操作系统JOS实习第五次报告

张弛 00848231,
zhangchitc@gmail.com

May 10, 2011

Contents

1 Introduction

我在实验中主要参考了华中科技大学邵志远老师写的JOS实习指导，在邵老师的主页上http://grid.hust.edu.cn/zyshao/OSEngineering.htm 可以找到。但是这次实验的指导远远不如lab1的指导详尽，所以我这里需要补充的内容会很多。

内联汇编请参考邵老师的第二章讲义，对于语法讲解的很详细。

2 File system preliminaries

2.1 On-Disk File System Structure

2.1.1 Sectors and Blocks

2.1.2 Superblocks

2.1.3 The Block Bitmap: Managing Free Disk Blocks

2.1.4 File Meta-data

2.1.5 Directories versus Regular Files

3 The File System

3.1 Disk Access

```
Exercise 1. Modify your kernel's environment initialization function,
env_alloc in env.c, so that it gives environment 1 I/O privilege, but
never gives that privilege to any other environment.

Make sure you can start the file environment without causing a General
Protection fault. You should pass the "fs i/o" test in make grade.
```

材料中已经说的很明确了，只要在envs[1]被创建时候将EFLAGS置位即可：

kern/env.c: env_alloc ()

```c
	// If this is the file server (e == &envs[1]) give it I/O privileges.
	// LAB 5: Your code here.
	if (e == &envs[1])
		e->env_tf.tf_eflags |= FL_IOPL_3;
```

材料里后来提了一句：

```
Do you have to do anything else to ensure that this I/O privilege setting
is saved and restored properly when you subsequently switch from one
environment to another? Make sure you understand how this environment
state is handled.
```

然后我特意去看了一下进程切换的代码，关键部分应该是kern/env.c中的`env_run()`

kern/env.c

```c
void
env_pop_tf(struct Trapframe *tf)
{
        __asm __volatile("movl %0,%%esp\n"
                "\tpopal\n"
                "\tpopl %%es\n"
                "\tpopl %%ds\n"
                "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
                "\tiret"
                : : "g" (tf) : "memory");
        panic("iret failed");  /* mostly to placate the compiler */
}

void
env_run(struct Env *e)
{
    if (curenv != e) {
        curenv = e;
        curenv->env_runs ++;
        lcr3 (curenv->env_cr3);
    }

    env_pop_tf (&curenv->env_tf);
}
```

寄存器的恢复是在`env_pop_tf ()`中完成的，里面好像没有恢复eflags，但是我在lab3的报告里将`env_pop_tf ()`的过程进行了详细的说明，其中popal指令从栈中恢复了所有的通用寄存器，然后是在iret指令中恢复了eip, cs以及eflags寄存器。

在继续作下面的部分前，我们先好好看一下文件系统实现的一些细节，看到：fs/fs.h

fs/fs.h

```c
#include <inc/fs.h>
#include <inc/lib.h>

#define SECTSIZE        512                     // bytes per disk sector
#define BLKSECTS        (BLKSIZE / SECTSIZE)    // sectors per block

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP + (n*BLKSIZE). */
#define DISKMAP         0x10000000

```

```
/* Maximum disk size we can handle (3GB) */
#define DISKSIZE        0xC0000000

struct Super *super;            // superblock
uint32_t *bitmap;               // bitmap blocks mapped in memory

/* ide.c */
bool    ide_probe_disk1(void);
void    ide_set_disk(int diskno);
int     ide_read(uint32_t secno, void *dst, size_t nsecs);
int     ide_write(uint32_t secno, const void *src, size_t nsecs);

/* bc.c */
void*   diskaddr(uint32_t blockno);
bool    va_is_mapped(void *va);
bool    va_is_dirty(void *va);
void    flush_block(void *addr);
void    bc_init(void);

/* fs.c */
void    fs_init(void);
int     file_get_block(struct File *f, uint32_t file_blockno, char **pblk);
int     file_create(const char *path, struct File **f);
int     file_open(const char *path, struct File **f);
ssize_t file_read(struct File *f, void *buf, size_t count, off_t offset);
int     file_write(struct File *f, const void *buf, size_t count, off_t offset);
int     file_set_size(struct File *f, off_t newsize);
void    file_flush(struct File *f);
int     file_remove(const char *path);
void    fs_sync(void);

/* int  map_block(uint32_t); */
bool    block_is_free(uint32_t blockno);
int     alloc_block(void);

/* test.c */
void    fs_test(void);
```

从这里可以看到文件系统实现细节被分成了三个大的模块：

- ide.c: 提供IDE磁盘的驱动，比如对特定扇区(sector)的读写以及切换操作磁盘（master和slave）
- bc.c: 提供磁盘的块缓存实现机制，这个后面会详细说明。大体意义是因为磁盘最大可以支持到3G，而这么大的磁盘空间不可能是被同时使用的，所以当用户请求读写一块磁盘区域时，将其加载到文件系统进程的虚拟地址里，这样就可以用比较小的内存操作很大一块磁盘。而bc（磁盘块缓存）就是专门为文件系统服务进程实现这部分功能的模块
- fs.c: 文件系统的核心功能，比如文件的增删和读写

倒是材料里提到的fs/serv.c（真正的文件系统服务器进程的实现）我们可以稍等一下再来关注。

3.2 The Block Cache

这里描述了磁盘块缓冲的具体机制：

因为JOS支持的磁盘大小最大在3GB左右，所以我们可以使用类似lab4中实现fork的COW页面机制，也就是

1. 用文件系统服务进程的虚拟地址空间（4GB）对应到磁盘的地址空间上（3GB）
2. 初始文件系统服务进程里什么页面都没映射，如果要访问一个磁盘的地址空间，则发生页错误
3. 在页错误处理程序中，在内存中申请一个块的空间映射到相应的文件系统虚拟地址上，然后去实际的物理磁盘上读取这个区域的东西到这个内存区域上，然后恢复文件系统服务进程

这样就使用用户进程的机制完成了对于物理磁盘的读写机制，并且尽量少节省了内存。当然这里也有一个取巧的地方就是用虚拟地址空间模拟磁盘地址空间，但是材料中也提到了：

> It would be awkward for a real file system implementation on a 32-bit machine to do this since modern disks are larger than 3GB.

因为一般机器硬盘显然不止3GB，但是一个32位机器虚拟地址只有4GB的地址空间，所以这里JOS的做法是为了方便而取了巧。

> **Exercise 2.** Implement the bc_pgfault and flush_block functions in fs/bc.c. bc_pgfault is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) addr may not be aligned to a block boundary and (2) ide_read operates in sectors, not blocks.
>
> The flush_block function should write a block out to disk if necessary. flush_block shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the PTE_D "dirty" bit is set in the vpt entry. (The PTE_D bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, flush_block should clear the PTE_D bit using sys_page_map.
>
> Use make grade to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap" for a score of 20/100.

首先我们要实现的是磁盘块缓冲的页面处理部分和写回部分，根据前面的铺垫，这两个地方要作的具体工作应该都很清楚了，他们主要用到的函数是跟磁盘直接交互的ide驱动：

```
int ide_read(uint32_t secno, void *dst, size_t nsecs);
int ide_write(uint32_t secno, void *dst, size_t nsecs);
```

secno对应IDE磁盘上的扇区编号，dst为当前文件系统服务程序空间中的对应地址，nsecs为读写的扇区数。了解完以后相应的编码就很简单了：

fc/bc.c

```c
static void
bc_pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
        uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
        int r;

        // Check that the fault was within the block cache region
        if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
                panic("page fault in FS: eip %08x, va %08x, err %04x",
                      utf->utf_eip, addr, utf->utf_err);


        // PGSIZE = BLKSIZE
        addr = ROUNDDOWN (addr, PGSIZE);

        if ((r = sys_page_alloc (0, addr, PTE_U|PTE_P|PTE_W)) < 0)
            panic ("bc_pgfault: page allocation failed : %e", r);

        // read the whole block[blockno]
        ide_read (blockno * BLKSECTS, addr, BLKSECTS);



        // Sanity check the block number. (exercise for the reader:
        // why do we do this *after* reading the block in?)
        if (super && blockno >= super->s_nblocks)
                panic("reading non-existent block %08x\n", blockno);

        // Check that the block we read was allocated.
        if (bitmap && block_is_free(blockno))
                panic("reading free block %08x\n", blockno);
}

void
flush_block(void *addr)
{
        uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

        if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
                panic("flush_block of bad va %08x", addr);

        // LAB 5: Your code here.

        addr = ROUNDDOWN (addr, PGSIZE);

        int r;

        if (va_is_mapped (addr) && va_is_dirty (addr)) {
            ide_write (blockno * BLKSECTS, addr, BLKSECTS);

            if ((r = sys_page_map (0, addr, 0, addr, PTE_USER)) < 0)
                panic ("flush_block: page mapping failed : %e", r);
        }
}
```

做完这部分以后进行make qemu可以通过check_bc, check_super 以及check_bitmap三个测试。他们是在fs/fs.c中的`fs_init()`中完成的：

fs/fs.c: fs_init()

```c
void
fs_init(void)
{
        static_assert(sizeof(struct File) == 256);

        // Find a JOS disk.  Use the second IDE disk (number 1) if available.
        if (ide_probe_disk1())
                ide_set_disk(1);
        else
                ide_set_disk(0);

        bc_init();

        // Set "super" to point to the super block.
        super = diskaddr(1);
        // Set "bitmap" to the beginning of the first bitmap block.
        bitmap = diskaddr(2);

        check_super();
        check_bitmap();
}
```

其中bc_init()就是简单的安装一下页错误处理程序。主要是这里设置起了文件系统的超级块Super，可以看到文件系统将第Super块指向了文件系统的Block 1，然后块位图指向了Block 2。这里对应了在2.1.3中呈现的那张磁盘规划图。

块位图是没有相关结构的（因为就是直接读取特定二进制位），关于Super结构的具体定义在inc/fs.h中:

inc/fs.h

```c
// See COPYRIGHT for copyright information.

#ifndef JOS_INC_FS_H
#define JOS_INC_FS_H

#include <inc/types.h>
#include <inc/mmu.h>

// File nodes (both in-memory and on-disk)

// Bytes per file system block - same as page size
#define BLKSIZE         PGSIZE
#define BLKBITSIZE      (BLKSIZE * 8)

// Maximum size of a filename (a single path component), including null
// Must be a multiple of 4
#define MAXNAMELEN      128

// Maximum size of a complete pathname, including null
#define MAXPATHLEN      1024

// Number of block pointers in a File descriptor
#define NDIRECT         10
// Number of direct block pointers in an indirect block
#define NINDIRECT       (BLKSIZE / 4)

#define MAXFILESIZE     ((NDIRECT + NINDIRECT) * BLKSIZE)

struct File {
        char f_name[MAXNAMELEN];        // filename
        off_t f_size;                   // file size in bytes
```

```
        uint32_t f_type;                // file type

        // Block pointers.
        // A block is allocated iff its value is != 0.
        uint32_t f_direct[NDIRECT];     // direct blocks
        uint32_t f_indirect;            // indirect block

        // Pad out to 256 bytes; must do arithmetic in case we're compiling
        // fsformat on a 64-bit machine.
        uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4];
} __attribute__((packed));      // required only on some 64-bit machines

// An inode block contains exactly BLKFILES 'struct File's
#define BLKFILES        (BLKSIZE / sizeof(struct File))

// File types
#define FTYPE_REG       0       // Regular file
#define FTYPE_DIR       1       // Directory


// File system super-block (both in-memory and on-disk)

#define FS_MAGIC        0x4A0530AE      // related vaguely to 'J\0S!'

struct Super {
        uint32_t s_magic;               // Magic number: FS_MAGIC
        uint32_t s_nblocks;             // Total number of blocks on disk
        struct File s_root;             // Root directory node
};
```

- 首先几个常数要读清楚，可以熟悉JOS的一些细节规定
- 一个File的大小为256 bytes，所以一个块中可以放下4个File结构。其具体的域在MIT材料的第一段预备知识中已经有过详细叙述

从这里可以看到一个Super其实没有占用一个块大小，大概就是一个File大小（256）加两个DWORDS，而块位图则占用了8个块大小（BLKBITSIZE）。后面马上我们就要对相关结构进行处理。

但是这里还有一个问题存在我的脑子中，就是我们这里只对Super指针的地址进行了赋值，那实际内存区域里存的东西是什么时候被初始化的呢？我找了很久，终于在张顺廷湿胸的提醒下去看了看fs/fsformat.c，才恍然大悟：

1. 首先，Super的指针赋值为：

fs/fs.c: fs_init()

```
        // Set "super" to point to the super block.
        super = diskaddr(1);
}
```

diskaddr定义在fs/bc.c中：

fs/bc.c: diskaddr()

```c
void*
diskaddr(uint32_t blockno)
{
	if (blockno == 0 || (super && blockno >= super->s_nblocks))
		panic("bad block number %08x in diskaddr", blockno);
	return (char*) (DISKMAP + blockno * BLKSIZE);
}
```

可见Super的位置是虚拟地址空间的第一块，当这个块被访问的时候，自然会使用磁盘块缓存机制读取到用户空间中，读取来源是IDE磁盘。

2. 这个IDE磁盘是哪里来的呢？在JOS里是以镜像文件由QEMU模拟成IDE磁盘的，产生方式在fs/Makefrag里有详细过程：

fs/Makefrag

```makefile

OBJDIRS += fs

FSOFILES :=		$(OBJDIR)/fs/ide.o \
			$(OBJDIR)/fs/bc.o \
			$(OBJDIR)/fs/fs.o \
			$(OBJDIR)/fs/serv.o \
			$(OBJDIR)/fs/test.o \

USERAPPS :=		$(OBJDIR)/user/init

FSIMGTXTFILES :=	fs/newmotd \
			fs/motd



FSIMGFILES := $(FSIMGTXTFILES) $(USERAPPS)

$(OBJDIR)/fs/%.o: fs/%.c fs/fs.h inc/lib.h
	@echo + cc[USER] $<
	@mkdir -p $(@D)
	$(V)$(CC) -nostdinc $(USER_CFLAGS) -c -o $@ $<

$(OBJDIR)/fs/fs: $(FSOFILES) $(OBJDIR)/lib/entry.o $(OBJDIR)/lib/libjos.a user/user.ld
	@echo + ld $@
	$(V)mkdir -p $(@D)
	$(V)$(LD) -o $@ $(ULDFLAGS) $(LDFLAGS) -nostdlib \
		$(OBJDIR)/lib/entry.o $(FSOFILES) \
		-L$(OBJDIR)/lib -ljos $(GCC_LIB)
	$(V)$(OBJDUMP) -S $@ >$@.asm

# How to build the file system image
$(OBJDIR)/fs/fsformat: fs/fsformat.c
	@echo + mk $(OBJDIR)/fs/fsformat
	$(V)mkdir -p $(@D)
	$(V)gcc $(USER_CFLAGS) -o $(OBJDIR)/fs/fsformat fs/fsformat.c

$(OBJDIR)/fs/clean-fs.img: $(OBJDIR)/fs/fsformat $(FSIMGFILES)
	@echo + mk $(OBJDIR)/fs/clean-fs.img
	$(V)mkdir -p $(@D)
	$(V)$(OBJDIR)/fs/fsformat $(OBJDIR)/fs/clean-fs.img 1024 $(FSIMGFILES)

$(OBJDIR)/fs/fs.img: $(OBJDIR)/fs/clean-fs.img
	@echo + cp $(OBJDIR)/fs/clean-fs.img $@
	$(V)cp $(OBJDIR)/fs/clean-fs.img $@
```

```
46
47 all: $(OBJDIR)/fs/fs.img
48
49 #all: $(addsuffix .sym, $(USERAPPS))
50
51 #all: $(addsuffix .asm, $(USERAPPS))
```

注意从32行开始的编译：

(a) 第32行：将fs/fsformat.c编译成可执行文件

(b) 第38行：使用fs/fsformat可执行文件产生镜像文件fs/clean-fs.img

(c) 第43行：将fs/clean-fs.img复制成真正的磁盘镜像文件fs/fs.img

我们主要关注第二步是如何产生fs/clean-fs.img的，可以看到它使用的命令为：

fs/Makefrag

```
41 $(V)$(OBJDIR)/fs/fsformat $(OBJDIR)/fs/clean-fs.img 1024 $(FSIMGFILES)
```

可以看到这个fs/fsformat接受了1024和一堆文件系统的目标文件生成了clean-fs.img，那么我们好奇这个fs/fsformat是干吗的呢？打开fs/fsformat.c来看看。

3. fs/fsformat.c是一个创建磁盘的工具，我们就看看它的主函数:

fs/fsformat.c: main()

```
int
main(int argc, char **argv)
{
	int i;
	char *s;
	struct Dir root;

	assert(BLKSIZE % sizeof(struct File) == 0);

	if (argc < 3)
		usage();

	nblocks = strtol(argv[2], &s, 0);
	if (*s || s == argv[2] || nblocks < 2 || nblocks > 1024)
		usage();

	opendisk(argv[1]);

	startdir(&super->s_root, &root);
	for (i = 3; i < argc; i++)
		writefile(&root, argv[i]);
	finishdir(&root);

	finishdisk();
	return 0;
}
```

可以从代码里看到它作了这么几件事：

(a) 使用opendisk创建一个磁盘文件，超级块在这里被初始化
(b) 使用startdir创建根目录，并初始化超级块
(c) 使用writefile将目标文件写入磁盘映像
(d) 使用finishdir将根目录写入磁盘映像
(e) 使用finishdisk将块位图设置为正确的值，完成磁盘映像的创建

其中opendisk就是我们初始化Super超级块的地方。

4. 来看一下具体代码：

fs/fsformat.c: opendisk()

```
void
opendisk(const char *name)
{
	int r, diskfd, nbitblocks;

	if ((diskfd = open(name, O_RDWR | O_CREAT, 0666)) < 0)
		panic("open %s: %s", name, strerror(errno));

	if ((r = ftruncate(diskfd, 0)) < 0
	    || (r = ftruncate(diskfd, nblocks * BLKSIZE)) < 0)
		panic("truncate %s: %s", name, strerror(errno));

	if ((diskmap = mmap(NULL, nblocks * BLKSIZE, PROT_READ|PROT_WRITE,
			    MAP_SHARED, diskfd, 0)) == MAP_FAILED)
		panic("mmap %s: %s", name, strerror(errno));

	close(diskfd);

	diskpos = diskmap;
	alloc(BLKSIZE);
	super = alloc(BLKSIZE);
	super->s_magic = FS_MAGIC;
	super->s_nblocks = nblocks;
	super->s_root.f_type = FTYPE_DIR;
	strcpy(super->s_root.f_name, "/");

	nbitblocks = (nblocks + BLKBITSIZE - 1) / BLKBITSIZE;
	bitmap = alloc(nbitblocks);
	memset(bitmap, 0xFF, nbitblocks * BLKSIZE);
}
```

这里创建了映像文件，并为Super和bitmap分配好了空间（在文件中留出相应大小的空间）

其他的函数我们就不细看了，有兴趣的话可以自行研究。主要是通过Super的例子，我们看到了一个物理磁盘被创建以及其相应的所有细节被设置好的全过程。可以在后面的部分中弄清楚文件系统每个模块的来龙去脉。

3.3 The Block Bitmap

```
Exercise 3. Use free_block as a model to implement alloc_block, which
should find a free disk block in the bitmap, mark it used, and return the
number of that block. When you allocate a block, you should immediately
flush the changed bitmap block to disk with flush_block, to help file
system consistency.

Use make grade to test your code. Your code should now pass "alloc_block"
for a score of 25/100.
```

这部分很简单，代码中告诉我们了在内存中直接查找bitmap的时间远远小于读取IDE磁盘的时间，可以不用在乎效率：

fs/fs.c: alloc_block()

```c
alloc_block(void)
{
    int blockno;
    for (blockno = 0; blockno < super->s_nblocks; blockno++)
        if (block_is_free (blockno)) {
            bitmap[blockno/32] ^= 1 << (blockno%32);
            flush_block (bitmap);
            return blockno;
        }

    return -E_NO_DISK;
}
```

3.4 File Operations

在进行下面的工作之前，先了解一下fs/fs.c中提供的各个函数的功能：

fs/fs.c

```c
/* public */
void    fs_init(void);
int     file_get_block(struct File *f, uint32_t file_blockno, char **pblk);
int     file_create(const char *path, struct File **f);
int     file_open(const char *path, struct File **f);
ssize_t file_read(struct File *f, void *buf, size_t count, off_t offset);
int     file_write(struct File *f, const void *buf, size_t count, off_t offset);
int     file_set_size(struct File *f, off_t newsize);
void    file_flush(struct File *f);
int     file_remove(const char *path);
void    fs_sync(void);
bool    block_is_free(uint32_t blockno);
int     alloc_block(void);


/* static */
static int file_block_walk(
        struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)

static int dir_lookup(
        struct File *dir, const char *name, struct File **file)

static int dir_alloc_file(
```

```
        struct File *dir, struct File **file)

static const char* skip_slash(const char *p)

static int walk_path(
        const char *path, struct File **pdir, struct File **pf, char *lastelem)

static int file_free_block(struct File *f, uint32_t filebno)

static void file_truncate_blocks(struct File *f, off_t newsize)
```

这里前面一部分public的函数应该无论是从函数名还是参数上都是比较明确的，就不再解释了，主要需要解释的是下面一部分静态函数：

file_block_walk(*f, filebno, ppdiskbno, alloc) ：
寻找一个文件结构f中的第filebno个块指向的硬盘块编号放入ppdiskbno，即如果filebno小于NDIRECT，则返回属于f_direct[NDIRECT]中的相应链接，否则返回f_indirect中查找的块。

如果alloc为真且相应硬盘块不存在，则分配一个。

当我们要将一个修改后的文件flush回硬盘，就需要使用这个函数找一个文件中链接的所有磁盘块，将他们都flush_block

dir_lookup(*dir, *name, **file) ：
这个很明显了

dir_alloc_file(*dir, **file) ：
在*dir对应的File结构中分配一个File的指针链接给*file，用于添加文件的操作。

skip_slash(*p) ：
用于路径中的字符串处理，调过斜杠。

walk_path(*path, **pdir, **pf, *lastlem) ：
*path为从根目录开始描述的文件名，如果成功找到了文件，则把相应的文件File结构赋值给*pf，其所在目录的File结构赋值给**pdir，lastlem为失败时最后剩下的文件名字。

file_free_block(*f, filebno) ：
释放一个文件中的第filebno个磁盘块。此函数在file_truncate_blocks中被调用

file_truncate_blocks(*f, newsize) ：
将文件设置为缩小后的新大小，清空那些被释放的物理块。

弄清楚函数功能以后就可以开始下面的工作了

Exercise 4. Implement file_block_walk and file_get_block. file_block_walk maps from a block offset within a file to the pointer for that block in the struct File or the indirect block, very much like what pgdir_walk did for page tables. file_get_block goes one step further and maps to the actual disk block, allocating a new one if necessary.

Use make grade to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file rewrite" for a score of 40/100.

先来看`file_block_walk()`：

fs/fs.c: file_block_walk()

```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc
    )
{
    int r;
    if (filebno >= NDIRECT + NINDIRECT)
        return -E_INVAL;

    if (filebno < NDIRECT) {
        if (ppdiskbno)
            *ppdiskbno = f->f_direct + filebno;
        return 0;
    }

    if (!alloc && !f->f_indirect)
        return -E_NOT_FOUND;

    if (!f->f_indirect) {
        if ((r = alloc_block ()) < 0)
            return -E_NO_DISK;
        f->f_indirect = r;

        memset (diskaddr (r), 0, BLKSIZE);
        flush_block (diskaddr(r));
    }

    if (ppdiskbno)
        *ppdiskbno = (uint32_t *) diskaddr (f->f_indirect) + filebno - NDIRECT;

    return 0;
}
```

这里涉及到了对文件中对于磁盘块链接的操作，一定要明确一个概念：File结构中无论是f_direct还是f_indirect，**他们存储的都是指向的物理磁盘块的编号！如果要对指向的磁盘块进行读写，那么必须用diskaddr 转换成文件系统地址空间后才可以进行相应的操作**。

这里最需要注意的是如果申请了一个INDIRECT的链接块，一定要记得将其清空，并写回到磁盘中。

继续看`file_get_block()`:

fs/fs.c: file_get_block()

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
```

```
{
    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 -E_NO_DISK;

        *pdiskbno = r;
        memset (diskaddr (r), 0, BLKSIZE);
        flush_block (diskaddr (r));
    }

    *blk = diskaddr (*pdiskbno);

    return 0;
}
```

这里有一个地方代码注释里没有说得太清楚，就是blk最后应该指向得到的blk对应在文件系统地址空间中的地址，所以最后需要用diskaddr转换。同样和上面的函数一样要记得写回新申请的物理块数据。

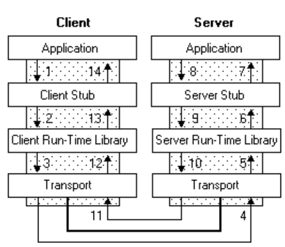
3.5 Client/Server File System Access

MIT的材料中在这部分开始实现文件系统的服务器端以及客户端的代码，两者通过RPC（Remote Process Call）实现通信。

3.5.1 How RPC Works

这部分内容在MIT材料上看到的时候就不是太理解，所以我专门去找了一下资料，下面的资料是在MSDN找到的，摘抄了一部分，详细的可以参考http://msdn.microsoft.com/en-us/library/aa374358(v=VS.85).aspx

The RPC tools make it appear to users as though a client directly calls a procedure located in a remote server program. The client and server each have their own address spaces; that is, each has its own memory resource allocated to data used by the procedure. The following figure illustrates the RPC architecture.



As the illustration shows, the client application calls a local stub procedure instead of the actual code implementing the procedure. Stubs are compiled and linked with the client application. Instead of containing

the actual code that implements the remote procedure, the client stub code:

1. Retrieves the required parameters from the client address space.
2. Translates the parameters as needed into a standard NDR format for transmission over the network.
3. Calls functions in the RPC client run-time library to send the request and its parameters to the server.

The server performs the following steps to call the remote procedure.

1. The server RPC run-time library functions accept the request and call the server stub procedure.
2. The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs.
3. The server stub calls the actual procedure on the server.

The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is complete, a similar sequence of steps returns the data to the client.

1. The remote procedure returns its data to the server stub.
2. The server stub converts output parameters to the format required for transmission over the network and returns them to the RPC run-time library functions.
3. The server RPC run-time library functions transmit the data on the network to the client computer.

The client completes the process by accepting the data over the network and returning it to the calling function.

1. The client RPC run-time library receives the remote-procedure return values and returns them to the client stub.
2. The client stub converts the data from its NDR to the format used by the client computer. The stub writes data into the client memory and returns the result to the calling program on the client.
3. The calling procedure continues as if the procedure had been called on the same computer.

The run-time libraries are provided in two parts: an import library, which is linked with the application and the RPC run-time library, which is implemented as a dynamic-link library (DLL).

The server application contains calls to the server run-time library functions which register the server's interface and allow the server to accept remote procedure calls. The server application also contains the application-specific remote procedures that are called by the client applications.

这个介绍大致能看清楚RPC的工作原理。对应到JOS，他们之间的联系是：

- RPC的最底层传输层可以是network，这里我们JOS只是在IPC上实现的RPC
- JOS中的Server Stub即fs/serv.c，Server Run-Time Library即fs/fs.c
- JOS中的Client Stub即lib/fd.c，用于封装文件传输的细节，实际上一个文件可以对应一个实际文件，也可以是Socket，Pipe之类的
- JOS中的Client Run-Time Library即lib/file.c，在这里对应真实文件系统的函数调用和数据传输

3.5.2 JOS C/S File System Access

Exercise 5. Implement serve_read in fs/serv.c and devfile_read in lib/file.c.

serve_read's heavy lifting will be done by the already-implemented file_read in fs/fs.c (which, in turn, is just a bunch of calls to file_get_block). serve_read just has to provide the RPC interface for file reading. Look at the comments and code in serve_set_size to get a general idea of how the server functions should be structured.

Likewise, devfile_read should pack its arguments into fsipcbuf for serve_read, call fsipc, and handle the result.

Use make grade to test your code. Your code should pass "lib/file.c" and "file_read" for a score of 50/100.

这个Exercise里面我们主要来关注服务器端程序的架构，下个Exercise里会对客户端程序的结构进行说明。

首先有几个特别重要的结构需要了解，看到inc/fs.h

inc/fs.h

```c

// Definitions for requests from clients to file system
enum {
	FSREQ_OPEN = 1,
	FSREQ_SET_SIZE,
	// Read returns a Fsret_read on the request page
	FSREQ_READ,
	FSREQ_WRITE,
	// Stat returns a Fsret_stat on the request page
	FSREQ_STAT,
	FSREQ_FLUSH,
	FSREQ_REMOVE,
	FSREQ_SYNC
};

union Fsipc {
	struct Fsreq_open {
		char req_path[MAXPATHLEN];
		int req_omode;
	} open;
	struct Fsreq_set_size {
		int req_fileid;
		off_t req_size;
	} set_size;
	struct Fsreq_read {
		int req_fileid;
		size_t req_n;
	} read;
	struct Fsret_read {
		char ret_buf[PGSIZE];
	} readRet;
	struct Fsreq_write {
		int req_fileid;
		size_t req_n;
		char req_buf[PGSIZE - (sizeof(int) + sizeof(size_t))];
	} write;
	struct Fsreq_stat {
		int req_fileid;
	} stat;
	struct Fsret_stat {
```

```
                char ret_name[MAXNAMELEN];
                off_t ret_size;
                int ret_isdir;
        } statRet;
        struct Fsreq_flush {
                int req_fileid;
        } flush;
        struct Fsreq_remove {
                char req_path[MAXPATHLEN];
        } remove;
};
```

这里需要了解union Fsipc，文件系统中客户端和服务器端通过IPC进行通信，那么通信的数据格式就是union Fsipc，它里面的每一个成员对应一种文件系统的操作请求。每次客户端发来请求，都会将参数放入一个union Fsipc映射到一个物理页传递给服务器端，这里就涉及到文件服务器程序的地址空间的布局：

```
/*
 *     4 Gig -------->  +------------------------------+
 *                      :              .               :
 *                      :              .               :
 *                      :              .               :
 *     USTACKTOP  --->  +------------------------------+ 0xeebfe000
 *                      |      Normal User Stack       | RW/RW  PGSIZE
 *                      +------------------------------+ 0xeebfd000
 *                      :              .               :
 *                      :              .               :
 *                      +------------------------------+
 *                      |             1024             | 1024 x PGSIZE
 *                      |          struct Fd *         |
 *  DISKMAP + DISKSIZE  +------------------------------+ 0xd0000000
 *                      |                              |
 *                      |                              |
 *                      |      3GB IDE Disk Space      |
 *                      |                              |
 *                      |                              |
 *     DISKMAP    --->  +------------------------------+ 0x10000000
 *                      |      union Fsipc *fsreq      | RW/RW  PGSIZE
 *     fsreq      --->  +------------------------------+ 0x0ffff000
 *                      :              .               :
 *                      :              .               :
 *                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
 *                      |     Program Data & Heap      |
 *     UTEXT -------->  +------------------------------+ 0x00800000
 *     PFTEMP ------->  |       Empty Memory (*)       |        PTSIZE
 *                      |                              |
 *     UTEMP -------->  +------------------------------+ 0x00400000      --+
 *                      |       Empty Memory (*)       |                   |
 *                      | - - - - - - - - - - - - - - -|                   |
 *                      |  User STAB Data (optional)   |                 PTSIZE
 *     USTABDATA ---->  +------------------------------+ 0x00200000        |
 *                      |       Empty Memory (*)       |                   |
 *     0 ------------>  +------------------------------+                 --+
 *
 */
```

上图我们已经明确的画出了服务器端程序的虚拟空间示意图，需要注意的仅仅是它和普通用户程序不同的地方：

[DISKMAP, DISKMAP + DISKSIZE) ：

前面提到过很多次了，这部分空间映射了3GB的对应IDE磁盘空间。

[0x0ffff000, DISKMAP)：
一次IPC请求的union Fsipc放置的地址空间。

[0xd0000000, 0xd0000000 + 1024 × PGSIZE)：
这片空间有1024个物理页，每个物理页对应一个struct Fd，这个我们留在后面讲。

其他区域：
和普通用户程序一样，UTEXT往上是代码段和部分数据，USTACKTOP下面是用户栈，往上是操作系统预留的系统栈和环境等等。

打开fs/serv.c，可以看到它定义的唯一一个全局变量opentab：

fs/serv.c

```c
#include <inc/x86.h>
#include <inc/string.h>

#include "fs.h"

#define debug 0

struct OpenFile {
        uint32_t o_fileid;      // file id
        struct File *o_file;    // mapped descriptor for open file
        int o_mode;             // open mode
        struct Fd *o_fd;        // Fd page
};

// Max number of open files in the file system at once
#define MAXOPEN         1024
#define FILEVA          0xD0000000

// initialize to force into data section
struct OpenFile opentab[MAXOPEN] = {
        { 0, 0, 1, 0 }
};

// Virtual address at which to receive page mappings containing client requests.
union Fsipc *fsreq = (union Fsipc *)0x0ffff000;
```

文件系统默认最大同时可以打开的文件个数为1024，所以有1024个struct Openfile，这个结构是服务器程序维护的一个对应，它将一个文件系统的真实文件struct File和用户客户端打开的文件描述符struct Fd对应到一起（具体struct Fd代表的意义见下段）。每个被打开的文件对应的struct Fd都被映射到FILEVA上往上的一个物理页，服务器程序和打开这个文件的客户程序共享这个物理页。客户端程序和文件系统服务器通信时就通过o_fileid来指定要操作的文件。

具体的struct Fd被定义在inc/fd.h中：

inc/fd.h

```c
struct FdFile {
        int id;
};
```

```c

struct Fd {
	int fd_dev_id;
	off_t fd_offset;
	int fd_omode;
	union {
		// File server files
		struct FdFile fd_file;
	};
};
```

它是一个抽象层，因为JOS和Linux一样，所有的IO都是文件，所以用户看到的都是Fd代表的文件，但是Fd会记录其对应的具体对象，比如真实文件、Socket和管道等等，因为我们现在只有文件，所以看到union里只有一个FdFile，后面如果有其他类型的对象加入，那么union里会有其他的内容。

kern/: ()

kern/: ()

kern/: ()

Exercise 6. Implement serve_write in fs/serv.c and devfile_write in lib/file.c.

Use make grade to test your code. Your code should pass "file_write" and "file_read after file_write" for a score of 60/100.

3.6 Client-Side File Operations

记得说client端的地址空间

和

lib带来的文件头entry.S

3.7 Spawning Processes

kern/: ()

kern/: ()

kern/: ()