# xv6-2021 实验报告——2152343 何慧琳

## 环境搭建

①安装Linux的操作系统Ubuntu22.04

②在linux的环境下安装Bochs

在安装之前，先下载这四个插件

sudo apt-get install build-essential

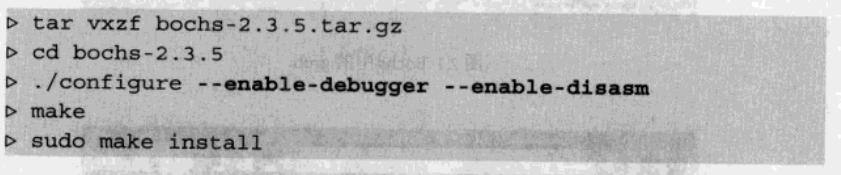
sudo apt-get install xorg-dev

sudo apt-get install bison

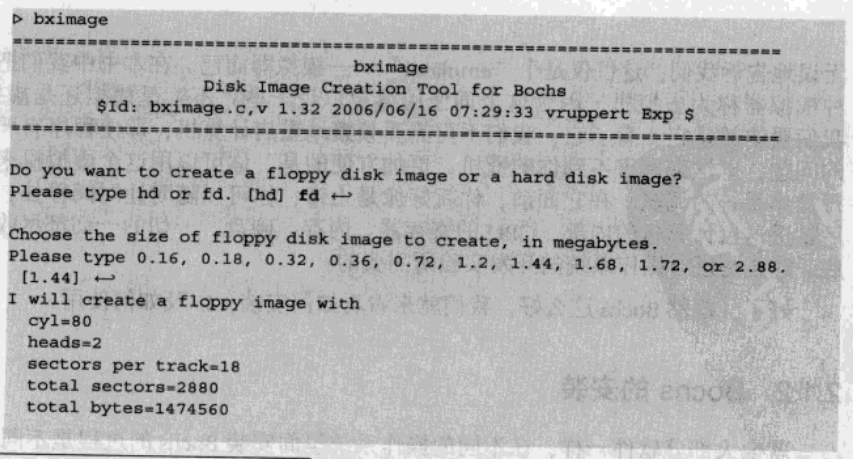
sudo apt-get install libgtk2.0-dev

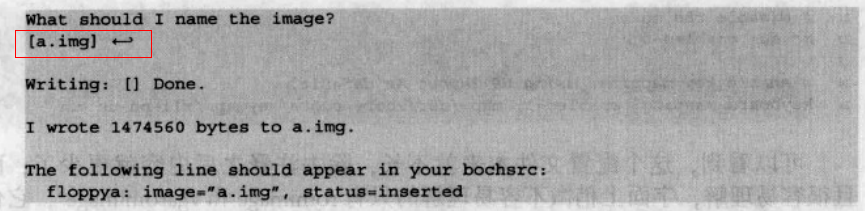
到bochs的官网上下载bochs

然后按书上的下面步骤执行



③创建软盘映像

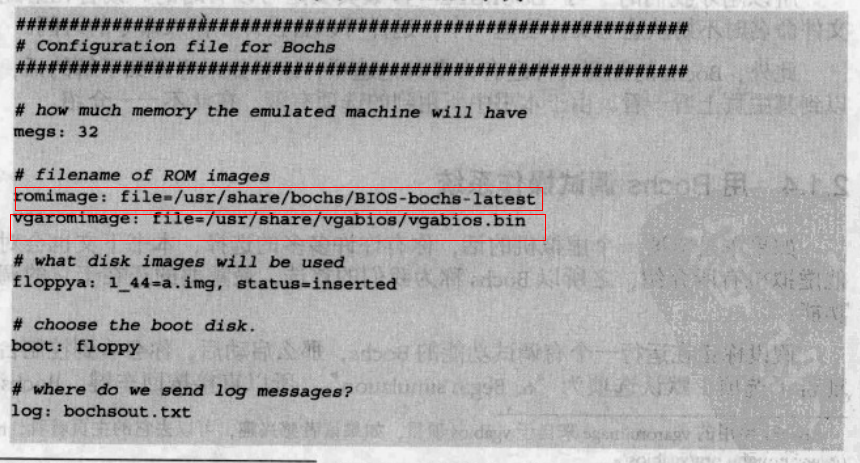


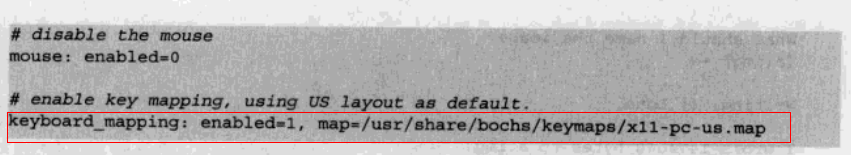


然后将扇区写进软盘：



然后需要新建一个配置文件，如下





然后输入



即可。

④导入实验所需软件：

$ sudo apt-get update && sudo apt-get upgrade

$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

⑤根据实验网站上的tools指示，检测环境配置是否成功。

## 代码托管链接

<https://github.com/linda7511/xv6>

## Lab0：Xv6 and Unix utilities

### sleep(easy)

#### 实验目的

实现xv6的UNIX程序sleep，熟悉xv6实验流程。

#### 实验步骤

①创建user/sleep.c

②在Makefile中UPROGS下添加$U/\_sleep\

③编写sleep.c

④先后输入指令make qemu, sleep 10运行测试程序



#### 实验中遇到的问题和解决办法

一开始我不知道怎么将我的代码上传到linux子系统中运行，通过上网查询我开始理清ubuntu双系统内的结构和一些使用方法，并跟着网上的指引完成了代码的上传和测试并成功通过了。

#### 实验心得

通过这次试验最大的收获是走通了xv6实验流程，对ubuntu的使用有了进一步地了解。

此外在实验过程中我得知xv6中sleep()函数是用户级的函数，位于 user/user.h 头文件中。它是一个库函数，用于用户级程序中的休眠操作。该函数通过调用系统调用 sys\_sleep() 将当前进程置于休眠状态，等待一段指定的时间。sys\_sleep() 函数是内核级的系统调用，位于 kernel/sysproc.c 文件中。它是一个操作系统内核函数，用于处理进程休眠的系统调用。当用户程序调用 sleep() 函数时，sleep() 函数会通过系统调用机制将请求传递给内核，内核会执行相应的操作以使进程休眠。在 sys\_sleep() 函数中，内核会更新进程状态，并调度其他可运行的进程运行。

### pingpong(easy)

#### 实验目的

熟悉管道的原理及操作，熟悉进程的创建及父子进程的关系。

编写一个程序，使用UNIX系统调用在一对管道上的两个进程之间“pingpong”一个字节，每个方向一个。父级应该向子级发送一个字节；子级应该打印“＜pid＞：received ping”，其中＜pid＞是其进程ID，将管道上的字节写入父级，然后退出；父级应该读取子级的字节，打印“＜pid＞：received-pong”，然后退出。

#### 实验步骤

编写pingpong.c。先用pipe创建一对管道分别实现父进程到子进程和子进程到父进程的通信，再用fork创建一个子进程，然后父子进程分别对管道进行读写。

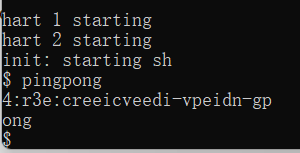
将pingpong.c添加到user目录下，在Makefile文件的UPROGS 中添加$U\_pingpong\

键入make qemu和pingpong命令测试。

#### 实验中遇到的问题和解决办法

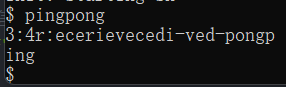
一开始我审题不清，以为是父进程向管道1写ping，子进程从管道1读父进程写的ping；子进程向管道2写pong，父进程从管道2读子进程写的pong。于是有了以下代码和结果。





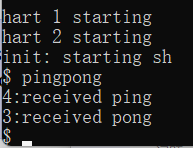
看到这个结果我有些震惊，于是去搜了父进程和子进程的关系，得知在fork之后，是父进程先执行，然后一个时间片到达之后就是子进程再执行了。所以出现以上结果是时间片不够大，父子进程来回切换执行的结果。此时我还没意识到我审错题了所以继续在上面代码基础上改进，我觉得要想获得预期结果，在某一进程准备打印读到的内容时应该阻止另一进程的读并打印，于是有了以下的代码和结果。但是我后来意识到这个改进实际是无用的，因为之所以出现串行打印本就是时间片不够大，一个进程打印了一些内容后必须切换到另一个进程并打印。





在我重新看了一遍题目后，我重新写了代码，于是得到了正确的结果。





#### 实验心得

通过这个实验我加深了对管道概念的理解，进一步熟悉了fork，pipe，read，write等函数的使用，对父子进程的关系也有了进一步的认识。这个实验标着easy但对于刚上手的我来说却有点艰难，现在有些地方也还有些云里雾里，毕竟这是爬坡的过程，我相信到后面会渐渐拨云见雾的。

### primes(moderate/hard)

#### 实验目的

熟悉管道创建级读写和进程创建的使用，进一步加深对管道、进程相关实现及原理的理解。

使用pipe和fork来设置管道。第一个进程将数字2到35写入到管道中。对于每个素数，创建一个进程，该进程通过一个管道从其左邻居读取，并通过另一个管道向其右邻居写入。由于xv6具有有限数量的文件描述符和进程，因此第一个进程可以在35处停止。

#### 实验步骤

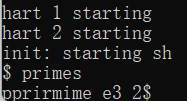
编写primes.c代码：创建管道p1，向其中写入2-35同时生成素数数组，向p1写入0作为此次结尾标志。进入大循环，创建p2管道，从p1中读出第一个数并输出；读第二个数不是0则fork创建子进程，原进程继续读p1并将每个从p1读出的数写入p2直到读到0，子进程将p1管道替换为p2管道，父进程变成祖父进程，子进程变成父进程，continue主循环；读到的第二个数是0则close两个管道的读写端并退出主循环。

将primes.c添加到user目录下，在Makefile文件的UPROGS 中添加$U\_primes\

#### 实验中遇到的问题和解决办法

一开始我以为全局变量会被父子进程共享，对进程的并行考虑也比较粗糙，于是有了下面的代码和结果。



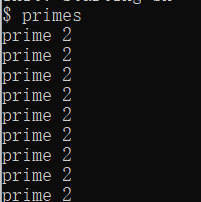


看到这个结果后我一直在想怎么让两个进程的printf不会串行输出，首先是想像pingpong.c中那样在printf之前先wait子进程结束，但是这样的话就要使最新创建的子进程最先输出一个素数，这样就可能要用递归，有点难；于是我又想printf有没有什么可以下手的地方，于是去查阅资料，知道了：printf是向标准输出(stdout)输出要打印的东西，是行缓冲，遇到\n才会打印出来，且当fork创建子进程时若父进程的stdout中还有内容会被子进程也复制一份（参考printfhttps://blog.csdn.net/world\_hello\_100/article/details/8066797）。于是这个想法也作废了。

我又进行了一些测验终于开始怀疑我的循环条件无效。并产生疑问父子进程全局变量，一个进程中被改变会在另一个进程中更新吗？通过上网查询获得了以下结果：

我开始参考网上的代码，并对我的代码进行了修改：



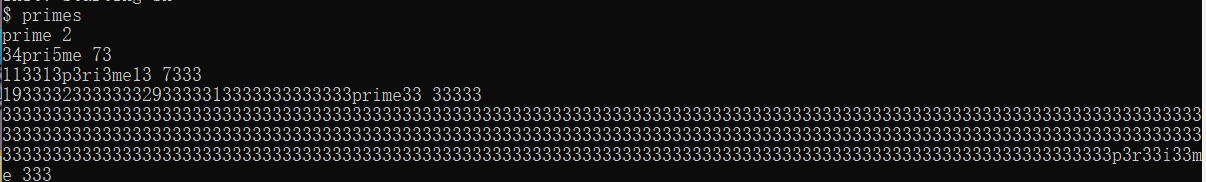


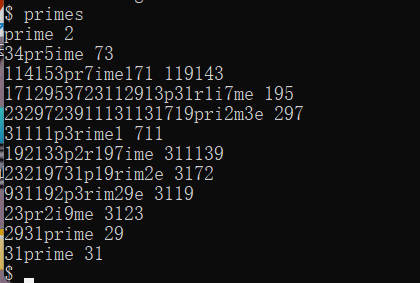
检查后我发现我的代码中子进程会不停被fork，在这个过程中我开始疑问子进程是从哪里开始执行的，后来知道“子进程和父进程都从调用fork函数的下一条开始执行”。加上fork的条件后终于获得了正确结果。





上面是我的最终代码，但是在最终代码之前（将第2行、倒数第二行的注释删掉），由于第二行这个close语句我的输出结果如下第一张图片。在此之前我以为close是一个很随意的操作，再次使用read/write函数可以直接正常读或写，但是这个结果让我知道close并不是一个可以随便用的函数，也让我开始思考close究竟是怎么实现的，以及为什么会出现以下结果，但是目前还没有头绪，先往后看看吧。





#### 实验心得

这个实验做起来有点困难，但是它逼我去进一步弄清管道、父子进程执行顺序等机理，虽然最后我还是参考了别人的代码，但是在探索、思考的过程我收获了很多，虽然我的代码中用的还是原始的自底向上的判断方法，但我通过学习别人的代码还学到了一个素数筛法。

### find(moderate)

#### 实验目的

熟悉目录路径组成，了解查找路径的实现方法。

编写一个简单版本的UNIX查找程序：查找指定目录下具有指定名称的所有文件并输出其路径。

#### 实验步骤

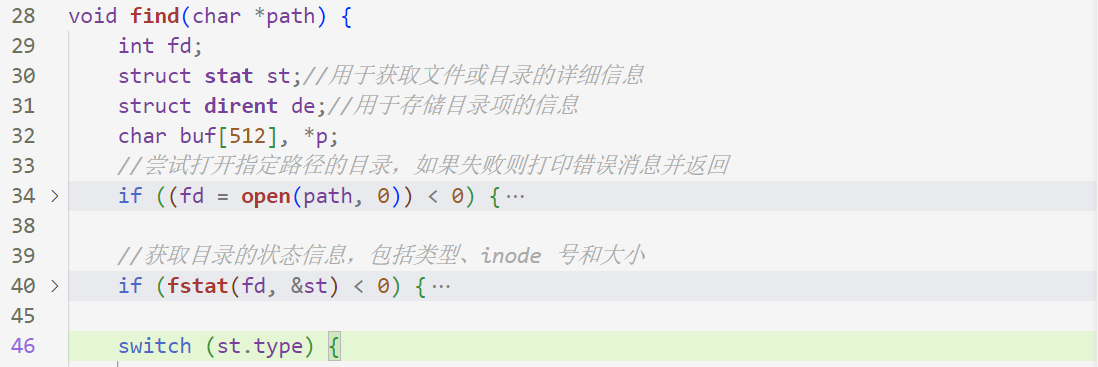
编写find.c代码：查找具有指定名称的文件用到深度优先搜索方法，遍历当前目录下的每一个文件，遇到普通文件则直接比较其文件名是否与目标文件名相同，相同则输出这一路径；遇到目录文件则进入目录，递归搜索这一目录。在主函数中获取find指令后面的搜索起始目录和目标文件名，调用find函数进行搜索。代码中还定义了fmtname函数来从一个路径字符串中获取该路径下的文件名。

将find.c添加到user目录下，在Makefile文件的UPROGS 中添加$U\_find\

#### 实验中遇到的问题和解决办法

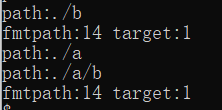
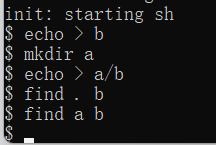
这个实验一开始我有点看不明白题意。我先按照提示看了xv6提供的ls.c代码和echo.c代码，上网查询后知道了echo的作用，以及题目中echo > b大概是通过输出重定向空字符串到b从而自动创建了一个名为b的空文件。但我还是不解题目find . b什么意思，看了网上别人的题解后知道了find a b就是在a目录下查找名为b的文件。

理解好题意后我开始在xv6代码的基础上进行修改，将原来的ls.c代码中的ls函数替换为find函数。





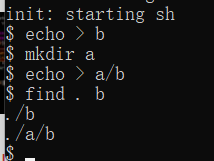
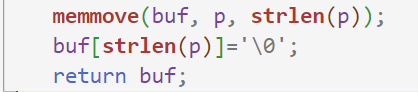
但是第一次测试结果如下图，没有任何路径输出。通过在一些地方加测试输出语句后我发现问题应该出在我fmtname(path)返回的字符串上，因为fmtname后的文件名长得和target一样但是strlen出来却不一样。



于是我发现是在fmtname函数中照搬了ls.c，将文件名以空格填充到DIRSIZ了。



纠正后便得到了正确的结果。



#### 实验心得

这次实验由于提供了参考代码所以进行得比较顺利。通过这次试验我了解到了echo、mkdir指令的作用和用法，知道了echo和ls的具体实现，重温了dfs和字符串的一些知识，对于目录组织也有了更清晰的认识。

### xargs(moderate)

#### 实验目的

了解xargs的使用，熟悉命令行、命令参数的读取，掌握main函数参数中argc、argv的内涵及使用。学习使用gets函数从标准输入行读取数据。

编写一个简单版本的UNIXxargs程序：从标准输入中读取行，并为每行运行一个命令，将该行作为命令的参数提供。

#### 实验步骤

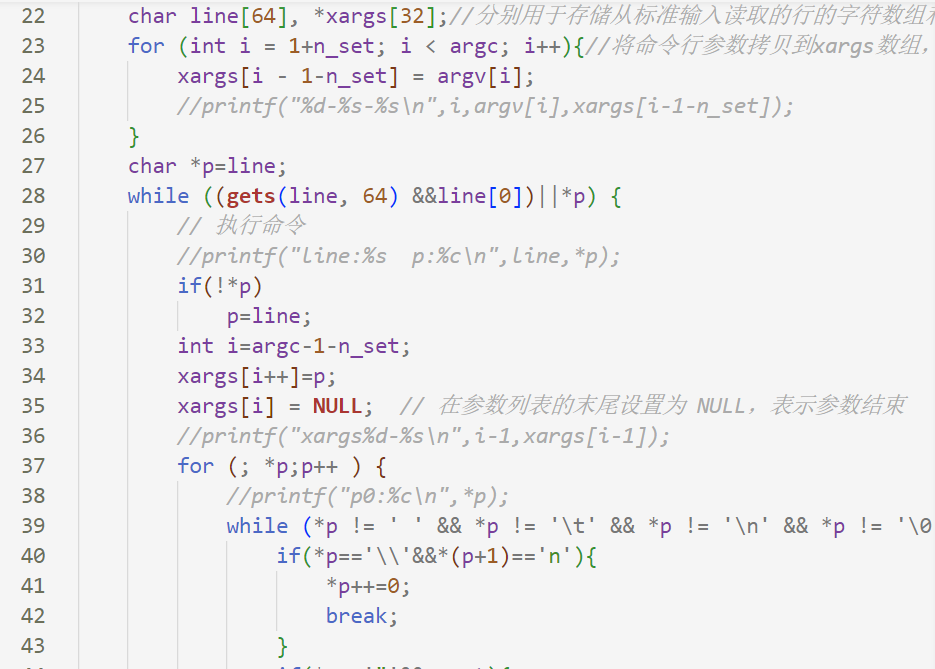
编写xargs.c代码：判断命令行中是否包含-n可选项，并相应地将argv中真正的参数复制到xargs数组中。读取输入行，以空格或\n分割命令行以获得各参数并将参数添加到xargs数组后面。完成参数获取后fork子进程将xargs参数传给exec执行command（即argv中的命令项）。父进程继续读取输入行中剩余的参数直到处理完毕。等待子进程退出后父进程也退出。

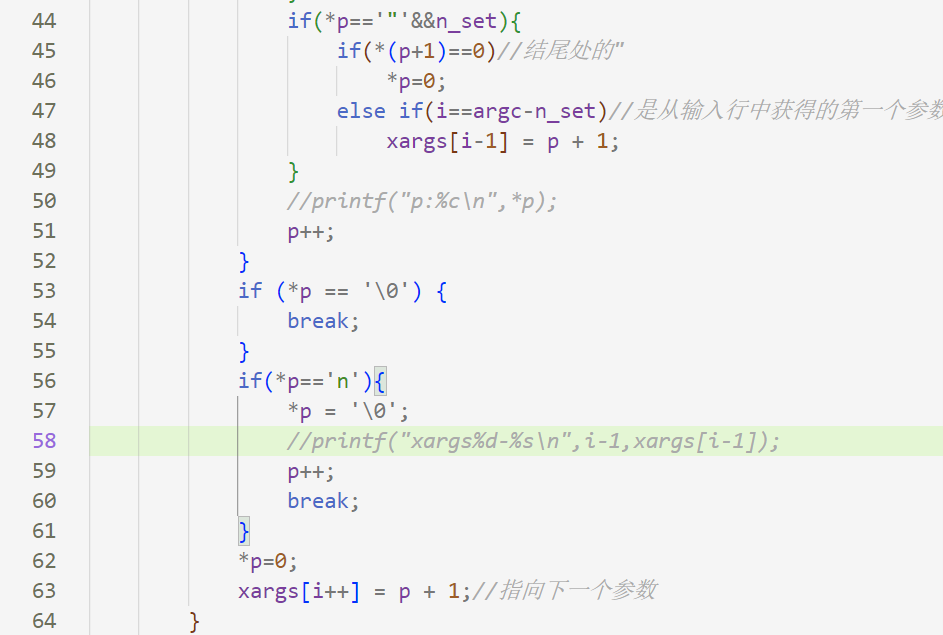
将xargs.c添加到user目录下，在Makefile文件的UPROGS 中添加$U\_xargs\

#### 实验中遇到的问题和解决办法

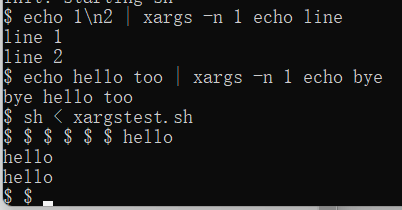
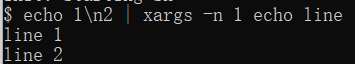
一开始除了echo 1\n2 | xargs -n 1 echo line这条命令输出结果不正确其它都正确，接下来我就在正确处理这条命令的探索过程中做了很多调试，在这一过程中最初的问题是我程序中的p指针读到\n时会有\*p=0，按理来说参数行从此处被分割，p+1指向2，但是我printf \*（p+1）时竟然也显示为NULL，于是我上网查，得到的结果是\n或者\0后面的字符并不会被忽略它们仍然存在，而且我想我的p是char\*类型的，原line的地址没有变化那么p+1理应仍指向2。后面我通过以%c输出\*p，突然反应过来p一次读一个字符，所以它只能分别读到\和n,于是我对代码进行了修改最终我终于获得了正确的结果，代码和结果如下：







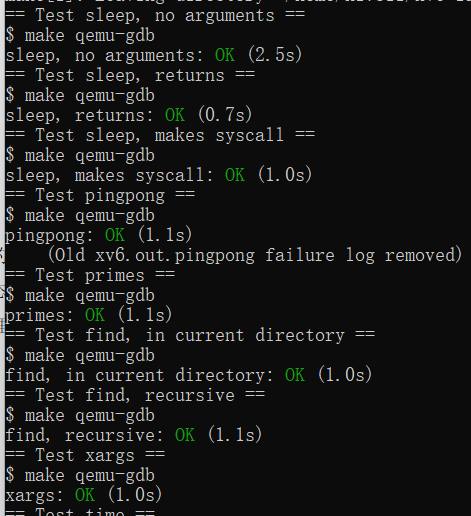




#### 实验心得

通过这个实验我对gets读取输入行的规则，argc、argv的含义，命令行的解析和读取都有了更深刻的认识，也重温了之前c语言的一些基础，对于xargs命令的用法也有了一定了解。

以下是lab0make grade的结果



## Lab1：System calls

### System call tracing(moderate)

#### 实验目的

了解操作系统的隔离性、用户态、内核态、系统调用的概念及实现。

创建一个新的跟踪系统调用来控制跟踪。它具有一个参数，一个整数“掩码”，其位指定要跟踪的系统调用。例如，要跟踪fork系统调用，程序调用trace（1<<SYS\_fork），其中SYS\_fork是kernel/syscall.h中的系统调用编号。如果系统调用的编号在掩码中设置，则必须修改xv6内核，以便在每个系统调用即将返回时打印出一行。该行应包含进程id、系统调用的名称和返回值。跟踪系统调用应启用对调用它的进程及其随后分支的任何子进程的跟踪，但不应影响其他进程。

#### 实验步骤

修改用户接口代码：在user/user.h中添加系统调用函数的定义“int trace(uint)”;在user/usys.pl中添加入口entry("trace");

修改内核代码：在kernel/syscall.h中定义系统调用号（添加#define SYS\_trace 22）；在kernel/syscall.c的syscalls函数指针数组中添加“[SYS\_trace] sys\_trace”;在proc结构体中添加一个uint trace\_mask成员变量，在proc.c中fork为子进程复制变量时将trace\_mask字段也复制给子进程；在sysproc.c中添加函数uint64 sys\_trace(void)，该函数通过argint函数读取参数赋值给mask变量，然后与trace\_mask字段位或即可；修改syscall函数，当系统调用号和trace\_mask匹配时输出相关信息；在syscall函数前定义syscalls\_name[]数组

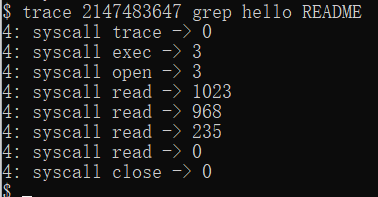
编写应用工具：创建user/trace.c，编写代码实现trace系统调用，设置要跟踪的程序的参数，exec系统调用运行需要跟踪的程序。

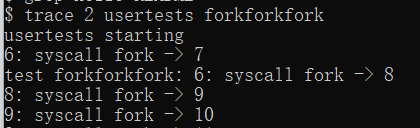
修改Makefile,将$U/\_trace\加到Makefile的UPROGS中

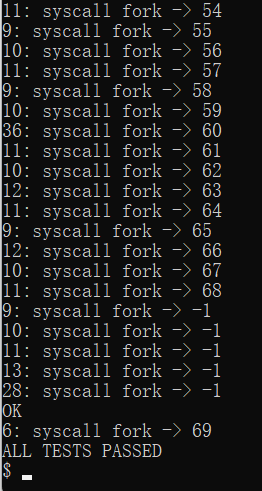
make qemu，进行测试

#### 实验中遇到的问题和解决办法

理解这次实验的内容是最大的问题，在实验前我看了xv6文档，并借助mit的课程视频进行理解，知道了操作系统的隔离性通过用户态和内核态间的切换来维护，对于陷入、系统调用等概念。但是对于这个实验具体如何实现还是一头雾水，在学习网上的教程之后终于直到了要创建这个跟踪系统调用需要在user代码中添加这个接口，在内核代码中实现这个追踪功能，再编写用户trace代码。







#### 实验心得

通过这个实验我对陷入、系统调用等概念有了进一步理解，通过阅读xv6文档我知道了fork函数的实现代码及li、ecall、ret等指令的作，对进程的装入、地址映射也有了一定了解；通过学习mit课堂视频我知道了s、o后缀的文件分别是c代码经gcc编译后形成的汇编语言文件、汇编语言经链接形成的二进制文件；通过查阅网上资料我对内核代码中与系统调用相关的部分也有了一定了解。

### Sysinfo(moderate)

#### 实验目的

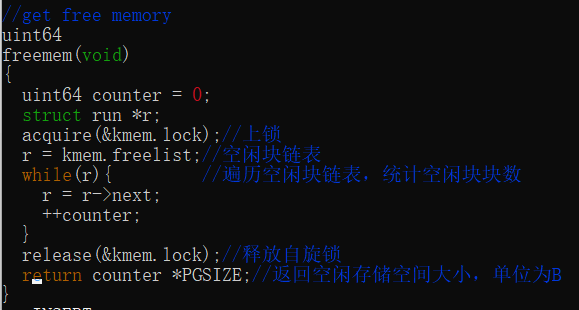
了解sysinfo的内容，了解空闲块链表和copyout的使用，熟悉锁的使用和pcb中的字段。

增加一个sysinfo系统调用，它收集有关运行系统的信息。该系统调用有一个参数，即指向结构sysinfo的指针（参见kernel/sysinfo.h）。内核为该结构的各个字段赋值：设置freemem字段为可用内存的字节数，设置nproc字段为状态是非UNUSED的进程数。实验提供了一个测试程序sysinfotest，如果输出“sysinfotest:OK”，则该任务通过。

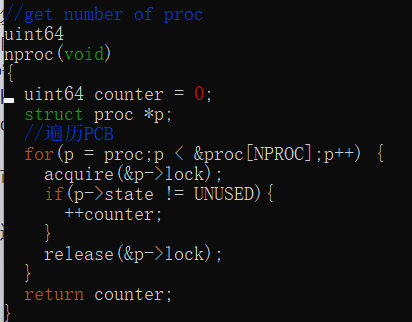
#### 实验步骤

修改用户代码接口：在user/user.h中声明struct sysinfo；在user/user.h中添加系统调用函数的定义（int sysinfo(struct sysinfo\*);在user/usys.pl中添加入口 entry("sysinfo")。

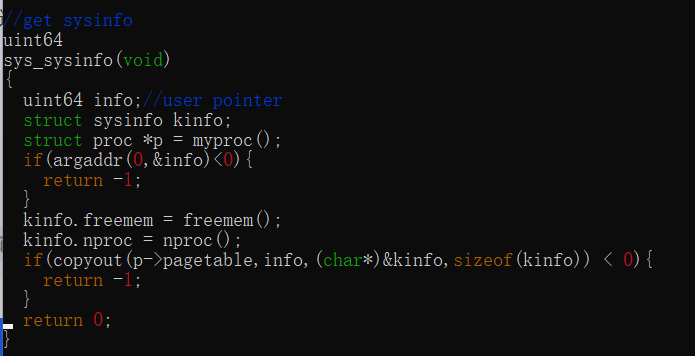
修改内核代码：在kernel/syscall.h中定义系统调用号#define SYS\_sysinfo 23；在kernel/syscall.c的syscalls函数指针数组中添加对应的函数及函数名；在kernel/defs.h中声明freemem、nproc函数；在kernel/kalloc.c中添加freemem函数并实现；



在kernel/proc.c中添加nproc函数并实现



在kernel/proc.c中添加sys\_sysinfo函数并实现



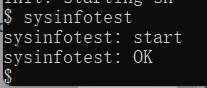
实验平台已提供了测试程序sysyinfotest，故无需编写c代码

修改Makefile,将$U/\_sysinfotest添加到Makefile的UPROGS中；make qemu进行测试。

#### 实验中遇到的问题和解决办法

一开始要实现freemem时我不知道空闲块链表该如何操作，在参考了kfree、kalloc两函数后知道了如何获取并使用它。同时通过观察它们的代码我还知道了此处要加锁，并且形成了编写内核代码时检查是否要加锁的警惕，上网查询资料总结出只有当某一变量只在进程在cpu上运行时才被访问时不用加锁，因为cpu是唯一的，其余都要警惕。

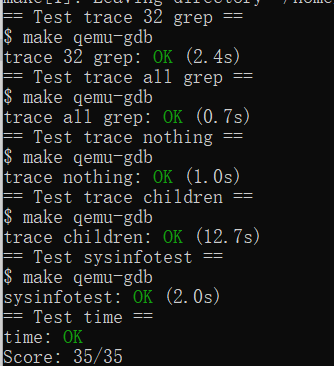
最终得到了预期的实验结果：



#### 实验心得

通过这两个实验，我对于增加一个系统调用的操作流程都更熟悉了。通过这个实验我对空闲块链表的定义及使用、对进程状态字段内容都有了进一步的认知，对于与进程状态相关的函数、定义在哪些文件里也有了一定的记忆。

以下是lab1 make grade结果：



## Lab2：page tables

### Speed up system calls(easy)

#### 实验目的

了解页表的原理和页表映射的实现。

有些操作系统(例如 Linux)通过在用户空间和内核之间的只读区域共享数据来加速某些系统调用。这样可以消除在执行这些系统调用时进行内核交叉的需要（以优化用户模式到内核模式的陷阱机制，对于某些系统调用不再需要切换模式）。此实验是为 xv6中的 getpid ()系统调用实现这种优化，进程id不会被轻易改变，但是需要经常获取，为优化其操作，我们应该将这个id放到这片区域中。

#### 实验步骤

①切换到pgtbl分支，查看USYSCALL定义和usyscall结构体定义

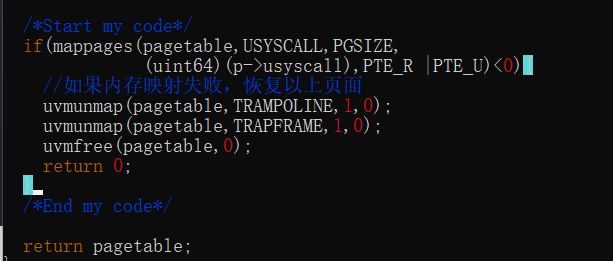


②在proc.h定义的proc结构体中添加变量“struc usyscall \*usyscall”，存储共享页的物理地址;

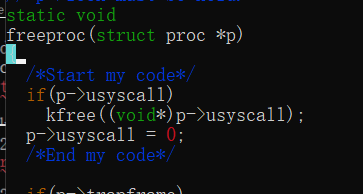
③在proc.c的allocproc()函数中，为usyscall分配一块内存并初始化：



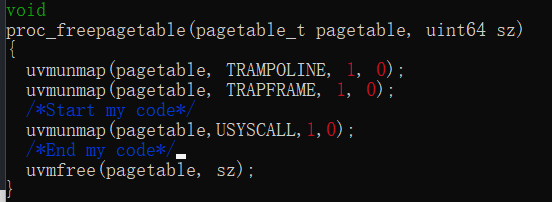
④在 kernel/proc.c 的 proc\_pagetable() 完成内存映射(mapping)：



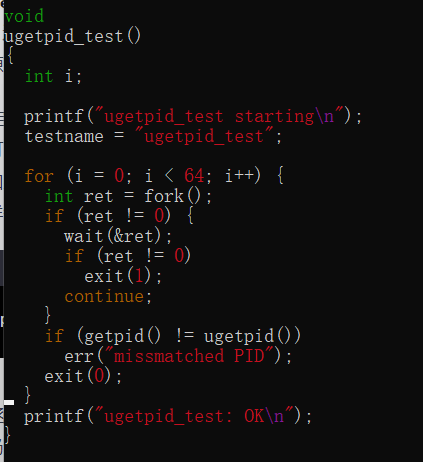
⑤在 freeproc() 函数中释放 p->usyscall 对应的物理页内存：

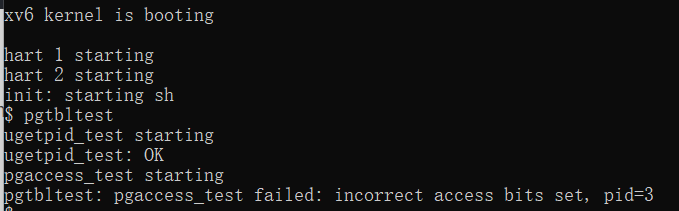


⑥在 proc\_freepagetable() 中释放页表中对应的页表项：



#### 实验中遇到的问题和解决办法

一开始当我添加好代码make qemu时，报错告诉我USYSCALL未定义，于是我到memlayout.h中去看确实没有定义USYSCALL，于是在里面补充了USYSCALL的定义，make qemu成功后我进行pgtbltest却exec failed,后来我意识到是因为我没有在一开始切换到pgtbl分支，于是我将分支切换到pgtbl，在这一个过程中我也对git的使用，对于工作区、暂存区，head区的概念有了更清晰的认识。切换成功后我就发现在这个分支中memlayout.h已经给出了USYSCALL和usyscall结构体的定义。在我在新的分支上再次make qemu时却报错：pgtbltest编译链接时找不到ugetpid函数的定义。于是我找到了pgtbltest.c中使用了ugetpid函数的ugetpid函数（如下图），也顺着头文件找到了ugetpid函数的声明，但是找不到ugetpid函数的定义，于是我怀疑用户空间中没有为我定义ugetpid函数，通过上网查询，我在ulib.c中自行添加了ugetpid函数。再次make qemu进行测试，发现测试成功。



#### 实验心得

这次实验的学习过程中，我重温了页表的概念，页表寻址的原理，知道了虚拟地址、物理地址、PTE的组成，了解了页表映射的一些规则，比如0x80000000以下用于映射设备接口，以上映射到DRAM，还了解了guard page的概念，了解了mapppages的用法：mappages(pagetable, va, size, pa, perm)是将物理地址范围pa到pa+size映射到虚拟地址范围va到va+size。它使用给定的页表pagetable和权限标志perm执行映射。

### Print a page table (easy)

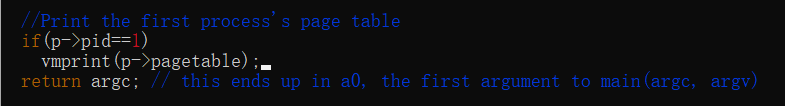
#### 实验目的

熟悉多级页表的机制

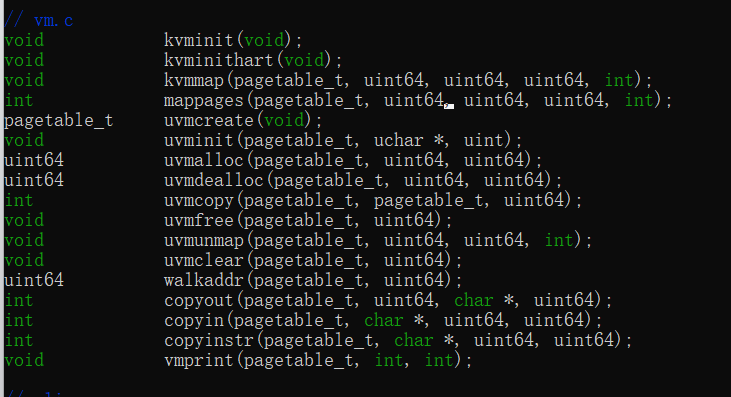
为了帮助我可视化RISC-V页表，也许为了帮助将来的调试，你该实验的任务是写一个函数来输出页表的内容。

#### 实验步骤

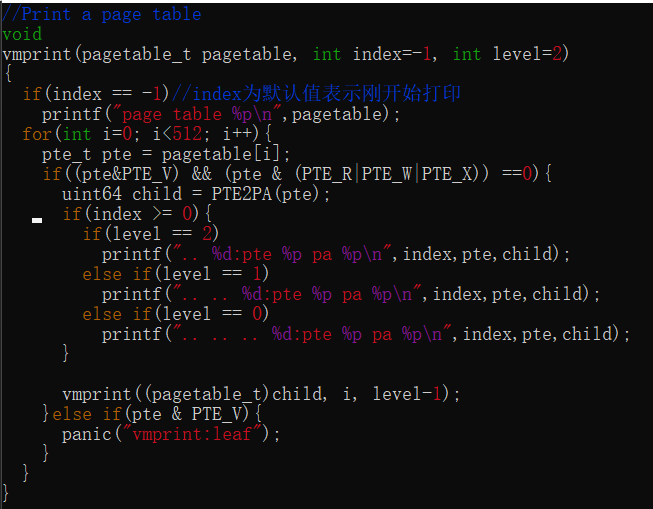
在exec.c的return argc前添加vmprint函数调用来打印第一个进程的页表



在defs.h中添加vmprint函数声明



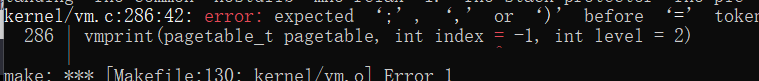
在vm.h中实现vmprint函数



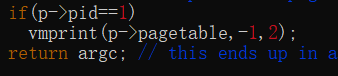
make qemu进行测试

#### 实验中遇到的问题和解决办法

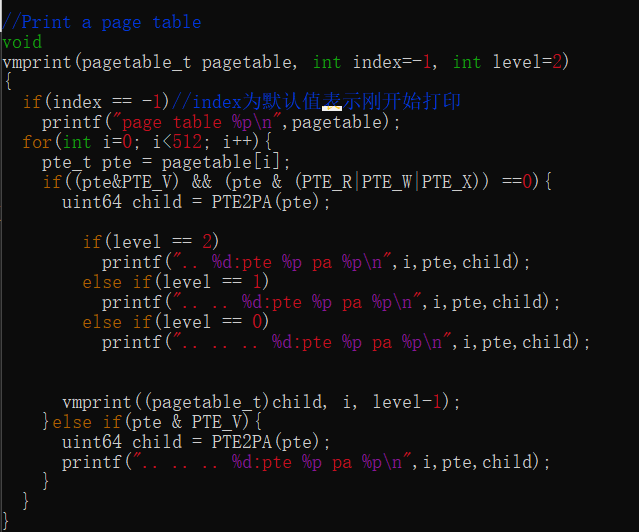
上面的vmprint函数是有误的，它并不是最终版本。上面的版本编译报错：



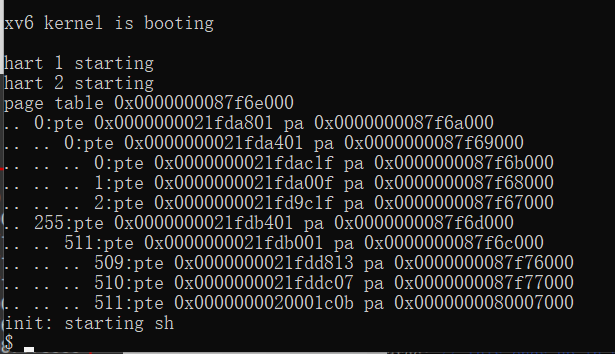
这个错误是因为在 C 语言中，函数的形参列表中不能设置默认参数值（default argument values），也就是不能在函数的声明或定义中给形参赋初值。为了解决这个问题，您可以修改函数的声明或定义，将默认参数值移除;然后在调用 vmprint 函数时，根据需要提供相应的参数值。



此外上面的代码中，当pte指向叶子页面时的处理我仿照freewalk进行处理，但是参考了网上的做法后觉得这个地方应该就当0级页表项处理输出。于是最终的函数如下：



实验结果如下：



实验结果分析：page table 0x0000000087f6e000表示根页表物理地址是0x0000000087f6e000；..0: pte 0x0000000021fda801 pa 0x0000000087f6a000表示根页目录项0，PTE内容为0x0000000021fda801，表示下一级（第二级）页表的物理页号为：0x87f6a, Flags为：0x0，bit 0 设置为1，PTE\_V有效。该物理页的起始物理地址为0x0000000087f6a0005

#### 实验心得

这个实验挺简单的，跟着提示一步步做条理挺清晰的。这个实验也让我回忆起我做操作系统第3次项目时的页表设计，让我想起页表是放在物理内存中的，这里的多级页表目录相当于当时提到的索引块的概念，又想起我当时设计页表时就是简单的用数组实现，现在我见识到了实际的页表设计起来有多复杂。通过这个实验我还学会了用%p来输出地址。

### Detecting which pages have been accessed (hard)

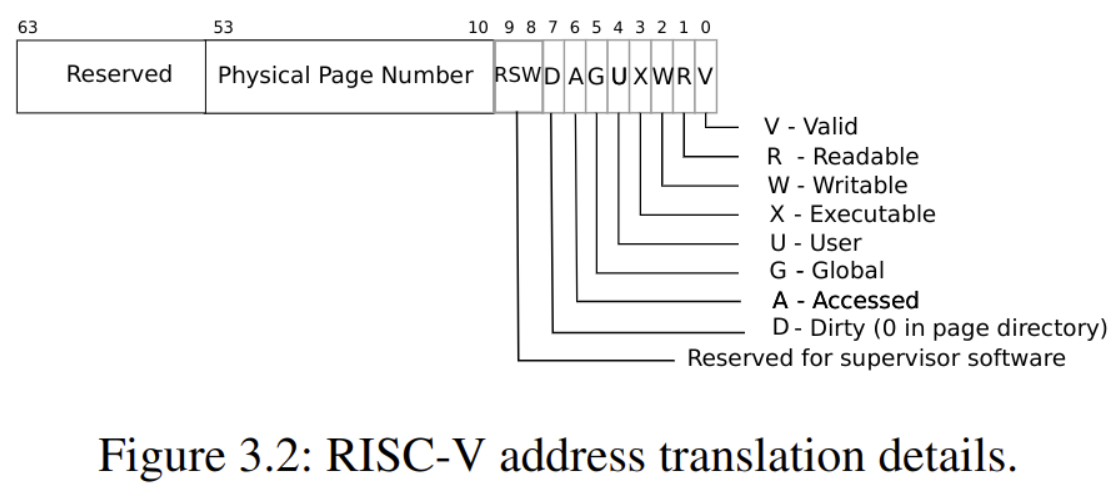
#### 实验目的

掌握PTE的内容组成，包括权限位的获取与设置，进一步熟悉页表的寻址逻辑及相关函数的使用

实现pgaccess()函数，它是一个报告已访问哪些页面的系统调用。该系统调用接受三个参数。第一个参数是需要检查第一个用户页面的起始虚拟地址。第二个参数是需要检查页数。最后一个参数用户缓冲区的地址，检查结果以位掩码（一种数据结构，每页使用一位，其中第一页对应于最低有效位）的形式存储在这个缓冲区中。

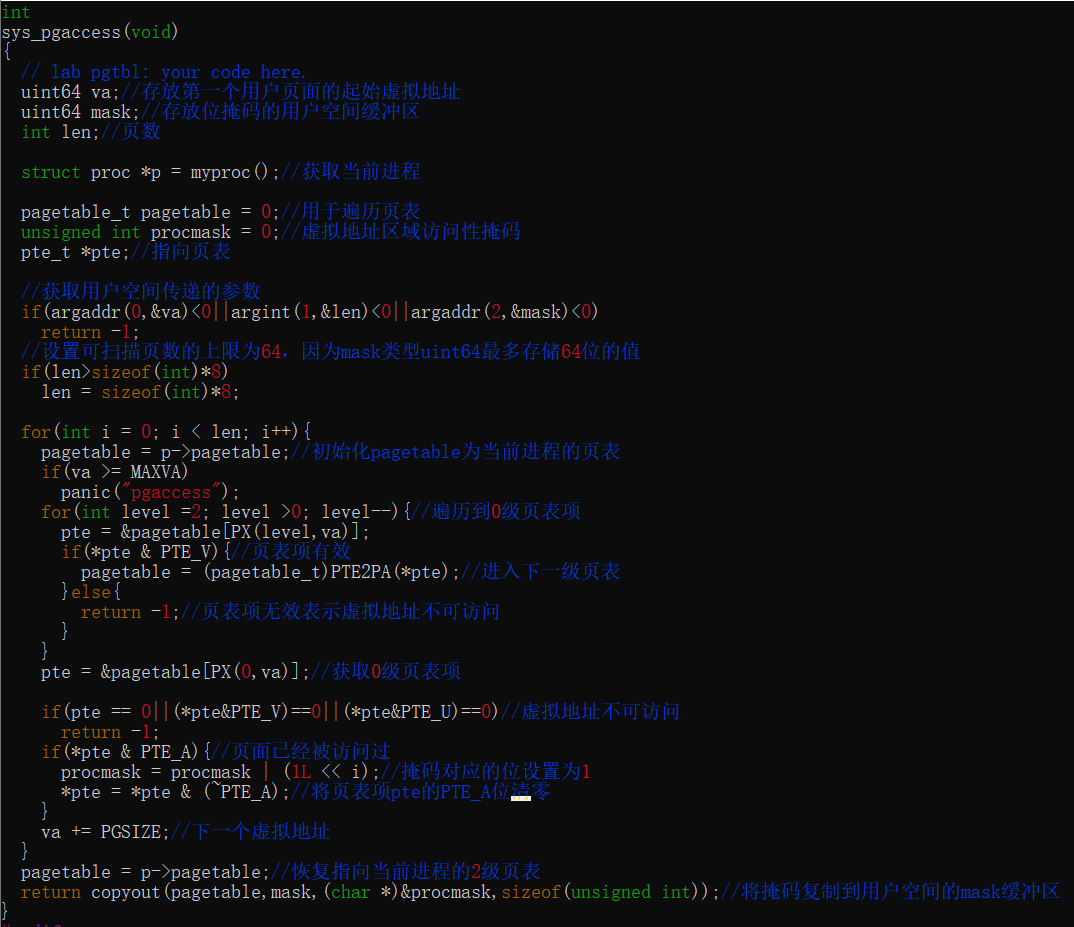
#### 实验步骤

在kernel/riscv.h中定义PTE\_A，根据下图知道访问位在第6位，于是将1L向左偏移6位得到PTE\_A





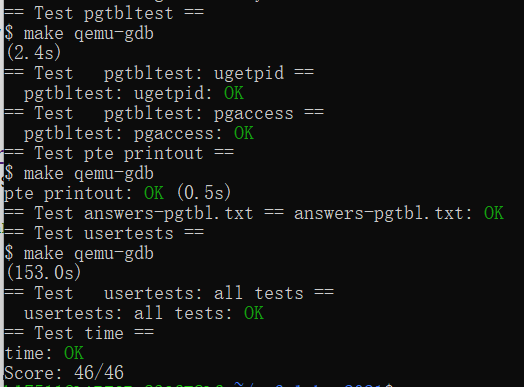
在kernel/sysproc.c中实现sys\_pgaccess函数



make grade检测实验是否通过。

#### 实验中遇到的问题和解决办法

当我去学习walk函数的时候我又对PTE，va感到有点迷惑了，当我仔细理清它们的关系之后终于有点拨云见雾了，并将我对它们的总结写在心得里面了。此外根据实验的指示我在检查后将PTE\_A清零了，但是我忘记了进入下一页遍历时更新va的值，这个问题解决后，我又发现我copyout的结果不对，通过比对发现是因为我在遍历结束后没有将pagetable恢复指向最初的页表，修改后遍便make grade通过了。



#### 实验心得

我学习walk时因为迷糊便总结了以下pa，pte的关系：根据物理地址pa可得到一个页表pagetable，页表中的某一项pagetable[i]存放的是pte，pte经PTE2PA映射可得到下一级页表的物理地址pa。通过这个实验我重新认识了argaddr、argint函数的作用和使用，就是将trampoline中的第i个参数传递给函数的第二个参数，这也加深了我对trampoline是用于保存用户态切换到内核态一些信息的记忆。这个实验还帮我复习了位运算，和copyout函数。而两个嵌套循环也让我对页的遍历，寻址更熟悉了。

## Lab3：Traps

### RISC-V assembly(easy)

#### 实验目的

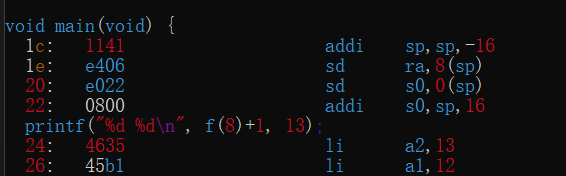
了解RISC-V的汇编，学会看汇编代码。

#### 实验步骤

跟着实验的问题学习call.asm文件，在回答问题探索问题的过程中学习。

#### 实验中遇到的问题和解决办法

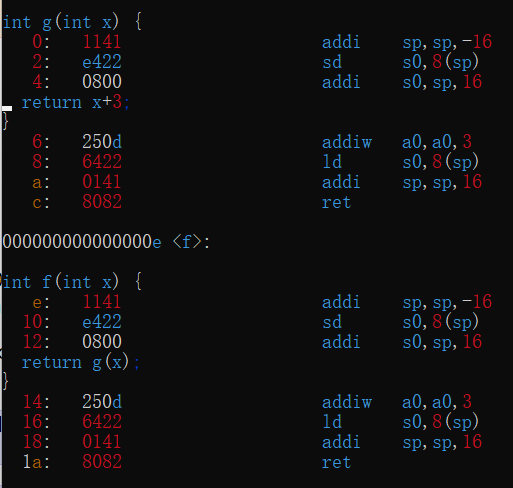
Q1: 哪些寄存器包含函数的参数?例如，哪个寄存器在main对printf的调用中保存了传参13 ?



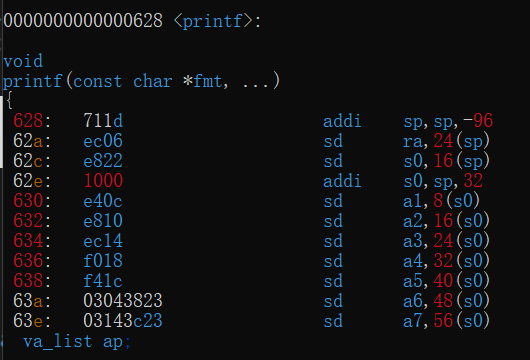
A: a0-a7包含函数的参数。由上图可看出寄存器a2保存13。

Q2:在main的程序集代码中，函数f的调用在哪里?调用g在哪里?(提示:编译器可能内联函数。)

A:查看asm文件，发现没有对f,g的调用，g被内联f函数（见下图），f函数被内联到main函数（见Q1中的图）。

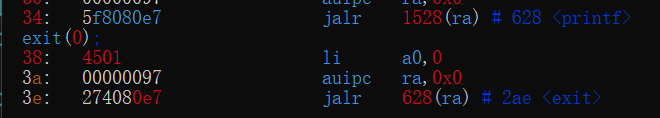


Q3：printf函数的地址在哪里？



A：在0x628

Q：main中，jalr跳转到printf之后，ra的值是多少？



A:ra保存的是函数调用后返回的地址，由上图可以看出ra的值是0x38, 对应的是0x34的下一条指令。当printf执行完成后，会让pc=寄存器ra的值，然后程序会执行main函数中的0x38这行的代码

Q: 输出是什么？xv6是小端的，如果要实现一样的输出：如果RISC-V是big-endian，为了产生相同的输出，你会把i设置为什么？您需要将57616更改为其他值吗？

A：

1. 输出是He110 World。57616，十六进制表示为 e110。小端：低地址在前，变量i会以如下方式存储在内存中:0x72 ('r') | 0x6c ('l') | 0x64 ('d') | 0x00。所以输出是He110 World。

2. 如果在大端上，为了产生相同的输出，i应该设置为0x726c6400

3. 而对于int的数据，则无论大小端输出的值都一样,所以不需要更改57616。

Q: printf("x=%d y=%d", 3); 输出结果是什么？

A：根据 C 语言的规则，当 printf 函数的参数数量少于格式控制符的数量时，它会读取内存中的随机值，并且结果是不确定的，可能输出任意的值。

#### 实验心得

当我make fs.img生成了asm文件后我终于对高级语言程序的编译链接过程有了更生动的记忆！当我打开asm文件时一开始还有点懵，但是当我看到熟悉的ld等指令时我反应过来这就是生成了一个汇编文件，里面都是汇编语言，由于计算机组成原理、计算机系统结构等课的铺垫我看到里面的内容并不觉得费解，感觉这个内容倒有点像是我们在计组实验的虚拟机上看到的那般。实验的倒数第二个问题提到小端大端，高程的时候讲过，所以一下就理解了。

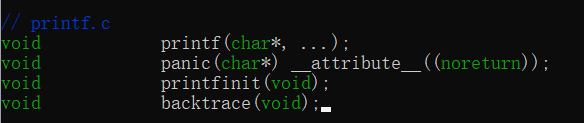
### Backtrace (moderate)

#### 实验目的

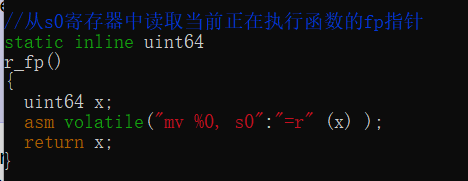
打印当前进程的调用链，熟悉stack frame的内部组成。

#### 实验步骤

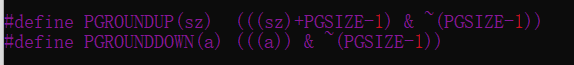
在kernel/defs.h中添加backtrace函数声明



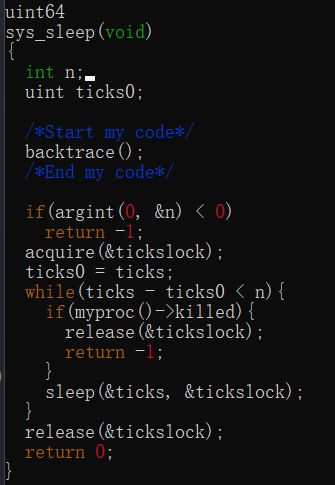
在kernel/riscv.h中添加如下函数



计算栈的最高地址和最低地址的两个表达式的定义



在kernel/sysproc.c中的sys\_sleep函数里添加backtrace调用

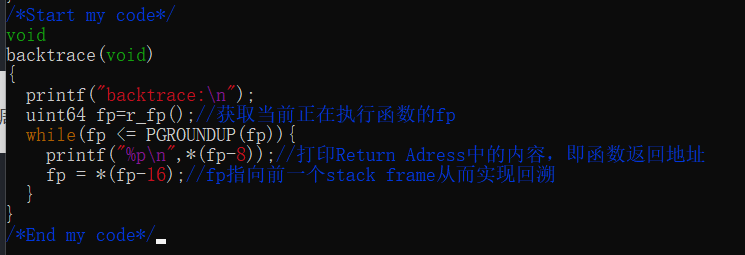


在kernel/printf.c中编写backtrace函数代码

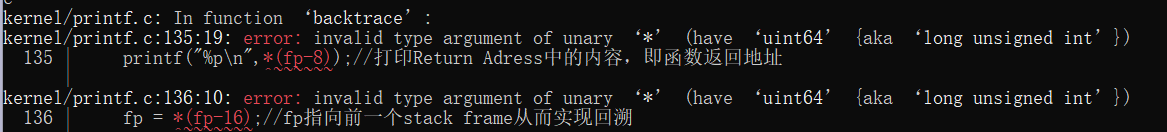


#### 实验中遇到的问题和解决办法

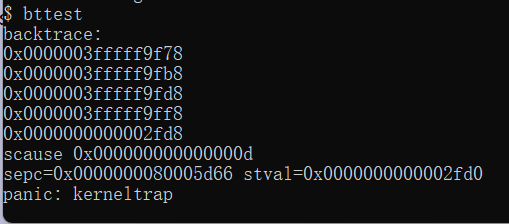
一开始我的backtrace代码如下



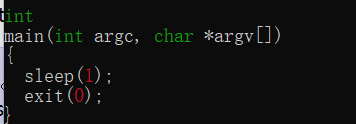
编译报错



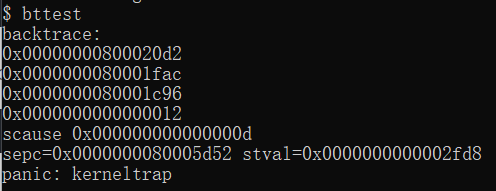
当我在两个\*前加上uint64之后编译通过但是bttest输出异常。



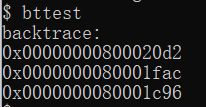
于是我去查看bttest.c想获得一些思路，发现其代码非常简单，只调用了sleep函数。

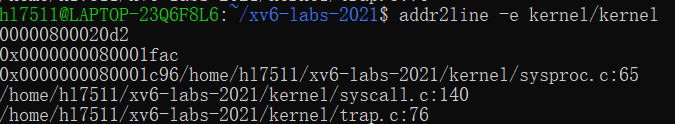


于是我想函数调用过程应该是这样的：bttest中的sleep函数触发了usertrap函数，其中又调用了syscall函数根据用户空间传递的系统调用号来调用sys\_sleep函数，sys\_sleep函数中又调用了backtrace函数。既然如此，输出就应该只有backtrace：以及三行地址才对，为什么上面出现了四行地址，而且地址都是0x0000003fffffxxxx，我听了mit的网课，根据教授所讲，这些地址接近于用户地址空间的顶部，属于trampoline。此外输出panic：kerneltrap提示我可能我的代码有一些导致内核trap的问题。

于是我首先怀疑我的输出地址时处理有误，因为地址不应该是0x0000003fffffxxxx。后来我终于发现是我对指针的理解不到位，printf("%p\n",(uint64 \*)(fp-8))输出会是fp-8处的地址而不是该地址存放的内容，因为(uint64 \*)(fp-8)只表示将 fp-8 地址强制转换为指向 64 位整数的指针，只有在前面再加个\*才能读取该指针指向的内容。这个问题纠正后，输出的地址的确正常多了。

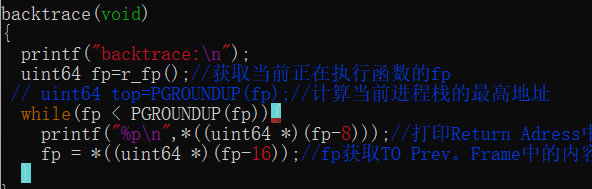
但是还是输出了四行地址，于是我去循环次数中找答案，发现我原来的代码中循环结束条件是fp<=top，但是等于不应该加上，否则到了top时fp还会想找前一个栈帧，我猜就是这里越界了导致kerneltrap，在这个地方也改正过后，我终于获得了正确答案。





#### 实验心得

做每个实验之前先看一下mit的视频都会对做实验很有帮助，通过这次学习我了解了用户地址空间的组成，知道了栈的结构，每一次函数调用都会产生一个栈帧，通过实验的hint我又得知xv6为每一个栈在xv6内核中分配了一页，所以我想一个进程拥有一个栈，而一个栈由若干个因进程中的函数调用而产生的栈帧组成，同时我还知道了如何计算栈地址的最高地址和最低地址。知道了这些，backtrace也就不难写了。此外我还发现只要在同一个栈中，哪怕指向的栈帧不同，PGROUNDUP（fp）计算出来的最高地址都相同，当我把代码改成下图那样时输出仍然正确。这个应该跟PGROUNDUP表达式有关只不过我没仔细看暂时看不懂。



### Alarm(hard)

#### 实验目的

进一步熟悉trap的全过程。

增加一个系统调用，该系统调用会在指定时间间隔后调用指定用户态的函数。

#### 实验步骤

在Makefile中添加“$U/\_alarmtest\”；

在user/user.h中添加两个函数声明：



在文件user/usys.pl中，添加如下内容：



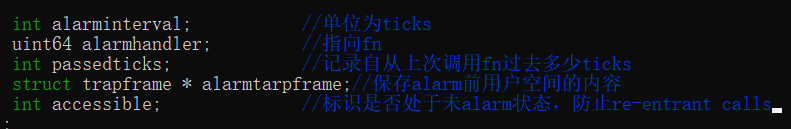
在文件kernel/syscall.c中，增加如下内容：



在文件kernel/syscall.h中，增加两个系统调用号的定义：



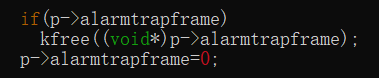
在kernel/proc.h的porc结构体中添加如下内容：



文件Kernel/proc.c，在函数allproc中为alarmtrapframe分配一页，并初始化与alarm有关的变量：



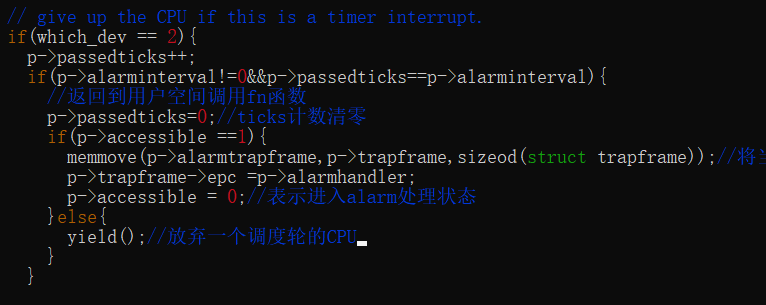
文件Kernel/proc.c，在函数freeproc中释放alarmtrapframe占用的物理内存：



在文件kernel/sysproc.c中，增加两个系统调用函数

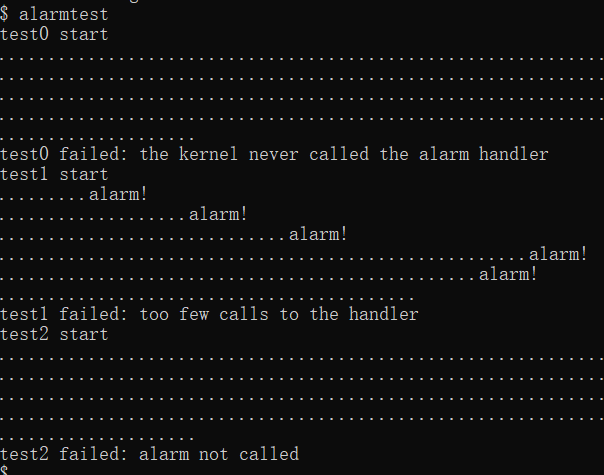


将kernel/trap.c中的usertrap代码修改为如下：

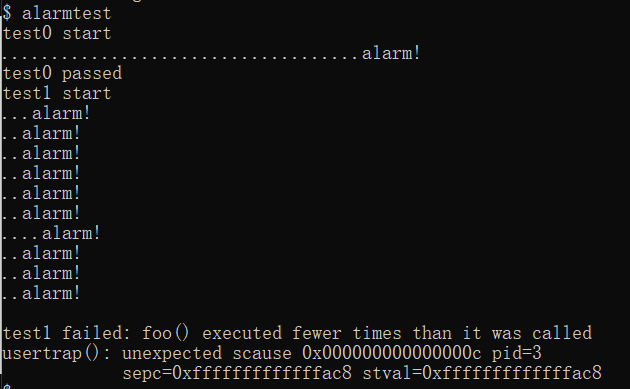


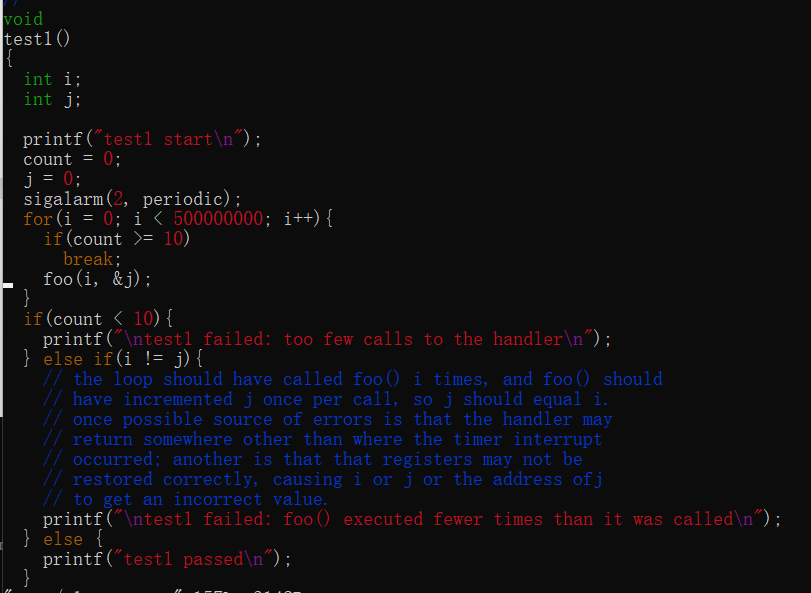
#### 实验中遇到的问题和解决办法

第一次测试我的结果如下

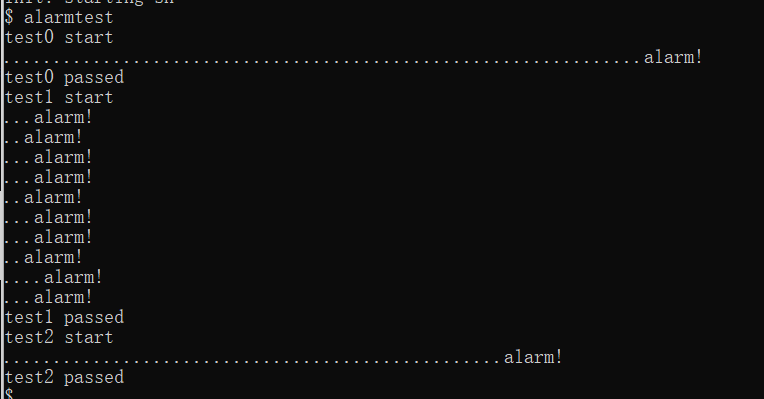


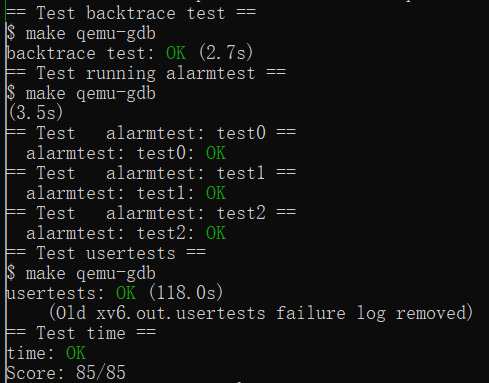
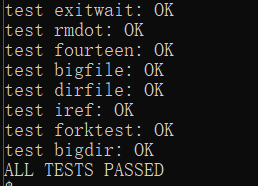
我准备打开gdb进行调试，这一过程中我的网络突然连不上了，于是直接退出程序重启电脑。重新进入ubuntu想要编译alarmtest.c文件时却发现该文件不见了，而之前alarmtest.c编译产生的文件如alarmtest.o等文件都在，我一时心急如焚，整个xv6-labs-2021中都找不到这个文件，也不知道如何恢复。这时候我突然想到我之前在traps分支的文件上做了修改后并没有提交修改，所以可以通过丢弃修改回到最初的版本来找回alarmtest.c，于是我执行git checkout .指令恢复到刚进入该分支时的版本，果不其然，找回了alarmtest.c。遗憾的是之前做的所有修改我都要重新再写一遍，不过还好每次我做了修改后都有截图记录，也正是根据我对之前编写代码的记录重新编写代码时发现之前freeproc释放alarmtrapframe后没有将其置零，但是我不知道是不是因为这个导致上图一次都未调用fn的结果。当我重新写过代码后得到如下结果



查看了test1的内容（如下图）后得知出现上图的原因可能有两种：

返回计时器中断以外的其他位置发生；另一个是寄存器可能没有正确恢复，导致i或j或j的地址以获得不正确的值。并且前面的结果后面还有不断地“...alrm”行输出，于是我突然发现我忘了在alarm时将accessible置0，这就导致sigreturn一直没有将alarmtrapframe中的内容复制给trapframe，即导致寄存器没有正确恢复，从而导致i,j地址不正确，在我加上置零语句后便得到了预期结果。





但是我上面的结果中test0的“.”的输出比官网上的多很多，我想这应该是因为我的CPUtimer interupt时间间隔比较长，使得过很久才能增加一次passedticks计数，从而当passedticks累积到alarminterval时test0函数已经输出了较多“.”。

#### 实验心得

这个实验首先阅读题意费了点时间，其次就是要改的文件很多，但其实编写代码并没有很复杂，这个实验让我知道编写操作系统时一定要心细，要全局考虑清楚，不然一点点的错误都会导致严重的后果。这个实验也让我对trap的过程有了更深刻的记忆，通过这一章的实验和学习，我认识到了很多trap所用到的软硬件结构包括内核中表明执行系统调用时计算机状态的PC,MODE,SATP,SEPC,STEVC等寄存器，函数调用时的栈桢，而这些都是os课上来不及展开的内容，总之收获颇丰。

## Lab4：Copy on-write

### Implement copy-on write(hard)

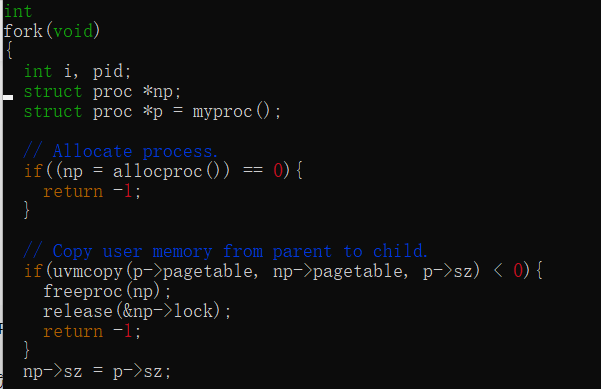
#### 实验目的

熟悉内存映射、内存分配的原理及相关函数，宏观把握进程对内存的使用，初步了解锁的使用。

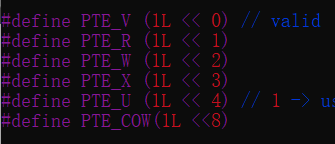
xv6中的fork() system call，复制parent进程所有的用户空间内存到child。

如果parent是非常大的，copying将花费很长时间。糟糕的是，这个工作通常是大量浪费的；例如，fork()后紧接着是exec()在child进程中，这将导致child会丢弃拷贝的内存，可能绝大多数都不使用。另一方面，如果parent和child使用一个page，并且其中一个或两个写，那么确实需要一个副本。于是让内存复制采取copy-on-write策略，即推迟对child的分配和拷贝物理内存页，直到拷贝确实需要。COW fork()仅仅给child创建一个pagetable，其用户内存的PTES指向parent的物理页。COW fork()标记parent和child的所有用户内存PTES是不可写的。当某个进程尝试写其中一个COW页时，cpu将强制一个page fault。此外一个给定物理页可能被多个进程的page table指向，仅应该在最后的指向消失时，才释放物理页。

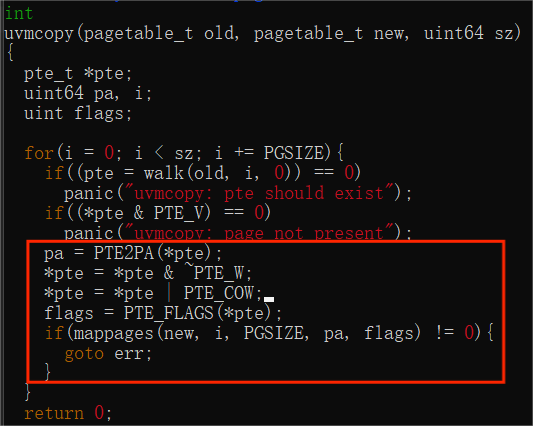
#### 实验步骤



修改kernel/riscv.h，新增PTE\_COW标志位,便于page fault时的判断处理，根据hints，可以使用RSW预留标志位即第8-9位来标志是否是COW page：



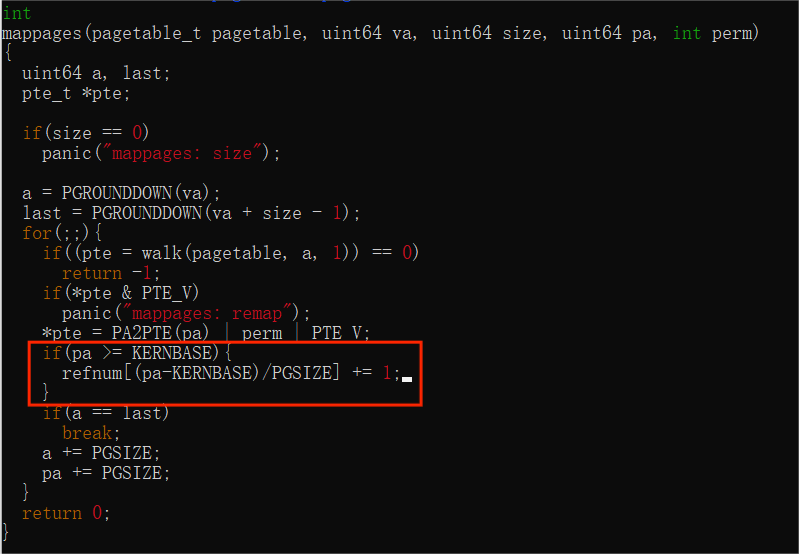
更改kernel/vm.c中的uvmcopy()来映射parent物理页到child，而不是分配新页。清除parent和child PTES的PTE\_W:



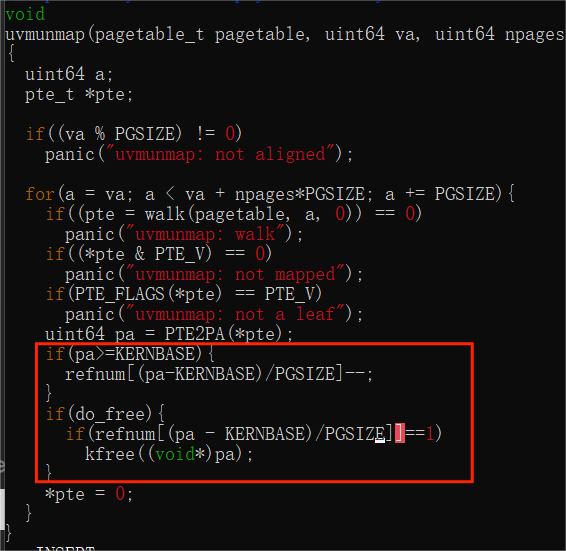
修改kernel/vm.c，新增int refnum[]来记录关联物理页的页表数量，其中PHYSTOP，KERNBASE指示了physical memory的起始和终止地址，它们相减除以PGSIZE即得到物理页的数量也即refnum数组大小：



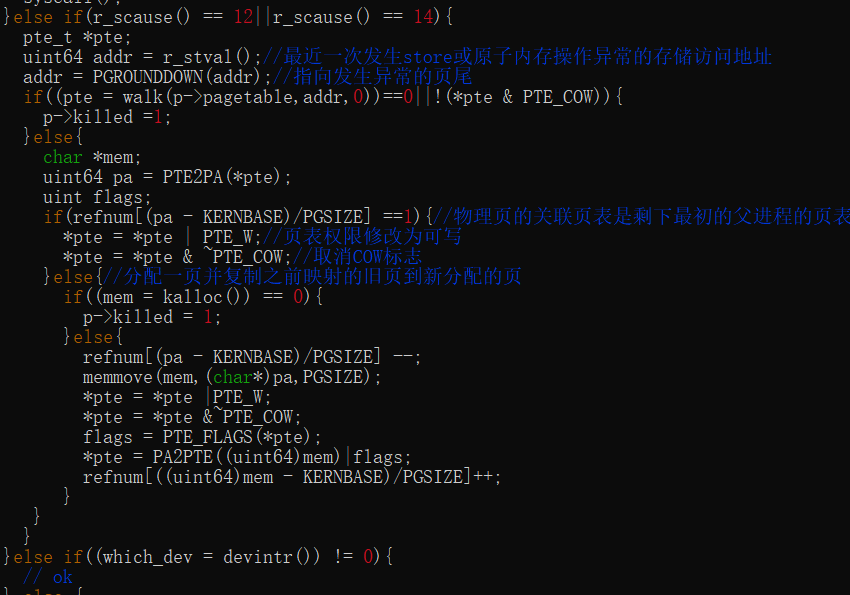
修改kernel/vm.c的mappages()，在页表与物理页绑定时，增加refNum对应元素计数：



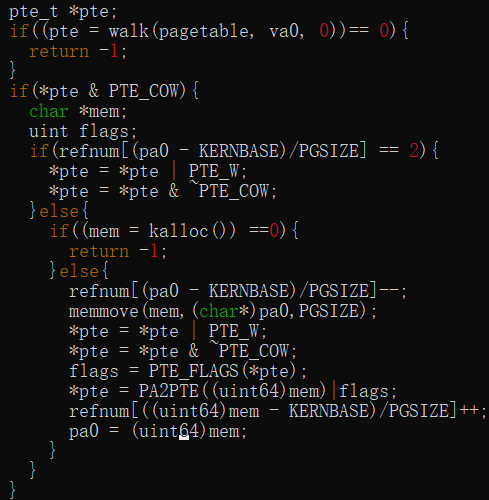
修改kernel/vm.c的uvmunmap()，在页表与物理页解绑时，减少refNum对应元素计数，当refNum==1即仅kernel pagetable持有时，释放内存：



修改kernel/trap.c的usertrap()，引入refnum，在发生page fault时，若该虚拟地址关联的PTE，表明关联的物理页是一个COW页，则新申请一个物理页，让此虚拟地址指向新物理页，并修改refNum计数：



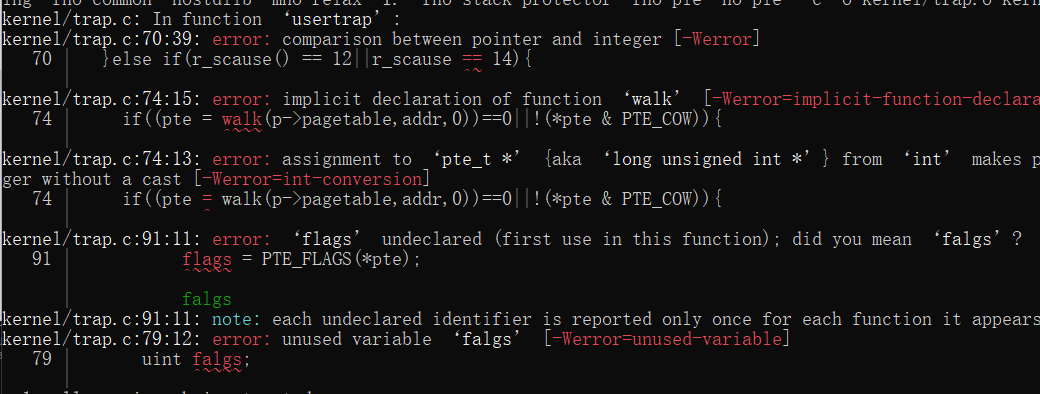
修改kernel/vm.c的copyout()，同kernel/trap.c的usertrap，在while循环中加入以下代码：



（非完整实验步骤，剩余实验步骤补充在3）④⑤）

#### 实验中遇到的问题和解决办法

①

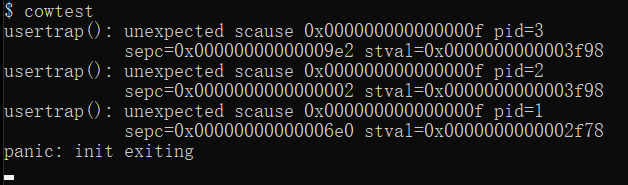


报错原因是在trap.c中使用了walk函数，但walk函数未被包含在defs.h头文件中，于是在kernel/defs.h中添加如下定义：

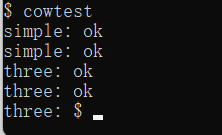
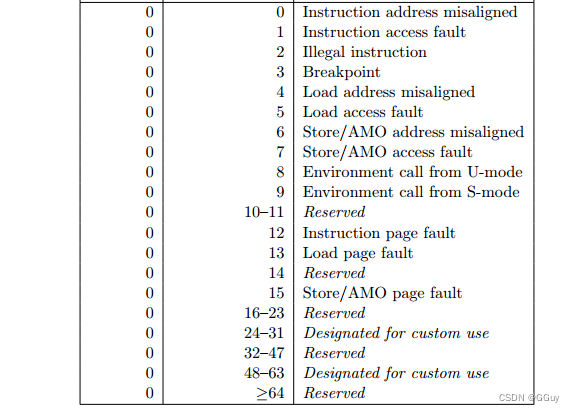


②

于是编译通过，但测试结果失败：

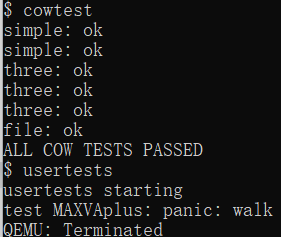


去usertrap函数中找问题，出现上面的输出，说明并没有检测到内存访问不命中，于是我怀疑是我r\_scause的判断条件值设置错误，gpt告诉我scause寄存器中值为12和14时分别表示Load page fault和Store/AMO page fault。后来经过查询才得知这两个错误对应的值应该为13和15（左下图为scause寄存器值表），改正后，结果如右下：



③

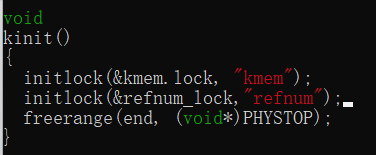
我将trap.c中的refnum==1判断语句改成==2了，因为内核页表和用户页表是分离的，ref为2是把用户页表和内核页表都映射到了同一页，此时没有其他进程引用这一页，因此==2时可以直接改为写权限，于是得到以下结果：



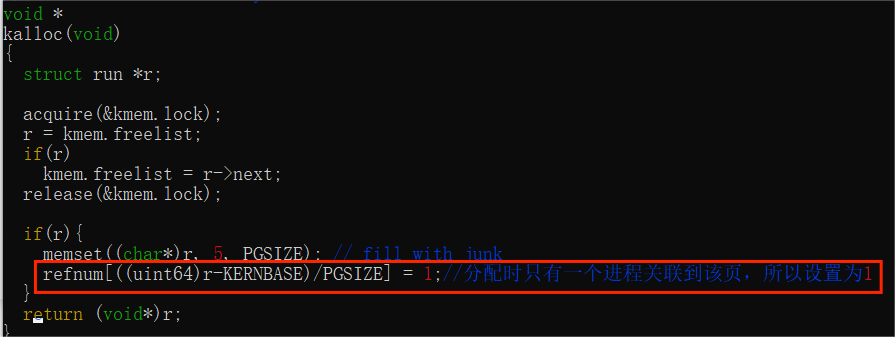
④

从上图可见usertests没有正常运行，我重新看了遍实验文档发现没有对kalloc、kfree、kinit修改，以及对refnum的处理没有上锁。于是添加以下实验步骤：

在kernel/kalloc.c中的kinit函数仿照着为refnum添加锁结构：



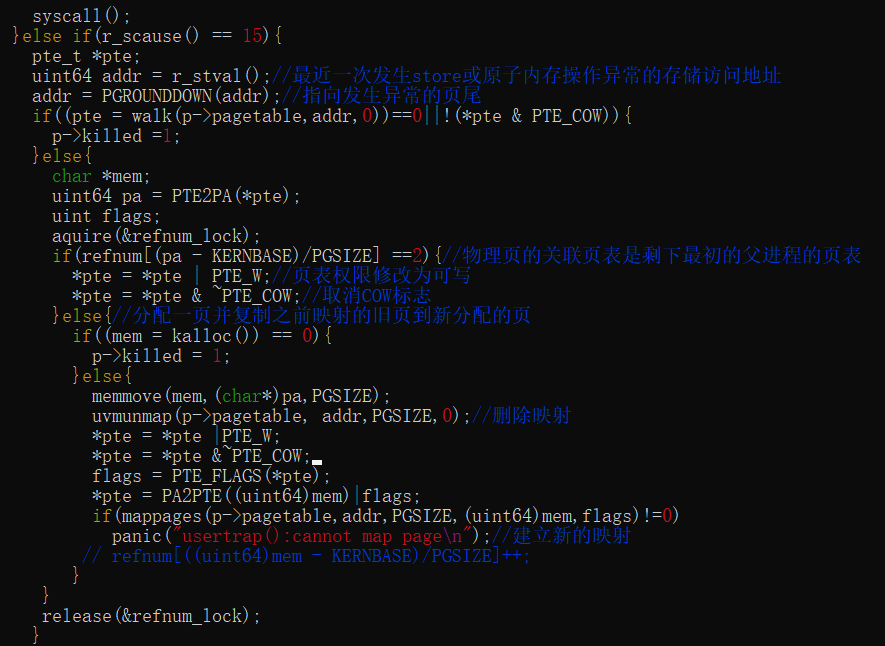
修改kernel/kalloc.c中的kalloc函数，使得当kalloc（）分配页面时，将页面的引用计数设置为1：



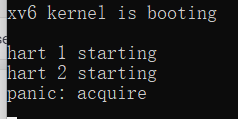
修改kernel/kalloc.c中的kfree函数，使得refnum减1，仅当refnum<=0时才释放该物理页。



再在usertrap、mappages中用到refnum的地方加上锁，修改后的两函数如下：

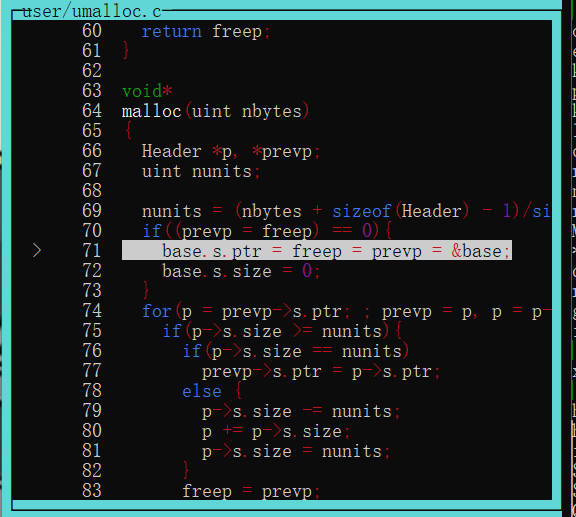
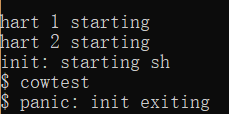


⑤此时make qemu出现以下结果：

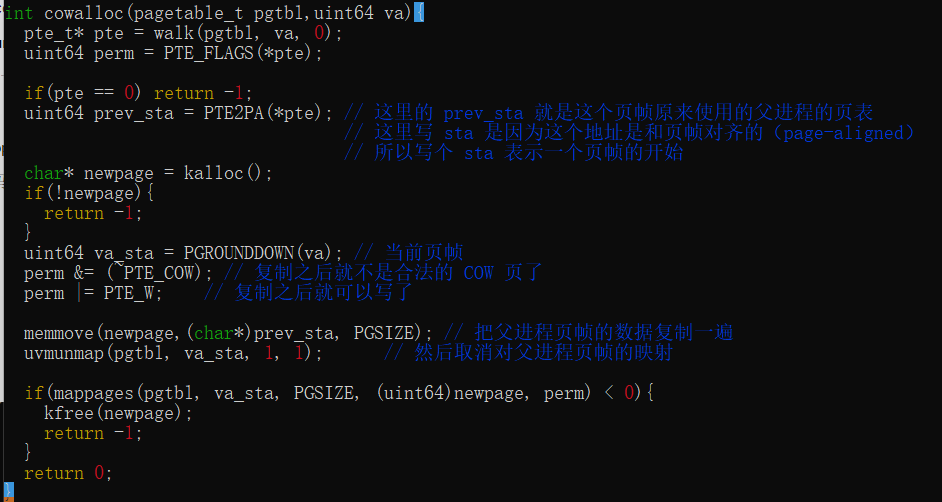


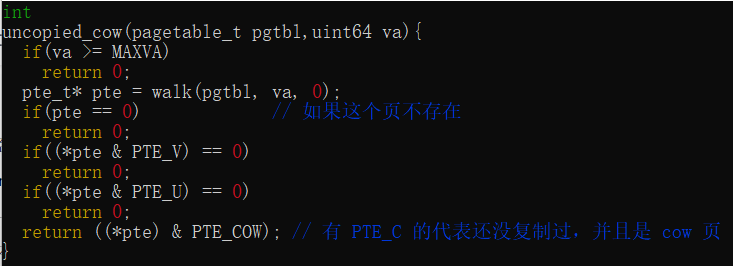
通过阅读main.c代码和查阅资料，我知道了上面结果的出现可能是当主核心设置 started = 1 后，其他核心被唤醒并输出 hart x starting，然后它们会继续执行 kvminithart() 和 trapinithart() 等初始化函数。然而，在这些初始化函数中可能存在对共享资源（如内存分页表）的竞争条件，因为其他核心可能尚未完全初始化。

panic: acquire 的错误可能是由于某个进程或线程尝试在其他核心还没有正确初始化共享资源的情况下访问它，导致了死锁或数据不一致。这是由于多核并行初始化所引起的问题。同构添加输出测试发现问题出在usertrap函数中分配新页并将旧页复制到新页的那一部分函数，在那一部分我终于发现是我调用uvmunmap时参数填错了，第三个参数npages我写成了PGSIZE，于是我将它改为了1。当我修改好后make qemu通过了但是当我执行cowtest时，出现左下的结果，经过gdb调试发现是在sbrk 函数增加程序的数据段大小时跳转到了右下图出，在该处之后发生了panic，所以应该是前面代码中内存分配存在问题。

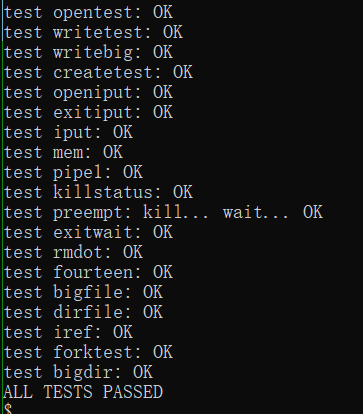
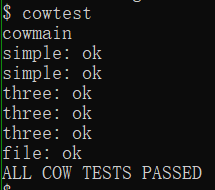


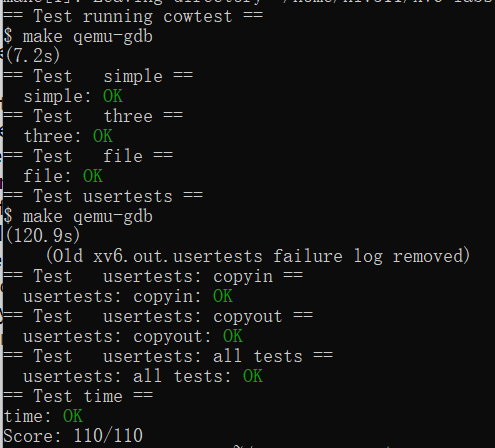
我重新检查与内存分配相关的代码发现我的uvmunmap与kfree逻辑上其实是重复的，因此我已经在kfree函数中做了对cow页的判断和处理，那么在uvmunmap时就只要if(do\_free)就进入kfree函数即可，并且我在usertrap中调用uvmunmap解除父进程页面与子进程页表绑定时将do\_free参数应总是设置为1。所以uvmunmap函数本不需要修改。随后我又参考网上的代码重新整理了我的usertrap函数、copyout函数，将它们逻辑重复的代码提取出来写在两个新建的函数cowalloc、uncopied\_cow中，这两个函数分别负责分配一个物理页给写失败的进程、判断某一页是否是未复制的cow页。





此外，我将mappages中对refnum计数加一的代码放到了umvcopy中调用mappages的代码后面。于是得到了正确结果。右下图为usertests结果。





#### 实验心得

耗时两天终于做出来了！好难，好累。刚开始的时候理解了题意我还觉得也没有那么难，无非就是fork的时候不分配新物理页，让父进程物理页直接映射到子进程页表，子进程要向物理页写数据时再将父进程的该页复制一份给子进程让它去写。但是实现起来真的不容易，编写好代码后，不停出现各种错误，从白天到黑夜将近调试了2天才成功。在这一过程中我学习了kernel/main.c代码，学习了scheduler代码，学习了spinlock锁的使用，学会了如何用make qemu-gdb调试代码，发现了qemu好像模拟了3个核，知道了在acquire一个锁和release该锁之间不能调用一个也用到该锁的函数，否则在执行内层函数时会检测到该进程已拥有该锁，发生死锁从而报错……总之每次报错都逼我学到了不少东西，我相信它们会为我之后的实验打下一定基础。

## Lab5：Multithreading

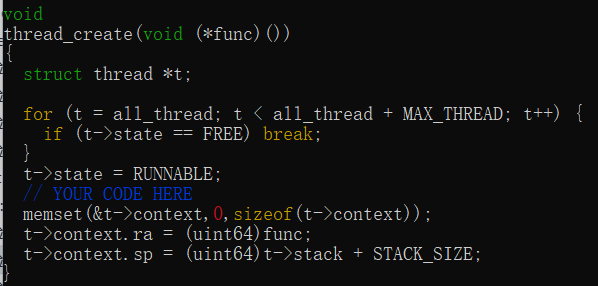
### Uthread: switching between threads (moderate)

#### 实验目的

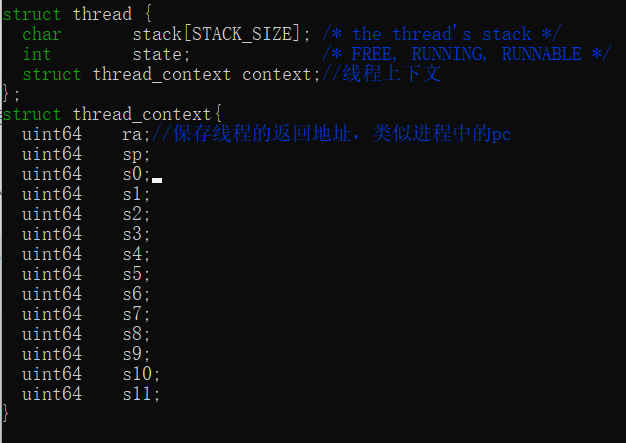
实现一个简单的用户级线程,掌握调度和上下文切换方法，了解多线程的概念。

#### 实验步骤

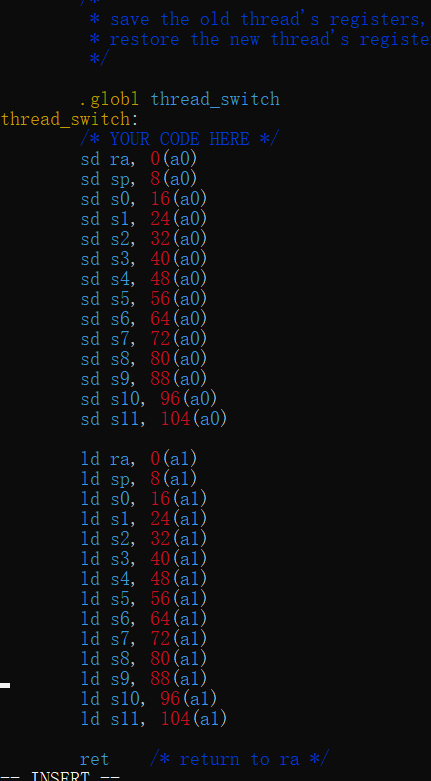
修改user//uthread.c中的 thread\_create 来保证当 thread\_schedule 第一次运行 thread\_create 创建出来的线程时，该线程就会在自己的 stack 上执行传递给 thread\_create 的函数：



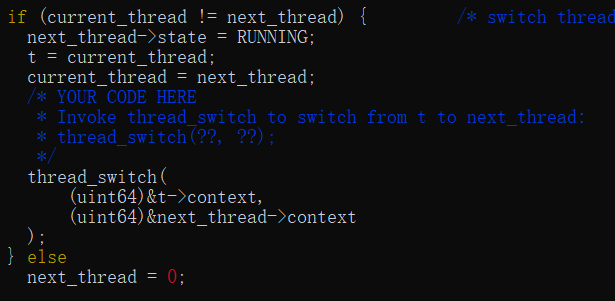
在user/uthread.c 中的thread结构体中增加context结构体保存上下文，并仿照kernel/proc.h 里面的 struct context定义struct thread\_context:



仿照 kernel/swtch.S在 user/uthread\_switch.S 中实现上下文的切换:

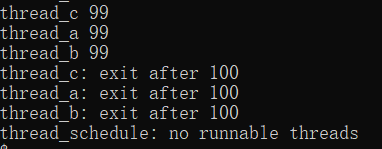


在user//uthread.c中的 thread\_schedule()中添加thread\_switch调用：



#### 实验中遇到的问题和解决办法

这次实验对我来说不算很困难，因为题目中给出的指示比较明确，代码中连哪里需要添加代码都标注好了，而且在copy-on-write实验时，在调试过程中我已经看过并学习到了kernel中的switch、scheduler函数，知道了内核线程的调度和上下文切换是如何进行的，仿照内核线程的处理过程这个实验也就不难实现，实验过程中除了一些简单的拼写错误就没有其它大问题了，以下是最终的实验结果：



#### 实验心得

其实上个cow实验我曾经想先放着最后再去看，于是先来看了这个实验的题目，看完后我开始进行相关学习，看了mit视频，我可以回答实验中的一个问题：“thread\_switch needs to save/restore only the callee-save registers. Why?”因为这类寄存器的值不可被重写，callee-save 寄存器是由被调用函数即 thread\_switch() 进行保存的, 在函数返回时已经丢失, 因此需要额外保存这些寄存器的内容。而另外一类calller寄存器的值可以被重写它们都会被保存在线程的堆栈上, 在切换后的线程上下文恢复时可以直接从切换后线程的堆栈上恢复 caller-save 寄存器的值。

### Using threads (moderate)

#### 实验目的

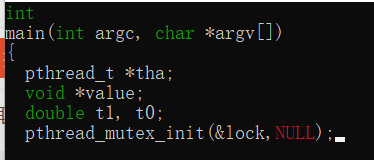
学习使用锁，了解锁的粒度对并行加速的作用。

#### 实验步骤

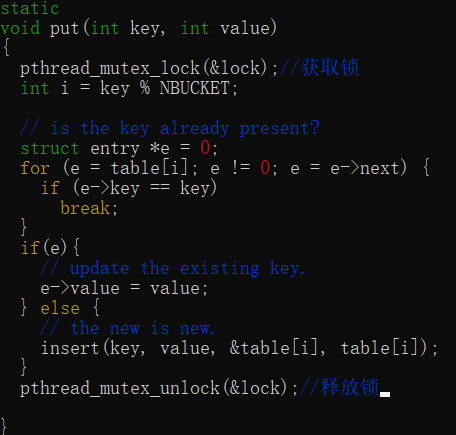
在notxv6/ph.c中声明一把锁



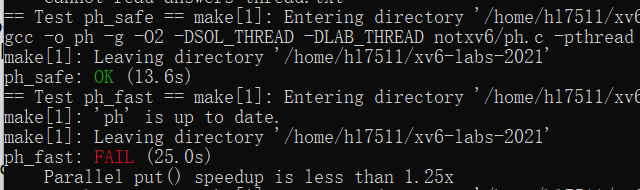
在notxv6/ph.c的main函数中初始化锁：



在notxv6/ph.c的put函数中，进入函数是加上锁，退出函数前释放锁，以实现put对哈希表的互斥访问：



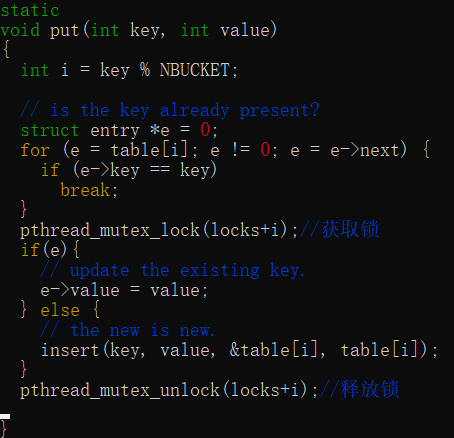
至此，实验结果如下：

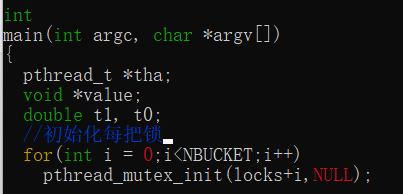


正如实验题目中所说现在还能通过ph\_safe，但是通不过ph\_fast。在某些情况下，并发put()操作在哈希表中读取或写入的内存没有重叠，因此不需要锁来相互保护。按照实验提示，为每个哈希桶加一个锁或许可以提高puts的并行加速。

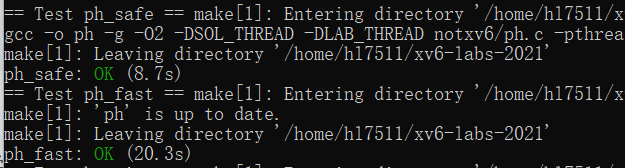
于是将上面的代码修改如下：







于是实验结果：



#### 实验中遇到的问题和解决办法

这个实验也不困难，而且在copy-on-write实验中我已经对锁的使用有了一些刻骨铭心的经验教训了，而这里只用在一个函数put中使用锁，所以没有很复杂，要添加的代码也就几句，这个实验总体没遇到什么问题。

#### 实验心得

这个实验很会循循善诱，先让我们知道多核模式下不同进程可能通过put函数对内存的哈希表同时访问从而造成数据丢失，于是让我们为哈希表加上锁来实现这一变量的互斥。当我们实现了put函数的安全操作后，又引导我们将锁的粒度减小，从一个哈希表一个锁减小到每一个桶一把锁，这样不同进程都要读写哈希表但是读写的是不同桶时它们可以互不干扰同时进行，从而提高了put速度。我系统结构课的小组汇报选题就是多核并行编程模型的探讨，我负责的部分是并行编程优化，当时我在查阅资料时看到提高并行编程速度的方法有减小锁的粒度，当时由于对锁理解还有点糊里糊涂，没懂这句话，现在通过实验我终于理解了。

### Barrier(moderate)

#### 实验目的

实现一个屏障点，使所有线程都到达这个点之后才能继续执行。

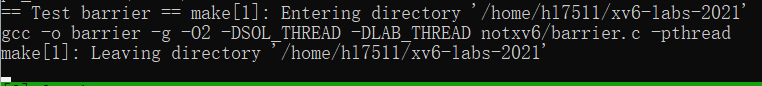
#### 实验步骤

在notxv6/barrier.c中的barrier函数中编写代码：

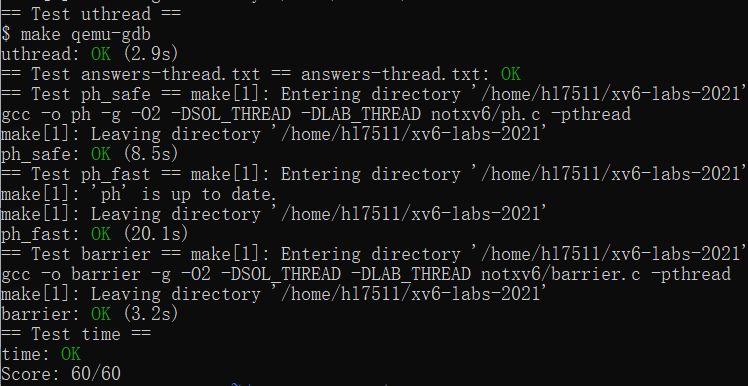


#### 实验中遇到的问题和解决办法

一开始我的代码中并没有在函数的最后一行unlock释放锁



导致我等了很久都没有出结果，我一开始没有unlock是因为我觉得每个线程都会先到else语句中释放锁并进入等待，但是我仔细想想后发现wait只在cond满足时才会释放锁并让thread进入等待，所以还应该在最后加上unlock语句，加上之后，结果如下：



#### 实验心得

这个实验也不算难，就是要细心，获取了锁要记得一定有个地方释放，不然就会像我刚开始一样进入死锁半天都没有出结果。通过这三个实验我认识了一些线程有关的工具，复习了一遍锁的使用，并开始在并行编程中注意到锁的粒度。

## Lab6：network driver

### network driver(hard)

#### 实验目的

了解驱动程序的概念和作用，了解用户与设备进行交互的底层原理。

本实验主要是写一个网卡驱动，在已有的框架下添加一些代码。完成两个函数，一个用于发送信息，另一个用于接收信息。

#### 实验步骤

完成kernel/e1000.c中的e1000\_transmit函数：

根据提示：

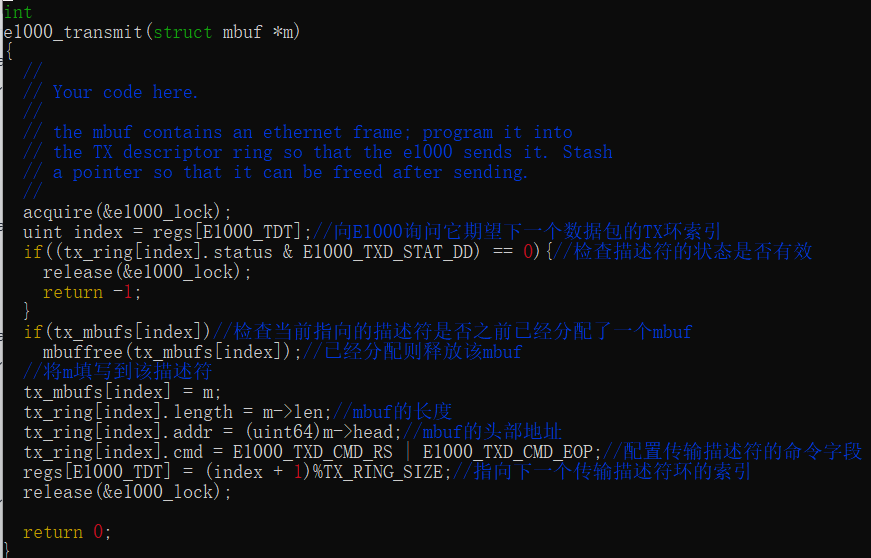
①通过读取 E1000\_TDT 控制寄存器，向 E1000 询问下一个期望发送数据包的传输描述符环的索引。

②检查传输描述符环是否溢出。如果在由 E1000\_TDT 索引的描述符中未设置 E1000\_TXD\_STAT\_DD 标志，表示 E1000 尚未完成相应的前一个传输请求，因此返回一个错误。否则，使用 mbuffree() 函数释放最后一个从该描述符传输的 mbuf（如果存在的话）。

③填充描述符。m->head 指向内存中数据包的内容，而 m->len 是数据包的长度。设置必要的 cmd 标志（查看 E1000 手册中的第 3.3 节），并储存 mbuf 的指针以便稍后释放。

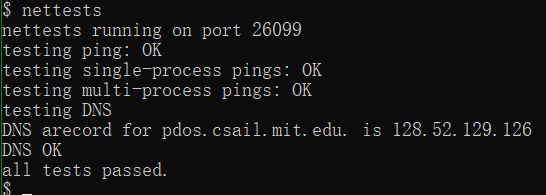
④通过将 E1000\_TDT 增加 1，并对 TX\_RING\_SIZE 取模来更新传输描述符环的位置。

⑤如果 e1000\_transmit() 成功将 mbuf 添加到传输描述符环中，返回 0。如果失败（例如没有可用的描述符来传输 mbuf），返回 -1，这样调用者就知道需要释放 mbuf。

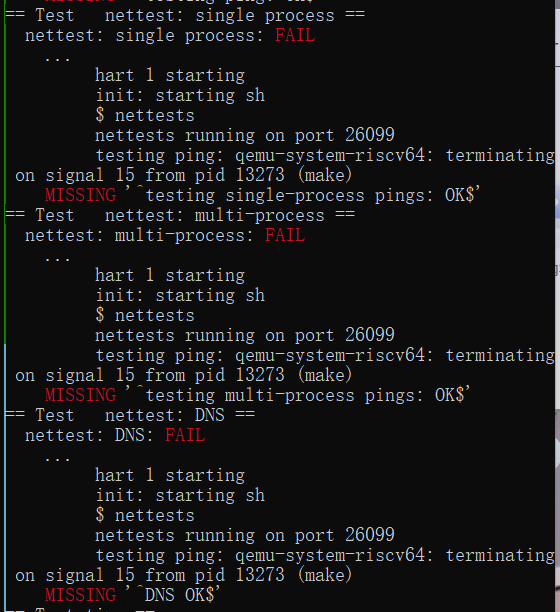


#### 实验中遇到的问题和解决办法

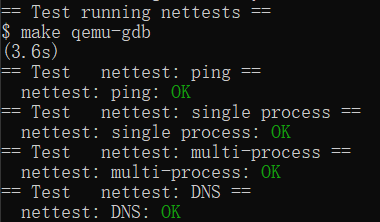
make qemu运行nettests通过了



但是make grade不通过



上面说missing“DNS OK”，但是我在make qemu nettests中的结果明明有这个消息，于是我怀疑不是程序的问题而是grade测试环境的问题，我一看网络，果然不知道什么时候断开了，于是我重启了电脑再次make gradde，果然成功了：



#### 实验心得

这次实验题目中的提示给的信息很多足够明确了，理解了要干什么，理解了实现逻辑跟着提示一步步去做就不难完成，只需要在同一个文件中的两个函数下添加一些代码。通过这个实验我学到了一些新东西，关于驱动程序和中断处理程序的。通过学习xv6文档我知道了用户在读一个文件的时候，这个文件的数据将会被拷贝两次。第一次是由驱动从硬盘拷贝到内核内存，之后通过 read 系统调用，从内核内存拷贝到用户内存。同理当在网络上发送数据的时候，数据也是被拷贝了两次：先是从用户内存到内核空间，然后是从内核空间拷贝到网络设备。把握这一点帮助了我理解题意，知道它的发送接收是从数据从哪传到哪。这次实验也提示我们锁的使用，我想写数据的时候应该是互斥的，但读不互斥所以在transmit函数中加了锁，在recv函数中没用锁，结果证明我的想法是行得通的。

## Lab7：Locks

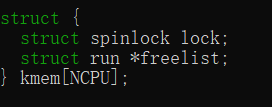
### Memory allocator (moderate)

#### 实验目的

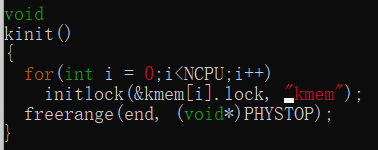
熟悉锁的使用和内存分配、空闲链表的使用，为了提高并行度让每个cpu有自己的freelist和访问它的锁。

#### 实验步骤

在kernel/kalloc.c中通过创建kmem数组让每个cpu有一个专属的锁和freelist：



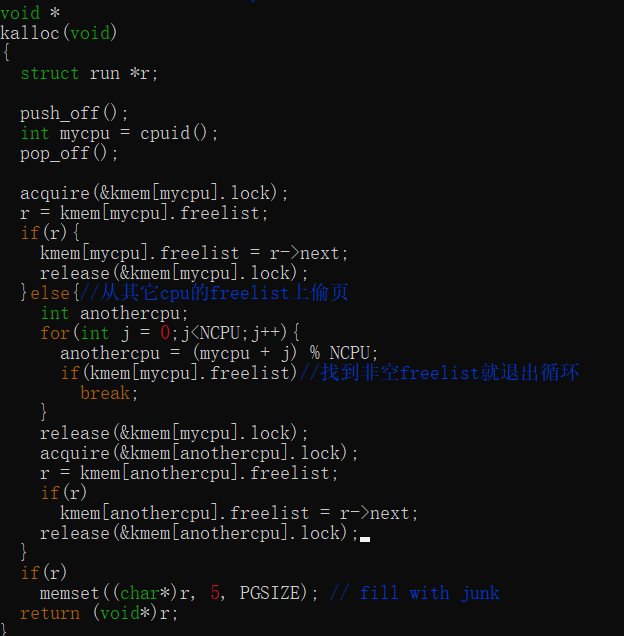
修改kinit，初始化每个CPU的锁



修改kfree：

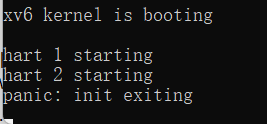


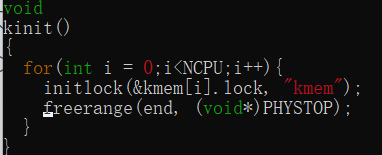
修改kalloc函数，让某一cpu没有空闲页的时候可以向其它cpu的freelist中偷一个过来：



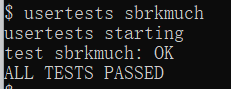
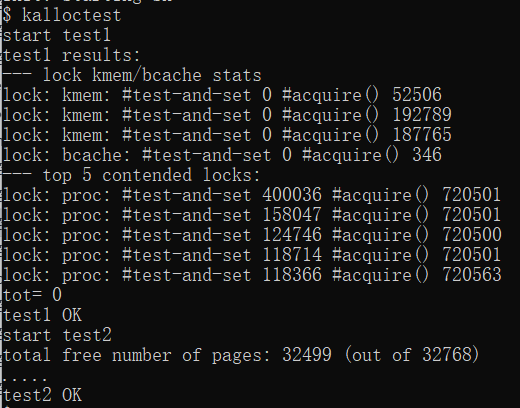
#### 实验中遇到的问题和解决办法

按照上面的步骤编写完代码后就出现了如下情况，之前遇到过，可能是内存分配出了问题我去检查代码，发现kinit中没有为每个cpu都freearange。

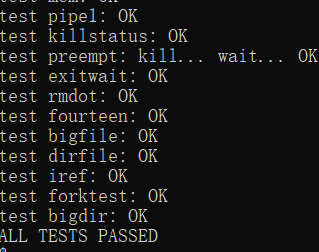


于是我将freearange调用包含到for循环里：

但结果还是和之前一样。我再仔细想了想觉得不应该把它加到循环里面，不然cpu0初始化的时候就会无意义地执行NCPU遍freearange。此外我突然发现我的kalloc中判断别的cpu上的空闲链表是否为空时用的数组索引写成了mycpu而不是anothercpu，这样就永远找不到一个可以偷的freelist了，于是其它cpu得不到任何内存分配，所以才会出现上面的报错。我将代码修改后结果如下：



usertests：



#### 实验心得

这次实验也没有很难，除了犯了个粗心错误其它没有什么问题。kalloc.c中的函数已经在copy-on-write实验中熟悉过了，所以这次写起来还比较得心应手，算是再复习了一遍其中的内容和锁的使用。

### Buffer cache (hard)

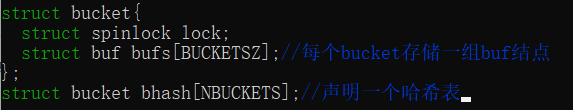
#### 实验目的

了解Buffer cache，通过减小锁的粒度，创建一个每个bucket一把锁的hash表提高并行访问disk的速度。

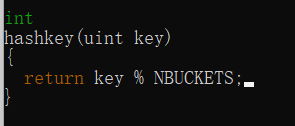
#### 实验步骤

创建一个每个桶有一把锁的hash表：

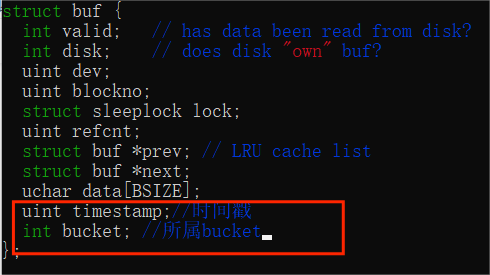




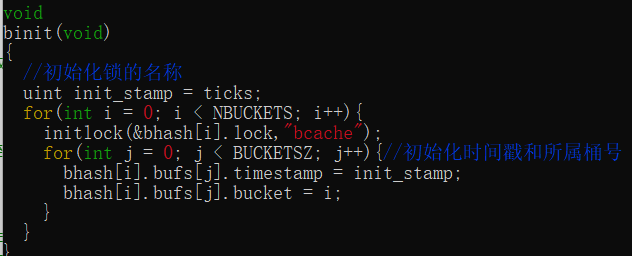
设置散列函数：



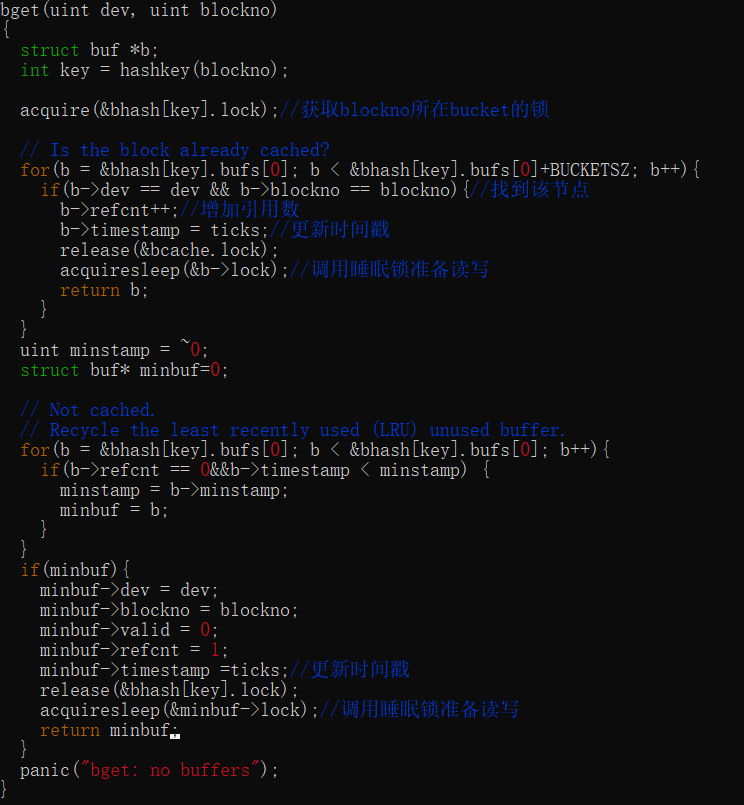
在kernel/buf.h中为buf结构体增添两项：



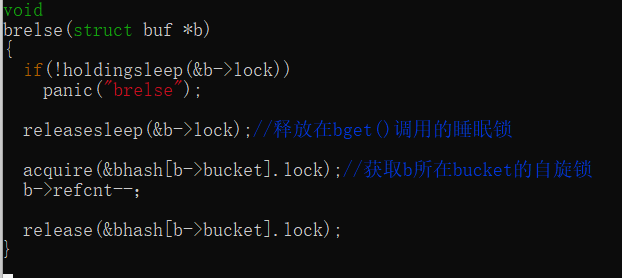
初始化hash表：



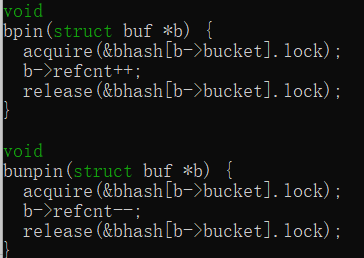
修改bget，首先根据blockno映射到相应的bucket，获取该bucket的自旋锁，然后遍历这个bucket里面的bufs，找到之后要更新时间戳，释放自旋锁调用睡眠锁。如果没有找到，就在所有refcnt为0的项里面搜索最近最少使用的即时间戳最小的，作为替换对象：



修改brelse：

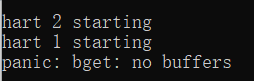


修改bpin和bunpin，将其中的bchache替换为bhash即可：

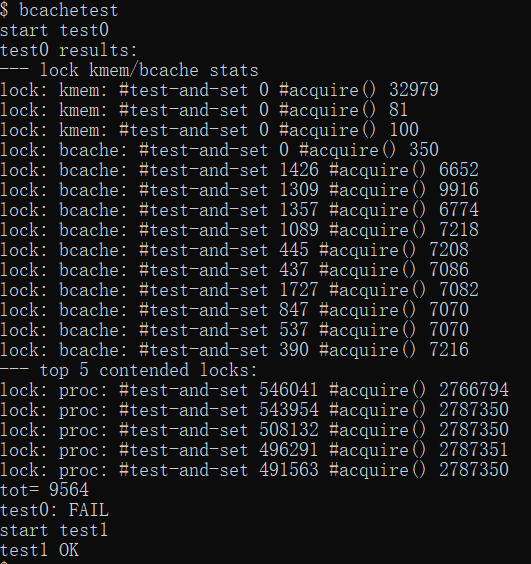


#### 实验中遇到的问题和解决办法

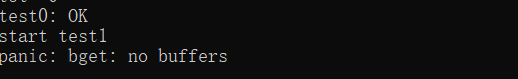
第一次运行结果：



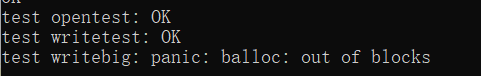
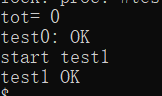
检查代码后发现第二个循环查找LRU时的循环结束条件写错了，应该是b < &bhash[key].bufs[0]+BUCKETSZ，导致bget失败。改正后，结果如下：



我再看了一遍题目，出现上面的结果说明我的acquire次数太多了，应该控制在500以内并且每次acquire尝试的次数最好接近于0，所以问题处在我NBUCKETS的数值即桶的数量的设置上，我原本想着要是个质数，要尽量大但又不要太大不然可能造成空间浪费，但是既然题目要求了，总缓存区数量为30（见param.h中NBUF定义）那我就将NBUCKETS设置为29，将BUCKETSZ设置为2吧，修改后，结果如下：

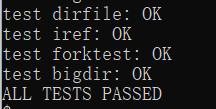


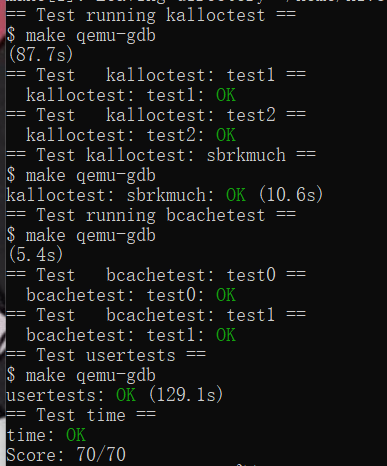
我看题目中提到”bcachetest's test1 uses more distinct blocks than there are buffers”，那我就将BUCKETSZ放大一点设置为4，结果终于通过了bcachetest（左下）但是usertests报错（右下）：



于是我怀疑我看错了param.h中的缓冲区大小，我再看了一遍里面的各定义然后通过上网查询知道了其中默认设置的 blocks 大小太少了，默认配置为 1000，需要将该值调大为 10000。

于是我将param.h中的FSSIZE值改为了10000。于是usertests也通过了：





#### 实验心得

这次实验题目看起来有点费力，因为都是没见过的函数和概念，于是我去看了一遍xv6相关部分的文档，看完后理解一些了，这其实跟之前一个实验中的思想一样，都是为hash表中的每个桶分配一把锁，通过减小锁的粒度减少acquire冲突次数提高并行度。通过这个实验我还对文件系统有了新的认识，见识了它的层级结构，知道了文件系统也会采用一个叫做buffer cache的结构，这其实和内存cache一样，都是靠高速缓存提高访问磁盘/内存的速度，

## Lab8：File system

### Large files (moderate)

#### 实验目的

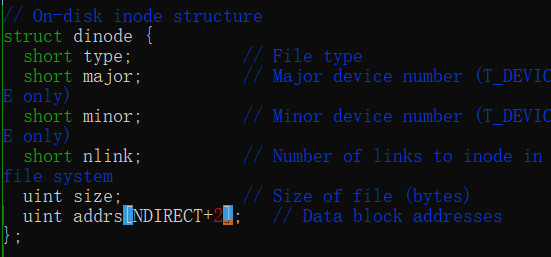
熟悉文件物理结构中索引结构多级索引的映射、索引规则。通过在已有的12个直接索引+1个一次间接索引的基础上改为11个直接索引+1个一次间接索引+1个二次间接索引来扩大文件系统大小。

#### 实验步骤

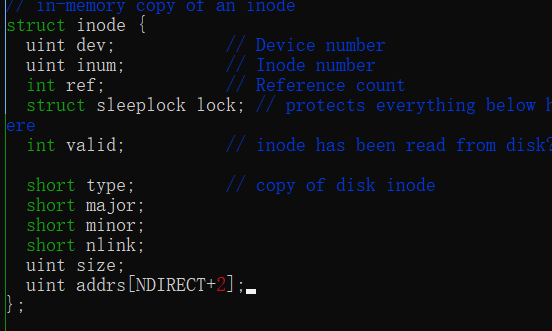
将kernel/fs.h中的NIDRECT修改为11：



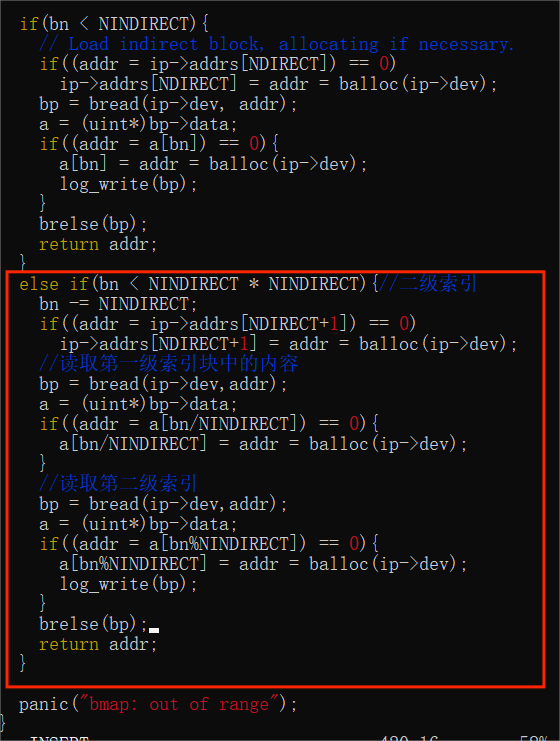
修改dinoed结构体中addrs[]大小为NDIRECT个直接索引，两个简介索引块（一个一级索引，一个二级索引）：



kernel/file.h中的inode中的addrs修改同上：

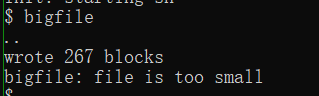


修改bmap中的内容，增加二级索引功能：

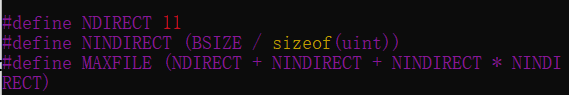


#### 实验中遇到的问题和解决办法

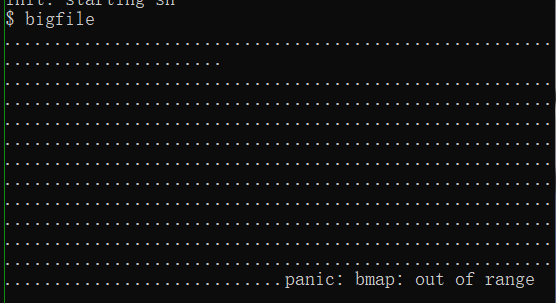
第一次运行结果如下：



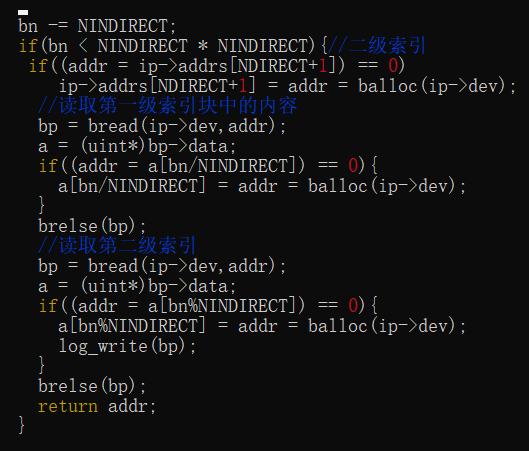
这个结果说明它似乎仍然按照之前的索引分配，于是我去检查fs.h中的相关定义，发现MAXFILE未修改，于是我将它修改如下



修改后运行结果如下：



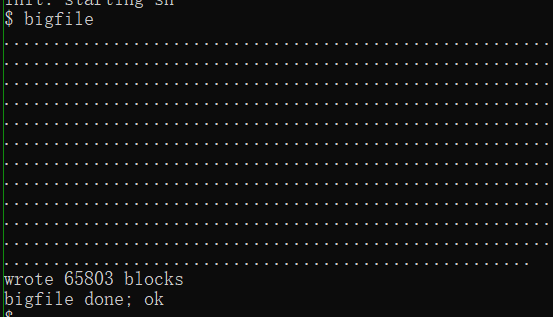
我去检查fs.c代码，发现二级索引进入判断条件有点奇怪，应该将bn先减去NINDIRECT之后在判断它是否小于NINDIRECT\*NINDIRECT，否则总共理想可用的11+NINDIRECT+NINDIRECT\*NINDIRECT个块中的最后NINDIRECT个大小将被忽略掉一直不能被使用，修改后代码如下：



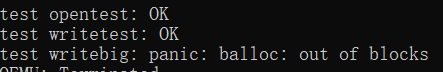
结果如下：



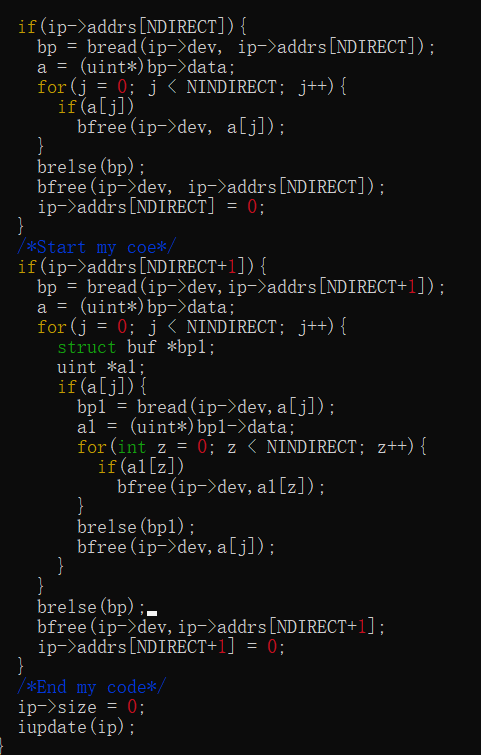
再次检查代码，发现获取第一级索引后没有log\_write(bp),加上后结果如下，终于成功了：



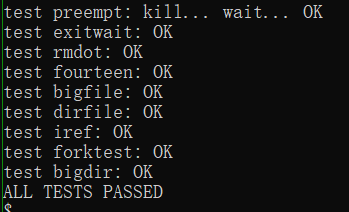
但是usertests失败了：



通过再次阅读题目中的提示，我发现我没有“Make sure itrunc frees all blocks of a file”，addr数组布局改变后应该在itrunc函数中也做相应的修改：



于是usertests也通过了：



#### 实验心得

这个实验不难，由于有过相关知识并且os大作业中做过文件索引结构，所以理解题意后实现起来并不难，在边查看边编写xv6中的代码的过程中比较发现它与我之前大作业的实现有哪些不同和哪些相同，其实逻辑都是相同的，只是我是直接在假想的物理空间开了个数组，读取索引直接读数组中的内容进行解析即可，而真实操作系统中是根据索引块地址到真实的磁盘去读索引块中的数据来获得下一级地址。

### Symbolic links (moderate)

#### 实验目的

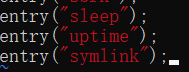
了解路径名查找工作原理，了解符号链接和硬链接的区别。

向xv6添加符号链接。符号链接（或软链接）是指按路径名链接的文件；当打开一个符号链接时，内核会跟随该链接指向引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。

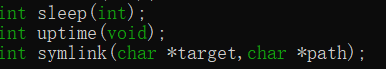
#### 实验步骤

在Makefile中添加“$U/\_symlinktest\”

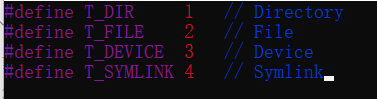
在user/usys.pl中增加symlink system call函数的入口：



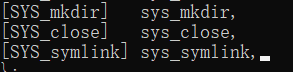
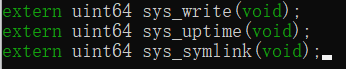
在user/user.hl中增加symlink system call函数的声明：



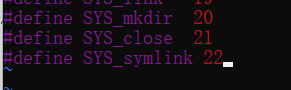
增加一个代表软连接的文件类型到kernel/stat.h中：



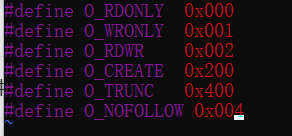
在syscall.c中添加一个叫symlink的指令：



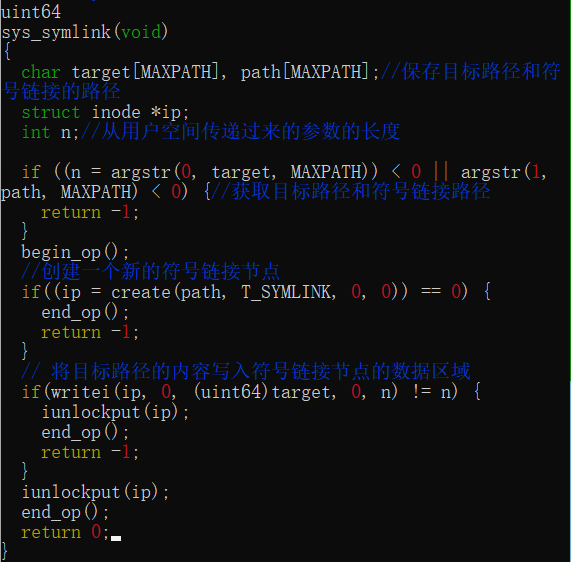
在syscall.h中添加一个系统调用号：



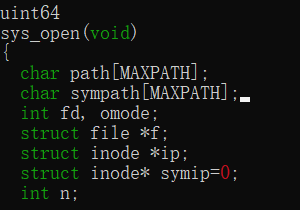
添加新的文件标志位 O\_NOFOLLOW 到 kernel/fcntl.h：

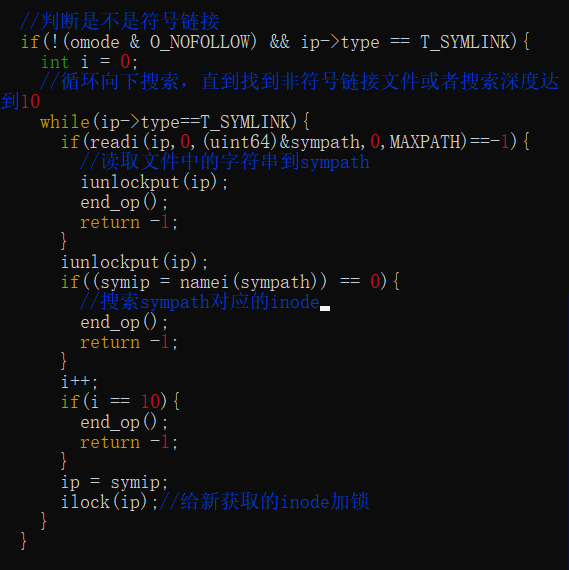


在sysfile.c中参考模仿sys\_link的程序去添加一个sys\_symlink指令。在link()中，把待链接的文件路径放在old里，把新的链接路径放在new里。link()使用namei查询old的inode，返回指针ip，增加ip的nlink数；使用nameiparent()查询new的父目录的inode，然后使用dirlink()把new加入它的父目录之中。相比link()，symlink()少了增加原来节点的nlink数，多了创建新的inode，写入target字符串的操作。因此再结合create()的操作来实现symlink()：



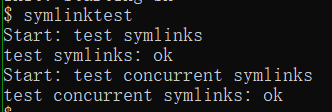
在sysfile.c中修改sys\_open，通过符号链接找到目标文件的inode：

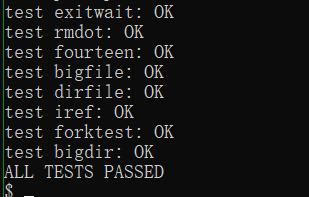


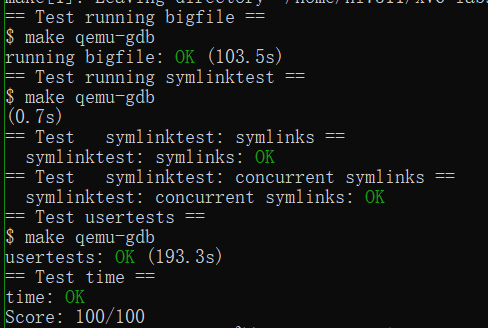


#### 实验中遇到的问题和解决办法

在经过一些语法报错修改后得到结果如下：







#### 实验心得

这个实验题目看起来有点费劲，因为没有接触过符号链接的概念，去看了文档，也没有提到这个概念，讲到了sys\_link这个系统调用但是没有详细讲，于是通过上网阅读sys\_link代码和查询资料完成了本次实验，只要理解了相关函数是干什么的怎么用，仿照着就不难完成，通过这个实验我认识到一些与文件路径处理相关的函数比如nami负责解析path并返回对应的节点。在open函数中看到对文件类型的判断时我想起了我第三次os大作业中打开并读取文件时的处理，先判断是普通文件还是目录文件再分别执行操作，而在对符号链接路径的循环逐层解析正是用了我当时获取路径文件名的处理时的方法。

## Lab9：mmap

### mmap(hard)

#### 实验目的

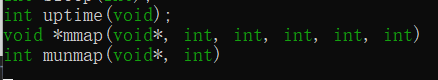
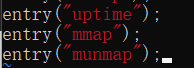
了解虚拟内存的概念和实现，熟悉进程的地址空间和虚拟地址空间到物理地址空间的映射。

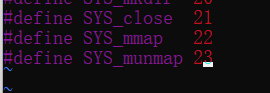
本实验的主要内容是把指定的文件fd映射到进程内存空间的某一个部分，映射建立之后，进程读写这块内存就像是在读写文件一样。

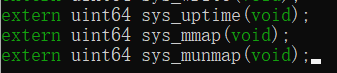
#### 实验步骤

①在 Makefile 中添加 $U/\_mmaptest

②添加有关 mmap 和 munmap 系统调用的定义声明. 包括 kernel/syscall.h, kernel/syscall.c, user/usys.pl 和 user/user.h：





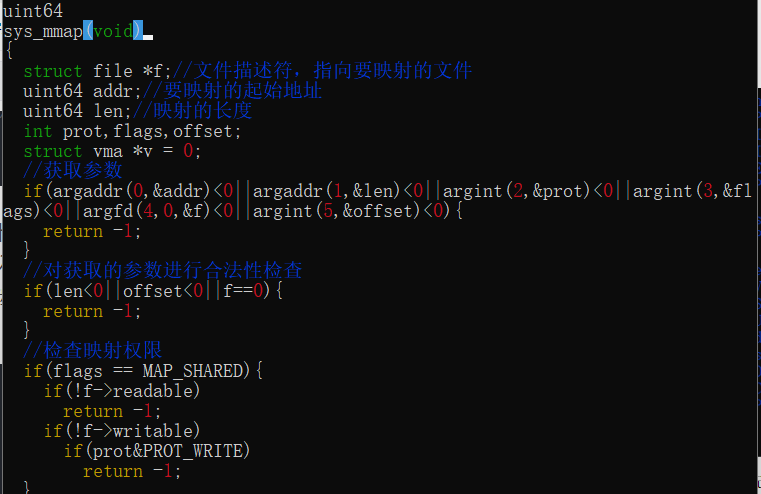


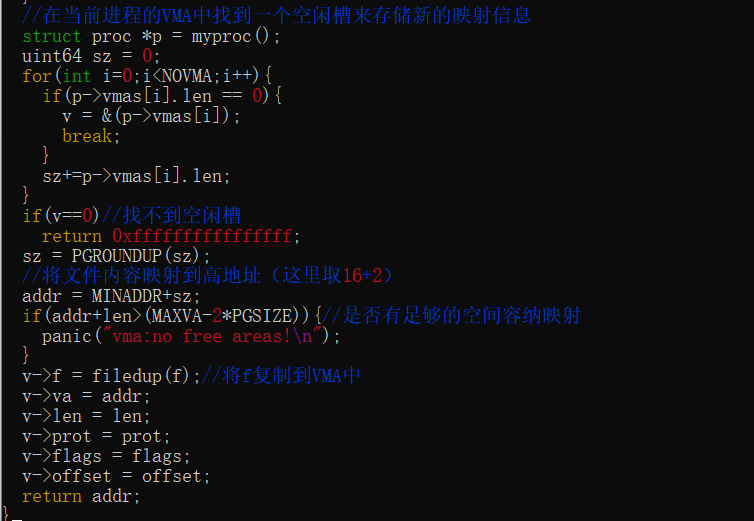


③定义VMA存储虚拟内存范围的起始地址、长度、权限、文件信息，其中flags=1表示MAP\_SHARED，为0表示MAP\_PRIVATE，并将VMA数组加入到每个进程中：



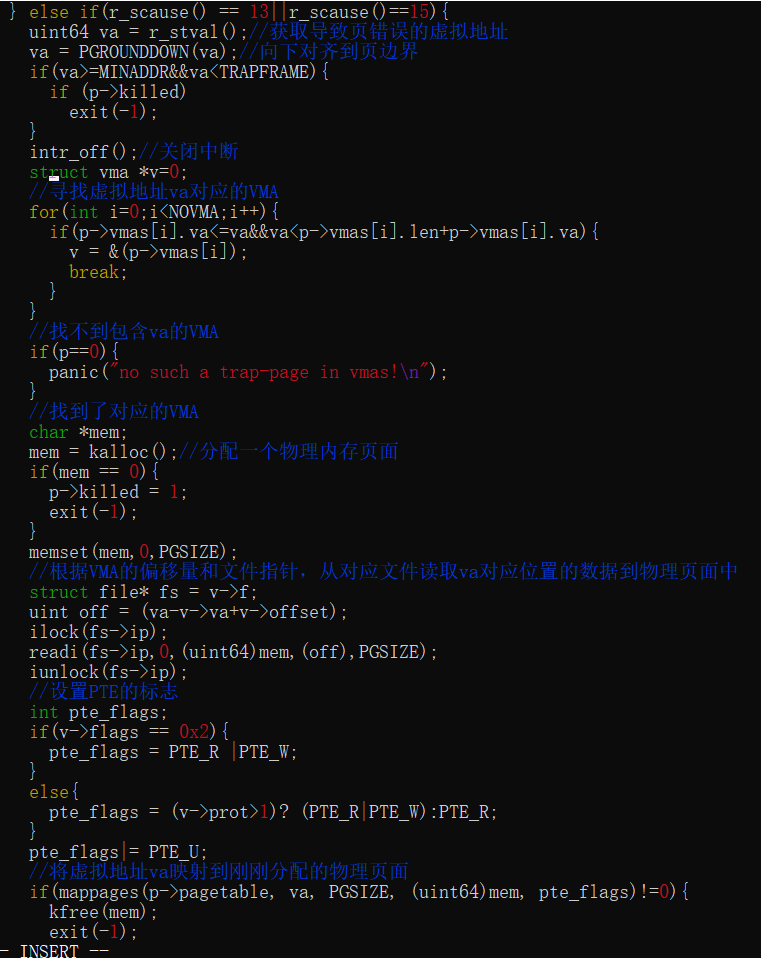
④在 kernel/sysfile.c 中实现mmap，首先将获取外部的输入，然后判断外部输入是否合理，接着在当前进程中找到一个空槽VMA，然后将外部输入放入该空槽中，最后返回该空槽存储文件指针的地址：





⑤这样之后，运行mmaptest，会触发页面故障，因为只是把文件地址返回，但并没有内容和读写权限，所以会触发trap，这里需要处理来自vma 的页面故障。

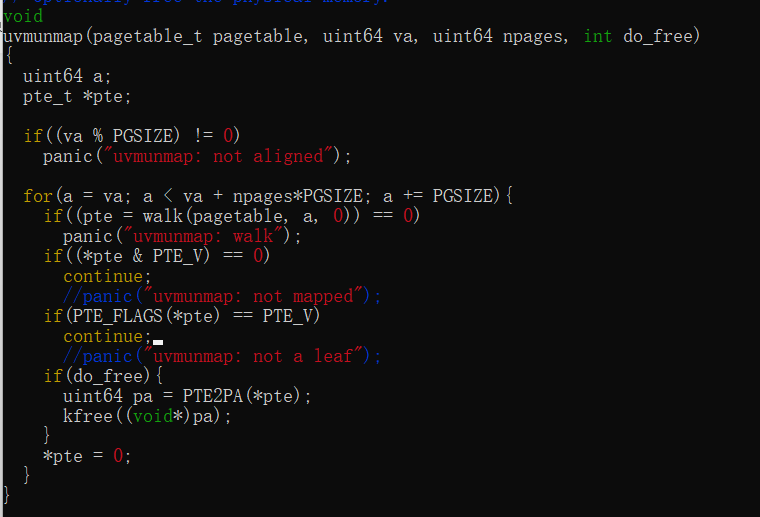
找到发生页面故障的vma，申请一页内存块将文件读入（这里需要注意文件读入时需要指定偏移量），然后将该内存块映射到该页面故障地址，添加适当的权限：



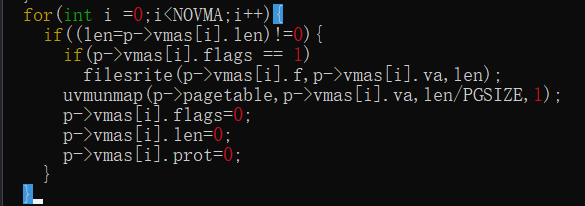
⑥实现munmap，方法与mmap类似：



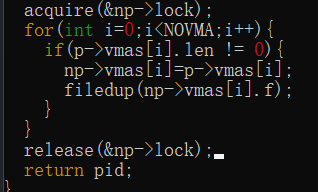
⑦修改uvmunmap，避免对无效页面进行取消映射，与cow中一样：



⑧修改kernel/proc.c中的exit使得进程结束后，将还在使用的vma进行释放，在exit中添加如下代码：



⑨修改 fork() 来保证子进程拥有对父进程 mmap() 的记录

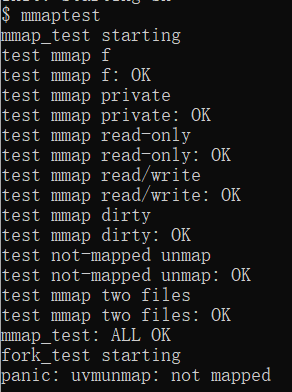


#### 实验中遇到的问题和解决办法

一开始出现了一堆编译报错，要么就是变量拼写错了，要么就是没有为新用到的函数和宏定义添加头文件，但是这个问题我始终找不到问题在哪，我觉得我的类型声明都没有问题，但是我想到之前每次出现这种无厘头的明明没有过改动的文件报错多半是改动过的文件里格式不对，于是我猛然发现我声明两个函数时没有加分号，于是问题就解决了。



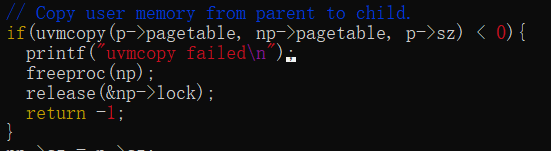
编译通过后运行出现以下错误：



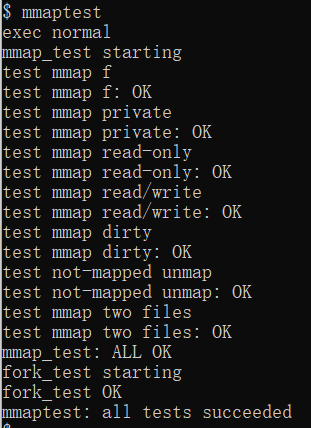
通过panic报错信息我发现是因为我没有相应地修改uvmunmap，于是增加了步骤中的第⑦步。

修改好后报错结果如下：



根据报错信息和发生报错的测试为fork，我循着fork的函数调用找到了问题应该出在fork→uvmcopy失败时的freeproc→proc\_freepagetable→uvmfree→freewalk。（通过分析我发现我fork添加代码段的位置不当，我将它放到了 np->state = RUNNABLE;前面），但是我在fork中诞生这个错误的地方加上输出测试时，输出结果并没有“uvmcopy failed”。

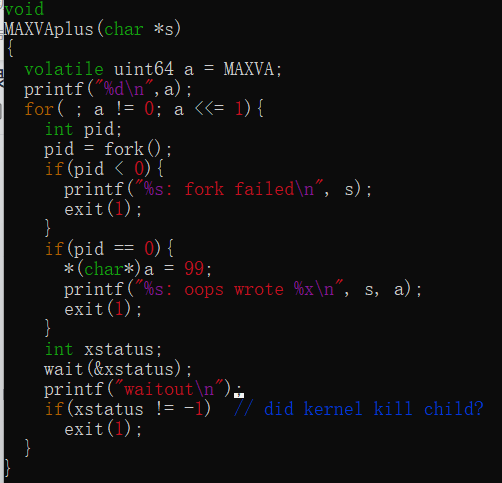
所以我又在其它调用了uvmfree的地方加上了输出测试。后来发现是进入了wait函数调用了freeproc然后产生的错误。这好像不是重点，但是这个至少说明不是由于fork uvmcopy失败导致的，而是页表项的处理不当，一个页表项指向了一个叶节点导致的。但是我怎么也找不出与map有关的错误，然后我突然想起题目中让我们在uvmunmap中忽略遇到叶子页面的情况，而在freewalk中正是遇到非有效页而panic，于是我仿照uvmunmap中的处理，让freewalk中也忽略这一情况，于是测试遍通过了。



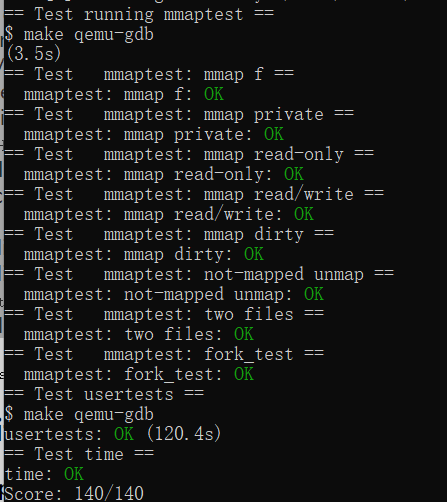
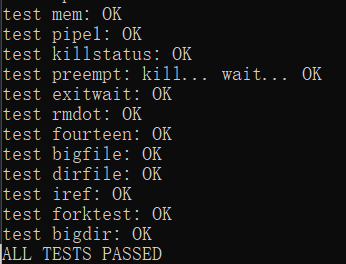
但是usertests卡在了第一个测试



通过调试我发现是下图中的wait陷入了无限循环，



经过查看wait函数可知在循环中没有找到处于 ZOMBIE 状态的子进程，并且当前进程没有其他子进程或当前进程本身已被标记为 killed，那么该函数将陷入死循环并永远不会返回。所以问题出在usertrap，检查后我将page fault的中r\_scause()==15删去，这一类的错误不属于vma页fault失误范畴，修改后jiu通过了。



#### 实验心得

这个实验也挺难的，耗时也挺久的，花费了整整一天多，一开始看这个题目有点费劲，但是借助一些博主的讲解后我大概理解这个题目是要模拟实现出一个虚拟内存VMA，通过将一些磁盘文件映射到物理地址空间，而进程中通过保存vma页表项信息，找到其虚拟地址映射到的物理地址从而读写该物理地址映射的文件，这样进程读写这块内存就像是在读写文件一样，从而实现了虚拟内存的功能。这次实验要写的代码很多，要自己增加两个函数，还要相应地修改fork、exit、uvmunmap甚至后来通过排错还发现不得不修改freewalk。虽然代码量比较大，但是思路上有cow实验可循，很多地方和那相似，或许也正是因为之前辛辛苦苦调了cow实验很久，这次才少走了很多弯路吧。到这里所有实验终于结束了，真开心啊，还挺不容易的，从7.10到7.30，从搭建环境到摸索出这个实验大概是个什么流程到熟悉urbuntu怎么用，再到学习xv6文档和mit视频，再到被迫学会gdb qemu调试代码，最后到现在的成功完成所有实验，尤其是放假回家后基本一直闭关整日写实验，虽然有点痛苦，但学到了不少课堂之外的东西、高阁以外的东西，到此本科os相关的知识的学习应该就到此止步了吧，很有收获，也希望这些东西不要忘掉，我觉得之后应该会经常用到，所以把它们（课堂上的和这次实验学到的知识）都记在笔记本上了，但是我知道我目前掌握的还不够全面，所以我还要持续学习，不断补充！