实验六 文件系统

马逸君 17300180070

前言

JOS的文件系统较UNIX操作系统中的真实文件系统简单，提供一些基础功能：创建、读写、删除文件。鉴于我们的文件系统只打算支持单用户，所以不包含文件所有权、权限。

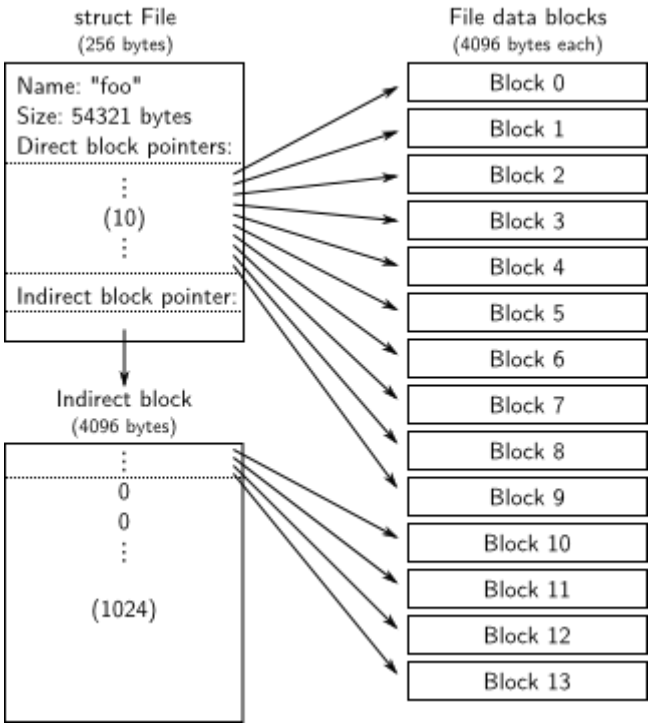
磁盘文件系统结构 鉴于我们的文件系统不打算支持硬链接，所以我们的文件系统将不使用I-node结构，而是在目录项中直接包含文件的元数据（这是因为这种情况下可以保证有且仅有一个目录项对应该文件）。文件系统隐藏底层文件块的散布，提供对文件随机读写的接口。我们的文件系统还支持让用户程序直接读取目录的元数据，这样，用户就可以自行进行目录扫描（例如实现一个ls程序）而不必通过特定的系统调用(system call)；这样做的缺点是会使得应用程序依赖于目录元数据的具体格式，从而很难在不修改（或者至少不重编译）应用程序的前提下修改文件系统的内部结构（所以现在大多数UNIX操作系统不鼓励这样做）。

扇区与块 块大小是扇区大小的整数倍，xv6中的块大小等于扇区大小，而大多数现代操作系统的块大小会更大，因为存储空间更便宜，而且在更大粒度上管理存储的效率更高。JOS中的扇区大小为512字节，我们的文件系统的块大小将设为4096字节，以匹配处理器的页大小。

超块 文件系统常在一些位于"非常好找的"位置的磁盘块（如第一块、最后一块）存放描述整个文件系统属性的元数据（如块大小、磁盘大小、找到根目录所需数据），这些磁盘块称为超块(superblock)。我们的文件系统有且仅有一个超块，它位于磁盘的块1（块0是boot sector和分区表）。

文件元信息 inc / fs.h中的struct File描述了我们的文件系统中的文件元数据的结构，包括文件的名称、大小、类型（常规文件或目录）以及指向文件块的指针。因为我们不使用I-node，因此元数据存储在目录项中。

如下图所示，struct File中的数组f_direct用于存储文件的前10个块的块号，称为直接块，可容纳大小不超过10 * 4096 = 40KB的文件。对大于40KB的文件，我们分配一个额外的磁盘块，称为间接块，以容纳4096/4 = 1024个额外的块号。我们的文件系统允许文件的大小最大为1034块，即4MB。为了支持更大的文件，真实文件系统通常支持双重和三重间接块。



目录文件与常规文件 struct File可以表示常规文件也可以表示目录，通过type字段来区分。文件系统以完全相同的方式管理常规文件和目录文件，除了它不会解释与常规文件关联的数据块的内容，而会将目录文件的内容解释为一系列描述目录中文件和子目录的File结构体。我们的文件系统中的超块包含一个File结构体（struct Super中的root字段），包含根目录的元数据。

1 磁盘访问

> i386_init通过将ENV_TYPE_FS类型传递给环境创建函数env_create来标识文件系统环境。在env.c中修改env_create，使其授予文件系统环境I/O权限，但不要将该权限授予其他环境。

env_create()

```
1 if (type == ENV_TYPE_FS) e->env_tf.tf_eflags |= FL_IOPL_3;
```

题面（英文原题）中已经说过，x86处理器使用EFLAGS寄存器中的IOPL位来确定是否允许保护模式代码执行特殊的设备I/O指令如我们的OUT指令，我们需要为文件系统环境赋予I/O特权来允许文件系统访问IDE磁盘寄存器。

不难猜想JOS应该为更改IOPL位预定义了一些常量。在终端中，我们执行如图所示的这样一条指令，在JOS目录下查找所有含"iopl"（大小写忽略）的字段，找到inc/mmu.h中定义的相关常量。

```
mayijun@ubuntu:~/oslab6$ ls * | awk -v a=$(pwd) '{print a"/"$0}' | grep -irn "io
pl"
kern/env.c:405: if (type == ENV_TYPE_FS) e->env_tf.tf_eflags |= FL_IOPL_3;
kern/syscall.c:129:// protection level 3 (CPL 3), interrupts enabled, and IOPL o
f 0.
inc/mmu.h:109:#define FL_IOPL_MASK      0x00003000      // I/O Privilege Level b
itmask
inc/mmu.h:110:#define FL_IOPL_0 0x00000000      //   IOPL == 0
inc/mmu.h:111:#define FL_IOPL_1 0x00001000      //   IOPL == 1
inc/mmu.h:112:#define FL_IOPL_2 0x00002000      //   IOPL == 2
inc/mmu.h:113:#define FL_IOPL_3 0x00003000      //   IOPL == 3
```

从i386手册[1]得知，in、ins、out、outs、cli、sti等特权级IO指令只有当CPL<=IOPL时才能执行。

8.3.1 I/O Privilege Level

Instructions that deal with I/O need to be restricted but also need to be executed by proce
(IOPL). The IOPL defines the privilege level needed to execute I/O-related instructions.

The following instructions can be executed only if CPL <= IOPL:

- IN -- Input
- INS -- Input String
- OUT -- Output
- OUTS -- Output String
- CLI -- Clear Interrupt-Enable Flag
- STI -- Set Interrupt-Enable

再结合题面中"全或无"的暗示和JOS lab3 [2]中关于用户环境寄存器的设定，就可以得到答案了。

问题

> 当从一个环境切换到另一个环境时，是否必须执行其他操作以确保正确保存和还原此I/O权限设置？为什么？

不需要。由[2]，切换到其他用户态环境时，系统将自动把上下文压栈，其中包括EFLAGS寄存器；而从其他用户态环境恢复时，系统弹出并恢复上下文，包括EFLAGS寄存器。所以系统可以自动保存和还原各环境的IO权限设置，且不同环境的设置互不影响。

2 块缓存

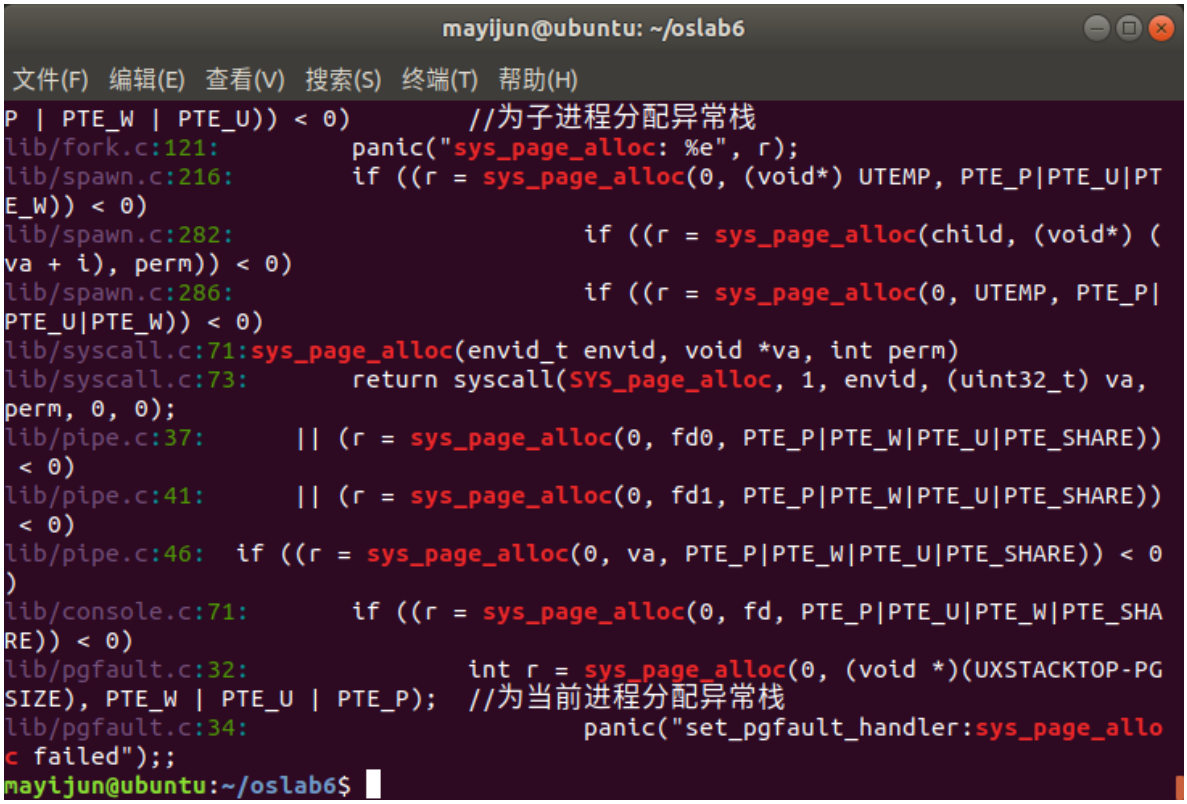
> 在fs / bc.c中实现bc_pgfault和flush_block函数。 bc_pgfault是一个页面错误处理程序，bc_pgfault的工作是从磁盘加载页面。编写此代码时，请记住（1）addr可能未与块边界对齐（2）ide_read在扇区而不是块中操作。
>
> 必要时，flush_block函数应将一个块写出到磁盘。如果该块甚至不在块缓存中或它不脏，则flush_block不应执行任何操作。我们将使用VM硬件来跟踪自从磁盘上一次读取或写入磁盘以来是否已修改了磁盘块。要查看某个块是否需要写入，我们可以看看是否在uvpt条目中设置了PTE_D" dirty"位。 将块写入磁盘后，flush_block应使用sys_page_map清除PTE_D位。

bc_pgfault()

在这个函数中，我们只需实现分配物理页面和从辅存读入页面两项内容。

鉴于addr可能未与块边界对齐，首先调用宏ROUNDDOWN将其对齐到PGSIZE的整数倍。

然后，仿照JOS中已有的示例（如下图所示在文件夹下查找"sys_page_alloc"），调用sys_page_alloc()函数分配一个物理页面于虚拟地址addr。若返回值为负，说明有错误，panic()并输出错误信息。



最后，根据题面提示，我们使用ide_read()函数，完成调入虚拟内存页面的工作。因为ide_read以扇区为单位，所以相应的参数中都需要乘上一个BLKSECTS（每块对应扇区数目）。若返回值为负则panic()。

```
    addr = ROUNDDOWN(addr, PGSIZE);
    if ((r = sys_page_alloc(0, addr, PTE_W | PTE_P | PTE_U)) < 0)
        panic("page alloc failed: %e, va %08x", r, addr);
    if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
        panic("ide read failed: %e, va %08x", r, addr);
```

flush_block()

如题面所述，首先我们还是需要将addr对齐到PGSIZE的整数倍。

然后用va_is_mapped()和va_is_dirty()两个函数判断该页是否需要被写出。

若需要写出，则调用ide_write()写出。ide_write()的用法与上面ide_read()相同，参数也需乘上一个BLKSECTS。返回值小于0则panic。

最后，我们调用sys_page_map()清除页表中该页对应的脏位(dirty bit)，因为在bc_pgfault()中已经给出了调用sys_page_map()来清除脏位的示范：

```
    // Clear the dirty bit for the disk block page since we just read the
    // block from disk
    if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
        panic("in bc_pgfault, sys_page_map: %e", r);
```

那么在flush_block()中仿照之即可。因为在bc_pgfault()中调入时是从(0, addr)映射到(0, addr)，那么在flush_block()中写出时则反过来，也是从(0, addr)映射到(0, addr)。

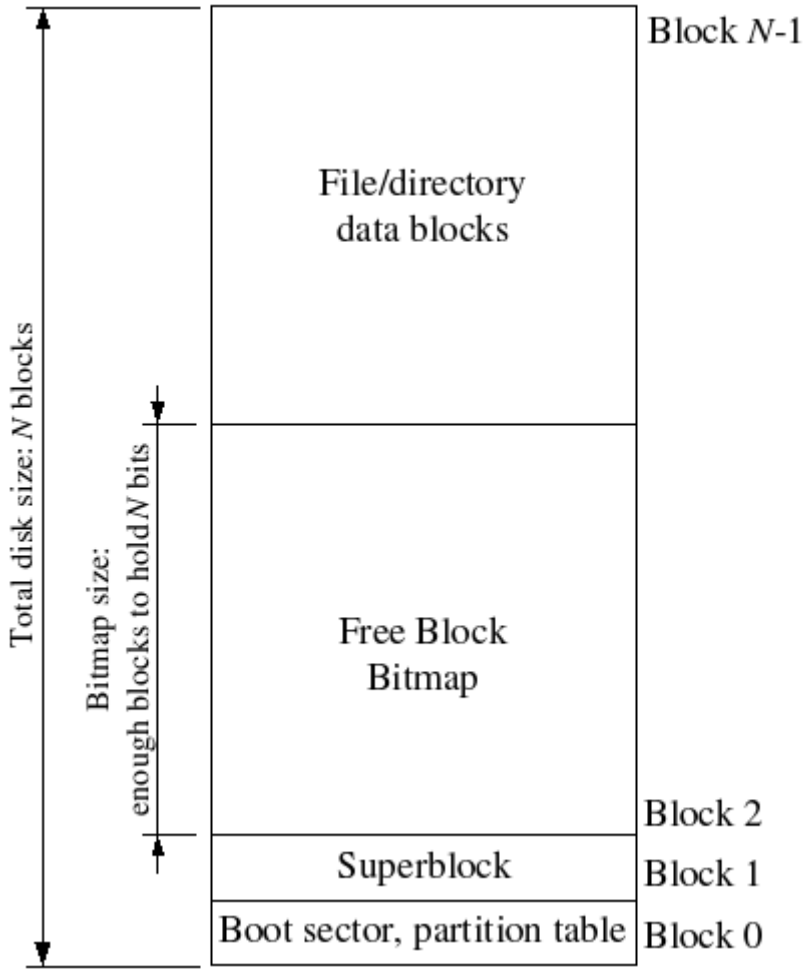
```
int r;
addr = ROUNDDOWN(addr, PGSIZE);
if (!va_is_mapped(addr) || !va_is_dirty(addr)) return;
if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
    panic("ide write failed: %e, va %08x", r, addr);
if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
    panic("in flush_block, sys_page_map: %e", r);
```

3 块位图(block bitmap)

> 使用free_block作为模板在fs / fs.c中实现alloc_block，后者应在bitmap中找到可用的磁盘块，将其标记为已使用，然后返回该块的编号。 分配该块时，应立即使用flush_block将更改后的bitmap块刷新到磁盘，以帮助文件系统保持一致。

前置知识

由lab5讲义我们知道，磁盘块的布局如下图所示，第0块为启动块和分区表，第1块为超块，第2块起为空闲块位表/位图，其大小为足够容纳N位，之后的部分存放文件和目录数据。



从free_block()函数中的相关语句（下图的行46）可以看出，因为每个位图项是32位长，所以每个位图项顺序地代表32个块的分配情况，所以我们用块号的低5位作为偏移量、剩余高位部分作为索引来访问该块在位图中对应的那一位，且该位为1代表空闲，为0代表正在使用。

```
39  // Mark a block free in the bitmap
40  void
41  free_block(uint32_t blockno)
42  {
43      // Blockno zero is the null pointer of block numbers.
44      if (blockno == 0)
45          panic("attempt to free zero block");
46      bitmap[blockno/32] |= 1<<(blockno%32);
47  }
```

alloc_block()

在知道磁盘块的布局和访问位图的方式之后，就变得十分简单。我们计算出位图占据了多少块，从位图部分后的第一块（也就是文件部分的第一块）开始循环，若遇到空闲块（用block_is_free()判定），就将其分配并返回块号；分配的过程是，将该块在位图中对应的一位置0，并flush。

```
1  // Search the bitmap for a free block and allocate it.  When you
2  // allocate a block, immediately flush the changed bitmap block
3  // to disk.
4  //
5  // Return block number allocated on success,
6  // -E_NO_DISK if we are out of blocks.
7  //
8  // Hint: use free_block as an example for manipulating the bitmap.
9  int
10 alloc_block(void)
11 {
12     // The bitmap consists of one or more blocks.  A single bitmap block
13     // contains the in-use bits for BLKBITSIZE blocks.  There are
14     // super->s_nblocks blocks in the disk altogether.
15
16     // LAB 5: Your code here.
17     int nblock_bitmap = (super->s_nblocks + BLKBITSIZE - 1) / BLKBITSIZE;
18
19     for (uint32_t i = 2 + nblock_bitmap; i < super->s_nblocks; i++)
20         if (block_is_free(i))
21         {
22             bitmap[i/32] &= ~(1 << (i%32));
23             flush_block(bitmap + i/32);
24             return i;
25         }
26
27     return -E_NO_DISK;
28 }
```

4 文件操作

> 实现file_block_walk和file_get_block。 file_block_walk从文件中的块偏移量映射到struct File或间接块中的指针，非常类似于pgdir_walk对页表所做的操作。 file_get_block更进一步，并映射到实际的磁盘块，并在必要时分配一个新的磁盘块。

file_block_walk()

分类讨论。

(1)首先如果filebno超出范围，直接返回错误-E_INVAL。

(2)然后处理最简单的一种情况，若访问的块号是直接块，可直接让*pdiskbno指向之。

然后是访问间接块，

(3)若对应的间接块已经分配，则让*pdiskno指向之，但指向的目标地址需要经过一定的推导：filebno是在文件的所有块中的顺序号，对应间接块中的块号应该是filebno - NDIRECT；f->f_indirect的值是间接块的块号，我们需要调用diskaddr()得到间接块的首地址，但因为diskaddr()的返回值是void*类型，需要强制类型转换成uint32_t*，否则运算的结果是错误的，无法通过测试；从而目标地址应等于(uint32_t*)diskaddr(f->f_indirect) + (filebno - NDIRECT)。

(4)最后是最复杂的情况，若对应的间接块未分配，(4.1)首先如果alloc=0则直接返回-E_NOT_FOUND；否则我们分配一个新的块（(4.2)若分配过程中出错，鉴于alloc_block()函数唯一可能的异常就是-E_NO_DISK，我们可以直接返回-E_NO_DISK），(4.3)将f->f_indirect指向该块，将其填充0初始化并将更改后的位图flush到磁盘（这两步需要调用diskaddr()将块号转换为对应的虚拟地址），就完成了间接块的分配，最后就归结到(3)。

```
// Find the disk block number slot for the 'filebno'th block in file 'f'.
// Set '*ppdiskbno' to point to that slot.
// The slot will be one of the f->f_direct[] entries,
// or an entry in the indirect block.
// When 'alloc' is set, this function will allocate an indirect block
// if necessary.
//
// Returns:
//  0 on success (but note that *ppdiskbno might equal 0).
//  -E_NOT_FOUND if the function needed to allocate an indirect block, but
//      alloc was 0.
//  -E_NO_DISK if there's no space on the disk for an indirect block.
//  -E_INVAL if filebno is out of range (it's >= NDIRECT + NINDIRECT).
//
// Analogy: This is like pgdir_walk for files.
// Hint: Don't forget to clear any block you allocate.
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno,
bool alloc)
{
       // LAB 5: Your code here.
       if (filebno >= NDIRECT + NINDIRECT) return -E_INVAL;  // 情况1
       if (filebno < NDIRECT)  // 情况2
       {
            *ppdiskbno = f->f_direct + filebno;
            return 0;
       }
       if (f->f_indirect == 0)  // 情况4
       {
            if (!alloc) return -E_NOT_FOUND;  // 情况4.1
            int r;
            if ((r = alloc_block()) < 0) return -E_NO_DISK;  // 情况4.2
            // 情况4.3
            // 需调用diskaddr()将块号转换为对应的虚拟地址
            f->f_indirect = r;
            memset(diskaddr(r), 0, BLKSIZE);
            flush_block(diskaddr(r));
       }
        // 情况3
       *ppdiskbno = (uint32_t*)diskaddr(f->f_indirect) + (filebno -
NDIRECT);
```

```
        return 0;
}
```

file_get_block()

首先调用刚刚完成的file_block_walk()，获得块号指针，若返回值小于0说明file_block_walk()出错，在file_get_block()中也直接将该返回值返回给调用者。

然后，若*pdiskbno的值为0，说明该块已经“指定”但尚未“分配”。仿照file_block_walk()中的流程进行块分配、初始化、flush。

最后就可以用*pdiskbno取得块号了，再套上一个diskaddr()就可以获得虚拟地址，用*blk指向之即可。

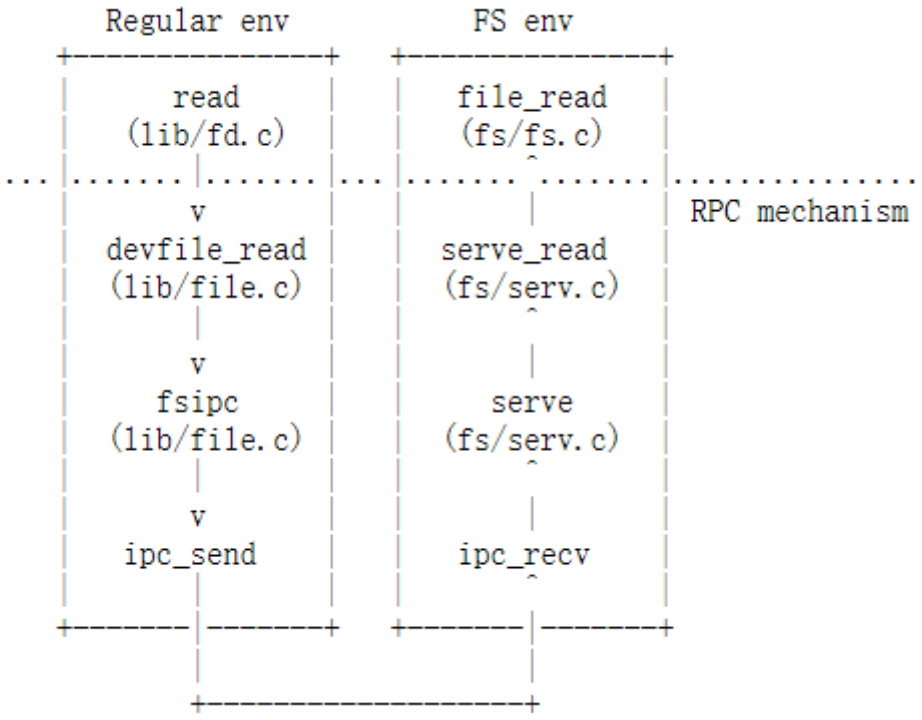
```
// Set *blk to the address in memory where the filebno'th
// block of file 'f' would be mapped.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_NO_DISK if a block needed to be allocated but the disk is full.
//  -E_INVAL if filebno is out of range.
//
// Hint: Use file_block_walk and alloc_block.
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
        // LAB 5: Your code here.
        int r;
        uint32_t* pdiskbno;

        if ((r = file_block_walk(f, filebno, &pdiskbno, 1)) < 0) return r;
        if (*pdiskbno == 0)
        {
             if ((r = alloc_block()) < 0) return r;
             *pdiskbno = r;
             memset(diskaddr(r), 0, BLKSIZE);
             flush_block(diskaddr(r));
        }

        *blk = diskaddr(*pdiskbno);
        return 0;

}
```

5 文件系统界面

> •在fs/serv.c中实现serve_read 。
>
> serve_read的大量操作将由fs / fs.c中已经实现的file_read来完成。 serve_read只需提供RPC接口即可读取文件。 查看serve_set_size中的注释和代码，以大致了解服务器功能的结构。

前置知识

由lab5讲义，由于其他环境无法直接调用文件系统环境的函数，因此我们将通过基于JOS的进程间通信(IPC)机制（参见[3]）的远程过程调用(RPC)实现对文件系统环境的公共(public)访问，具体如下图所示。



read（已经写好）是设备无关的，其将请求分派到适当的设备读取函数，在这种情况下为devfile_read，其实现专门针对磁盘的读取、将参数捆绑在请求结构中、调用fsipc发送IPC请求然后解包并返回结果。文件系统服务器循环执行serve()函数，不停地通过IPC接收请求，将请求分派到适当的处理函数，然后通过IPC发回结果；本例中，serve()将分派到serve_read()，后者将处理特定于读取请求的IPC详细信息，例如解压缩请求结构并最终调用file_read来实际执行文件读取。

serve_read()

因为紧邻serve_read()的serve_set_size()提供了一个样本，所以serve_read()可以参照之。

首先是打开文件，获得文件打开的结构体(struct OpenFile)，这部分与serve_set_size()别无二致。

然后，按照提示，我们调用file_read()完成实际底层读取文件的操作。第二个参数ret->ret_buf需要从inc/fs.h中阅读结构体struct Fsret_read的定义得知。第三个参数，考虑到RPC的返回值最长只能有一个页面长度，所以需要从req->req_n和PGSIZE中取小值；但是因为前置的devfile_read()函数（定义在lib/file.c中）已经assert了req_n <= PGSIZE，所以此处可以安全地直接使用req->req_n。第四个参数o->o_fd->fd_offset可从fs/fs.c中OpenFile、File和Fd结构体的定义得知，因为其中只有Fd保存了文件偏移量。

```
32  struct OpenFile {
33      uint32_t o_fileid;  // file id
34      struct File *o_file;    // mapped descriptor for open file
35      int o_mode;     // open mode
36      struct Fd *o_fd;    // Fd page
37  };
```

最后再将文件偏移量加以r（file_read()）的返回值，并返回之。

```
int r;
struct OpenFile* o;

if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
    return r;

if ((r = file_read(o->o_file, ret->ret_buf, req->req_n, o->o_fd->fd_offset)) < 0)
    return r;
o->o_fd->fd_offset += r;
return r;
```

6 文件系统界面(2)

> 在fs/serv.c中实现serve-write，在lib/file.c中实现devfile-write。

serve_write()

直接参照serve_read()实现即可。只需将file_read替换为file_write，将ret替换为req。

对file_write()的第三个参数，因为前置的devfile_write()函数（即将实现）已经保证req_n <= 缓冲区的最大大小，所以仍然可以安全地直接使用req->req_n。

```
int r;
struct OpenFile* o;

if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
    return r;

if ((r = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset)) < 0)
    return r;
o->o_fd->fd_offset += r;
return r;
```

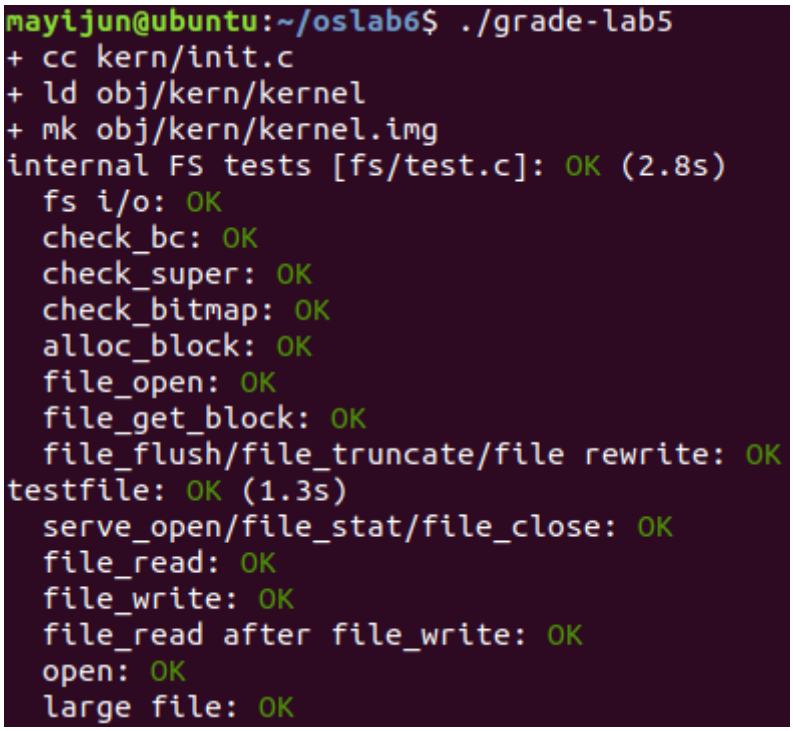
devfile_write()

根据提示，首先我们限制写入的字节数不超过写入请求中缓冲区的大小。

之后的部分可以参照紧邻devfile_write()的已写好的devfile_read()实现，只需将read都替换为write，并且将memmove()移到fsipc()前面即可。

```
// Write at most 'n' bytes from 'buf' to 'fd' at the current seek position.
//
// Returns:
//   The number of bytes successfully written.
//   < 0 on error.
static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
    // Make an FSREQ_WRITE request to the file system server.  Be
    // careful: fsipcbuf.write.req_buf is only so large, but
    // remember that write is always allowed to write *fewer*
    // bytes than requested.
    // LAB 5: Your code here
```

```
    int r;

    n = sizeof(fsipcbuf.write.req_buf) > n ? n :
sizeof(fsipcbuf.write.req_buf);
    fsipcbuf.write.req_fileid = fd->fd_file.id;
    fsipcbuf.write.req_n = n;
    memmove(fsipcbuf.write.req_buf, buf, n);
    if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
        return r;
    assert(r <= n);
    assert(r <= PGSIZE);
    return r;
}
```



[1] https://css.csail.mit.edu/6.858/2014/readings/i386/s08_03.htm

[2] https://pdos.csail.mit.edu/6.828/2018/labs/lab3/

[3] https://pdos.csail.mit.edu/6.828/2018/labs/lab4/