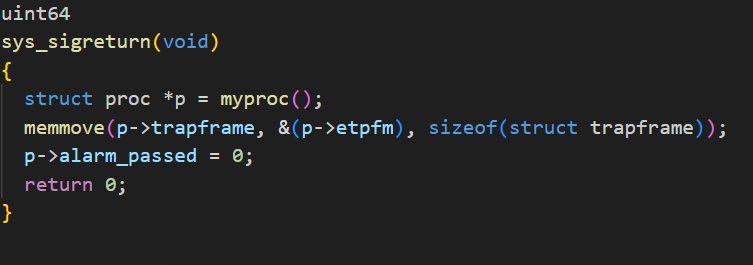
添加 sys\_sigalarm 系统调用：接收参数，为进程设置 alarm 的间隔时长和处理函数：



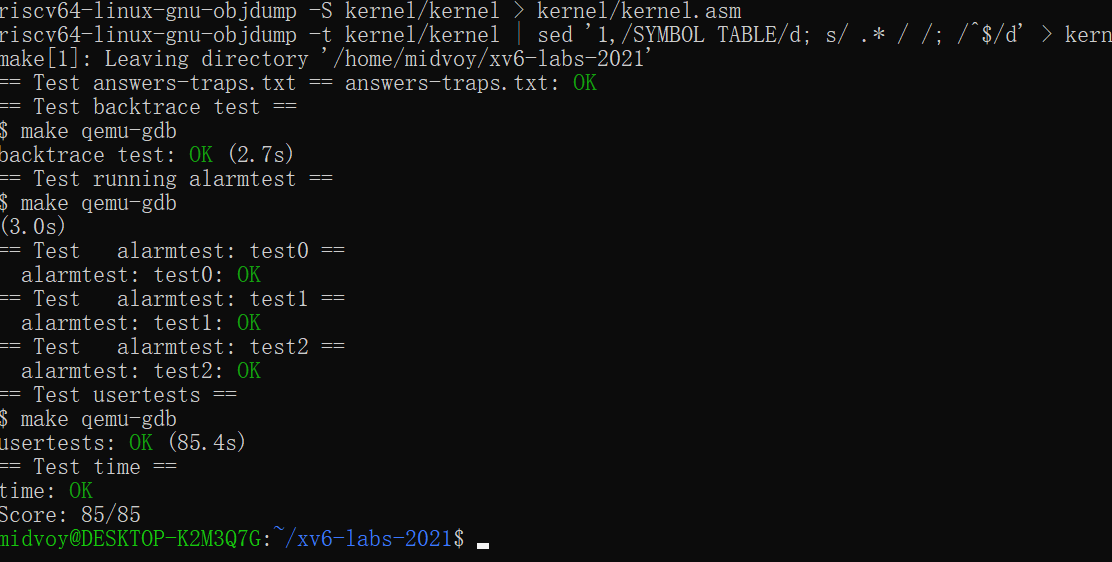
在 usertrap中处理时钟中断时，如果进程需要 alarm 就保存当前 trapframe 到 etpfm，调用 handler （把 handler 地址放到 trapframe->epc，回到用户空间之后就会运行该函数）：



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

待填

## 实验心得

待填

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

## Implement copy-on write(hard)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

# **Lab6: Multithreading**

## Uthread: switching between threads (moderate)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

## Using threads (moderate)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

## Barrier(moderate)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

# **Lab7: networking**

## Your Job (hard)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

# **Lab8: locks**

## Memory allocator (moderate)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

## Buffer cache (hard)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

# **Lab9: file system**

## Large files (moderate)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

## Symbolic links (moderate)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填

# **Lab10: mmap**

## mmap (hard)

## 实验目的

## 实验步骤

## 实验中遇到的问题和解决办法

待填

## 实验心得

待填