操作系统课程设计项目说明文档

MIT 操作系统实验

|  |  |
| --- | --- |
| 作者姓名： | 陈瑾旭 |
| 学 号： | 2152207 |
| 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

[操作系统课程设计项目说明文档 1](#_Toc21173)

[实验一 Xv6 and Unix utilities 5](#_Toc6188)

[1.1 实验目的 5](#_Toc11751)

[1.2 实验内容 5](#_Toc900)

[1.3 实验步骤 5](#_Toc26777)

[1.3.1 运行 xv6 5](#_Toc28203)

[1.3.2 实现 sleep 7](#_Toc7363)

[1.3.3 实现 pingpong 9](#_Toc23662)

[1.3.4 实 现 prime 10](#_Toc7071)

[1.3.5 实 现 find 12](#_Toc1402)

[1.3.6 实 现 xargs 14](#_Toc25235)

[1.3.7综合测试 15](#_Toc29055)

[1.4 实验小结 17](#_Toc12162)

[实验二 System calls 18](#_Toc23767)

[2.1 实验目的 18](#_Toc26544)

[2.2 实验内容 18](#_Toc29646)

[2.3 实验步骤 18](#_Toc5872)

[2.3.1 实现系统调用跟踪 18](#_Toc9671)

[2.3.2 实 现 sysinfo 25](#_Toc13971)

[2.3.3 综合测试 28](#_Toc13664)

[2.4 实验小结 28](#_Toc350)

[实验三 Page Tables 29](#_Toc15302)

[3.1 实验目的 29](#_Toc14819)

[3.2 实验内容 29](#_Toc19096)

[3.3 实验步骤 29](#_Toc11589)

[3.3.1 加速系统调用 29](#_Toc27523)

[3.3.2 打印页表结构 30](#_Toc15226)

[3.3.3 检测哪些页表被访问过 31](#_Toc28931)

[3.3.4 综合测试 33](#_Toc27618)

[3.4 实验小结 33](#_Toc31162)

[实验四 traps 34](#_Toc18347)

[4.1 实验目的 34](#_Toc27100)

[4.2 实验内容 34](#_Toc18417)

[4.3 实验步骤 34](#_Toc26024)

[4.3.1 RISC-V 程序集 34](#_Toc4818)

[4.3.2 实现 Backtrace 36](#_Toc5357)

[4.3.3 Alarm-test0()：invoke handler 37](#_Toc6191)

[4.3.4 Alarm-test1/test2(): resume interrupted code 38](#_Toc21008)

[4.3.5综合测试 40](#_Toc16369)

[4.4 实验小结 41](#_Toc2127)

[实验五 Copy on write 42](#_Toc13665)

[5.1实验目的 42](#_Toc10007)

[5.2实验内容 42](#_Toc1251)

[5.3实验步骤 42](#_Toc15377)

[5.4实验小结 47](#_Toc423)

[实验六 Multithreading 48](#_Toc21211)

[6.1实验目的 48](#_Toc24889)

[6.2实验内容 48](#_Toc20896)

[6.3实验步骤 48](#_Toc29496)

[6.3.1实现一个用户级进程的创建和切换 48](#_Toc15105)

[6. 3 .2使 用 UNIX pthread 线程库实现一个线程安全的哈希表 50](#_Toc27111)

[6. 3 . 3 实 现 barrier 函数 51](#_Toc19746)

[6.3.4测试成绩 53](#_Toc2971)

[6.4实验小结 55](#_Toc12263)

[实验七 net 56](#_Toc6989)

[7.1实验目的 5](#_Toc16014)6

[7.2实验内容 59](#_Toc16014)

[7.3实验步骤 59](#_Toc16014)

[7.4实验小结 59](#_Toc16014)

[实验八 Locks 59](#_Toc26199)

[8.1 实验目的 59](#_Toc2143)

[8.2 实验内容 59](#_Toc14037)

[8.3 实验步骤 59](#_Toc21115)

[8.3.1 Memory allocator (moderate) 59](#_Toc15211)

[8.3.2Buffer cache (hard) 61](#_Toc24709)

[8.3.3测试成绩 65](#_Toc22100)

[8.4 实验小结 66](#_Toc5698)

[实验九 File System 67](#_Toc10777)

[9.1 实验目的 67](#_Toc14242)

[9.2 实验内容 67](#_Toc26136)

[9.3 实验步骤 67](#_Toc6938)

[9.3.1 修改文件系统使其支持更大的文件存储 67](#_Toc29312)

[9.3.2 增加一个通过符号链接的系统调用 72](#_Toc32637)

[9.3.3 测试成绩 73](#_Toc31657)

[9.4 实验小结 75](#_Toc18351)

[实验十 Mmap 77](#_Toc25644)

[10.1 实验目的 77](#_Toc1655)

[10.2 实验内容 77](#_Toc3075)

[10.3 实验步骤 77](#_Toc31497)

[10.3.1 理解虚拟地址与 mmap 77](#_Toc8230)

[10.3.2 实现 mmap 系统调用 78](#_Toc18096)

[10.3.3 运行测试 80](#_Toc3108)

[10.4 实验小结 82](#_Toc24702)

# 实验一 Xv6 and Unix utilities

## 实验目的

1. 掌握了 xv6 操作系统的内核和用户部分，了解了操作系统的基本结构和功能，以及如何编写用户程序，并能够进行操作系统的简单配置和调试。

2. 理解了系统调用的概念和作用，掌握了如何使用系统调用来进行操作系统和用户程序之间的交互，能够使用系统调用来进行文件操作、进程管理、网络通信等操作。

3. 熟悉了父子进程的概念和原理，掌握了使用 fork() 函数来创建新进程的方法，并能够通过进程间通信来实现进程间的协作和数据共享。

4. 掌握了素数筛算法的原理和实现方法，了解了如何使用管道来实现进程间通信，并能够编写基于管道的素数筛程序，实现对一定范围内的素数的查找。

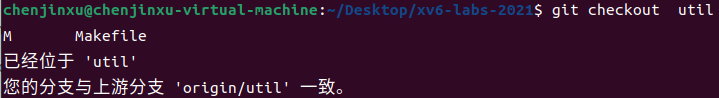
5. 完成了 find 和 xargs 程序的编写，这两个程序都是常见的用于文本搜索和命令行参数处理的工具。对于 find 程序，能够实现在指定目录下查找符合条件的文件，并输出文件路径；对于 xargs 程序，能够实现将标准输入中的文本作为命令行参数传递给指定命令，并执行该命令。

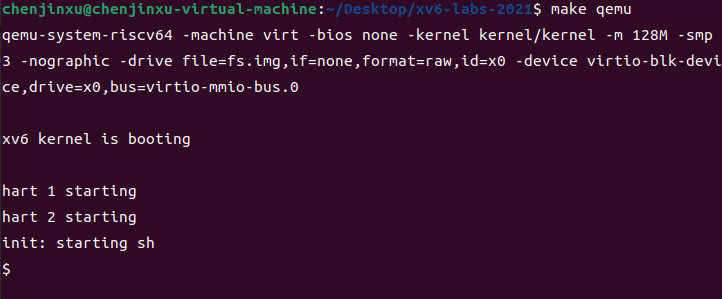
## 实验内容

1. 运行 xv6。
2. 为 xv6 实现 UNIX 程序 sleep：sleep 应该暂停一段用户所指定的时间。
3. 编写一个程序，使用 UNIX 系统调用在两个进程之间通过一对管道“乒乓” 一个字节，每个管道一个。
4. 使用管道编写并发版本的主筛。
5. 编写一个简单版本的 UNIX 查找程序：查找具有特定名称的目录树中的所有文件。
6. 编写一个简单版本的 UNIX xargs 程序：从标准输入中读取行并为每一行运行一个命令，将该行作为命令的参数提供。

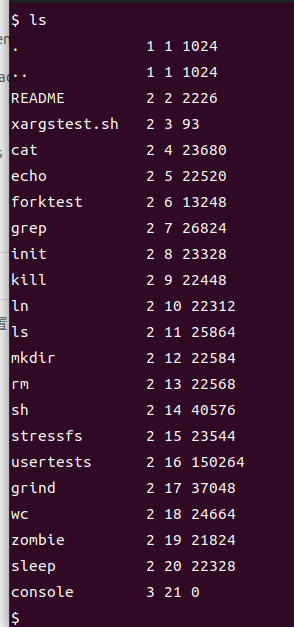
## 实验步骤

### 运行 xv6

1. 切换到 util 分支：
2. 利用 make qemu 指令运行 xv6

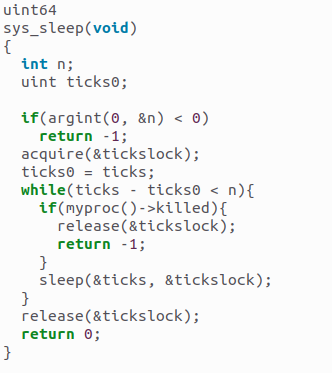


1. 使用 ls 命令查看文件：



### 实现 sleep

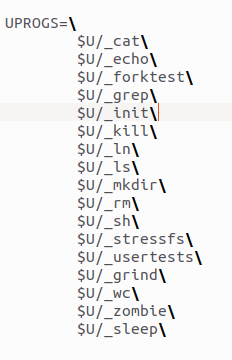
1. 查看 sleep 系统调用函数：



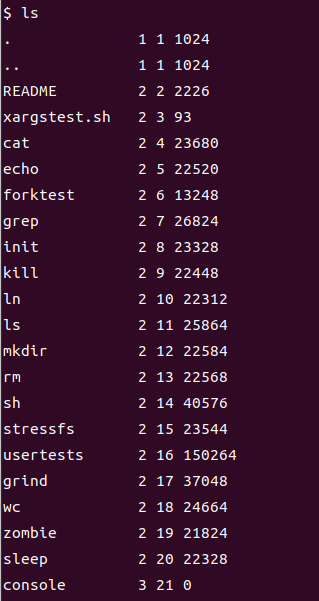
2.写一个简单的 C 程序（sleep.c），它接受一个命令行参数，表示要休眠的时间（以秒为单位），然后使程序休眠这段时间。程序首先检查是否有足够的命令行参数。如果没有，它将打印一条用法消息，并退出程序。然后，程序将第一个命令行参数（也就是指定的休眠时间）转换为整数，使用 sleep 函数来使程序休眠指定的秒数。最后，程序调用 exit 函数来正常退出程序。



3.在makefile里声明这个方法（加在了最后）



1. 重新make qemu，运行ls发现有sleep指令

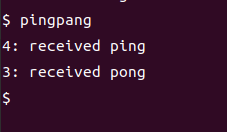


1. 运行sleep 10（停顿了一下）

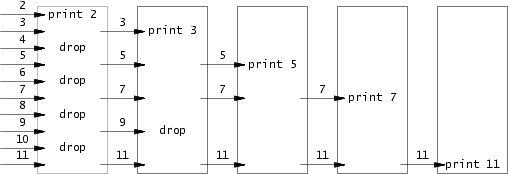


### 实现 pingpong

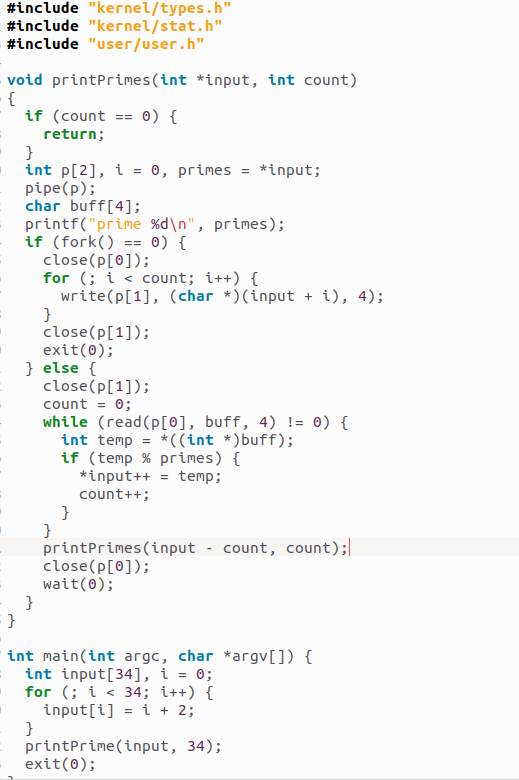
1. 首先实现代码，我们的程序需要实现两个进程之间的通信，通过管道来传输数据。程序首先创建了两个管道 p1 和 p2，每个管道都有两个文件描述符，一个用于读取数据，一个用于写入数据。然后，程序调用 fork() 函数来创建一个子进程。在子进程中，它关闭了管道 p1 的写端和管道 p2 的读端，然后从管道 p1 中读取数据到缓冲区 buffer 中，如果读取的数据长度不等于 len，则输出错误信息并退出程序。接着，程序输出接收到的数据信息，并将数据写入到管道 p2 的写端中，同样如果写入的数据长度不等于 len，则输出错误信息并退出程序。最后，子进程使用 exit(0) 函数正常退出。在父进程中，程序关闭了管道 p1 的读端和管道 p2 的写端，然后将数据写入到管道 p1 的写端中，如果写入的数据长度不等于 len，则输出错误信息并退出程序。接着，程序从管道 p2 的读端中读取数据到缓冲区 buffer 中，如果读取的数据长度不等于 len，则输出错误信息并退出程序。最后，程序输出接收到的数据信息，并使用 exit(0) 函数正常退出。
2. 将该方法加入Makefile（这里不放截图了，因为与上面展示的相似）
3. 运行pingpang



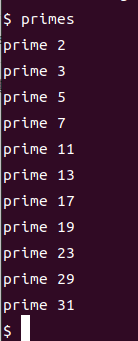
### 实现 prime

1. 首先根据官网给出的网址能了解关于素数筛的一些知识方便我们之后进行操作。

和上图阐述的原理一样，所以我们需要借助父子进程的并发性，同时利用上一小问中所学习到的管道通信机制来传输消息即可。

2.为了实现该功能，我实现了一个递归版的素数筛，使用了管道和 fork() 函数实现进程间通信。主函数中初始化了一个数组 input[]，其中包含了从 2 到 35 的所有自然数。然后，主函数调用 printPrimes 函数，并将 input 数组和其长度 34 作为参数传递给它。而printPrimes 函数首先从 input 数组中取出第一个元素作为当前素数 prime，并打印出它。然后，它创建一个管道 p，并在子进程中将 input 数组中不是 prime 的倍数的元素写入到管道p 中。子进程执行完毕后，父进程从管道 p 中读取数据，并将不是 prime 的倍数的元素从 input 数组中删除。最后，父进程递归调用 printPrime 函数，将更新后的 input 数组和元素个数作为参数传递给它。这样就完成了素数筛的功能。

1. 在 MakeFile 中添加 primes 之后，在 xv6 中运行 primes 命令，结果如下：

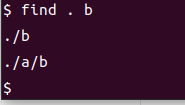


### 实现 find

1. 参考 ls 方法，我发现 find 实现方法的整体思路应当为深度优先搜索，因此在这种思路下我写了一个名为find的可以在指定目录下查找指定文件名的文件，并输出文件的路径的文件。程序的主函数“main”首先检查命令行输入参数的个数是否正确，如果不正确就输出错误信息并退出。我调用“find”函数来查找指定文件名的文件，传入的参数为指定的目录和要查找的文件。“find”函数首先打开指定目录，如果打开失败则输出错误信息并返回。然后获取目录的状态（包括类型、大小等信息），如果获取失败则输出错误信息并关闭文件描述符并返回。如果目录不是一个目录类型，则输出使用说明并返回。如果指定的路径名称长度太长，则输出错误信息并返回。然后程序使用“strcpy”函数将指定的路径名称复制到缓冲区“buf”中，并在路径名称后面添加一个“/”字符，以便在路径后面添加文件名。程序使用“while”循环来读取目录中的每个目录项，并将目录项的文件名添加到路径名称后面。如果目录项的“inum”字段为零，则说明该目录项无效，跳过并继续读取下一个目录项。程序调用“stat”函数来获取添加了文件名后的路径的文件状态信息。如果获取失败，则输出错误信息并继续读取下一个目录项。如果该目录项是一个目录类型，并且不是“.”和“..”目录，则递归调用“find”函数来查找该目录下的文件。如果该目录项的文件名与指定的文件名相同，则输出文件路径。最后，程序关闭文件描述符并返回。

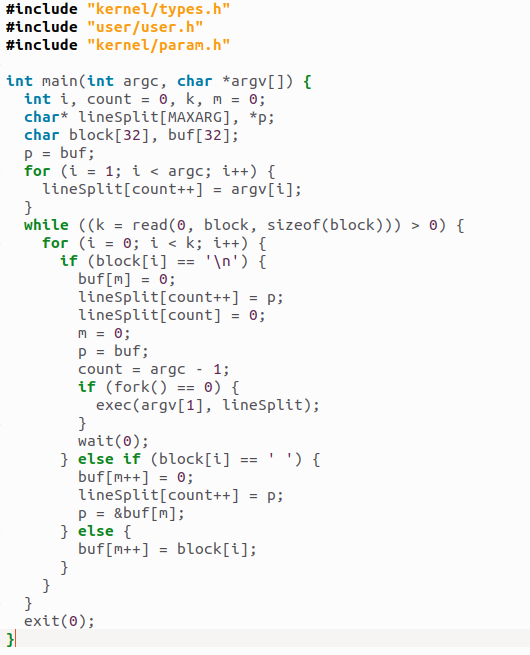


1. 完成上述代码后，在 MakeFile 文件中添加 find 方法。之后，便可以在 xv6 中使用 find 方法：



### 1.3.6 实现 xargs

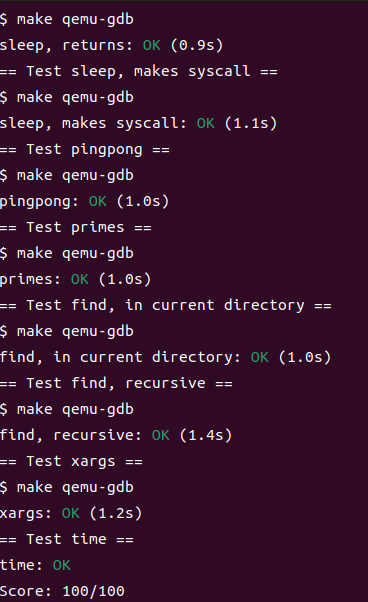
1. 通过阅读本题题目的要求，可以发现需要解决的主要问题是对字符串进行处理。其中，| 之前的结果会在缓冲流中。因此程序的主要流程如下：首先，程序读取命令行参数，并将除了程序名以外的所有参数存储到“lineSplit”数组中。然后，程序进入一个循环，不断从标准输入中读取数据块。对于每个读取的数据块，程序会遍历其中的每个字符，将字符存储到缓冲区“buf”中，直到遇到换行符“\n”为止。当遇到换行符时，程序将缓冲区中的数据作为参数传递给指定的命令，并在子进程中执行该命令，然后等待子进程结束。在读取数据块的过程中，如果遇到空格符“ ”，则说明当前字符之前的数据已经组成了一个参数，将该参数存储到“lineSplit”数组中，并将缓冲区指针“p”指向下一个参数的起始位置。如果遇到其他字符，则将该字符添加到缓冲区中，直到遇到空格符或换行符。重复执行以上步骤，直到读取完整行文本。程序退出。



2.将其添加到 MakeFile 文件中后，即可实现 xargs 功能：9a35b379921fb28d53a16b67d55c3a9

### 1.3.7综合测试

通过 make grade 进行本实验综合测试：



至此，本实验全部完成。

## 实验小结

通过本实验，我深入了解了 xv6 操作系统内核以及 qemu 模拟器的使用方法。在实验过程中，我遇到了一些困难，例如不了解管道的创建和关闭方法，对素数筛算法也不够熟悉等。但是通过和同学交流和网络资源的帮助，我成功地解决了这些问题。此外，我还初步了解了系统调用接口，这是操作系统和用户之间的一个重要接口，也认识到了操作系统内核和用户程序之间的区别。通过这个实验，我不仅掌握了一些实用的编程技巧和调试方法，也提高了自己的问题解决能力和与他人交流的能力。

# 实验二 System calls

## 实验目的

1. 进一步了解系统调用。
2. 掌握添加系统调用的方法。
3. 理解系统调用的工作原理和 xv6 内核的工作过程。

## 实验内容

1. 为 xv6 添加系统调用跟踪功能。
2. 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

### 实现系统调用跟踪

1. 首先我学习了实现系统调用跟踪的相关知识，首先归纳了以下几点需要实现的功能：“

1.1.在系统调用处理函数中添加跟踪代码。在每次系统调用发生时，记录当前进程的进程ID、系统调用编号、系统调用参数和返回值等信息。可以使用printf函数将这些信息输出到控制台或保存到文件中。

1.2.在进程创建函数中将跟踪功能继承给子进程。当一个进程调用fork函数创建一个子进程时，子进程会继承父进程的地址空间和所有属性，包括跟踪功能。这样可以确保子进程也能够被跟踪。

1.3.在跟踪函数中添加控制代码。跟踪功能应该可以被启用和关闭，以便在需要时打开或关闭跟踪功能。可以使用一个全局变量或系统调用来控制跟踪功能的开关。”

1. 接下来就是具体实现，首先在user/user.h中添加系统调用函数的定义。

6548a300c977744bfe65f06e9656a29

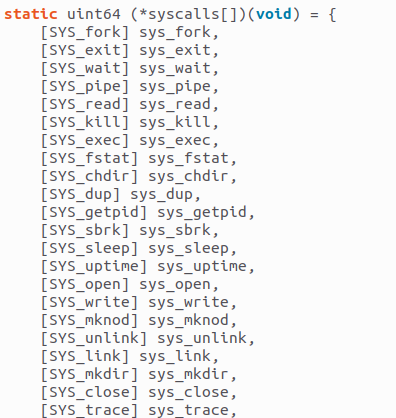
1. 在user/user.pl中添加入口

28b9c63ec70f0982ec50474f61fa8ca

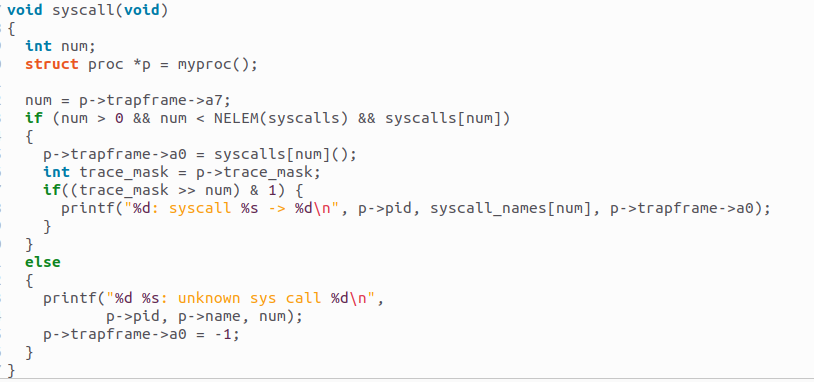
1. 在kernel/syscall.h中定义系统调用号

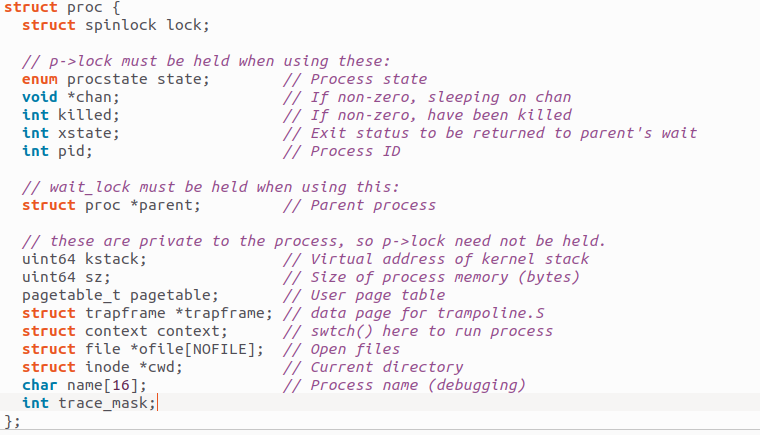
d0796ab4ac621132660a3eeebc79f84

1. 在kernel/syscall.c中的syscall函数指针数组中添加对应的函数和数组

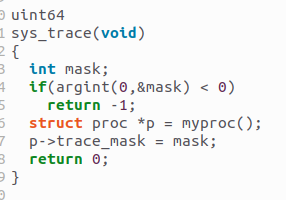
d1b1fd5dd700bc601370df60bcbf0bb

6.在syscall函数中，可读取trapframe->a7获取系统调用号，之后根据该系统调用号查找syscalls数组中的对应的处理函数并调用。当系统调用号和trace\_mask匹配就输出题目规定格式的信息。

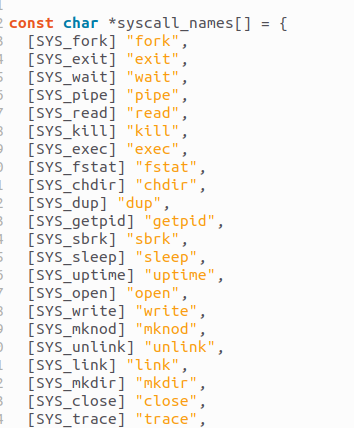
7.在proc结构体中添加一个trace\_mask字段用来标识，之后再创建子进程的fork函数中复制该字段到新进程。

b5ced937405d67cef90ca2bc4041b6e

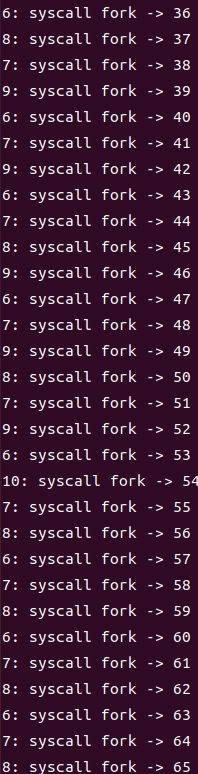
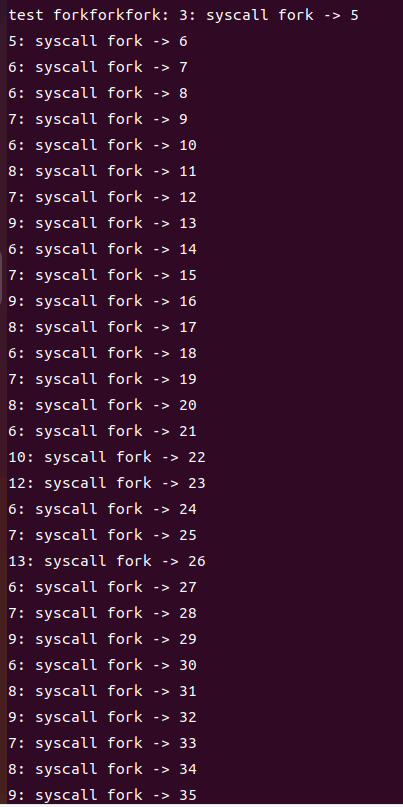
1. 系统调用sys\_trace的实现。在sysproc.c中添加函数uint64 sys\_trace（void）函数，该函数通过argint函数读取相应的参数并且赋值给trace\_mask变量。

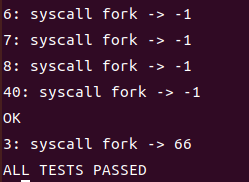


1. 增加一个新的数组，因为之前的syscall函数有用到



9.在 MakeFile 文件中添加 trace 后，即可成功运行：





### 实现 sysinfo

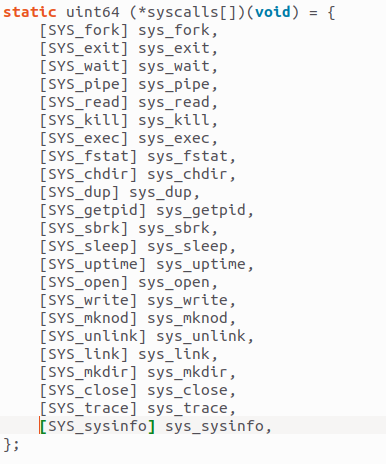
1. 首先看任务，他需要我们增加一个sysinfo系统调用，它收集有关运行系统的信息。
2. 然后根据上一个实验的经验，我们首先添加系统调用sysinfo与之前的trace系统调用相同，即在user/user.h中声明sysinfo（）的原型，同时预先声明struct sysinfo的存在。然后在user/usys.pl中添加入口。

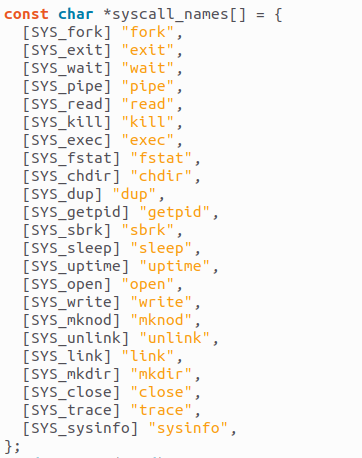
599441553f3e06b04078a0146f8c14e

07524c36154c961f647c207858bdeb5

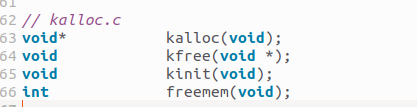


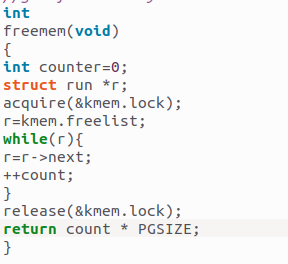
1. 然后是修改内核代码的部分，我们在kernel/syscall.h中定义系统调用编号，然后在kernel/syscall.c的syscal函数指针数组中添加对应的函数和函数名。



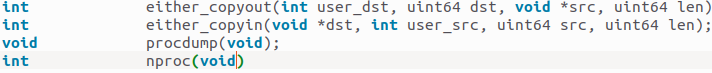
**f94fd3384ab53a667b1a590ed41aa33**

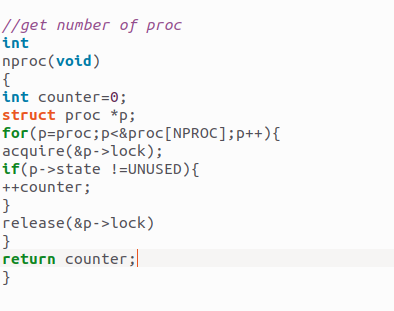
1. 接下来就是实现freemem函数的实现，通过阅读kalloc和kfree函数可知，我们只需要统计kmem.freelist链表的长度再乘以PGSIZE就可以得到空闲内存。于是我们现在kernel/defs.h中声明freemem函数。然后在kernel/kalloc.c中添加freemem函数。而函数的实现这段代码实现首先通过定义了一个计数器变量`counter`，然后获取内核内存管理锁`kmem.lock`以保证多线程访问的安全。接着，函数将指向空闲内存页链表的指针`kmem.freelist`赋值给结构体指针`r`。在while循环中，函数遍历链表，每次将指针`r`指向下一个节点，同时将计数器`counter`加1。最后，函数释放内存管理锁，计算并返回空闲内存的总大小，即空闲页数乘以每页的大小（`PGSIZE`）。





1. 接下来阅读procdump和相关代码知道xv6的进程结构体保存在proc[NPROC]数组中，而且字段还有五种状态。因此我们只需要遍历这个数组然后统计状态即可。因此我们首先声明nproc函数。然后在kernel/proc.c中添加nproc函数。这个函数的作用是获取系统中进程数量。函数首先定义了一个计数器变量counter，然后通过遍历进程表proc，对每个进程执行如下操作：获取该进程的锁p->lock以保证对该进程状态的读取不会被其他线程打断。检查该进程的状态p->state是否为UNUSED，如果不是，则代表该进程正在被使用，需要统计到进程数量中。

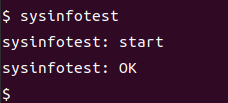




6.然后添加sys\_sysinfo（void）的系统调用代码

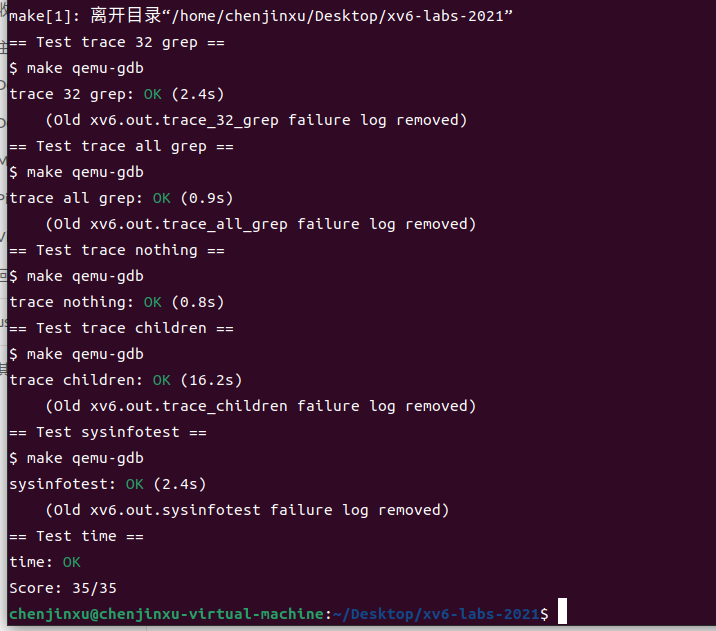


1. 在makefile文件中完成声明后，即完成了全部要求。测试如下



### 综合测试

我通过 make grade 进行本实验综合测试：



## 实验小结

在本次实验中，我通过深入学习系统调用的相关知识，掌握了系统调用的基本原理和使用方法。在实践中，我编写了一些简单的系统调用，并针对课程提供的测试程序进行了测试，从中了解到系统调用的具体使用场景和注意事项。

在第一个跟踪实验中，我需要仔细研究内核源代码，包括 syscall()等函数，以了解系统调用的执行过程和相关的内核数据结构。这个过程虽然比较繁琐，但是让我更深刻地理解了系统调用的实现原理和机制。

在第二部分实验中，我需要编写一个名为 sysinfo 的系统调用，用于获取系统信息。由于对进程的定义仅仅停留在课本上，所以一开始难以下手。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的方式。通过仿照已经存在的内核函数，我终于完成了该部分实验，从中深刻理解了系统调用的编写和使用方法。

# 总的来说，通过本次实验，我不仅更深入地了解了系统调用的相关知识，还通过实践掌握了系统调用的具体编写和使用方法。这也为我今后学习操作系统和计算机系统相关知识打下了比较扎实的基础，让我更加熟练地掌握了内核和用户空间的区别和联系。

# 实验三 Page Tables

## 实验目的

1. 了解 RISC-V 中的页表机制。
2. 深入理解逻辑地址到物理地址的映射关系，掌握如何实现一个页表映射。
3. 加深对操作系统内存管理的理解。

## 实验内容

1.加速系统调用

2.打印页表

3.检测哪些页面已经被访问

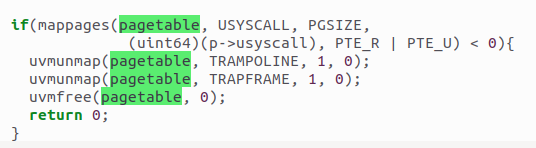
## 实验步骤

### 加速系统调用

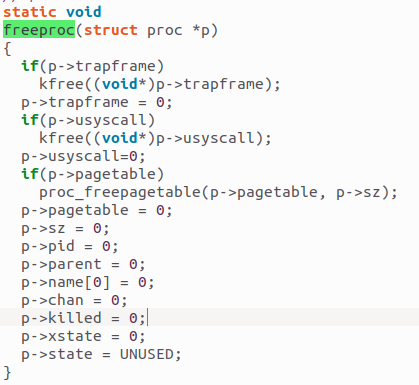
1. 首先看题目，大致上，这个加速系统调用的含义是说，要在用户进程的地址空间新映射一个页，这个页是用户和内核的共享只读数据区，而其用户空间地址是已经定义好的USYSCALL。页内存放的struct ussyscall结构体含有成员pid，我们可以在分配进程时将其pid存储进去，然后使用ugetpid(void)函数直接从USYSCALL处获取pid，而不是使用原来的getpid()系统调用，从而避免kernel crossings，达到加速效果。
2. 首先在kernel/proc.h中的struct proc增加一个一个域，存储共享内存块的物理地址

522412e316b7536461fd017fc1924af

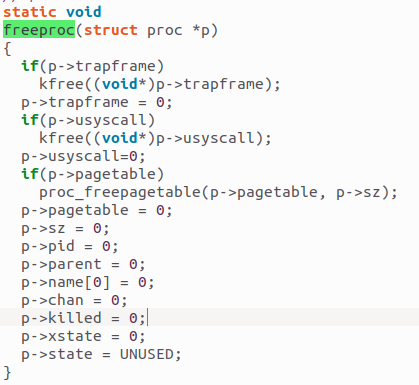
1. 在pagetable中映射一个用户态系统调用处理函数的页面，以便用户程序可以调用它。如果映射失败，函数将释放在之前分配的内存，并返回0。



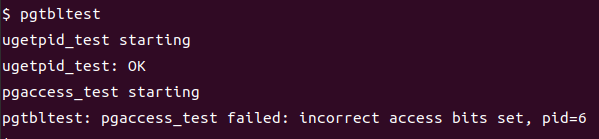
1. 在kernel/proc.c的freeproc函数中增加释放共享内存块



1. 在kernel/proc.c的proc\_freepagetable函数中增加一行释放页表中共享内存页项

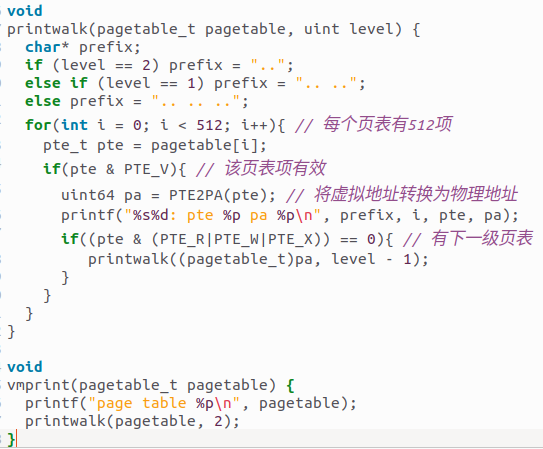


1. 运行得到结果



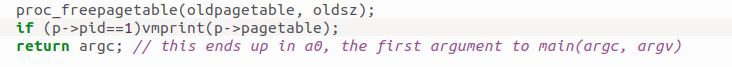


### 打印页表结构

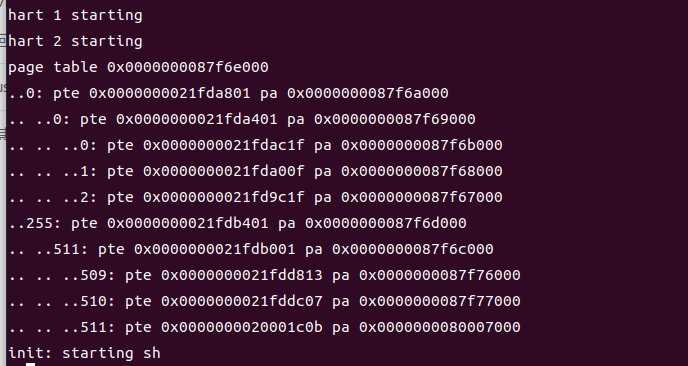
1. 本实验实现一个vmprint()函数来遍历页表并打印，在xv6中有个freewalk()函数，实现递归释放页表页，我模仿该函数来写vmprint()函数。
2. 在kernel/defs.h中定义vmprint的原型，以便可以从exec.c调用它

**b6724eaeb9c2d0f337c82b2ddb42958**

3.在exec.c中的返回argc之前插入if (p->pid==1) vmprint(p->pagetable)，以输出第一个进程的页表



1. make qemu，检测是否成功

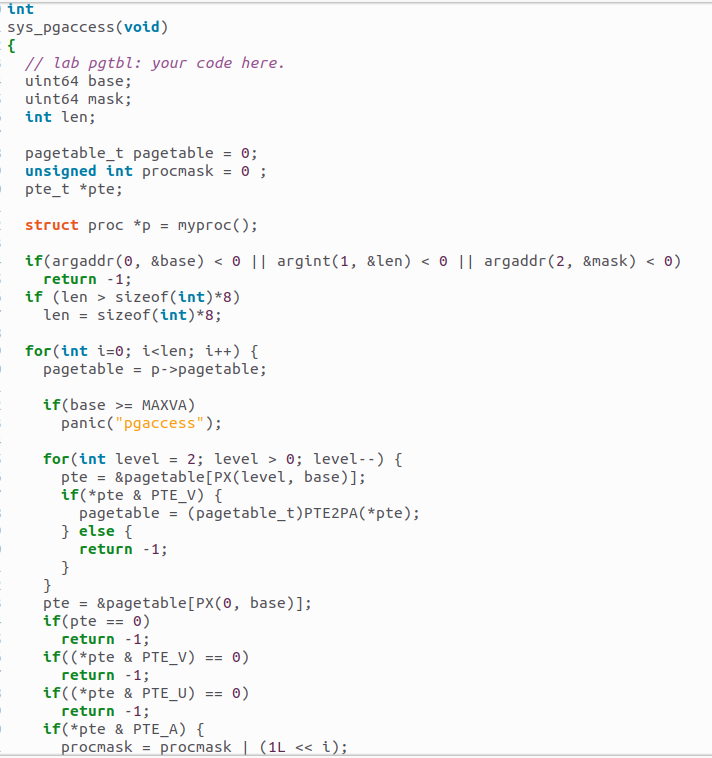


### 检测哪些页表被访问过

1. 首先看题目：“实现pgaccess()函数，它是一个报告已访问哪些页面的系统调用。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。当运行pgtbltest时，pgaccess测试用例通过，本部分实验你将获得满分。”
2. 我们第一步首先在kernel/riscv.h中定义常量PTE\_A。

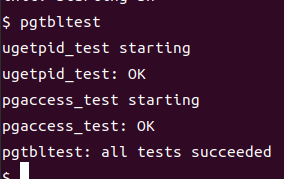


1. 在kernel/sysproc.c中编写sys\_pgaccess函数





4.测试



### 综合测试

## 实验小结

这个实验可以说是所有实验中花费时间最多的一个实验了。因为这次实验涉及到了操作系统中一个重要的概念，就是页表结构。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。

在本次实验的过程中，还涉及到了操作系统中地址的定位，在完成的过程中还要注意越界判断等。所以可以说，这次实验的完成是对能力的一个很好的提升。

# 实验四 traps

## 实验目的

1. 进一步了解系统调用所发挥的重要作用。
2. 掌握通过中断进行系统调用的过程及其所发挥的作用。
3. 了解简单的汇编语言，知道汇编语言是如何发挥作用的。
4. 理解 xv6 中的堆栈，同时试着实现一个用户级中断处理。

## 实验内容

1. 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。
2. 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。
3. 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

### RISC-V 程序集

1. 我先阅读 call.asm 的源代码，可以发现有函数 g、函数 f 和 main 函数，其中：main 函数调用如下：

exit(0);

a0,0x0

a0,a0,1968 # 7d8 <malloc+0xea> ra,0x0

1536(ra) # 630 <printf>

auipc

addi auipc jalr

li a2,13

li a1,12

4635

45b1

00000517

7b050513

00000097

600080e7

24:

26:

28:

2c:

30:

34:

printf("%d %d\n", f(8)+1, 13);

s0,sp,16

addi

addi sp,sp,-16

sd ra,8(sp) sd s0,0(sp)

1141

e406 e022

0800

1c:

1e:

20:

22:

void main(void) {

|  |  |  |
| --- | --- | --- |
| 38: | 4501 | li a0,0 |
| 3a: | 00000097 | auipc ra,0x0 |
| 3e: | 27e080e7 | jalr 638(ra) # 2b8 <exit> |

1. 哪一个寄存器存储了传递给函数的参数？例如，哪一个寄存器存储了 main 函数中调用 printf 函数的参数 13？

通过阅读源代码，我可以发现 RISC-V 中一共有 a0~a7 一共 8 个寄存器。在 main 函数调用 printf 函数的过程中，参数 13 被存储在了寄存器 a2 中。

1. 在汇编代码中，main 函数对函数 f 的调用在哪里？对函数 g 的调用在哪里？ 需要注意，编译器可能会进行函数内联。

通过阅读 main 函数，我会发现对于函数 f 调用是直接计算出了结果 12， 对于函数 g 的调用则是内联在了函数 f 中。

1. 函数 printf 位于什么位置？

通过阅读源代码，我发现了这一句：

34: 600080e7 jalr 1536(ra) # 630 <printf>

也即跳转到 ra+1536 的位置，此时 ra 的值即为 pc 的值，也即 printf 的地址为

0x30+1536=0x630。

1. 在 main 函数通过 jalr 跳转到 printf 函数后，此时 ra 寄存器中的值为多少？ 查询 jalr 指令，我发现：跳转并链接指令（jal）具有双重功能。若将下一条指令 PC + 4 的地址保存到目标寄存器中，通常是返回地址寄存器 ra，便可以用它来实现过程调用。如果使用零寄存器（x0）替换 ra 作为目标寄存器，则可以实现无条件跳转，因为 x0 不能更改。像分支一样，jal 将其 20 位分支地址乘以 2，进行符号扩展后再添加到 PC 上，便得到了跳转地址。跳转和链接指令的寄存器版本（jalr）同样是多用途的。它可以调用地址是动态计算出来的函数，或者也可以实现调用返回（只需 ra 作为源寄存器，零寄存器（x0）作为目的寄存器）。Switch 和 case 语句的地址跳转，也可以使用 jalr 指令，目的寄存器设为 x0。因此，ra 此时存储的应当是 printf 函数的返回地址，也即 0x38。
2. 我直接运行下面的代码：

unsigned int i = 0x00646c72;

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

输出结果为：He110 World。我不需要修改，因为编译器会自己进行转换。

1. 在下面的代码中，‘y=’后面所输出的值是什么？为什么会发生这一情况？

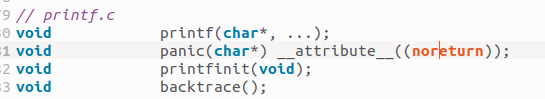
**printf**("x=%d y=%d", 3);

它所输出的是寄存器 a2 的值，这是因为 printf 会从 a2 寄存器读取参数作为

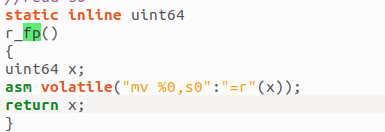
y 的值。

### 实现 Backtrace

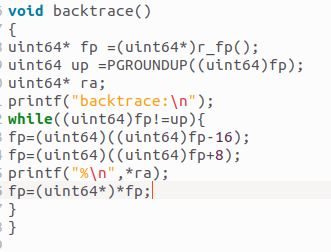
1. 首先在在kernel/defs.h添加定义void backtrace（）



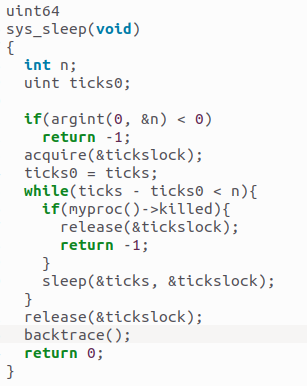
1. 修改kernel/riscv.h中增加r\_fp()的实现，使用内联汇编从寄存器 s0 中读取一个 64 位整数，并将其作为返回值返回,作用就是用来读取寄存器s0



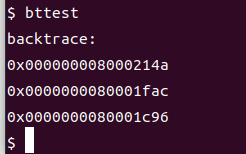
1. 在kernel/printf.c中增加backtrace()的实现，它使用帧指针链（Frame Pointer Chain）打印当前函数调用链的栈帧信息。



1. 在kernel/sysproc.c的sys\_sleep()函数中调用backtrace()



1. 最后，在上面的进入该函数之前利用寄存器存储下来当前进程状态，就完成了本次实验。测试如下：



### 4.3.3 Alarm-test0()：invoke handler

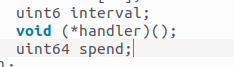
1.首先看实验的要求，在这个练习中，你将为xv6添加一个特性：在进程使用cpu时间时，定期发出警报。根据题目的提示信息，我们首先在Makefile加入相应的指令。



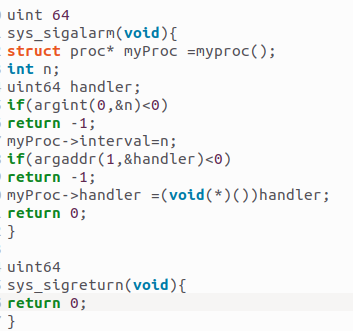
2.然后在usys.pl中加入之后用到的函数的声明，并修改user.h,syscall.h，syscall.c，proc.h文件在其中声明



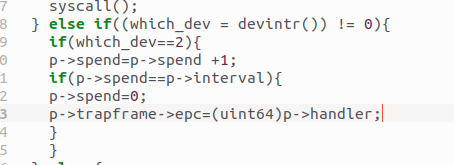
fe6cb53c45c3e48446dcefa48bccf1b

ccd938a0ed3a0026bd394a5d3190394166ec004d239e9f44b959e8479b3914

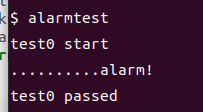
3.增加声明的两个函数，首先sys\_sigalarm() 函数，它的作用是设置当前进程的 alarm 时钟周期，并为当前进程设置 signal handler。该函数首先通过 myproc() 函数获取当前进程的指针，然后使用 argint() 和 argaddr() 函数从用户空间获取参数。如果参数获取失败，函数返回 -1。如果参数获取成功，则将 alarm 时钟周期和 signal handler 设置到当前进程的结构体中，并返回 0。然后是sys\_sigreturn() 函数，她的作用是从内核模式返回到用户模式，并将控制权交还给调用该系统调用的用户程序。该函数没有参数，只需要返回一个值，表示系统调用是否执行成功。在这里，该函数只是简单地返回 0。



1. 增加判断，作用是实现 alarm 时钟的功能，即在进程花费一定的时间后，自动触发 signal handler。通过这种方式，可以实现进程间通信和信号处理等功能。



1. 测试

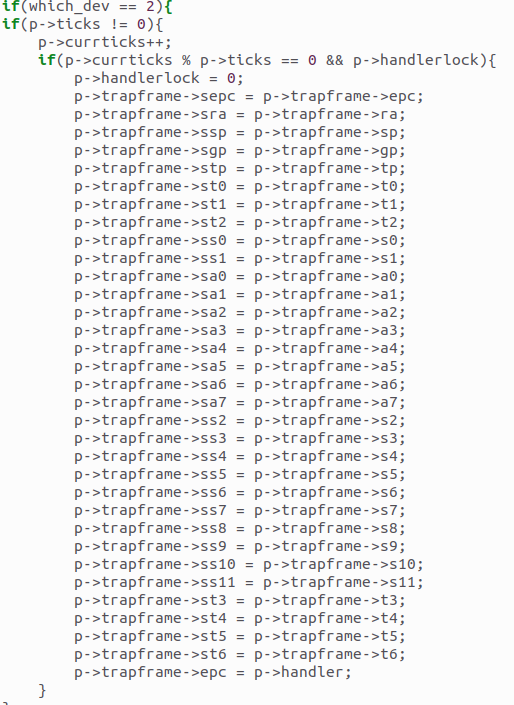


### 4.3.4 Alarm-test1/test2(): resume interrupted code

1.首先是修改kernel/proc.h

996d9f6661ebaf8930c4bc9a7c5eed8

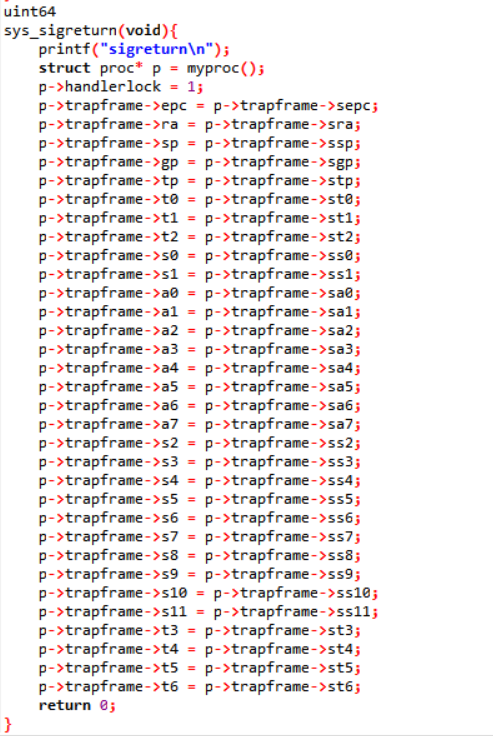
2.修改kernel/proc.c



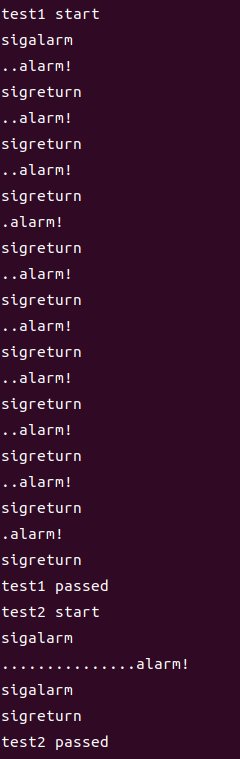
3.修改kernel/defs.h

bc3e9ba031900db218bcede39292f91

4.在kernel/sysproc.c写下函数

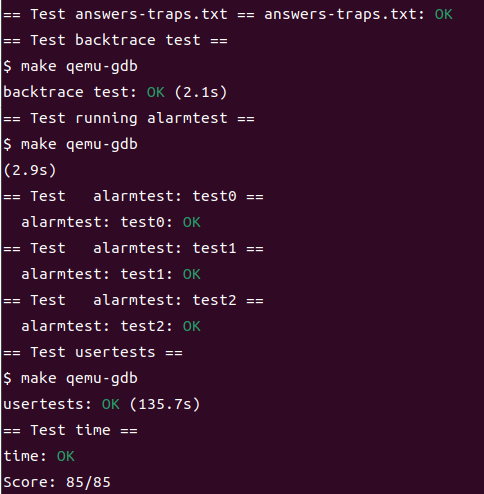


5.测试



### 4.3.5综合测试

我通过 make grade 进行本实验综合测试：



## 实验小结

这次实验主要是针对于操作系统中一个非常重要的中断机制而展开的。中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。

在本次实验的 alarm 中，我实际上就是通过实现中断现场的保存和处理结束后现场的恢复，实现了一个中断的过程。在这个过程中，其实我遇到的问题挺多的，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。

# 实验五 Copy on write

## 5.1实验目的

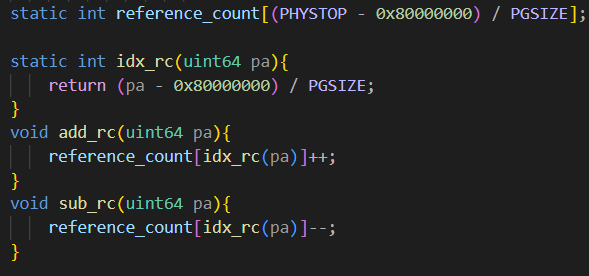
1. 理解什么是 Copy On Write（COW），其思想和作用是什么。
2. 熟悉简单 fork() 系统调用的工作机制，提出一种 COW 的实现手段。
3. 理解 xv6 中的 fork 函数，并修改内核程序，实现 COW。

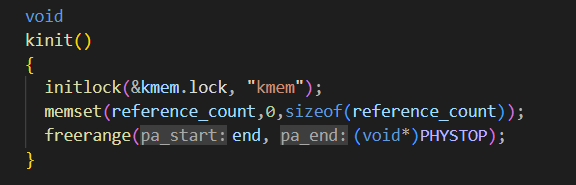
## 5.2实验内容

## 修改 xv6 内核代码，使系统支持 Copy on Write 功能。本实验要求为xv6系统增加 fork 函数写时复刻功能，即当 fork 完成时父子进程共享同一组只读物理内存页。当进程试图读写这些物理内存时，为该进程重新分配新的物理内存页，并将共享物理内存页的内容拷贝至新内存页。该实验与 Lazy 实验内容类似，但难度略微更大一些。

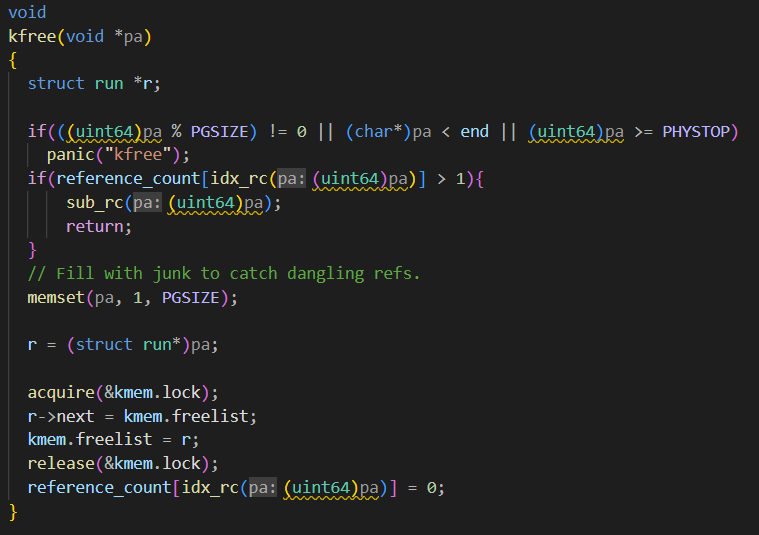
## 5.3实验步骤

1. 为了实现写时复刻功能，我们需要对物理内存页的被映射次数进行记录，以避免在物理内存页尚被进程使用时，被拥有其的另一个进程释放。在这里将物理内存页的引用计数实现安放在kernel/kalloc.c中，在这里，我们仅记录用户进程可能使用的物理内存范围，即0x80000000至PHYSTOP。使用
2. reference\_count 数组记录该范围的所有物理内存页的引用计数。对于特定的物理地址，可以通过其与0x80000000的偏移量除以PGSIZE得到其索引。在这里提供了对该数组进行更改的add\_rc 和 sub\_rc 接口函数。

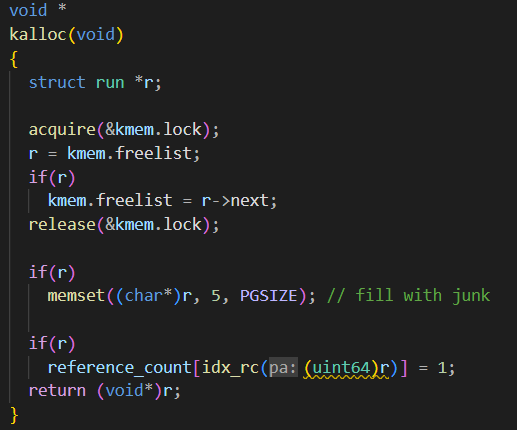


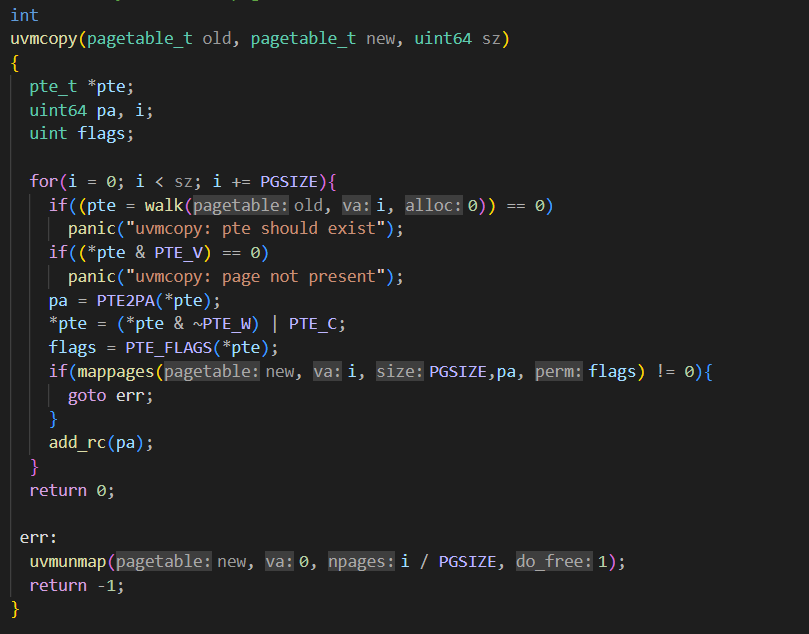
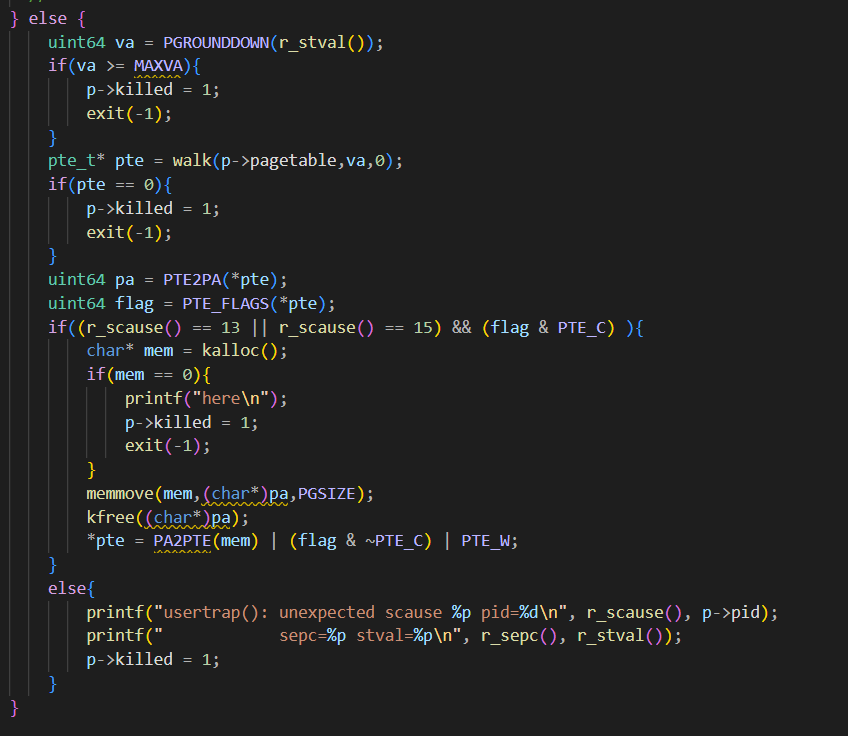


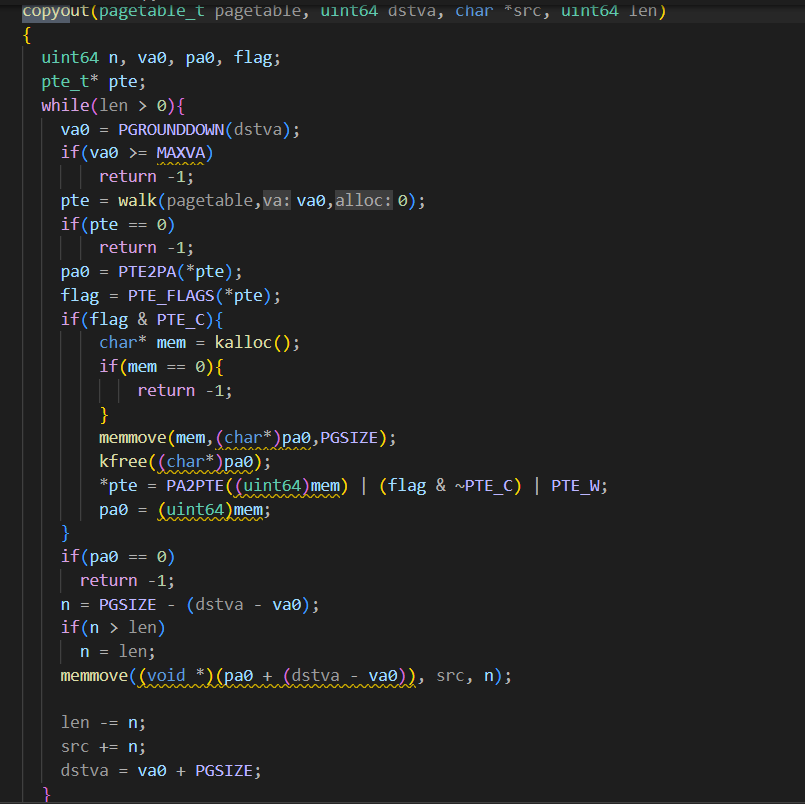
1. 在 kfree 函数中对所要释放的物理内存页的引用计数进行判断，如其引用计数大于1，则仅将其减一；如引用次数小于1（当为1时仅有一个虚拟内存页映射至该页，当为0时为 kinit 初始化调用kfree），则释放该页并将其添加至空闲列表：



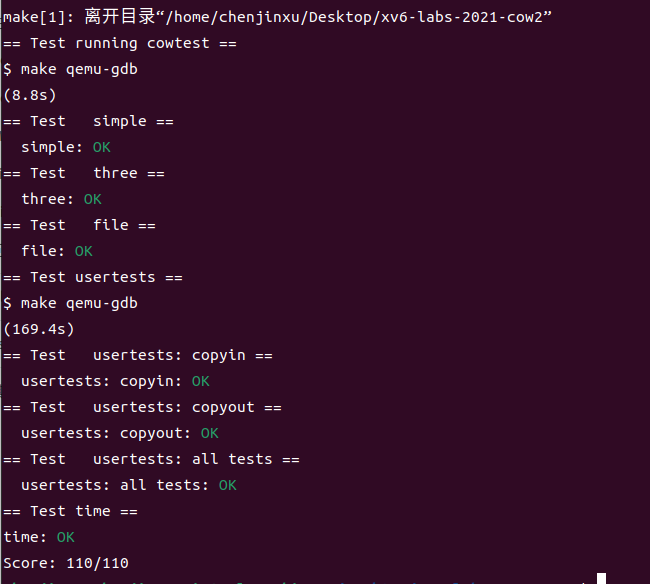
1. 在 kalloc 中，如空闲列表不为空，则将被分配的物理内存页的引用计数初始化为1：



1. 在 kernel/vm.c 中，我们更改 uvmcopy 以使得在 fork 时父子进程共享同一片物理内存。在这里，我们定义利用了页表项的用户保留flag项，规定 #define PTE\_C (1L << 5) ，以标记虚拟内存页是否为 COW 页（即被写时复刻的页）,在拷贝虚拟内存页时，我们仅将父进程的物理内存页映射至子进程的对应虚拟内存页，并为父子进程的虚拟内存页的页表项删去PTE\_W 标记、增添PTE\_C标记，并增加该物理内存页的引用计数（利用add\_rc）。
2. 在 kernel/trap.c 的 usertrap 中，我们捕捉 COW 页被写入的情况,首先，我们需要根据 r\_stval() 返回的引起中断的虚拟地址，判断该虚拟地址是否为异常地址。如非异常地址，则根据该虚拟地址提取出页表项的 FLAG 标记位和其所映射的物理地址pa。然后，我们需要利用 r\_scause() 判断中断原因是否为页错误导致的，并确认引起页错误的页是否为 COW 页，如为其他情况则终结当前进程。如果当前中断是由于写入了只读物理内存 COW 页导致的，则为当前虚拟内存页分配新的物理内存，并将旧的物理内存页的内容拷贝至新的物理内存页。之后，我们需要对被 COW 页映射的物理内存页调用kfree。在这里，如该物理内存页的引用计数为1，则释放了该页；如大于1，则递减了其引用计数。最后，我们需要将该虚拟内存页的页表项删去PTE\_C 标记，并增加PTE\_W 标记。

7.最后，我们需要为 copyout 增加与 usertrap 类似的判断代码，以防止对 COW 页的直接写入：

* + - 1. 完成之后，我在 qemu 模拟器中测试如下：



## 5.4实验小结

实验 5 和本次实验都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。

此外，虽然在实验任务中并未提到，但是我观察到 xv6 系统实现了分页式的内存管理，这使得父子进程间的内存可以按照页面为单位进行拷贝，这也提高 了一定的内存有效数据比例、降低了一定的拷贝开销（相较于直接复制整个进程）。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。

# 实验六 Multithreading

## 6.1实验目的

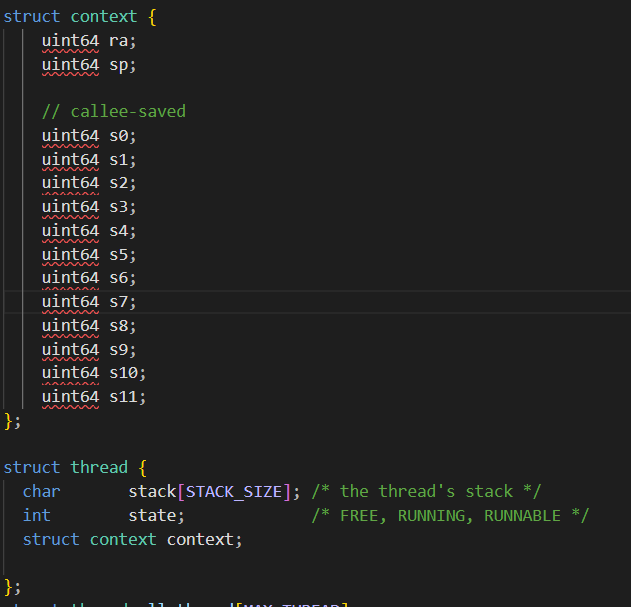
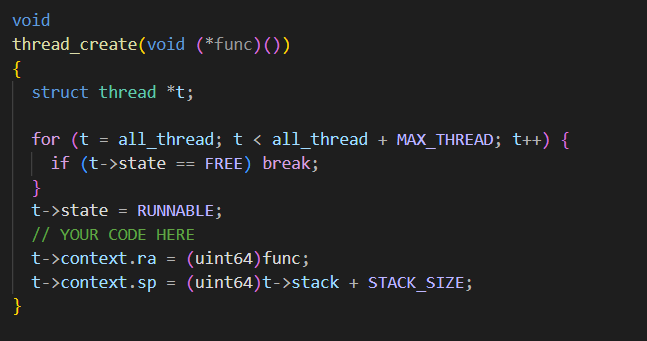
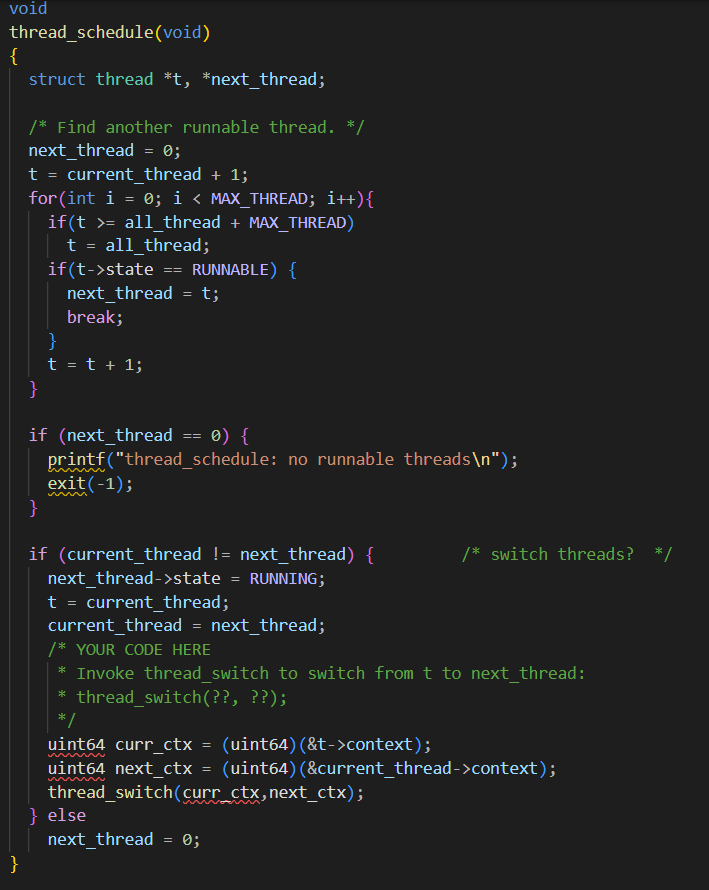
1. 熟悉多线程编程，理解进程切换方式和执行顺序。
2. 了解多核多线程对程序执行效率的影响。
3. 理解锁的含义，能够判断何时、何处应该加锁。
4. 回顾进程同步/互斥的模式并在程序中应用。

## 6.2实验内容

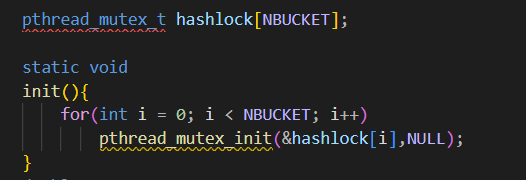
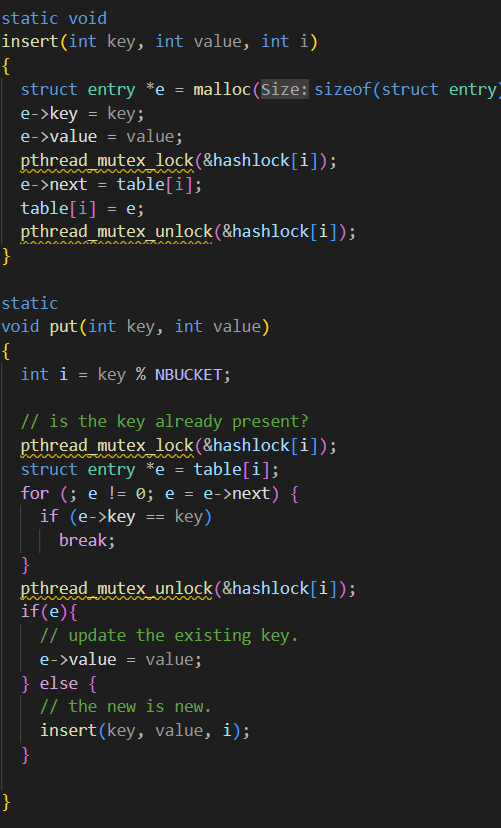
1. 实现一个用户级进程的创建和切换。
2. 使用 UNIX pthread 线程库实现一个线程安全的哈希表。
3. 实现 barrier 函数。其作用是：当进程到达 barrier 函数调用时，会开始等待其他进程，当所有进程都到达 barrier 时，才停止等待。

## 6.3实验步骤

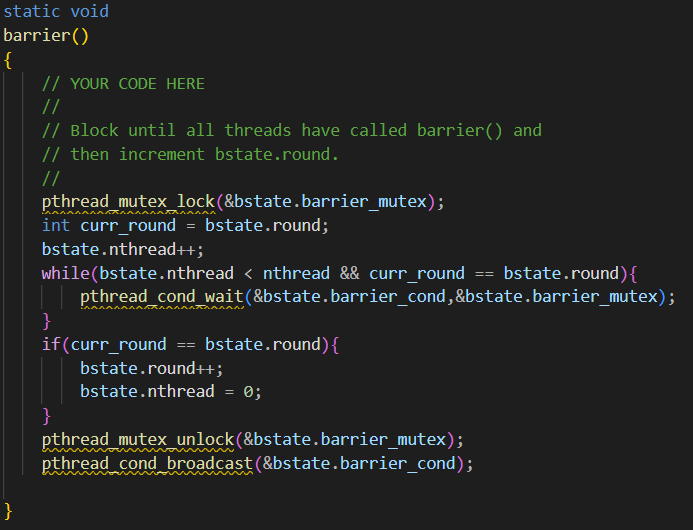
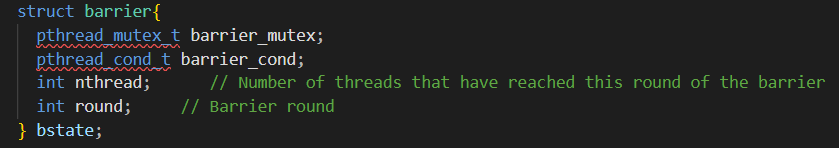
### 6.3.1实现一个用户级进程的创建和切换

1. 在本部分中，需要我们实现xv6用户级线程包。首先，为了在线程切换时保存当前线程上下文（即寄存器和程序计数器），需要在 thread 结构中增加 context 结构保存上下文，当 thread\_swithch 切换线程时，其将当前线程的上下文保存在其 thread 结构的 context 中，并将被调度线程 context 中的条目读入寄存器
2. 在 RISC-V 架构中，ra 寄存器保存了当前函数的返回地址。在 thread\_create 创建线程时，我们将 context->ra 置为线程所需执行函数的起始地址，以在 thread\_switch 中调用 ret 时，其控制流跳转至该线程的所需执行的函数：
3. 然后修改thread\_schedule 用于调度线程执行，其遍历所有线程，寻找状态为 RUNNABLE 的可执行线程，并调用 thread\_switch 实施具体的线程切换

### 6.3.2 使用 UNIX pthread 线程库实现一个线程安全的哈希表

1. 在本部分中，需要我们利用UNIX接口实现简易的并发哈希表，该哈希表需要在多个线程对其进行插入操作时保证线程安全。在这里，我们的保护方式是为每一个哈希桶分配一个互斥锁，当对一个桶中的元素进行遍历时，利用该锁对桶进行保护，首先是锁的初始化，init 函数被 main 函数调用：
2. 当 put 对于哈希表进行插入时，利用锁对针对链表结构的操作进行保护。在插入时对于链表结构的操作有部分，分别为 put 函数中遍历哈希桶的主键匹配，以及 insert 函数中对于哈希表的头插：

### 6.3.3实现 barrier 函数

1.在本部分中，需要利用UNIX接口实现 barrier 屏障功能，其功能为只有当参与屏障的所有线程均调用 barrier 函数时，任意线程才可以从 barrier 中返回，否则就阻塞在 barrier 中。具体的实现方式需要使用到UNIX中的同步原语 条件变量pthread\_cond\_t ，其功能与xv6中的sleep/wakeup机制相似。在这里，nthread 为用户输入的参与屏障的全部线程数、bstate.nthread 为当前达到屏障函数的线程数并被初始化为零、bstate.round 为屏障进行的轮数。在代码中，我们利用互斥锁 bstate.barrier\_mutex 保护 bstate 中的所有条目，并用条件变量bstate.barrier\_cond 实现线程间的同步。当一个线程调用 barrier 时，其对 bstate.nthread 进行递增，并记录当前的 bstate.round。然后，其进入循环判断，如当前进入屏障的线程数小于全部线程数，或者当前屏障轮数没有改变时，则调用 pthread\_cond\_wait 阻塞在 bstate.barrier\_cond条件变量上，并让出互斥锁使得其他线程得以调用 barrier 。当参与屏障的最后一个线程调用 barrier 时，其通过循环判断。在这里，将当前轮数 bstate.round 递增，使得其余被阻塞的线程可以跳出循环判断。并将 bstate.nthread 清零。在这里，值得注意的是该种同步方式使得相邻两轮的屏障之间的 bstate.nthread 不会互相影响。因为唤醒线程调用 pthread\_cond\_broadcast 前，其余所有线程都正在被阻塞，因此唤醒线程一定会在其余线程对 bstate.nthread 进行递增之前将其清零、也会在其余线程记录其当前轮数之前将 bstate.round 递增，其也一定是在该轮中最后一个对 bstate.nthread 进行递增的线程。因此，该种特性使得相邻两次屏障之间是相互独立的。当有线程达到该处时，说明该轮中所有线程均达到了 barrier ，因此调用pthread\_cond\_broadcast 唤醒其余阻塞进程。

### 6.3.4测试成绩

通过 make grade 进行本实验综合测试：至此，本实验全部完成。

## 6.4实验小结

本实验是关于多线程编程的。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。在操作系统课程中介绍了锁的概念和几个模型，并讨论了多线程存在的死锁问题。在实验中，实现了进程切换、运用锁来完成线程间的互斥（多线程哈希表）和运用锁和条件信号量来完成多线程之间的同步（barrier），三个小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。

# 实验七 net

7.1实验目的

本实验的目标是编写一个 xv6 设备驱动程序，用于网络接口卡（NIC）。通过完成这个实验，我将学习如何在 xv6 操作系统中实现网络通信功能，并理解网络设备驱动程序的工作原理

7.2实验内容

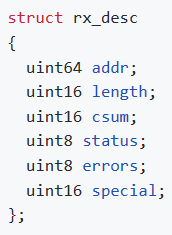
在这个实验中，你需要完成 kernel/e1000.c 文件中的 e1000\_transmit() 和 e1000\_recv() 两个函数，以使驱动程序能够发送和接收数据包。你需要根据提供的 E1000 软件开发手册，在这两个函数中编写相应的代码。

具体而言，需要完成以下任务：

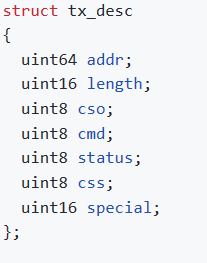
1. 在 e1000\_transmit() 函数中，实现将数据包发送到网络的功能。首先我需要了解 E1000 的寄存器和标志位的定义，以及数据包发送的过程。参考 E1000 软件开发手册的第 3.3 节和第 14 节，以及提供的代码和注释。
2. 在 e1000\_recv() 函数中，实现接收来自网络的数据包的功能。我需要了解数据包接收的过程以及相关的寄存器和标志位。参考 E1000 软件开发手册的第 3.2 节和第 14 节，以及提供的代码和注释。

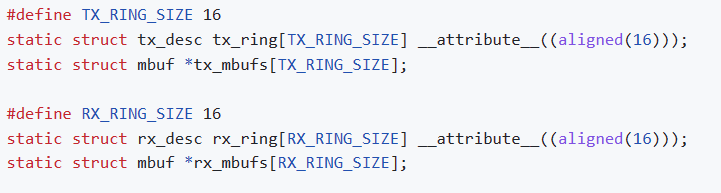
7.3实验步骤

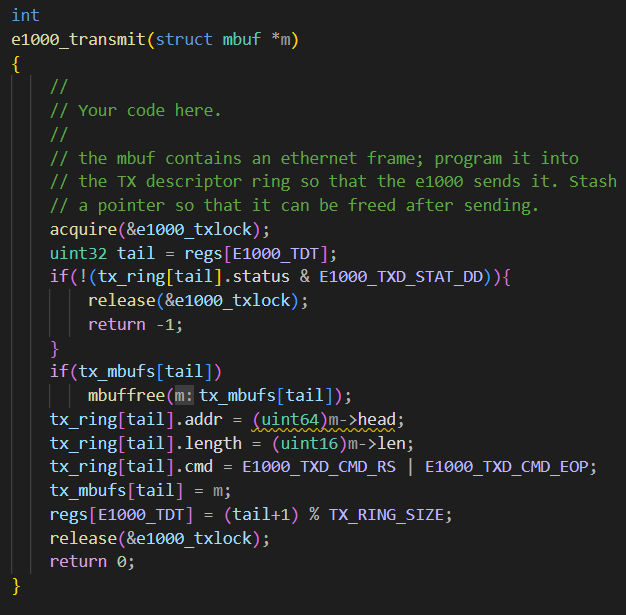
1.为了完成本实验，我们必须要了解E1000硬件的数据包接受与传输机制。在软件层面上，E1000的接受和传输主要都是基于一种名为 描述符 的数据结构完成的。描述符担任了硬件与软件数据传输中介的功能，并为E1000设备提供了DMA（直接内存访问）机制。在每个描述符中都存放了一个位于RAM内存的地址，其可被硬件和软件共同访问和操作。并且，为了提升E1000设备的大规模数据承载能力，E1000驱动程序维护了 描述符环 数据结构作为数据包传输和接受的缓冲机制。其中，主要的条目为addr、length 和 status。其中 addr 为该描述符所映射的地址，其内存由驱动程序软件预先分配，并当E1000设备接受到数据包时被硬件填充。length 为硬件填充的数据长度。status 中存放了当前描述符的状态，其中最重要的状态位为 E1000\_RXD\_STAT\_DD ，当硬件完成了对当前描述符的填充时，该位被置为1，以通知驱动程序软件将其提取。

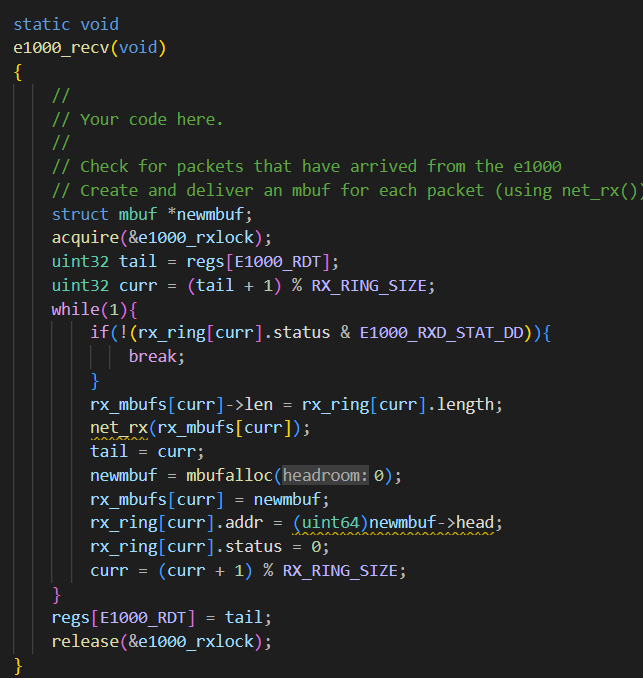


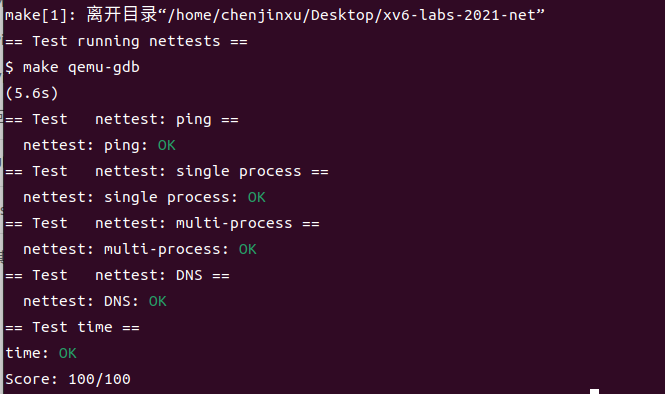
2.传输描述符的数据结构如下所示，其中，主要的条目为addr、length 、cmd和 status。addr 为该描述符所映射的内存，其内容被软件填充，并被硬件所提取；length 为软件填充的数据长度；cmd 控制了该描述符的行为，当 E1000\_TXD\_CMD\_EOP 被设置时，表示当前描述符为用于存储一块数据包的最后一个描述符（当数据包较大时可由多个描述符共同存储），当 E1000\_TXD\_CMD\_RS 被设置时，硬件在提取当前描述符的内容时，会自动将该描述符的status 置为 E1000\_TXD\_STAT\_DD 。



1. 其与接受描述符环的结构类似，队列的长度被软件初始化并存放在 E1000\_TDLEN 寄存器中。Head 指向了当前硬件所要提取的描述符，被存放在 TDH 中并初始化为 0，并被硬件自动更新；Tail 指向了当前软件所要填充的描述符，被存放在 TDT 中并初始化为 0，并被驱动程序更新。在这里，接受和传输描述符所指向的内存段被构造成了 mbuf 缓冲区数据结构以方便被其他函数操作，且驱动程序为接受和传输描述符环各分配了一个存放 mbuf 缓冲区指针的数组：
2. 在本实验中，驱动程序的初始化及一些支撑数据结构已经被编写完成，在这里只需填充用于接受和传输数据包的主要驱动程序函数即可。其中，用于传输数据包的函数为 int e1000\_transmit(struct mbuf \*m) ，其由驱动程序主动调用来传输数据包，其实现如下。在这里，使用互斥锁 e1000\_txlock 保护传输描述符环，其在 e1000\_init 中被初始化。

通过提取 TDT 寄存器的值，得到当前所需填充的描述符索引，并检查 status 判断其是否已经被硬件提取完毕或未被填充（在e1000\_init 中 status 被初始化为 0）。如当前描述符被填充且未被硬件提取完毕，则释放锁并返回 -1。如当前描述符未被填充或已被硬件提取完毕，则释放被该描述符所指向的缓冲区。并将当前描述符的 addr 置为函数传入的缓冲区的 head 地址（其中存放了缓冲区的具体数据）；将 length 置为缓冲区所表示的数据长度；并将 cmd 置为 RS | EOP（原因见上文）。最后，递增 TDT 寄存器的值。

1. 用于接受数据的函数为 static void e1000\_recv(void)。当操作系统接受到数据包时，其将产生设备中断并在 devintr 中调用 e1000\_intr ，该函数调用 e1000\_recv 完成对数据包的具体接受工作，其实现如下。在这里，使用互斥锁 e1000\_rxlock 保护接受描述符环，其在 e1000\_init 中被初始化。首先提取 RDT 得到最后一个应被硬件填充的描述符索引，并将 curr 置为 RDT 所指向的下一个描述符索引，其为可能被硬件填充并待软件接受的第一个描述符。然后，陷入循环以提取所有被硬件填充的描述符。值得注意的是，在软件进行提取的过程中，硬件也可以同时对描述符进行填充，因此该方法可应对次数超出环大小的硬件填充。在循环中，通过检查 status 以判断当前描述符是否已被硬件填充，如未填充则跳出循环结束函数。如当前描述符被硬件填充，则通过缓冲区中的实际数据长度更新描述符的 length 字段，并调用 nex\_rx 将该缓冲区的内容发生至网络堆栈使得操作系统可以将其提取。在完成 new\_rx 后，将 tail 置为 curr 以记录最后一个被软件所提取的描述符索引。然后，为该描述符分配新的缓冲区并将描述符的 status 和 addr 字段更新。最后，将 curr 更新为下一个描述符的索引。在循环结束后，将 tail 回写至 RDT。
2. 实验完成后进行make grde



7.4实验小结

本次实验的主要目的是编写一个 xv6 操作系统的设备驱动程序，用于网络接口卡（NIC）。通过完成这个实验，我旨在学习如何在 xv6 操作系统中实现网络通信功能，并深入理解网络设备驱动程序的工作原理。在实验的具体内容中，我需要完成 kernel/e1000.c 文件中的 e1000\_transmit() 和 e1000\_recv() 两个函数，以实现驱动程序能够发送和接收数据包的功能。为此，我需要依据提供的 E1000 软件开发手册，编写相应的代码。在 e1000\_transmit() 函数中，我的任务是实现将数据包发送到网络的功能。首先，我了解了 E1000 寄存器和标志位的定义，以及数据包发送的过程。我参考了 E1000 软件开发手册的第 3.3 节和第 14 节，以及提供的代码和注释来进行编码。另外，在 e1000\_recv() 函数中，我成功实现接收来自网络的数据包的功能。为此，我学习了数据包接收的过程以及相关的寄存器和标志位。我参考 E1000 软件开发手册的第 3.2 节和第 14 节，以及提供的代码和注释来进行编码。通过完成这些任务，我深入了解了网络设备驱动程序的工作原理，并获得在 xv6 操作系统中实现网络通信功能的实践经验。这将有助于我扩展对操作系统和网络通信的理解，并提升我的编程技能。

# 实验八 Locks

## 实验目的

1. 理解锁竞争，包括锁竞争产生的原因以及解决方法。
2. 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。
3. 加深对于使用锁来达到进程互斥方法的掌握程度。

## 实验内容

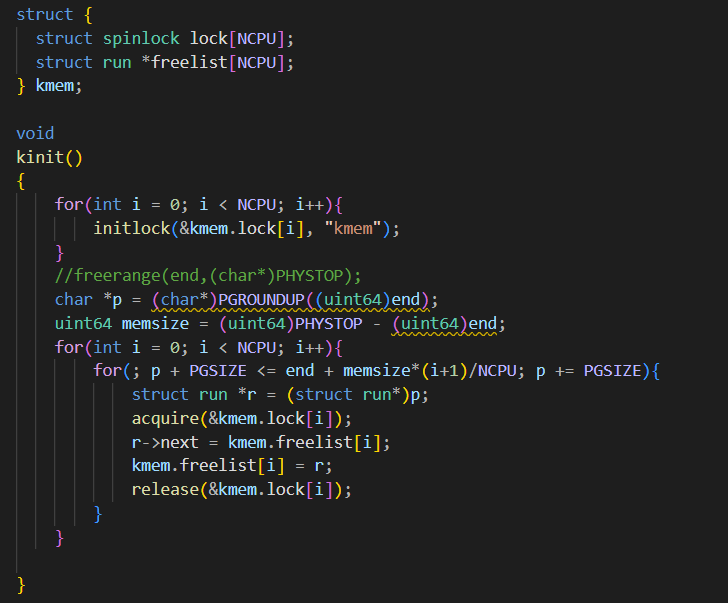
1. Memory allocator (moderate)
2. 重新设计系统管理内存的方式，实现每个 CPU 管理一个空闲链表和一个锁。 修改系统的 IO 缓冲区，通过散列的方式完成对缓冲区和锁的分割。

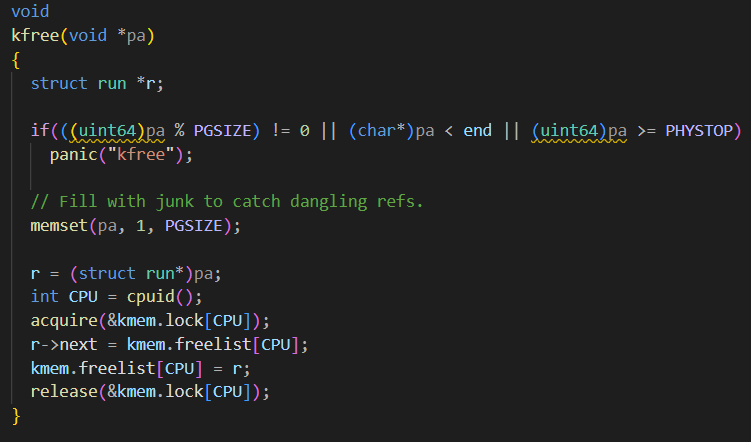
## 实验步骤

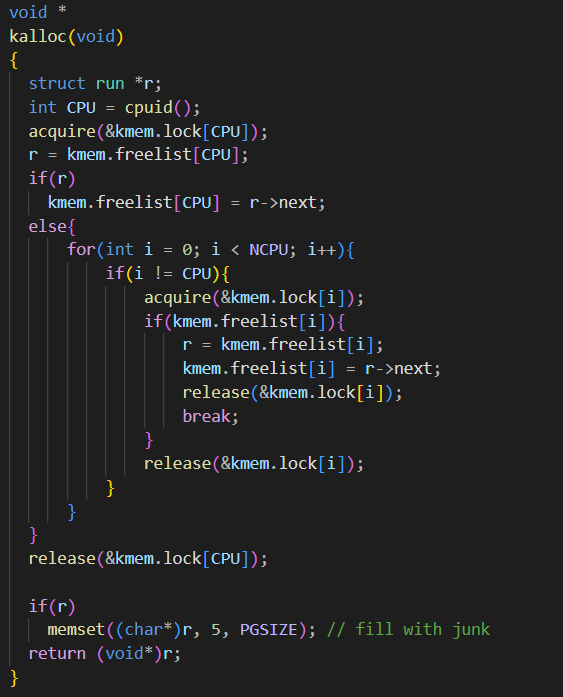
### Memory allocator (moderate)

1.在本部分中，我们需要将内存分配器的锁粒度从全局锁降低为每CPU一个的小粒度锁，并使得各CPU分别拥有并使用它们的空闲列表，使得各CPU的内存分配是相互独立的，以提升并行性。当一个CPU的空闲列表为空时，该CPU需要在其他CPU的空闲列表中试图寻找空闲内存块，这一点为本部分的主要复杂度来源。

首先，我们需要在 kmem 结构中为每个CPU各自分配一个空闲列表及互斥锁。在 kinit 中，我们将空闲内存块均匀地添加在各个空闲列表中，并初始化各空闲列表的互斥锁。



2.改变kfree函数，其在对应空闲列表锁的保护下，将空闲块添加至该空闲列表

3.在 kalloc 中，我们首先获取本CPU的互斥锁，并尝试从本CPU的空闲列表中获取空闲块。如本CPU的空闲列表为空，则在其余CPU的互斥锁保护下，尝试在其空闲列表中获取空闲块。

### 8.3.2Buffer cache (hard)

1.在本部分中，需要降低磁盘缓冲区的锁粒度，以提升其并行性。基本思想是，原本磁盘缓冲区通过单一空闲链表以管理缓存区，因此在 bget 和 brelse 必须获取全局锁以保护缓冲区。因此，我们可以通过哈希表的方式来管理缓冲区，使得一组 dev 及 blockno 与一个哈希桶单一映射，通过此方法我们仅需在 bget 和 brelse 中保持哈希桶的锁，既可以保证对应缓冲区保持并发安全。首先，我们在 bcache 中存放若干哈希桶及其对应的锁，并定义一系列用于在哈希表中增添、查找、删除元素的功能函数。值得注意的是，我们不在对哈希表操作的功能函数中使用互斥锁，而在 bget 中根据特定情况使用锁：

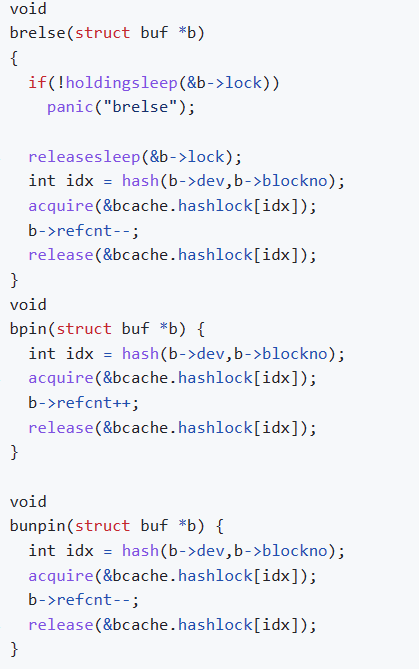


2.bget 函数是本部分的难点所在。其难点主要在于，在哈希表中查找元素及在查找失败在哈希表中驱逐、分配元素的工作必须是原子的。在这里，解决该问题的基本思想是利用哈希桶的锁，保证对缓冲区中磁盘块、哈希桶的操作是并行安全的。其原理在于，各磁盘块都根据 其 dev 和 blockno 被组织到了各自的哈希桶中，即磁盘块与哈希桶具有单一映射关系，因此对于哈希桶的锁保护也可以同时保护对应的磁盘块。其具体代码如下：

### 2c88b3c30b421583069a543396e17da

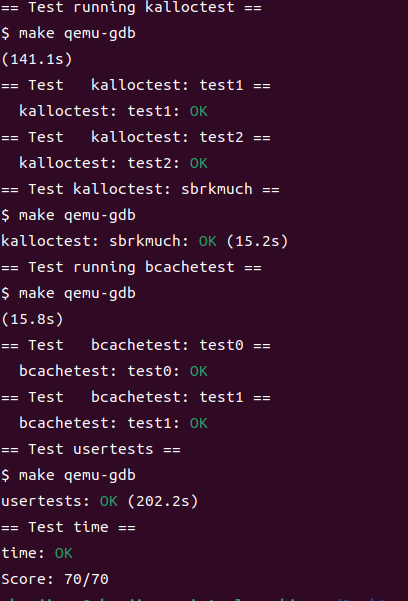
首先根据所需查询的磁盘块的 dev 和 blockno 得到对应的哈希桶，并在哈希桶锁的保护下利用 find 搜索缓存区中是否缓存了该磁盘块。如查找成功，则在哈希桶锁的保护下递增 b->refcnt。最后，释放哈希桶锁、获得磁盘块睡眠锁并返回。在这里，为了方便区分，将所需查询的磁盘块所对应的哈希桶称为查询哈希桶，将当前缓冲区中可用缓存块所对应的哈希桶成为驱逐哈希桶。如未查找成功，则不释放查询哈希桶锁，以保证在查询哈希桶中查找及在查询哈希桶插入操作是原子的。然后，遍历bcache.buf 中各缓存块，在其对应的哈希桶锁的保护下检查其是否可用（b->refcnt 为0）。在这里值得注意的是，如bcache.buf 中的缓存块是否映射至查询哈希桶，以避免重复获取该锁导致死锁。如当前缓存块可用（在这里为了简单采用了序列化驱逐的方式，而并未采用LRU方式），则在驱逐哈希桶中驱逐该缓存块中原有的磁盘块，并将所需查询的磁盘块信息填入该缓冲块，并插入至查询哈希桶。最后，释放驱逐哈希桶的锁和查询哈希桶的锁。如当前缓存快不可用，则释放该缓存块所对应哈希桶的锁。

1. 在 brelse 中，释放磁盘块的睡眠锁，并在其对应哈希桶锁的保护下递减b->refcnt（在 bpin 和 bunpin 中同理）：



### 8.3.3测试成绩

通过 make grade 进行本实验综合测试：



## 实验小结

实验 7 与本实验都是关于并行编程中问题的性能优化。用户可以实现用户级的多线程程序来更好地使用处理器资源，而分时系统通过多个进程的并发运行可以让多个程序看起来像同时在执行。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）。

我在两个实验中分别体会了控制进程的同步/互斥（实验 7）和分割资源来使锁竞争得到缓解、提高资源并行利用率。实验同时还涉及到了关中断、散列、资源分配等其他方面的细节，关联了我之前课程中学习的知识。从功能性角度来说，一个操作系统只需要拥有运行程序的能力，与用户交互的系统也最多需要增加并发运行这一个功能限制。而在实验中通过对资源的划分实现了提高系统效率和资源利用率这些非功能性需求，在一个面向现实世界的操作系统中，非功能性需求一样有重要地位。

# 实验九 File System

## 实验目的

1. 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。
2. 掌握通过增加索引级数的文索引扩展方式。
3. 理解文件映射和打开的方式，了解 xv6 系统的文件系统层级架构。
4. 理解硬链接和软链接的实现方式。

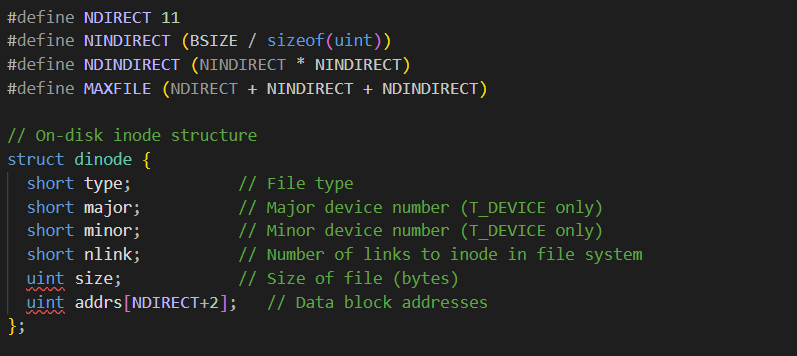
## 实验内容

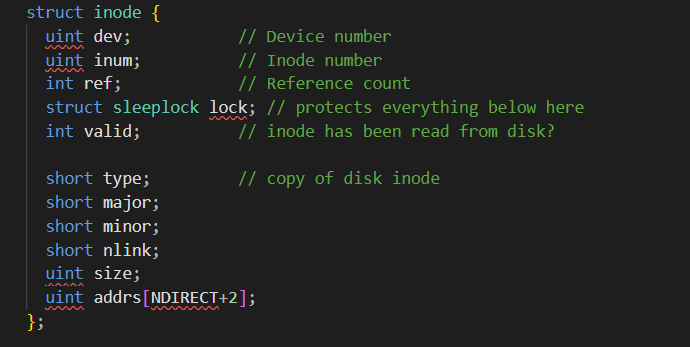
1. 修改文件系统使其支持更大的文件存储。具体做法是实现二级间接的索引节点（inode）。
2. 增加一个通过符号链接的系统调用。将目标符号与系统路径链接，使对符号文件的操作都同步到对相应路径文件的操作。

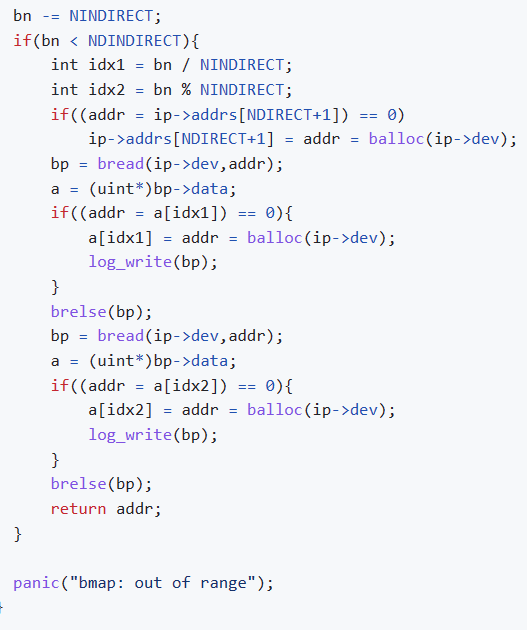
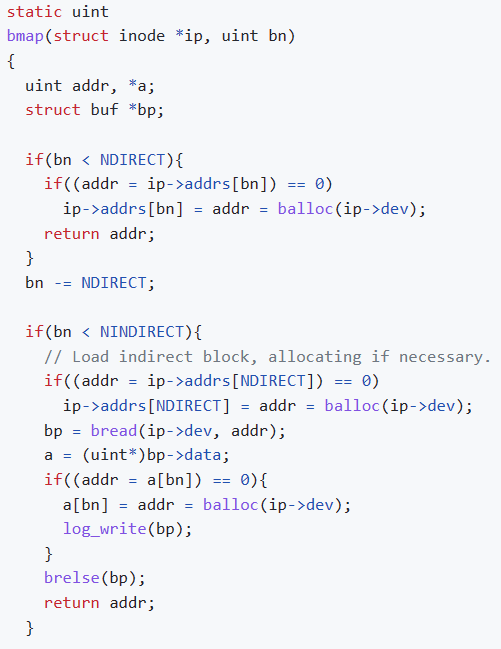
## 实验步骤

### 修改文件系统使其支持更大的文件存储

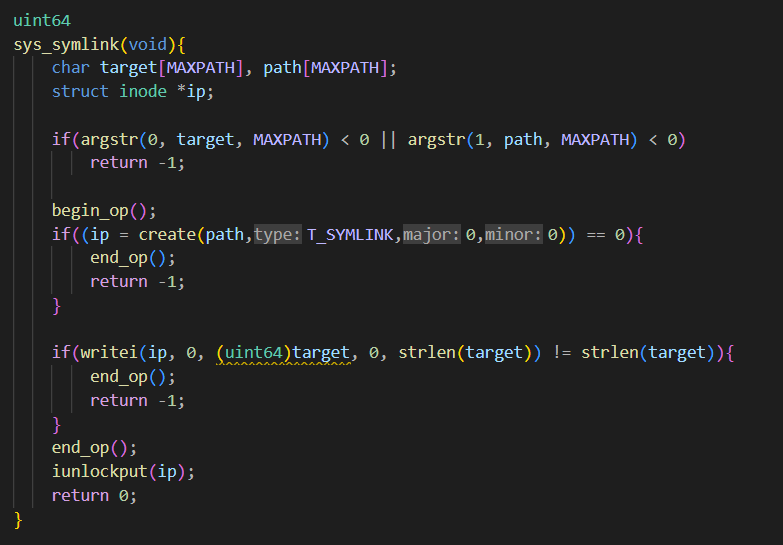
1. 在本部分中需要为 inode 提供双重间接块，使得xv6中的最大文件大小被增加。其原理与 inode 中的原有间接块类似：在双重间接块中，存放了 NINDIRECT 个间接块，而一个间接块中又存放了 NINDIRECT 个直接磁盘块，因此在双重间接块中即可表示 NINDIRECT\*NINDIRECT 个直接磁盘块，其使得xv6的最大文件大小大大增加。为了实现双重间接块，我们需要对 dinode 、 inode 及一系列相关宏进行更改：

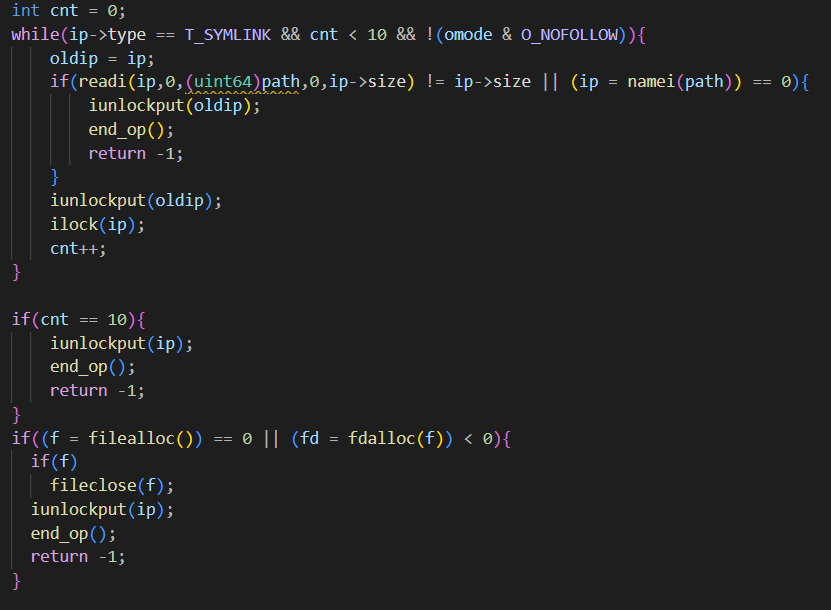




1. 之后，我们需要对 bmap 和 itrunc 函数中对 inode 的 addrs 操作方法进行更改：  
   
2. 在 btrunc 中，我们需要搜索双重间接块中的所有子间接块，以及子间接块中的所有直接块，并释放其中被分配的块

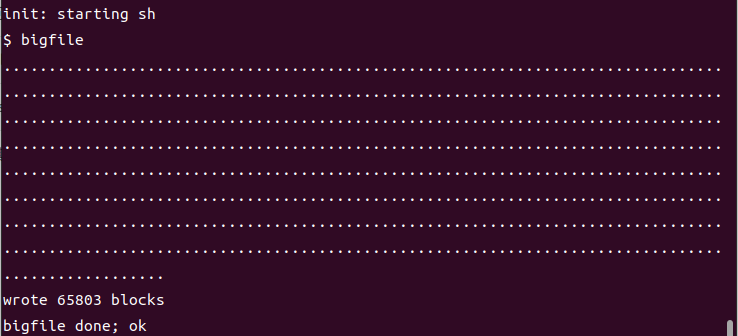
### 增加一个通过符号链接的系统调用

1. 在本部分中，需要为xv6提供符号链接功能。在UNIX操作系统中，文件链接可以被分为硬链接和符号链接（软连接）两种。其中硬链接即为原xv6中的 sys\_link 功能，其为链接得到的新文件分配与被链接的旧文件相同的 inode，因 inode 中存放了构成文件的各磁盘块的块号，因此硬链接得到了原有文件的物理副本，且仅当硬链接文件和原文件均被删除时，构成文件的磁盘块才会被释放；符号连接则仅在新文件中存放了被链接文件的路径名，其链接方式更加轻量且可跨文件系统、跨主机链接，但当被链接文件被删除时符号链接文件也随之失效。符号链接的具体的实现并不复杂，仅需在系统调用 sys\_symlink 中创建类型为 T\_SYMLINK （在 kernel/stat.h 中定义）的文件，并在该文件中存放被链接文件的路径名即可：
2. 除此以外，我们还需对 sys\_open 的行为进行修正，在寻找或创建 inode 后，分配 file 和文件描述符前，需要根据 O\_NOFOLLOW （kernel/fcntl.h中定义）判断是否对符号链接进行跟踪。在这里，首先对当前 inode 类型进行判断，如未指定 O\_NOFOLLOW、当前文件类型为 T\_SYMLINK 、且跟随次数小于10，则对该符号链接进行跟随。跟随方式即为从符号链接中读取其指向的路径名，并将当前 inode 指定为路径名所对应的文件即可。如无法找到该文件或跟随次数超过10，则 sys\_open 返回 -1。

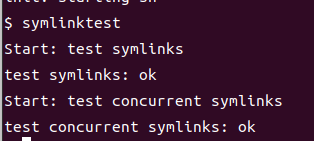


### 测试成绩

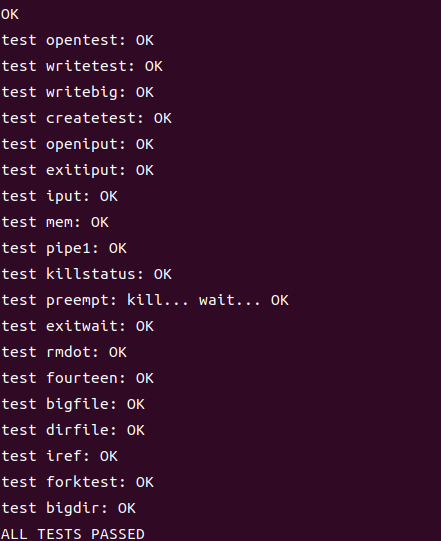
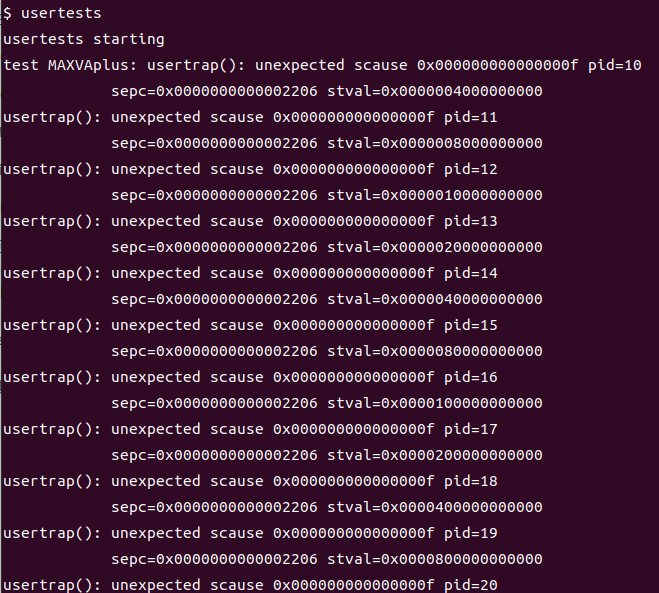
运行bigfile，通过



运行symlinktest



运行 usertests，全部通过。



至此，实验九全部完成。

## 实验小结

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。

第一个部分中的拓展文件大小是一个典型的程序数据结构设计思想，可以看作是对于一个高度为 1 的树，将其某一些节点拓展成指向相似的节点的非叶子结点，从而扩展这个数的容量。

第二个部分是实现系统文件软链接相关的内容。这个部分在课程内要求不高， 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的理解。

通过这次实验，我不仅仅扩展了 xv6 文件系统的功能，还提高了我对操作系统和计算机组成原理中相关概念的理解。这次实验提供了有趣而具有挑战性的任务，帮助我们加深对文件系统设计和软链接的理解，并提升了我的编程技能和对计算机系统的整体认识。

# 实验十 Mmap

## 实验目的

1. 了解使用虚拟内存的原因和好处。
2. 理解创建和释放文件映射的方式。
3. 梳理文件内存映射的生命周期。

## 实验内容

添加 mmap 与 munmap 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。本实验的实验内容只有一条，但是难度被标为困难（hard），因此我还是将实验步骤细分，这样更利于实验的理解和报告的组织。细分后的实验分为四步：

* + 1. 理解虚拟地址映射和访问的方式，了解 mmap 系统调用的实际工作流程。
    2. 实现 mmap 系统调用。
    3. 实现 munmap 系统调用。

4.修改其他相关的函数。

## 实验步骤

### 理解虚拟地址与 mmap

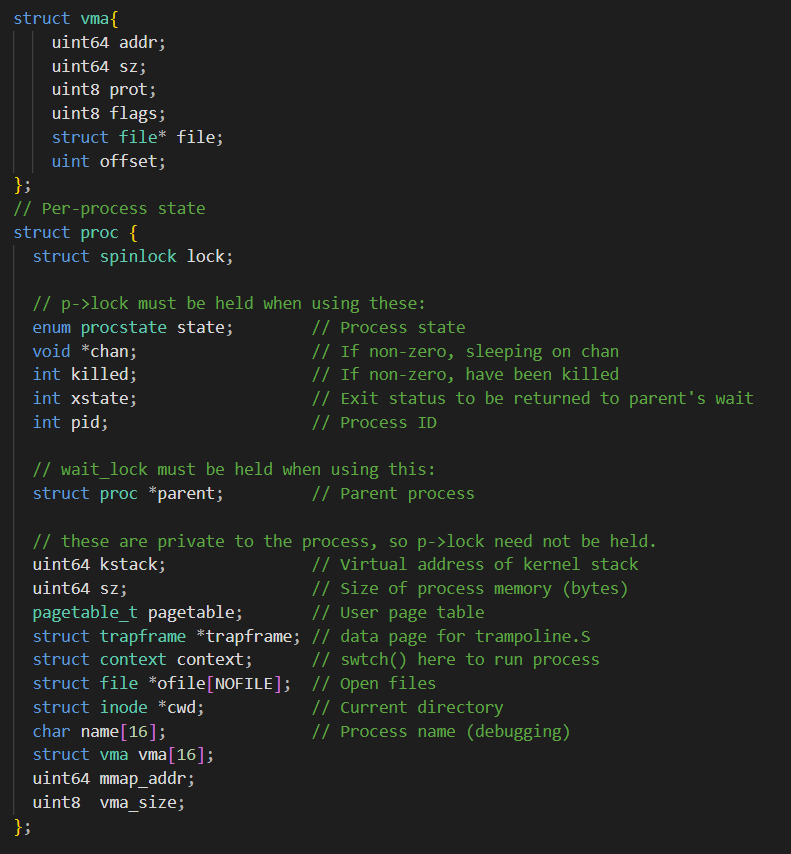
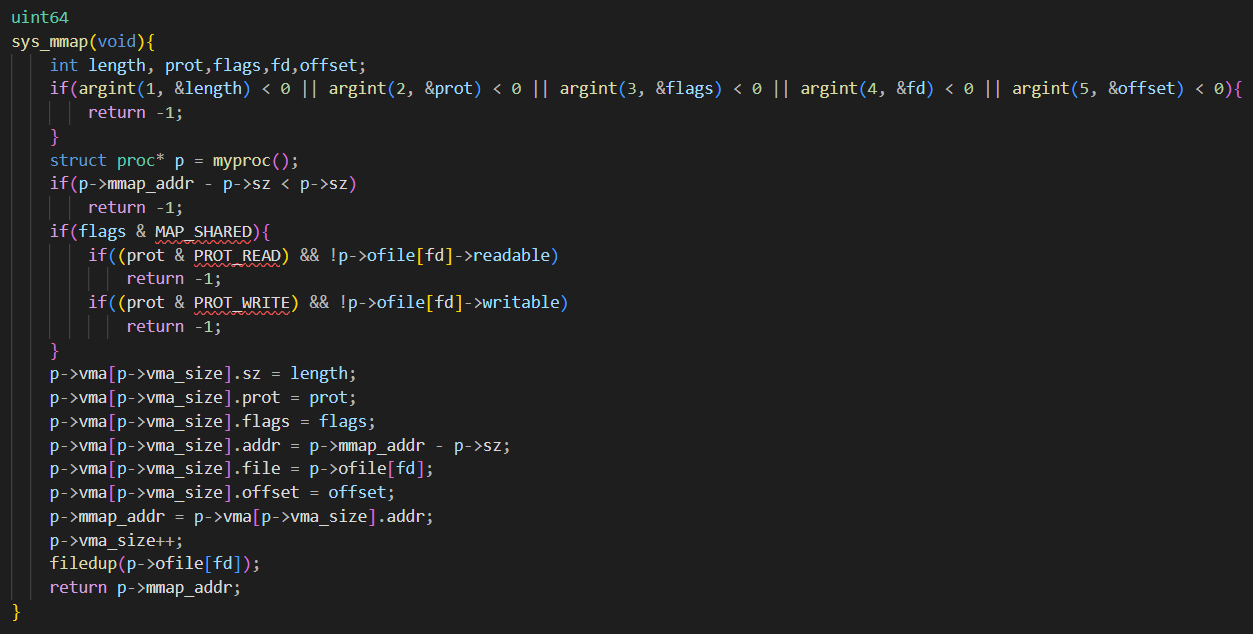
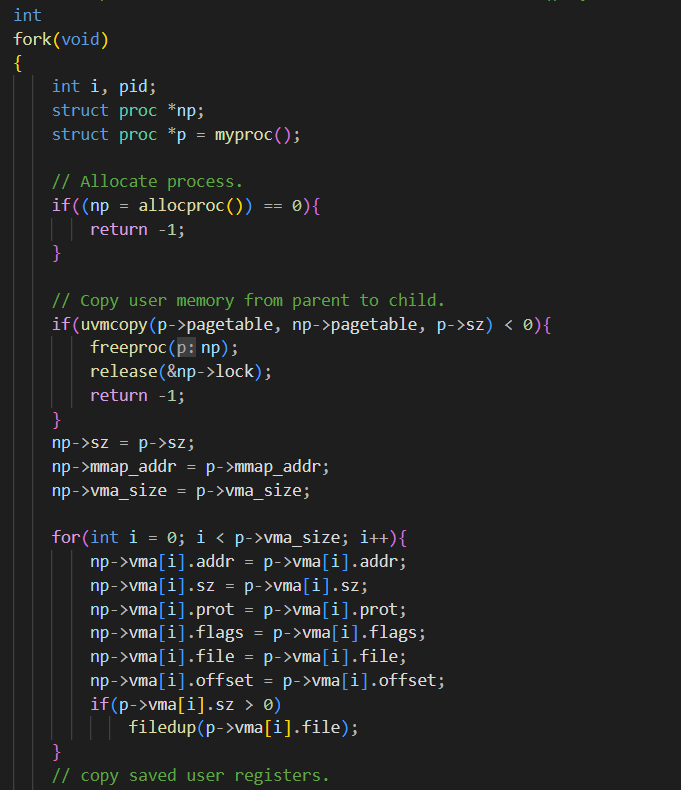
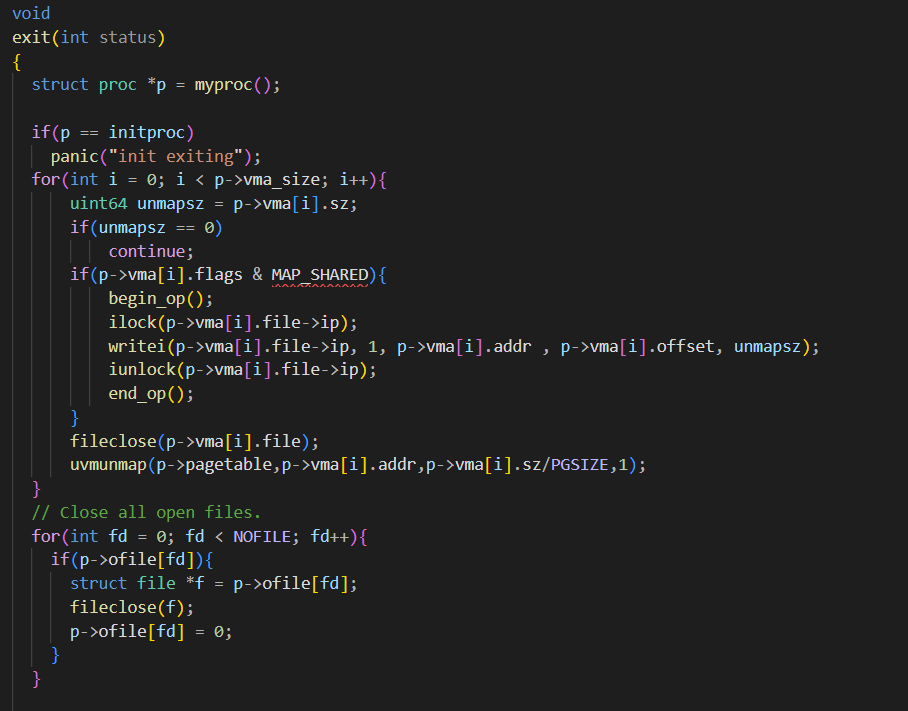
虚拟地址是虚拟内存的地址，虚拟内存的提出是为了解决多进程系统中同时运行多个程序的内存空间问题。直接使用物理地址的系统进程之间的地址空间不能隔离，并且可以同时运行的进程数量收到内存空间的限制。为了解决这个问题， 通过对进程地址空间的划分（分段或是分页），人们提出了一种在不使用某一部分进程数据时，就将其调到磁盘的内存管理策略，虚拟内存的技术也由此诞生。使用虚拟内存技术的操作系统中，进程地址空间都很大，并且允许不同进程使用相同的虚拟地址。在寻址时再对这些虚拟地址进行翻译转化。

mmap 允许用户将文件映射到进程的地址空间内，实现了文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文

件磁盘上。通过 mmap 的文件操作会更加高效，因为常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝；而 mmap 操控文件只需要从磁盘到用户主存的一次数据拷贝过程。

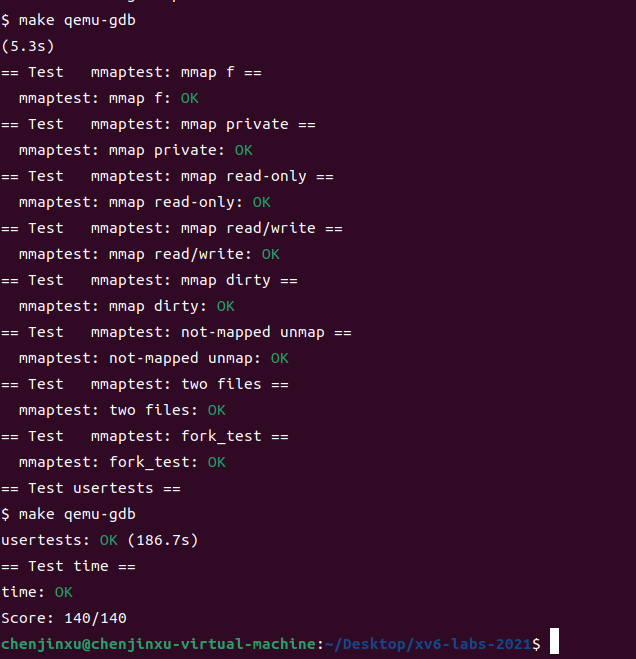
mmap 会记录映射的文件，并将文件信息填充到对应进程中。它并不会直接打开映射文件或是创建物理内存映射文件，而是在访问内存产生缺页中断时，再将文件内容懒加载到内存中。系统调用会在进程的地址空间中寻找一个并未使用的区域来映射文件，并将其指针填入进程记录的文件虚拟地址区域，供下次使用。

### 实现 mmap 系统调用

1. 首先要将存储映射缓冲区存放在用户进程空间的最顶端，即紧邻 TRAMFRAME 页的高虚拟内存部分。首先，我们需要为进程控制块 proc 增加新的元素，用于记录 mmap 的调用情况：
2. 在 sys\_mmap 系统调用中，我们在进程控制块中填写与 vma 有关的信息：
3. 然后我们需要对 exit 和 fork 进行更改，使得其对存储映射缓冲区进行合理的操作，在 exit 中，我们对存储映射缓存区的所有内容进行类 munmap 操作。在这里将对进程有关 vma 的情况操作在 freeproc 中集中处理，在 exit 中，我们对存储映射缓存区的所有内容进行类 munmap 操作。在这里将对进程有关 vma 的情况操作在 freeproc 中集中处理：

### 10.3.3 运行测试

运行 make grade，实验通过。



至此，该实验全部完成。

## 实验小结

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。这次实验的内容较多，覆盖了 xv6 系统中虚拟内存、文件地址、懒加载等知识点。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。

实验中需要调用的系统函数众多，这也是在实现两个系统调用时比较难以理解的部分。例如在映射一个页面时需要调用 mappages，而在取消映射关系时则会调用 uvmunmap 函数。此外，该实验涉及到了许多关于文件的操作，都让阅读了很多相关的源码和 xv6-book 相应的章节。

这个实验让我印象最深刻的地方在于对于页面懒加载的实现以及vma 数据结构的设计。懒加载这种方式在实验 5（Lazy allocation）和实验 6（Copy on write）中都有涉及，中心思想是不能一次性加载、拷贝所有内存，因为大多时候用不到。这种方式的泛用性印证了懒加载思想的重要地位。而在设计 vma 数据结构时，除了记录地址、长度、权限、文件等描述中提醒的字段之外，还记录了该内存映射地址是否正在使用、文件的共享标记这两个必要的字段。提出这两个字段是开始编写 mmap 系统调用时发现实现中有缺漏或是困难，才发现有字段缺失。

本次试验做起来阻力较大，但是收获也很多。我认为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。

项目代码github链接：[RidenShogunei/MIT-2021: 暑假小学期实验 (github.com)](https://github.com/RidenShogunei/MIT-2021)