JOS 实验五 实验记录

作者：卓达城

邮箱：zhuodc@qq.com

（w f lfj and 1874）

开始本实验之前请使用svn把代码合并。

本实验编程技巧不多，不想lab4，用了很多很多技巧，好像是在卖弄什么似的，但是代码量大和逻辑复杂，总的来说分成四大块，分别是服务器模块，用户模块还有底层模块和通讯模块。

以下先按照底层模块，服务器模块（包含通讯模块），用户模块（包含通讯模块），**四个模块如何合作**的顺序详细叙述。

底层模块（fs.c)

磁盘结构：

Directory or file

bitmap

super 1

super 0

super 0 这里我们不用管，是bootloader。

super 1 记录着磁盘的基本信息（魔数和总块数）和根目录文件。

bitmap 记录这那些块可用那些不可用（已用或者未用）。

File结构可以表示文件或者目录，在jos里面目录是记录文件和目录信息的文件。

如果一个文件是目录的话，那么这个文件里面记录的是File结构数组。

文件系统里面有两个概念，一个是实际块号，一个是文件块号，实际块号就是块在磁盘的位置，也可以在服务器进程中转化为虚拟地址。文件块号是相对文件而言的，文件的第一块块号为0

磁盘

File结构解释：

File 这个里面又放着

1024个块号，其中

前10个没有用。

一个block是4k，一个文件最大可以放1024个block，所以一个文件最大是4M

........

direct 0

name

direct 1

direct 2

...........

direct 2

...........

indirect

...........

。

。

如果文件的大小小于10个block，那么文件块号就放在direct记录的块里面，如果文件大于10个block，那么文件的10个块以上的文件块号就放在indirect里面，indirect block是指向一个块的，块里面放在块号的信息，这里要注意的是indirect block块的前十个块号是没有用的，为什么作者要这样做，这是因为要保证文件最大是4M，jos文件最大只能是4M。

服务器模块：

先看线性地址（虚拟地址）的分布情况

0xffffffff

0xf0000000 KERNBASE

opentab是一个OpenFile数组，具体在内存那个位置不知道，由编译器决定，编译器应该足够聪明不至于让它跟上面提到的地方重复了。

堆栈等其它信息

0.5G

4K

4K

............

0xD0000000

opentab[i].o\_fd

(FILEVA)

这里为何要4k一个呢？

原因很简单，因为这里要用作页映射，是进程通信需要的。

bitmap

这一段是用来映射3G磁盘的，每从磁盘读一块磁盘块，服务器进程就会把它映射到这一段空间中。（4K为单位）然后映射给用户模块。

super

bootloader

block2对应的地址（4K)

block1对应的地址（4K)

block0对应的地址（4K)

0x10000000

4k

(DISKMAP)

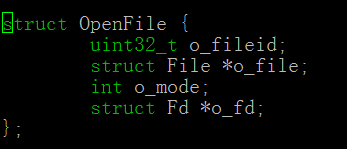
0x0ffff000

这里是用来接收客户端进程发来的请求的

(REQVA)

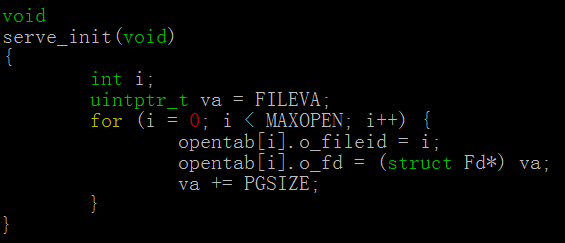
0x00000000

这里最关键的结构式OpenFile结构



其中o\_file是用来操作底层文件系统的，

o\_fd是用来传到客户进程的，客户进程根据o\_fd的信息操作文件。



这段代码就是把o\_fd映射到FILEVA以上的地址，以便进程之间的通讯，具体见服务器模块的图。

用户模块：

0xffffffff 线性地址（物理地址）

0xf0000000 KERNBASE

其它信息

这里是用来存储映射过来的文件的，每个文件最大为4M。（就是映射地址）一共4M\*32=128M

4M

4M

.

.

0xD000000 FILEBASE

.........

这里是用来存储Fd结构的。这里一开始没有分配地址空间，只是在用的时候才分配。通过使用检查页表是否存在的方式判断是否被分配或者使用。

4K

4M

4K

4K

FDTABLE

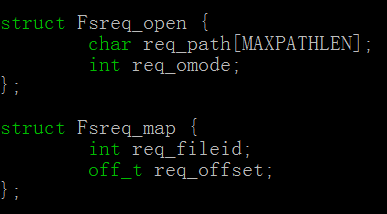
0x0000000

这里最关键的结构式Fd

用户进程通过Fd保存的信息实现相关的文件操作。（例如dev是保存相关操作函数的指针）

通信模块：

在inc/fs.h中定义了一些通信的数据结构。



通过这些数据结构，服务器进程对底层文件系统进行相应的操作。

发送函数如下：



第一个参数是操作类型，第二个是要传递的参数，第三个就是fd（服务器进程把openfile的o\_fd映射到这里），第四个就是权限。

**四个模块的合作**

**大体上是：用户需要文件操作--->用户进程向服务器进程发出请求--->服务器进程操作底层文件模块--->服务器进程向用户进程发出已经完成操作的信息，并把相关信息传递回去--->用户进程获得相关文件。**

**底层文件用file结构操作，用户用fd结构，服务器用OpenFile把两者联系起来。**

先从客户端进程开始（icode）

**客户端：**

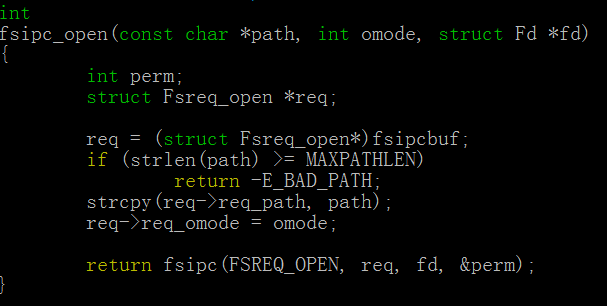
第一个函数（icode.c)



这个函数的作用就是打开一个文件并获得句柄。（file.c)



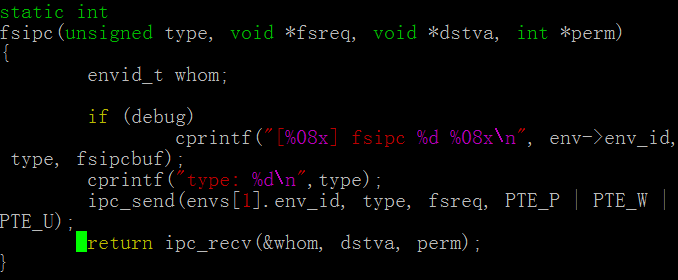
分配一个句柄(fsip.c)



方框里面的内容是设置req，serve将根据req这个参数对底层文件系统进行操作。

然后发送到服务器端。这里FSREQ\_OPEN是告诉服务器进程要进行什么样的操作，req就是这个操作需要的参数，fd是地址，服务器进程进行完相关的处理之后就会把内容映射到fd的地址中，fd具体在哪里请看客户端模块的图。

现在进入fsipc函数，代码如下（fsipc.c)：

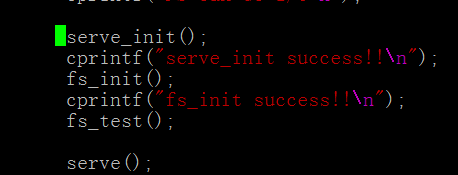


具体操作是向服务器进程发送该发送的数据,注意，这里的ipc\_send会循环发送，知道服务器进程响应为止。

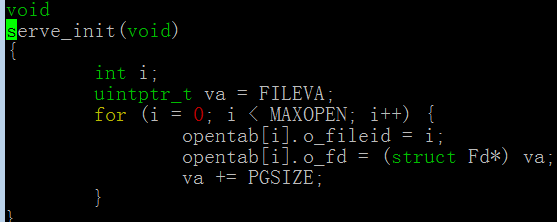
然后进入阻塞状态，ipc\_recv函数。

**现在进入服务器模块：**

第一个函数：serve\_init() (serv.c)



进入serve\_init函数，此函数的功能是初始化服务器进程。



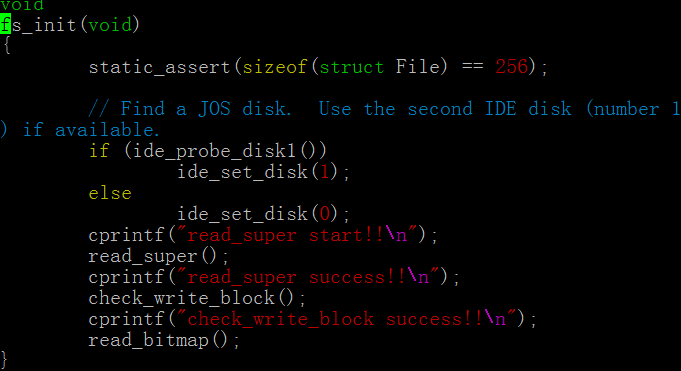
初始化opentab，并把o\_fd映射到FILEVA以上的地址，具体详见服务器模块图。

然后进入底层文件系统初始化函数

**底层模块：**



这个函数的具体功能是读出super块很bitmap

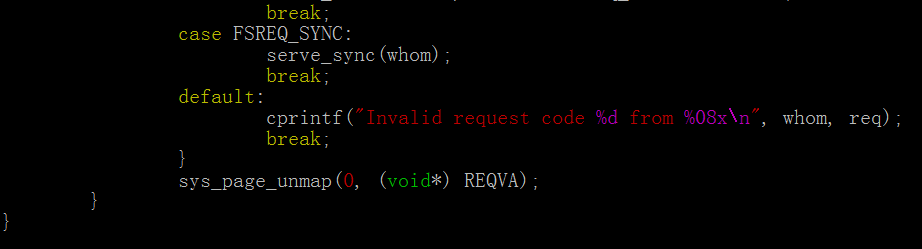


然后是fs\_test函数，这个函数可以删除，这里就不讨论了。

**退出底层模块：**

现在进入serve（）函数





这个函数是一个无限循环。

当运行这个函数的时候该进程会进入阻塞状态，然后等待用户模块向它发送信息，然后进行相关处理。（第一块代码是使进程进入阻塞状态，然后进入switch，准备进行相应的处理）

按照上面用户模块发来的信息，应该是进入serve\_open：



现在进入serve\_open函数（serv.c)

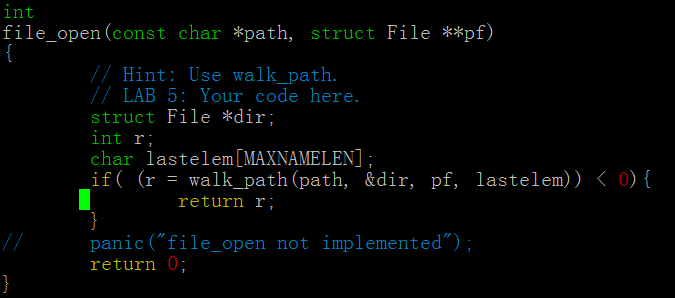


先分配一个openfile，openfile到底是什么请看服务器端模块图。

现在进入**底层模块**：（fs.c)



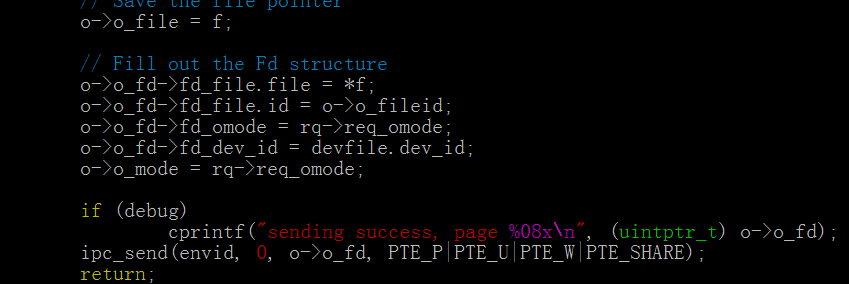
进入file\_open函数



根据路径从磁盘中读出file。并放在pf中。

退出**底层模块**。

进入**服务器模块**：

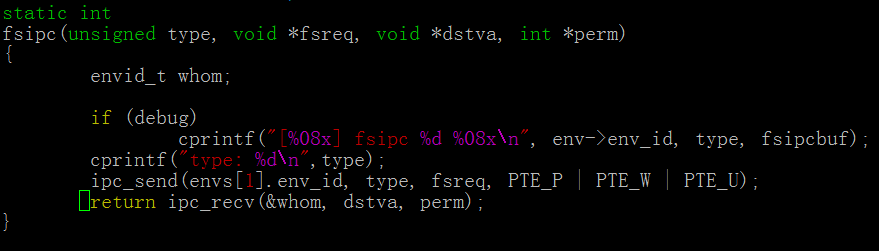


设置Openfile，然后发送回用户模块。然后服务器模块返回serve函数，并进入阻塞状态，等待用户模块向它发送消息。

退出**服务器模块**

进入**用户模块**：

fsipc函数（fsipc.c)



ipc\_recv函数返回，用户进程继续运行。

至此一次文件操作完成。

有些文件操作需要用到dev这个结构，这个结构就是把用户模块的文件基本操作函数指针放在里面。但是具体原理是一样的。

到此为止，文件系统的框架应该已经了解。下面进行深入理解。

开始做这个实验之前，我们必须先读相应的代码。

名词：

服务器进程指的是映射磁盘块的进程。

文件：fs.c

char\* diskaddr(uint32\_t blockno)

把块号转换成服务器进程内存相对应的虚拟地址。

bool va\_is\_mapped(void \*va)

判断虚拟地址va在服务器进程中是否被映射。

bool block\_is\_mapped(uint32\_t blockno)

判断blockno指定的block是否在服务器进程中被映射。

bool va\_is\_dirty(void \*va)

判断va的内容是否被修改了

bool block\_is\_dirty(uint32\_t blockno)

判断blockno对应的内容（这些内容必须应经映射到服务器进程）是否被修改。

int map\_block(uint32\_t blockno)

实际上就是分配一个页面个blockno对应的地址

static int read\_block(uint32\_t blockno, char \*\*blk)

先映射一个页到服务器进程的地址空间，然后把blockno对应的块读到内存中。

void write\_block(uint32\_t blockno)

判断blockno对应的内存页是不是脏页，如果不是，直接返回，如果是，写到磁盘中。

void unmap\_block(uint32\_t blockno)

从服务器进程中把一个页的映射删除，但是这个页必须是干净的（相对于脏页）或者是bitmap里面标记为free的（被删除了）。

bool block\_is\_free(uint32\_t blockno)

判断磁盘块对应的bitmap位是否为0,1表示已经用了，0表示没有用。

int alloc\_block\_num(void)

分配一个磁盘块，返回磁盘块号，然后把bitmap写回磁盘。

int alloc\_block(void)

分配一个磁盘块，然后把磁盘块映射到相应地址。

void read\_super(void)

把超级块读到服务器进程的地址中。

void read\_bitmap(void)

把bitmap读到服务器进程的地址空间中。

void check\_write\_block(void)

这个函数是用来检测write\_block函数是否正常。通过把块1复制到块0，然后修改块1，写入磁盘，然后读出，然后比较，然后把块0的内容放回去，然后super =块 1

void fs\_init(void)

先检查有没有磁盘1(还有一个是0），如果有设置diskno为1，这个全局变量在ide\_read里面会用到，用来判断写到那一个磁盘上。

读超级块。

检查写函数

读bitmap

int file\_block\_walk(struct File \*f, uint32\_t filebno, uint32\_t \*\*ppdiskbno, bool alloc)

寻找文件f中块号为filebno的块（文件中第一块为0），ppdiskbno是指向f\_direct数组的元素或者f\_indirect指向的块的元素。

alloc是没有块的时候是否分配一个新块。

\*ppdiskbno是实际块号的地址

这里有一句代码：

因为super记录着文件系统的根目录的文件的。

int file\_map\_block(struct File \*f, uint32\_t filebno, uint32\_t \*diskbno, bool alloc)

给文件f的第filebno个块映射。

alloc是否需要新建一个块

diskbno返回的实际块号。

int file\_clear\_block(struct File \*f, uint32\_t filebno)

清除f的一个块。

int file\_get\_block(struct File \*f, uint32\_t filebno, char \*\*blk)

从磁盘读一个块，并且映射到对应的服务器进程空间。

\*blk为虚拟内存地址。

如果没有就分配一个块。

int file\_dirty(struct File \*f, off\_t offset)

使offset/BLKSIZE个块为脏读，通过自己写自己，然后cpu自动在内存中标记为脏。

int dir\_lookup(struct File \*dir, const char \*name, struct File \*\*file)

在dir中找到为name的文件，并把它放到\*file中

int dir\_alloc\_file(struct File \*dir, struct File \*\*file)

在指定的目录dir下分配一个file

static inline const char\* skip\_slash(const char \*p)

略过 /

static int walk\_path(const char \*path, struct File \*\*pdir, struct File \*\*pf, char \*lastelem)

根据路径遍历文件系统，如果有文件\*pf = 要找的文件

如果没有，把剩下的路径复制到lastelem中。

int file\_create(const char \*path, struct File \*\*pf)

建立一个path文件

int file\_open(const char \*path, struct File \*\*pf)

打开一个文件。

static void file\_truncate\_blocks(struct File \*f, off\_t newsize)

把文件大小变为newsize

int file\_set\_size(struct File \*f, off\_t newsize)

把文件f设置成newsize大小

void file\_flush(struct File \*f)

把文件的内容同步到磁盘上。这里可以通过file\_map\_block查找实际块号。

void fs\_sync(void)

同步硬盘上所有数据。

void file\_close(struct File \*f)

关闭一个文件

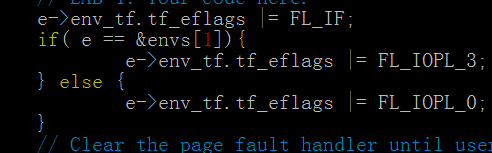
int file\_remove(const char \*path)

删除一个文件

文件系统相对前面的虚拟内存系统来说是相对简单的，下面开始实验：

EX1:

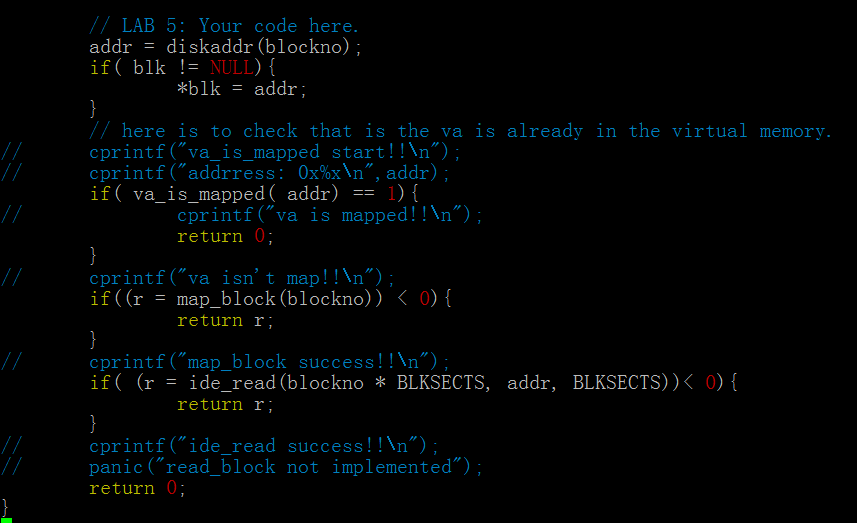
在env.c 的env\_alloc中添加以下代码：



这里涉及一些cpu的硬件特性，eflag的FL\_IOPL位，FL\_IOPL表明当前段（任务）的特权级为多少才可以访问IO，如果是3，就表明当前段（任务）的特权级为3,2,1都可以访问I/O，如果为0，就表明当前段的特权级必须为0才能访问I/O。

EX2:

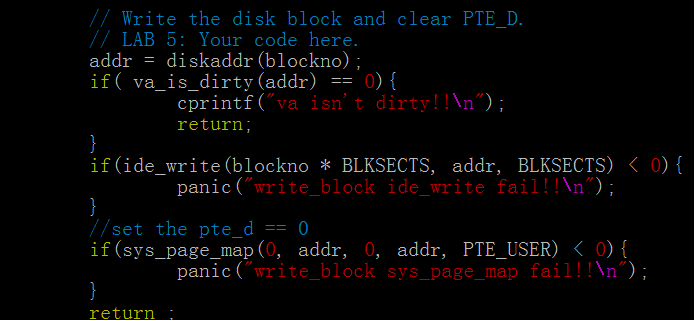
read\_block代码如下：



如果addr已经在内存中有映射，函数返回，不用任何操作。如果没有，根据blockno分配内存页，并进行影射，然后把磁盘上的数据映射到内存上。blockno对应那一个内存地址，根据服务器模块图理解。

根据上面的函数解释看这里的代码应该很容易懂的，这里不再累述。

write\_block



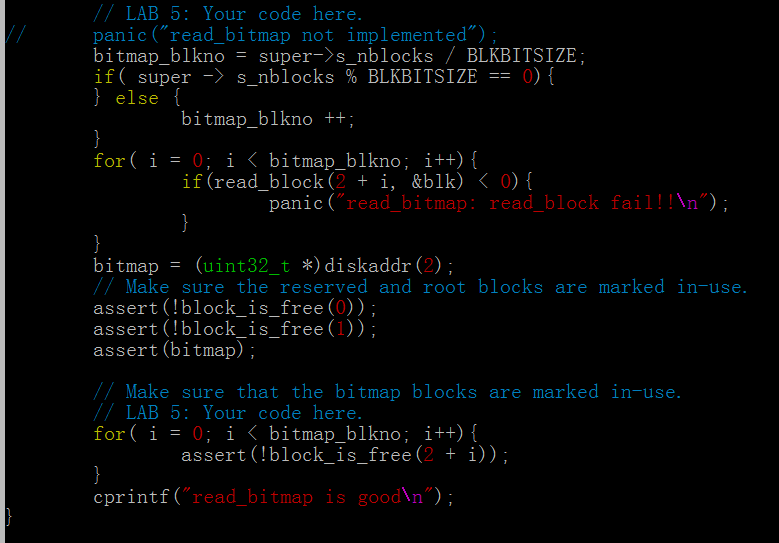
先判断是否是脏页，如果是，就写入，并清除PTE\_D位，如果不是直接返回。

这里PTE\_D位是什么时候设置的呢？这又是cpu做的事情，当一个内存也被访问的时候，cpu就会对其进行标记，把PTE\_D位置1，但是cpu从来不会主动把PTE\_D位置0，所以必须由程序设定。

这里可以用sys\_page\_map来清除PTE\_D，PTE\_D是脏读位。

EX3:

read\_bitmap

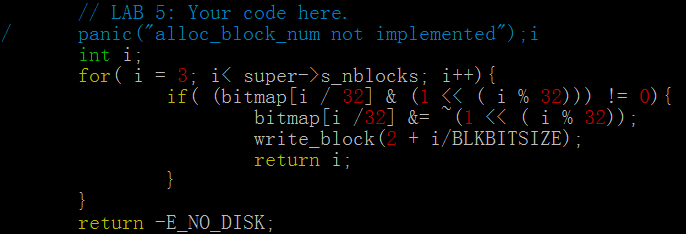


第一块代码是用来求出一共需要多少块才能存储bitmap。

第二块代码是用来检查存储bitmap的块是否标记为已用，在bitmap里面0表示没有用，1表示已经用了。

EX4:

alloc\_block\_num

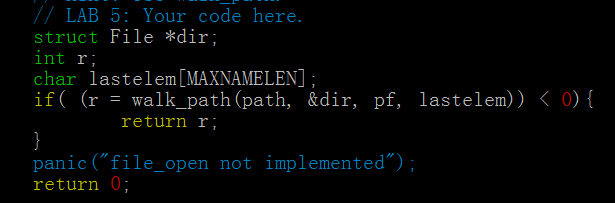


这个函数是从bitmap中找一个空闲的块，然后分配出去，但是这里要注意，一旦bitmap被修改了，必须马上写入磁盘，保证同步。

这里bitmap的每一位表示一个块，所以要用到位操作，具体为何这样写，自己模仿程序走一下就会知道。

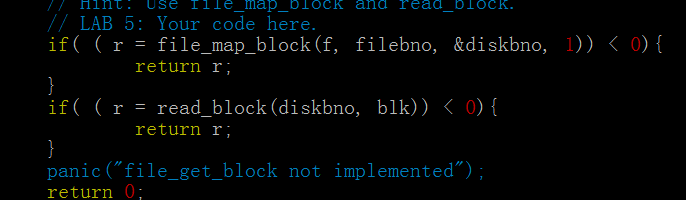
EX5:

file\_open



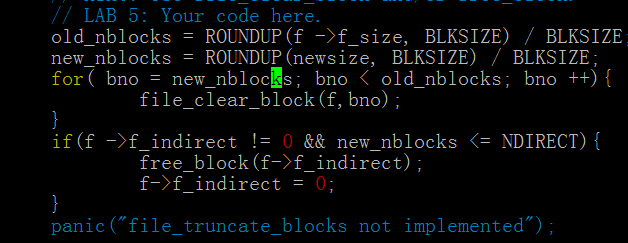
打开一个文件。

file\_get\_block



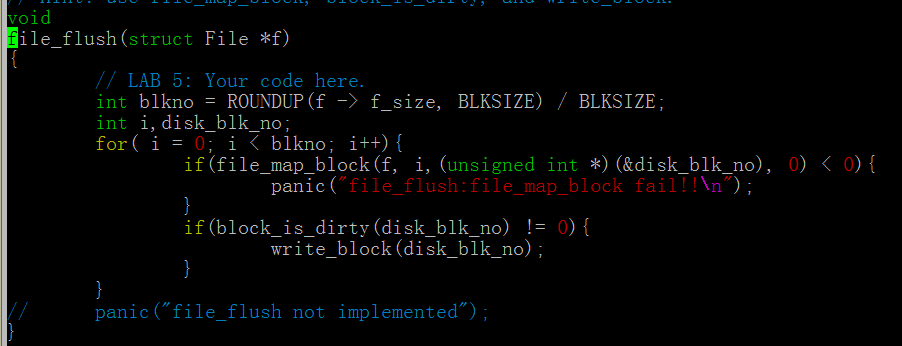
取得一个block

file\_truncate\_blocks：



把文件的大小设置为新大小。

file\_flush



把文件同步到磁盘上，就是写入磁盘。

以上为底层文件系统的实现。

---------------------------------好累啊！！-----------------------------------------------------------^\_^

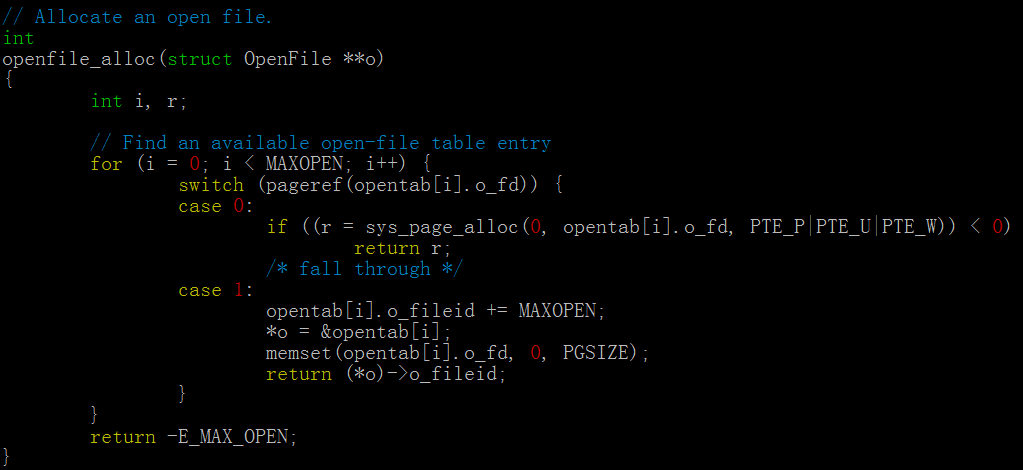
现在进入服务器进程

serv.c

void serve\_init(void)

这个函数初始化服务器，实际上就是初始化opentab和完成o\_fd的映射。

int openfile\_alloc（struct OpenFile \*\*o）



这个函数的无论什么情况都会执行case 1的内容，但是case 0的内容不一定会执行。

当什么时候会执行case 0呢，就是没有分配o\_fd页的时候，o\_fd对应的地址在服务器模块的图中可以看到。

但是case 1又是怎么解释。

当分配一个页到o\_fd的时候，pp\_ref = 1，当它映射到用户进程的时候，pp\_ref = 2，当用户进程关闭或者关闭文件的时候，pp\_ref = 1，因为用户模式中的fd\_close中是会unmap页的。



但是在服务器进程中serve\_close是不会把page unmap的，也就是pp\_ref是不会等于0的。

所以当再次调用这个函数的时候，就不用重新分配页。这样可以提高速度，但是如果在前一分钟打开30个文件，在之后的时间只打开一个文件的话，会造成页的浪费。如果要解决这个问题的话可以在serve\_close中加上page unmap。

int openfile\_lookup(envid\_t envid, uint32\_t fileid, struct OpenFile \*\*po)

这个函数就是根据fileid寻找openfile。

void serve\_open(envid\_t envid, struct Fsreq\_open \*rq)

打开一个文件，并填充OpenFile中的信息，然后发回去。

void serve\_set\_size(envid\_t envid, struct Fsreq\_set\_size \*rq)

设置文件的大小

void serve\_map(envid\_t envid, struct Fsreq\_map \*rq)

把服务器进程的一个block映射到用户进程。

void serve\_close(envid\_t envid, struct Fsreq\_close \*rq)

关闭一个文件。

void serve\_remove(envid\_t envid, struct Fsreq\_remove \*rq)

删除一个文件。

void serve\_dirty(envid\_t envid, struct Fsreq\_dirty \*rq)

把一个页标记为脏页

void serve\_sync(envid\_t envid)

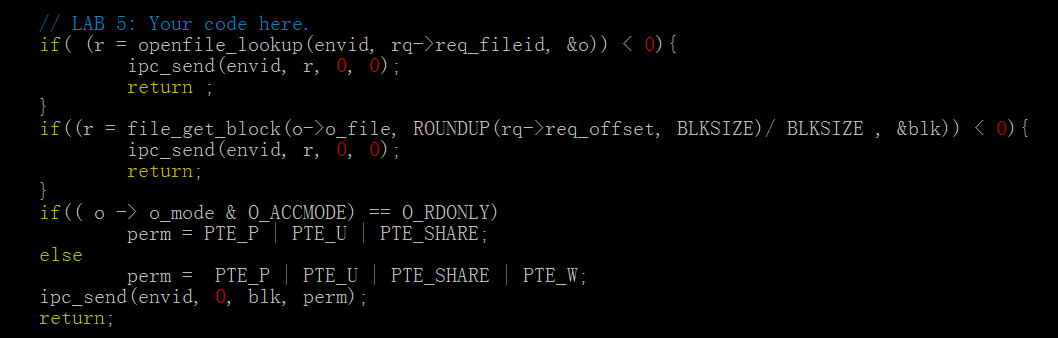
把所有文件同步到磁盘。

void serve(void)

服务器进程主要函数，用来分配用户进程发来的请求。

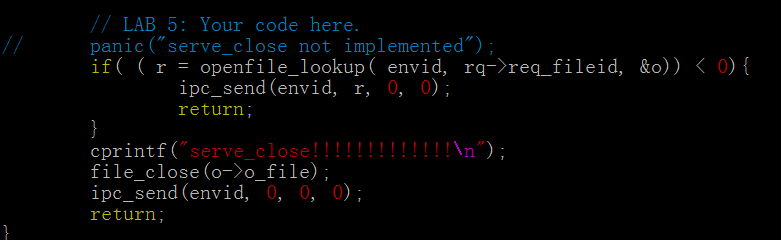
EX6 （serv.c)

void serve\_map(envid\_t envid, struct Fsreq\_map \*rq)



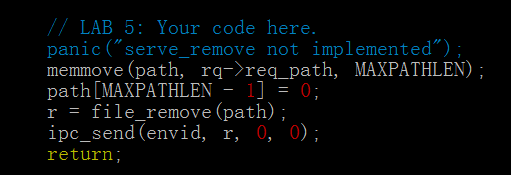
这个函数把一个block映射到用户进程。

void serve\_close(envid\_t envid, struct Fsreq\_close \*rq)



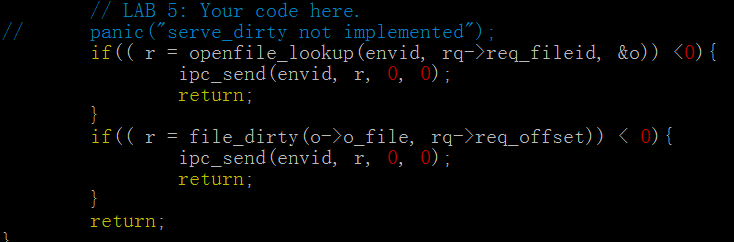
这个函数关闭一个文件。这里我们应该把所有页映射删除，但是为了提高效率，我们可以不做这一步。但是这样会造成物理页不够用。这里可以在fs.c中加入一个file\_unmap，然后把关闭的文件unmap掉，这样会好一点，但是考虑到jos就算在这里加入减少物理内存页的使用，也是无法解决它物理内存页分配完就会挂掉的问题，而且作者也没有要求，所以这里就不加了。

void serve\_remove(envid\_t envid, struct Fsreq\_remove \*rq)



这个函数的功能是删除一个文件。

void serve\_dirty(envid\_t envid, struct Fsreq\_dirty \*rq)



把一个页标记为脏页。

现在进入用户模块：（fd.c)

INDEX2DATA(i)这个宏是把fd装换成放数据的地方。具体情况用户模块图。

char\* fd2data(struct Fd \*fd)

得到对应fd的映射的数据的地址，具体请看用户模块图。

int fd2num(struct Fd \*fd)

fd转成索引号。

int fd\_alloc(struct Fd \*\*fd\_store)

分配一个fd（这里可以翻译成文件句柄吧）。

int fd\_lookup(int fdnum, struct Fd \*\*fd\_store)

寻找一个fd。

int fd\_close(struct Fd \*fd, bool must\_exist)

关闭一个文件句柄

int dev\_lookup(int dev\_id, struct Dev \*\*dev)

根据dev\_id寻找对应的dev，这里只有一个dev，dev里面放的是磁盘基本操作的函数的地址。

int close(int fdnum)

关闭一个句柄，注意这里有page\_unmap。

void close\_all(void)

关闭所有文件。

int dup(int oldfdnum, int newfdnum)

复制一个文件句柄，并把对应的内容也映射到新的文件句柄的相应的数据区(FD2DATA）。具体请看用户模块图。

ssize\_t read(int fdnum, void \*buf, size\_t n)

从文件中读数据到buf。

ssize\_t readn(int fdnum, void \*buf, size\_t n)

这个函数暂时没有用。

ssize\_t write(int fdnum, const void \*buf, size\_t n)

写入文件。

int seek(int fdnum, off\_t offset)

设置fd的offset

int ftruncate(int fdnum, off\_t newsize)

修改文件的大小。

int fstat(int fdnum, struct Stat \*stat)

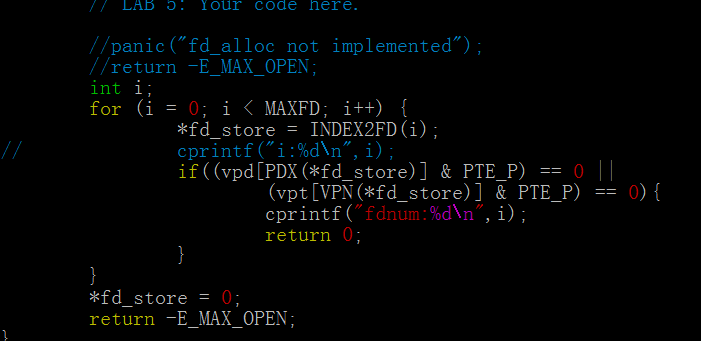
设置stat结构。

int stat(const char \*path, struct Stat \*stat)

设置stat结构。

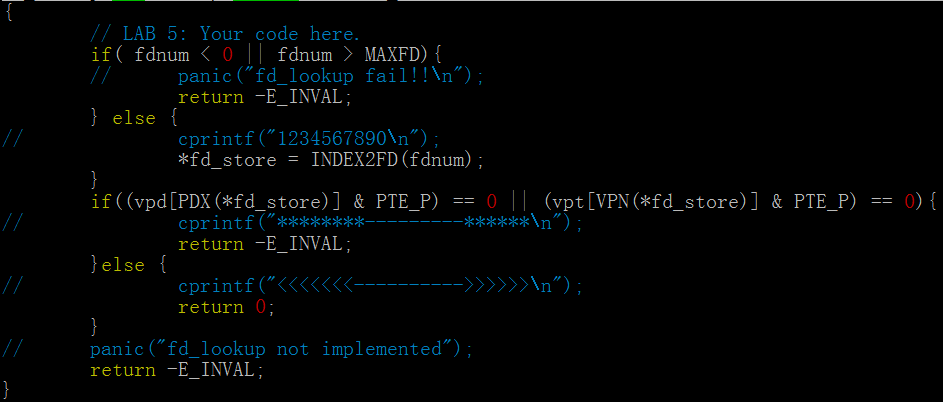
EX7 fd.c

int fd\_alloc(struct Fd \*\*fd\_store)



通过检查页是否分配来确定fd有没有被分配。

int fd\_lookup(int fdnum, struct Fd \*\*fd\_store)

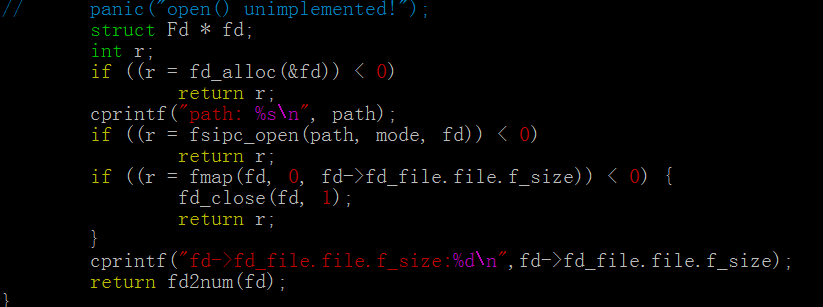


根据fdnum查找一个fd。

EX 8

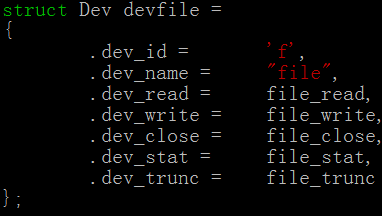
file.c

open函数



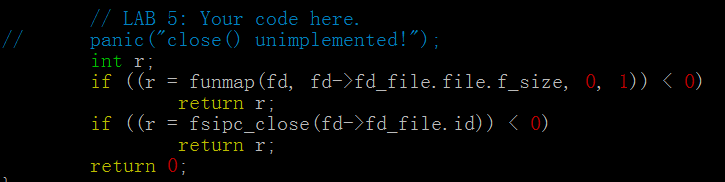
这里要注意啦，open的时候会把文件的内容都map到fd对应的数据区里面，具体请看用户模块图，文件最大为4M。

下面是dev结构，记录着基本操作函数的指针。



EX 9 file.c

static int file\_close(struct Fd \*fd)



这里会把fdunmap掉，跟serv.c是不一样的。然后fd对应数据的映射也会被unmap（）。

funmap会使相应文件的所有页变为脏，并且unmap掉。

然后fsipc\_close会令服务器把所有脏页写回磁盘。

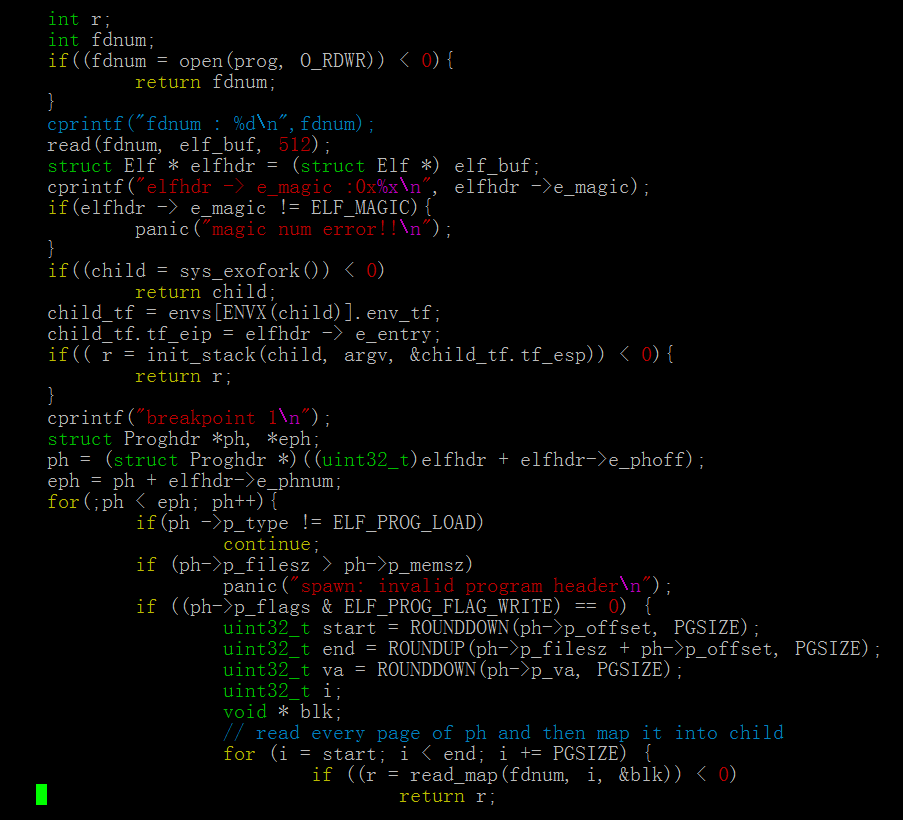
EX10

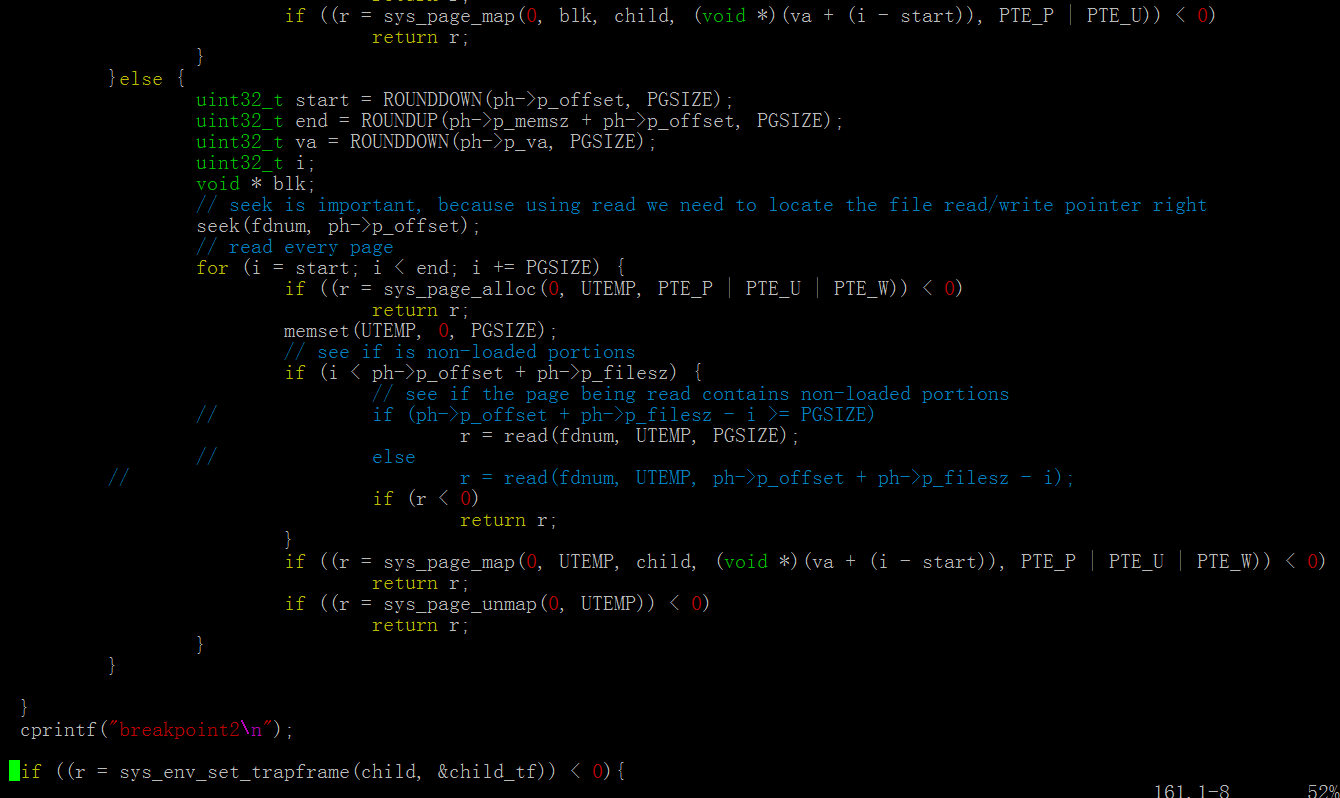
spawn函数

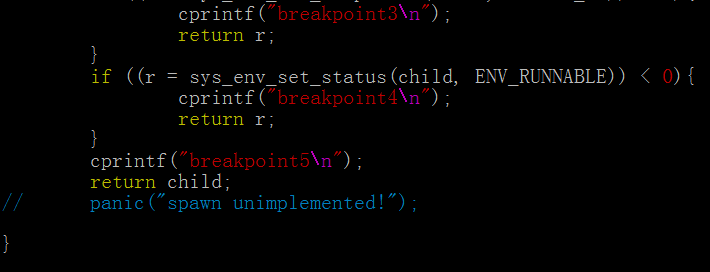
这个函数的作用是把程序从磁盘读出，并放到jos上运行。

加载的时候请注意：可写的段需要分配物理页，不可写的直接映射就可以了。

具体代码如下：



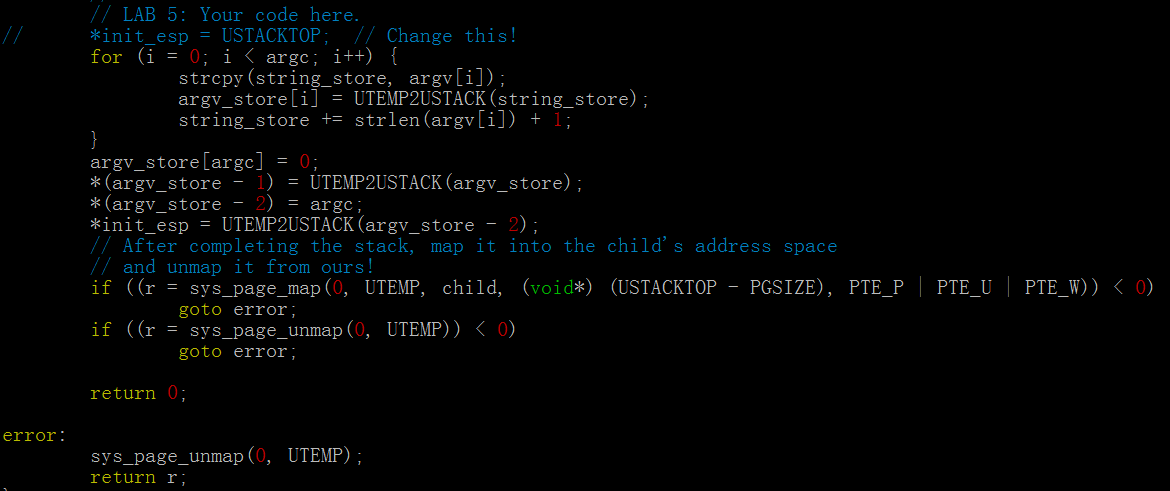




EX11

spawn.c

init\_stack函数



这个函数就是初始化我们运行的程序的堆栈。至于为什么要这样初始化，请了解编译器的内容，参数压栈是从右到左的。

初始化后的堆栈如下：

USTACKTOP

arc

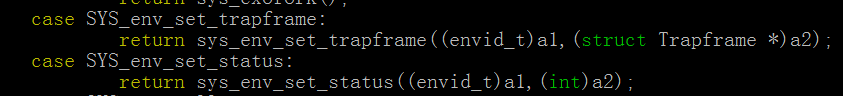
最右边的参数

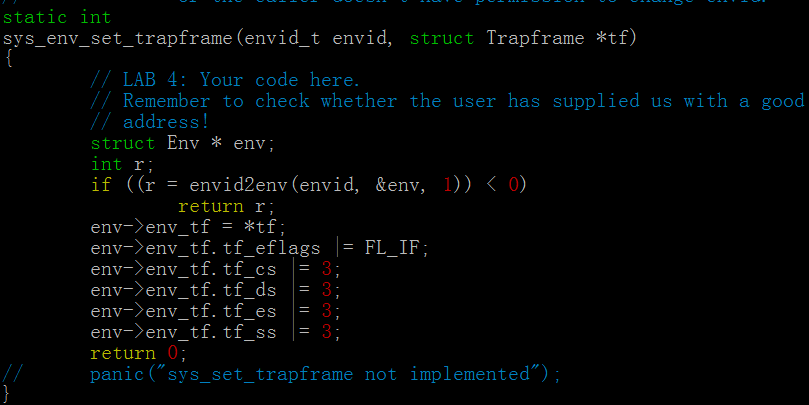
string\_store

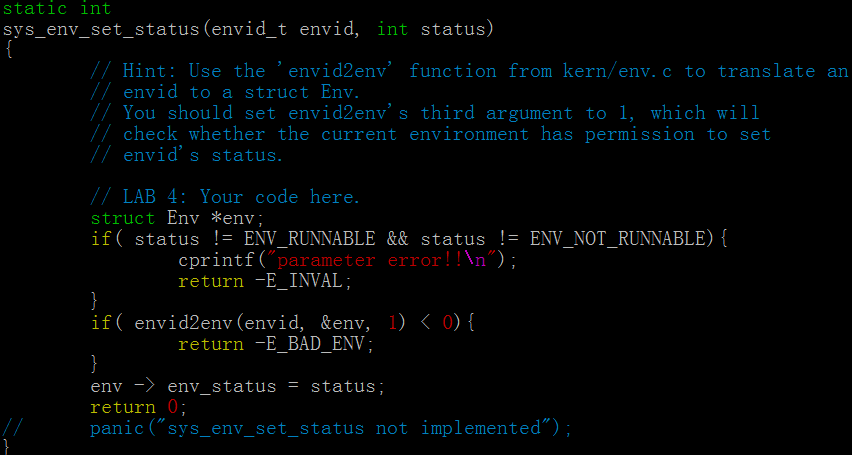
argv

完成上述代码以后还要加几个系统中断。

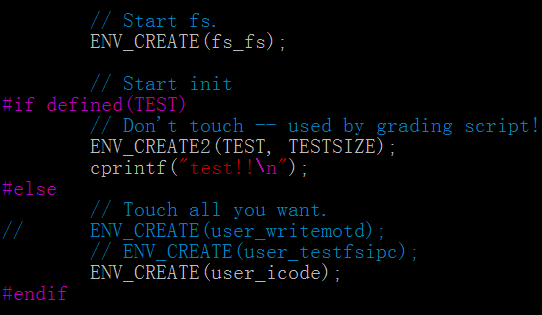
这里为何要这么做请了解前面的实验。







然后再init.c加上以下代码：



实验完成。