## LAB5

这个实验开始进入实现文件系统。

首先先注释掉两处地方，然后验证原先的primes、forktree和pingpong都能正常运行。

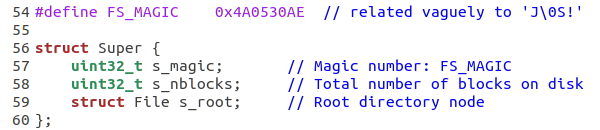
接下来取消注释，开始lab5。

这个简化的文件系统和UNIX文件系统的一个重要区别是不包含i节点，而是直接把文件的元数据存储在文件目录项中。（本来是文件目录项只包含文件名和指向i节点的指针，目的是支持硬链接）。

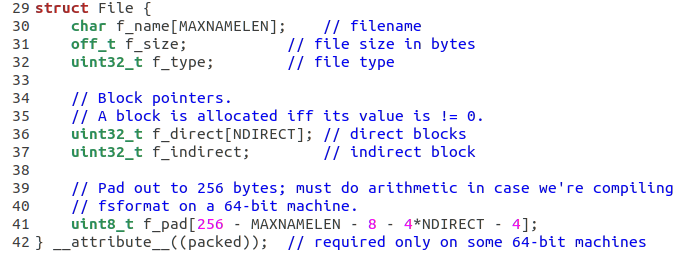
该文件系统也支持对目录文件的read操作，所以可以直接用read来实现ls命令，而不像在教学期间的实验课上要用各种系统调用来获取一个目录的信息。（这好处是便于编程，但坏处是系统难以支持多种文件系统，因为在不同的文件系统下同一个应用程序难以保持兼容性）

区分sector（扇区）和block（块）的概念，前者是硬件（硬盘）的属性，后者是操作系统的属性。实验中扇区大小是512字节，理论上块大小必须是扇区大小的整数倍，所以JOS的块大小是4096字节，即和页大小4KB一致，便于管理。UNIX xv6操作系统的文件系统块大小为512B，和扇区大小保持一致。注意xv6和JOS的区别。

超级块的数据结构在inc/fs.h中可以查看，实验中默认磁盘块1号是超级块（0号是引导块）。



可以看到，根目录文件的内容本身就放在超级块里，而不是存放指针。



文件元数据项的大小保证是256B，因为最后用了f\_pad数组来填充。这样一个块（4KB）可以有16个元数据项。

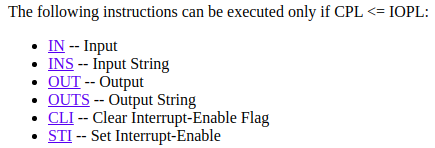
JOS使用10个直接索引块和1个间接索引块，因此最多支持1034块大小文件（略大于4MB）。

这个实验不要求实现整个完整的文件系统，而只需要实现一些关键部分。因此，对于各种已经提供的代码和接口一定要熟悉，要耐心花时间看代码。

IDE接口（Integrated Drive Electronics）是电子集成驱动器，是把“硬盘控制器”与“盘体”集成在一起的硬盘驱动器。把盘体与控制器集成在一起减少硬盘接口的电缆数目与长度，数据传输的可靠性得到增强，硬盘制造起来变得更容易的技术。

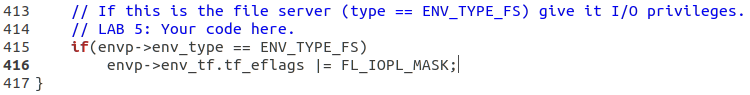
#### Exercise 1

这里终于出现了FL\_IOPL\_MASK标志位



查看x86手册可知，IOPL设置了能够当前能够执行I/O操作所需的特权，所以设置成IOPL=3的话就表示用户态也可以执行I/O操作。

当然，在之前的实验中，因为不清楚这一点，所以把所有的环境都加上了FL\_IOPL\_MASK，在这里需要去掉，然后通过检查环境的类型来配置。



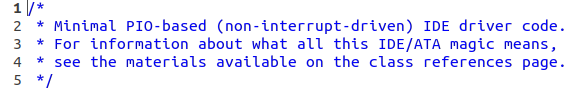


现在能够通过fs i/o测试了。但其实在之前的那种错误实现下，也能够通过测试，只不过那种情况所有的用户态环境都能够进行i/o操作。

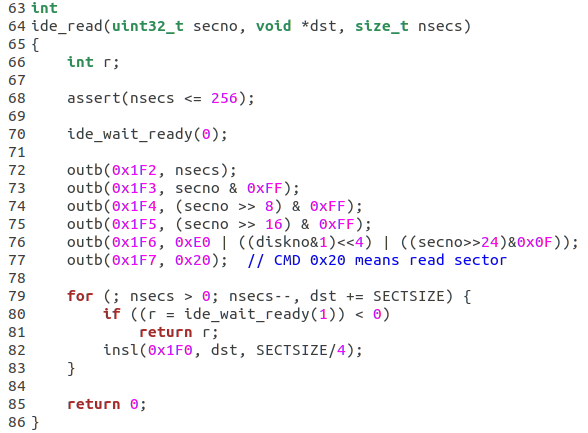
#### Exercise 2

在fs/bc.c里面实现了buffer cache

Fs/ide.c里面实现了从磁盘读取数据的接口，先熟悉一下。



在ide.c里面提到了class reference page，可以去找一下。

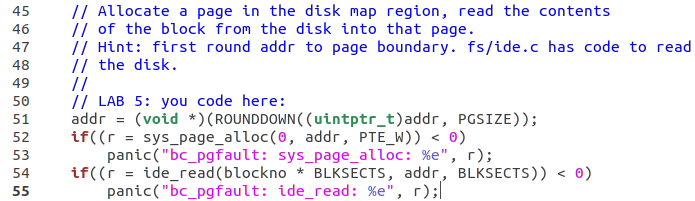


上面这段代码就是用于读取硬盘数据的，其实在lab 1里面已经分析过了，大概理解在做什么。

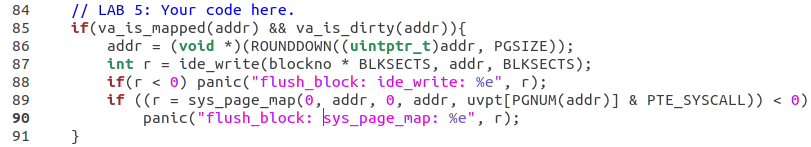
需要注意的是一旦硬盘处于ready状态，就能从0x01f0端口连续读取一个扇区大小的数据。Insl指令是如何从大小为8 bit的寄存器中直接读取32位数据，以及如何连续读取数据，这其实有些迷惑。可以这么认为，端口只是x86给OS提供的抽象接口，并不是一个端口就对对应一个寄存器，物理上可能对应多个寄存器。至于如何读取数据，这其实是x86硬件的底层实现决定的，对于上层的OS软件来说只需要编写相应的指令即可。

所以，这个exercise要注意的地方也就是块号到扇区号的转换，理论上应该是8倍的关系（已经提供了宏BLKSECTS）。

先实现bc\_pgfault，注意对返回值作检查。



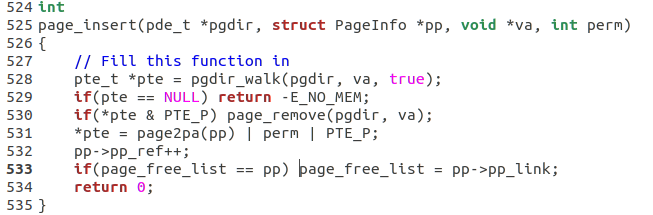
然后完成flush\_block函数



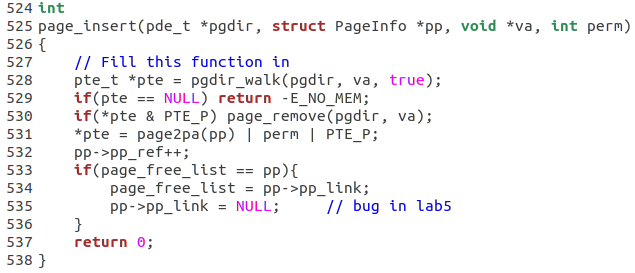
结果出现了pp->pp\_link不是NULL的panic

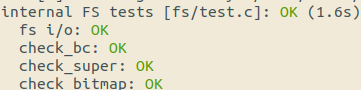


经过检查，发现是当年lab2就留下的坑

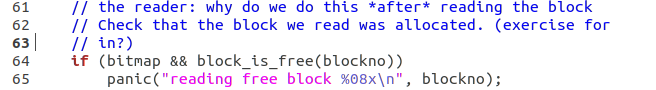


其实这个地方发现pp位于page\_free\_list上面之后，应当在page\_free\_list上去除pp。不应当仅仅把page\_free\_list，修改，还需要把pp->pp\_link的值也修改成NULL！

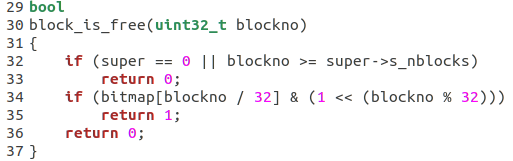




这样就能够通过测试了



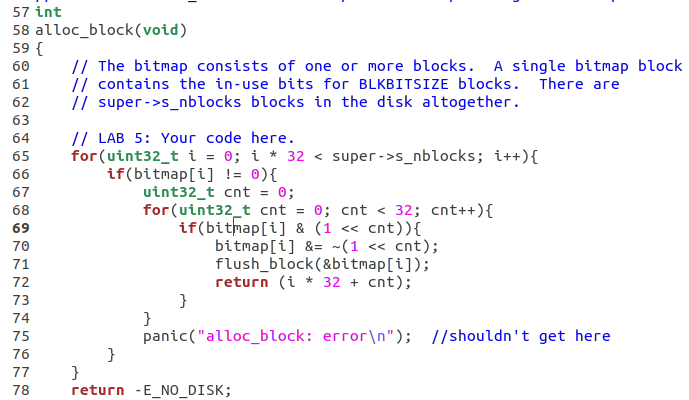
这里有一个问题，为什么要读入之后再检查。其实就是要考虑到bitmap本身也是存放在硬盘上的，因此这个缺页处理函数处理的可能就是bitmap本身的缺页。如果是这种情况的话，缺页处理函数先企图获取bitmap数组的值，然后造成缺页中断，中断处理函数又先企图获取bitmap数组的值，这样就会在缺页嵌套中断中死循环，直到用户异常栈耗尽。



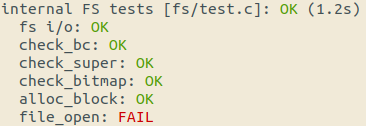
实验中的位图使用int型数组表示，所以要找到具体的一块对应的位，要先除以32，然后再与上偏移。

#### Exercise 3

实现block\_alloc，这里不清楚能不能用x & -x的方法来计算，因为感觉这样只能计算出对应位置的幂，还需要取log才能得到结果，不如直接移位运算。



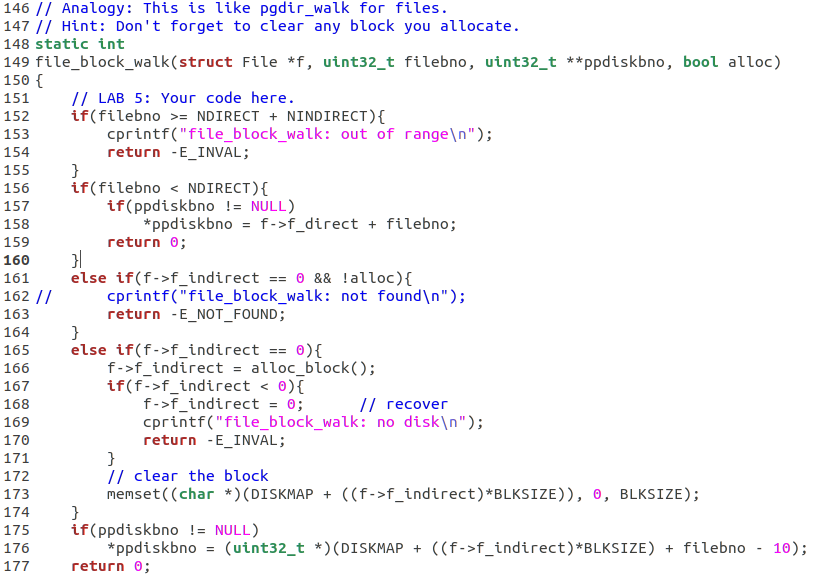
现在能通过alloc\_block的测试了



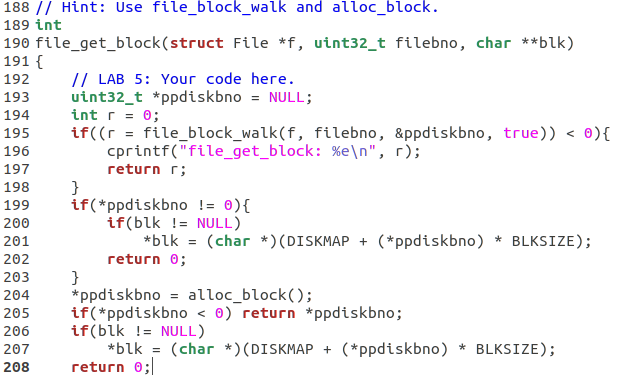
#### Exercise 4

这个练习要求先通读fs/fs.c，了解每个函数的功能，然后再实现尚未完成的函数。

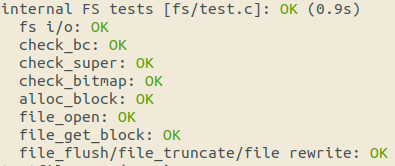
在file\_block\_walk上面的注释里面有提到clear any block you allocate，不清楚是不是用memset清0，应该是清0，否则的话原来的数据其实都无效的，但用户不知道。新增的页是所谓的二级索引表，里面都是0的话其实就表示没有数据（因为块号0是无效的，表示boot引导区）



接下来实现file\_get\_block，只要对文件系统概念清晰，完成这个是不难的。

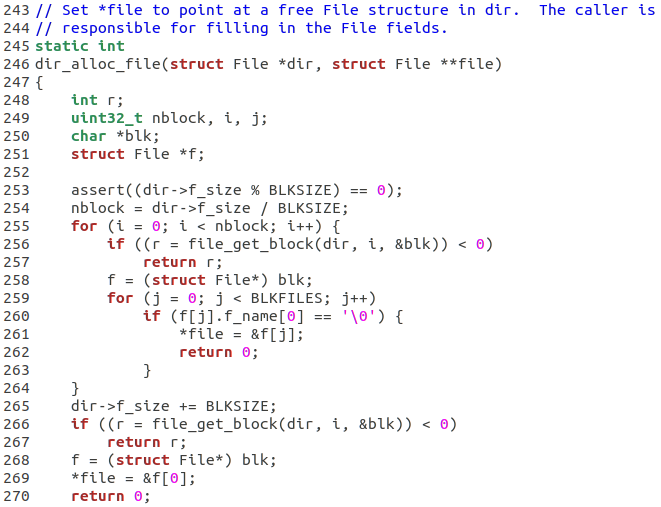


很轻松就通过了测试。



Dir\_lookup函数，就是在dir目录中寻找与字符串name匹配的文件结构，比较好理解。

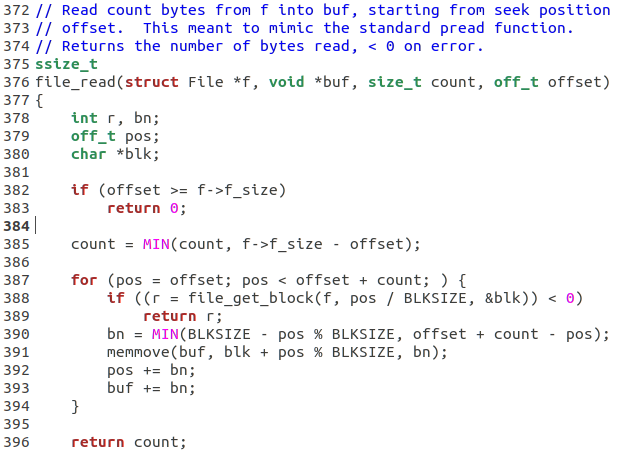
下面的dir\_alloc\_file函数有一点要注意，它是在dir中寻找空闲文件结构，这个空闲的意思就是指文件名字符串为空，就叫空闲，而不是说整个File结构全0。所以分配了空间结构之后，调用者一定要自己配置相关的数据项，否则原来的数据是无效的，会造成混乱。



Walk\_path函数比较复杂，但其实呢就是解析路径字符串，然后寻找对应的文件。如果能找到文件，就正常返回。如果只找到了文件所在的目录，但这个目录里却没找到文件，那就目录要记录。否则，就返回没找到。

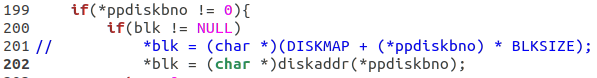
File\_create函数调用之后，需要调用者再额外进行配置，例如文件类型、大小之类的。

File\_read实现了pread标准接口，这里一方面保证没轮循环复制的内存是按页对齐的，又保证了不会超过文件的大小，也不会越界，还是比较精妙的。



File\_wrtie也是一样的，但是其内部没有flush，所以应该调用者需要注意flush。

还有要注意，获取一个块的虚拟地址可以用diskaddr函数来完成。

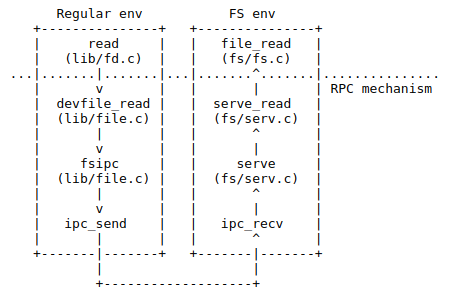


所以上面的代码还可以进一步简化

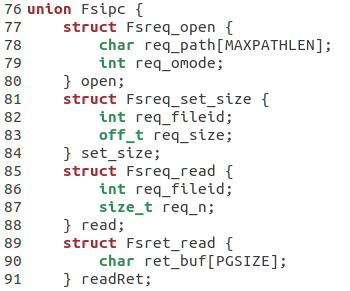
还有注意这个实验中出现了一些static\_assert的使用，可以正好学习一下。

#### RPC

接下来需要掌握RPC（远程调用方式，remove procedure call）的概念。它最终需要使用进程间通信来实现。这里可以清楚地认识到，文件系统本身是一个进程，只不过它是由操作系统自行建立的，对I/O有较高权限的进程。普通进程请求文件系统服务，必须使用进程间通信的方式实现。

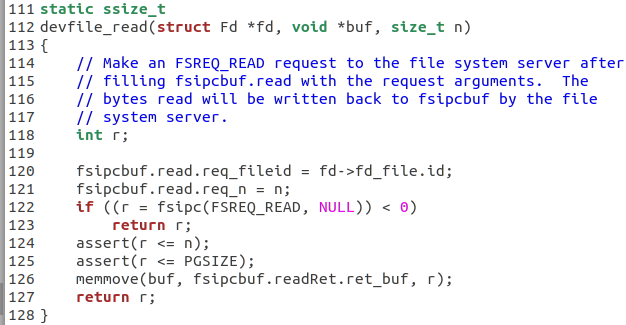


这个调用非常的精妙！首先，普通用户进程有一个全局枚举变量，枚举类型的部分如下。

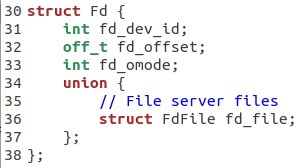


当普通进程需要向文件系统发起请求时，就把请求号作为ipc的一个参数，然后附带传送的共享页就是上面这个枚举体。枚举体保证占一个页的大小。

例如普通进程需要读取一个文件的数据，那就提供文件描述符和需要读取的字节数，然后通过ipc发给文件系统，文件系统对其进行响应，就是把对应的页读取出来，存入共享的这个枚举体对应的内存中，然后返回成功读取的字节数。

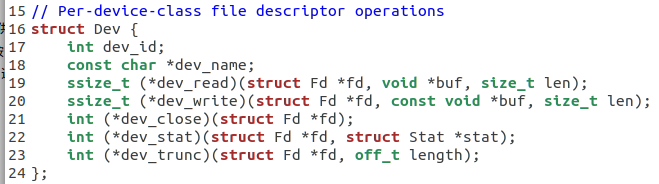


对于用户来说，在顶层首先需要调用open函数，提供路径字符串path和打开模式mode。查看fd.c可知，每个文件描述符结构虽然不大，但是按页对齐存放，就是为了将文件描述符作为共享页参数传递给文件系统，让文件系统修改文件描述符。这里要注意区分文件句柄和fd\_file，文件句柄是一个整数，表示的是该文件对于普通用户进程来说的句柄（即相对于0xD0000000的页偏移量），而fd\_file是文件系统给这个文件的编号，是便于文件系统找到这个文件。



文件描述符的第一项是设备id，也就是表明该文件属于何种设备。

设备数据类型定义在fd.h头文件中，每个结构描述了一种设备。

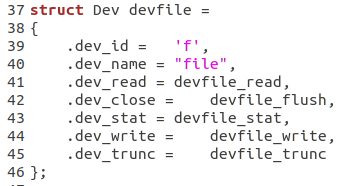


系统中一共有哪些设备呢？在同一文件里，可以看到相应的定义。



这里，devfile就表示磁盘设备，devcons表示控制台终端设备，devpipe表示管道。

然后，磁盘设备相关的操作都在哪里呢？就在file.c里面。

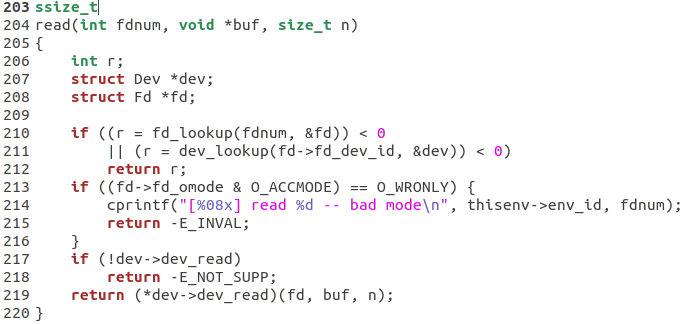


所以，用户可以先用open函数，向文件系统请求服务。然后文件系统如果根据路径能找到相应的文件的话，就把文件的类型写到open函数提供的文件描述符上。

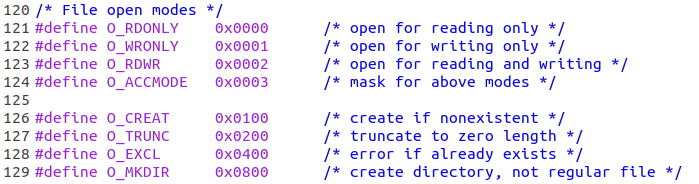
用户成功地open了一个文件之后，就可以用这个文件描述符来作相应的操作了。为什么要打开了才能进行操作呢？例如read函数，它需要根据设备的类型来调用不同的函数，所以首先是根据文件描述符来获取需要打开的文件的类型以及相关的函数。

例如read，最后是返回(\*dev->dev\_read)(fd, buf, n)，其实就是调用了该设备的函数。

结合上面给出的devfile的定义可知，这里就是调用了devfile\_read函数。



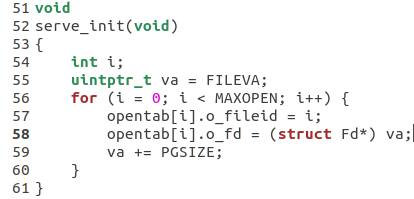
在lib.h里面定义了相关的模式常量



用户发来的请求页存放在这里

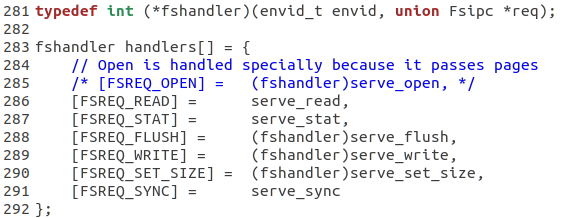


经过serve\_init函数的初始化可知，fileid就是对于文件系统的opentab表的索引，而每个表项的fd就是从0xD0000000开始的1024个页中的某一页的地址。初始化时它们是按序排列的。

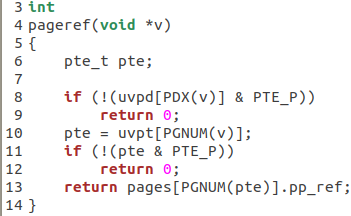


对于请求处理函数，同样建立一个函数指针数组，存放这些函数的首地址。

对于open函数单独处理，因为该函数需要返回的页不是fsipc的union页。



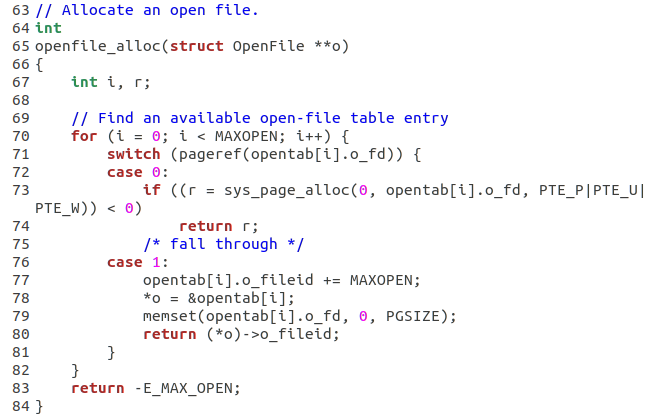
在lib/pageref.c里面，有一个孤零零的函数。这个函数返回当前用户对于某个虚拟地址这一页的引用数。



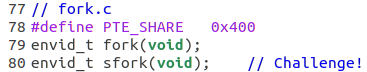
这样，openfile\_alloc函数就可以看懂了。Case 0就表示这一页已经分配出去了，case 1表示这一页分配出去了一个。这里发现，如果引用计数小于等于1的情况，该文件都属于尚未打开。看来得好好理解一下。

引用计数为1，说明这一页要么只在文件系统中映射了，要么只在普通用户映射了。无论哪种情况，都说明这是一个无效的打开文件句柄，所以相当于这个结构没用，可以再分配。

然后注意那一行注释/\* fall through \*/，表示case 0 中如果r==0的话，就会继续执行case 1的代码。因此，正常情况下的o\_fileid值一定大于MAXOPEN，且初始时会清空o\_fd。

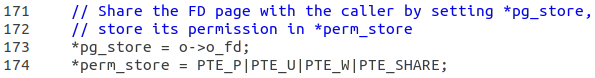


还有一件事



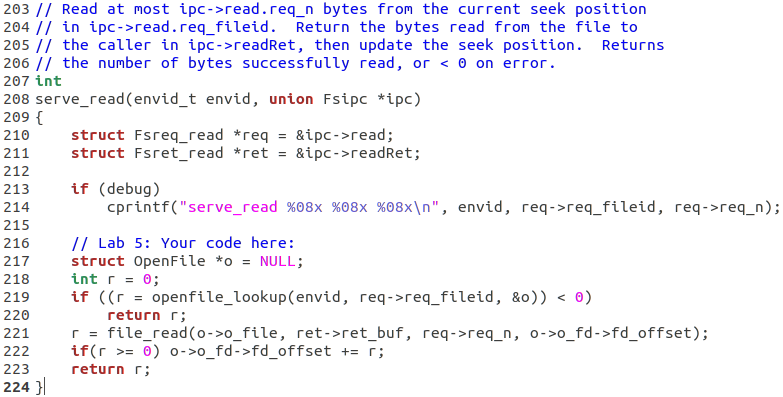
在lib.h里面找到了PTE\_SHARE的定义，这也是PTE中AVAIL3位中的一位，怀疑可能和lab4中的challenge6的实现有关系。先看一下当前的实验中是如何使用这个PTE\_SHARE的。

在serve\_open的最后，添加了PTE\_SHARE这个权限，但暂时不知道在哪里使用它。

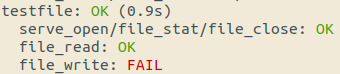


#### Exercise 5

读取内容成功之后，要记得修改文件描述符中的offset。可见文件描述符的另一个作用就是维护offset偏移量，使得用户不需要关心文件的偏移而能够顺序地读取文件中的内容。

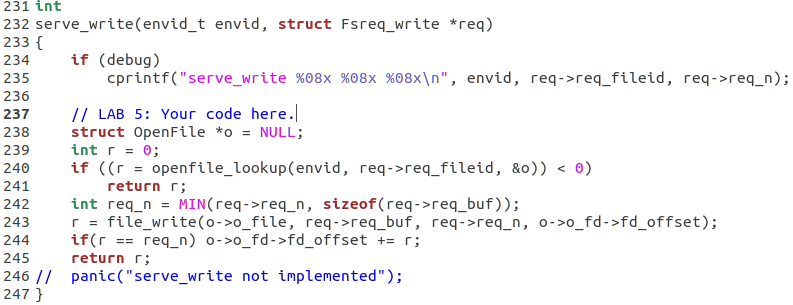


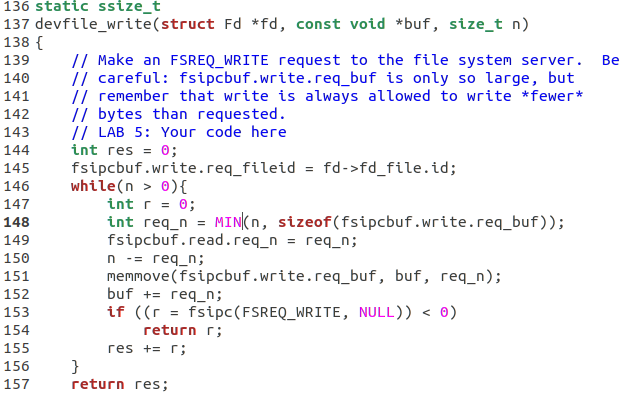
通过了file\_read的测试，然后接下来完成serve\_write函数。



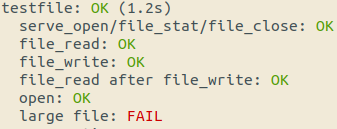
#### Exercise 6

要注意别忘了完成devfile\_write函数。

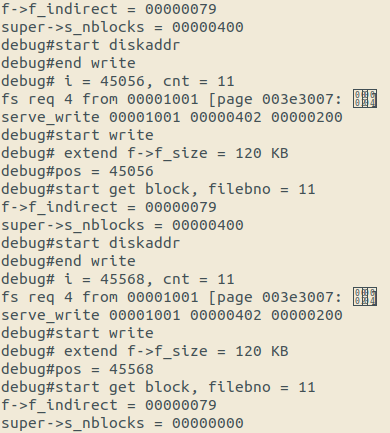




但是没有90分，因为large file还不行。



非常诡异



超级块的数据竟然被修改了！

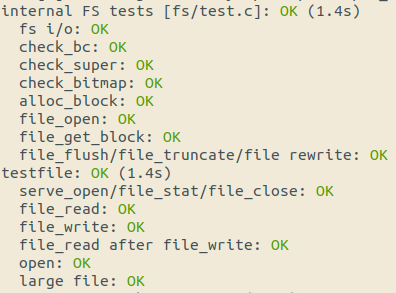
原因就在于file\_block\_walk的这个地方，diskaddr的返回值是void \*，所以空指针和整数相加减值就是加减相应的值。应该把void \*先转换成uint32\_t \*，然后再进行加减，这样结果才是正确的！所以说指针很容易造成错误，特别是再加上强制类型转换的时候。



应当修改为先强制类型转换



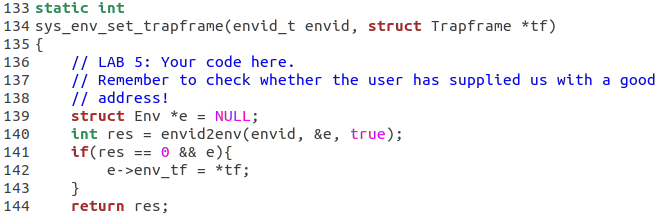
但还是只有85分，这就不清楚了，照理已经通过了这部分的测试了。



先继续完成下面的练习吧。

#### Exercise 7

只需要完成sys\_env\_set\_trapframe就可以，感觉没啥内容。本来spawn.c里面应该有不少函数需要实现，但是在2016年的实验里面已经由老师给出了，可能是考虑到太难了。



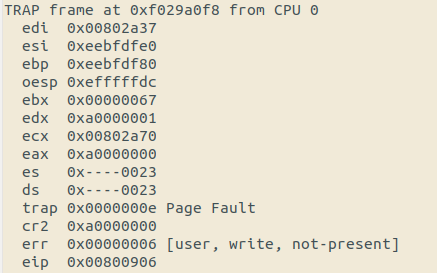


#### Exercise 8

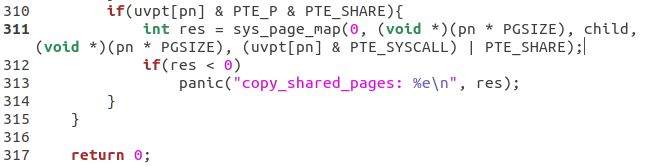
Fork和spawn的进程之间需要共享文件描述符。

遇到写问题，无法通过testpteshare

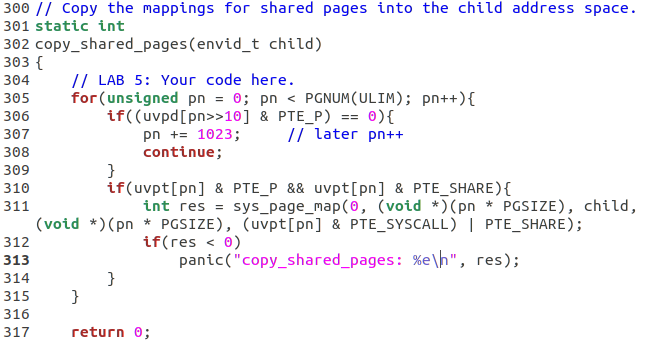
运行的话，会出现user, write, not-present



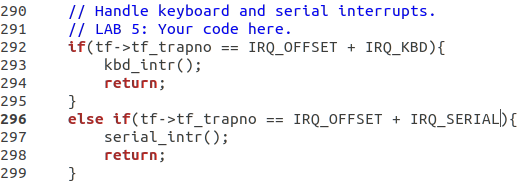
因为这里有问题，uvpt[pn] & PTE\_P & PTE\_SHARE显然是恒0 的值。



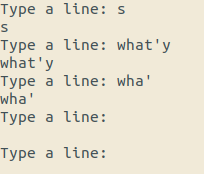
改成下面这样



#### Exercise 9

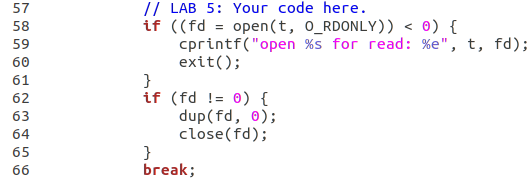


不知道怎么让它停下来。



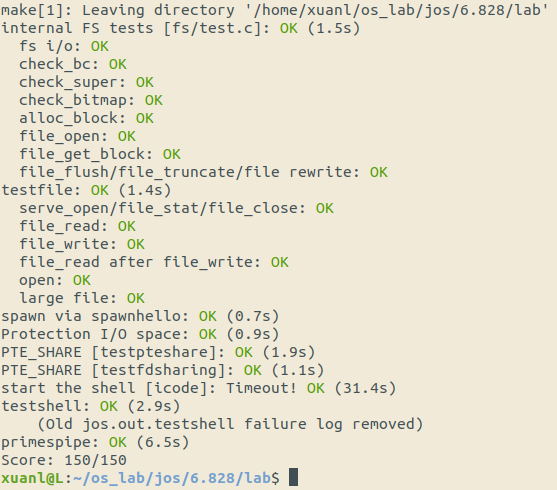
#### Exercise 10

这个shell的实现比较复杂，但要写的部分比较简单，基本复制 > 的实现就行了。但要注意的是mode需要改一下，里面不能有CREATE和TRUNCK，只能是O\_RDONLY，这和重定向输出是不同的，输出的话对应的文件需要情况，而输入的话文件不能修改。



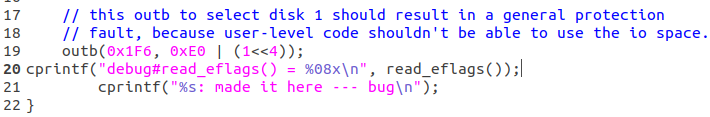
最后终于通过了所有的测试，太不容易了。

这样的话，就基本完成了操作系统课设了。之后的lab6不在课设要求范围内。



#### 疑惑1

这个faultio.c的用户测试很奇怪，加上那句debug的cprintf之后就能通过测试，但不加的话，自己测试没问题，但是无法通过测试脚本的测试。



反正只要会输出和eflags相关的值的话，就能通过测试，否则不能，这就非常奇怪了。这可能和makefile文件有关系，但一时也没法深究了。