操作系统实习报告

曹胜操 - 1500012838

Lab 5: File system, Spawn and Shell

在这一Lab中，我们将实现`spawn`，载入并运行磁盘上的可执行文件，然后就可以在JOS中运行shell了。我们首先需要一个可读写的文件系统。

我们将实现一个简单的单用户的文件系统，有着最基本的一些功能，足以在JOS中发挥作用。这个文件系统不会像大多数UNIX文件系统一样使用inode来保存文件元数据，而是在对应的目录项中保存。文件和目录都是由一系列的数据块组成，可以分散于磁盘上的各处。文件系统将这背后的细节隐藏起来，提供一个方便的读写接口。我们的文件系统还允许用户环境直接读取目录中的元数据，而不用通过一些特殊的调用。

大多数磁盘是不能以字节为单位进行读写的，而是以512B大小的扇区(sector)为单位的。文件系统分配和储存文件的单位是块(block)，其大小是扇区大小的整数倍。我们的文件系统使用4096B大小的块，正好也是内存中页的大小。还有一些特殊的、容易被读取的超级块(superblock)被用于存储文件系统的元数据，如块大小、磁盘大小等。我们的文件系统将会有1个这样的超级块，位于磁盘上的块1。块0通常为Boot loader和分区表等保留，所以不使用它。这个超级块的布局可见`inc/fs.h`中的`struct Super`，其内容很简单：

```
// 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
};
```

文件的元数据的布局可见`struct File`，主要包括文件名、大小、类型，以及组成文件的块的指针。这个元数据储存于目录项中：

```
// 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
```

其中的`f_direct`数组中的指针指向文件的前`NDIRECT`个块，这部分支持的文件大小为40KB。如果文件更大，`f_indirect`指向的块中存储着`NINDIRECT`个块的指针，可以额外支持4M大小的文件。这里的指针实际上是磁盘中的块编号。在真实的文件系统中，要支持更大的文件需要double/triple-indirect块。从这个`struct File`的角度来看，文件和目录的区别并不大。后者的块中是文件或者子目录的元数据。在我们的文件系统的超级块中有一个特殊的`File`，代表根目录。一个`struct File`的大小为256B，每个块可以放下16个这样的结构体。

我们将实现的只是文件系统中的一小部分。首先我们将在用户级文件系统环境中实现一个基于PIO (Programmed I/O)磁盘驱动。x86处理器通过EFLAGS中的IOPL位来确定保护模式下的代码是否可以执行特殊的I/O指令，比如`in`和`out`。因此我们需要给这个文件系统环境这样的权限，而其它的环境则不能有。

> Exercise 1. `i386_init`通过在`env_create`创建环境时传入`ENV_TYPE_FS`这个类型来确定文件系统环境。修改它，给这个文件系统环境以特殊的I/O权限。

在`i386_init()`中，完成SMP启动后就创建了文件系统环境：

```
	// Start fs.
	ENV_CREATE(fs_fs, ENV_TYPE_FS);
```

我们在`kern/env.c`中的`env_create()`中添加以下代码来赋予权限：

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

> Question 1. 需要做一些特殊的处理来保证I/O权限在上下文切换被正确地保存和恢复吗?

不需要，因为这个权限存于`tf_eflags`中。这个寄存器中的内容在进入trap的时候会被处理器保存，返回时通过`iret`恢复。

在`GNUmakefile`中，可以看到QEMU使用文件`obj/kern/kernel.img`作为磁盘0的映像（类似于Windows中的C盘），使用`obj/fs/fs.img`作为磁盘1的映像（类似于D盘）。磁盘0只用于启动内核。

我们的文件系统将实现一个块缓存机制。我们只支持不大于3GB的磁盘，在文件系统环境的地址空间中留存了`DISKMAP = 0x10000000`到`DISKMAP + DISKMAX = 0xD0000000`作为磁盘的映射，例如磁盘块1被映射到虚拟地址`0x10001000`。`fs/bc.c`中的`diskaddr()`实现了磁盘编号到虚拟地址的转换。我们并不会直接把整个磁盘都读入内存，而是先建立好映射，在需要的时候才读入。

> Exercise 2. 实现`fs/bc.c`中的`bc_pgfault`和`flush_block`。前者处理缺页，从磁盘中读入需要的内容。需要注意的是`addr`并不一定是块对齐的，`ide_read`处理的基本单位是扇区。后者在必要时将块写回磁盘。如果该块根本不在缓存中，或者并没有被写过，就不需要进行写入操作。检查`uvpt`项中的`PTE_D`位即可。写回后要把该位清零。

`bc_pgfault()`实现如下：

```
// Fault any disk block that is read in to memory by
// loading it from disk.
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);

    // Sanity check the block number.
    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n" , blockno);

    // Allocate a page in the disk map region, read the contents
    // of the block from the disk into that page.
    // Hint: first round addr to page boundary. fs/ide.c has code to read
    // the disk.
    //
    // LAB 5: you code here:
    addr = ROUNDDOWN(addr, PGSIZE);
    if ((r = sys_page_alloc(0, addr, PTE_U | PTE_W | PTE_P)) < 0)
        panic("bc_pgfault: %e!\n" , r);
    ide_read(blockno * BLKSECTS, addr, BLKSECTS);

    // 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);

    // Check that the block we read was allocated. (exercise for
    // the reader: why do we do this *after* reading the block
    // in?)
    if (bitmap && block_is_free(blockno))
        panic("reading free block %08x\n" , blockno);
}
```

做完必要的检查后，我们在内存中分配一个页，然后用`ide_read()`来读入磁盘数据。代码中留下的问题我们将在后面回答。`flush_block()`实现如下：

```
// Flush the contents of the block containing VA out to disk if
// necessary, then clear the PTE_D bit using sys_page_map.
// If the block is not in the block cache or is not dirty, does
// nothing.
// Hint: Use va_is_mapped, va_is_dirty, and ide_write.
// Hint: Use the PTE_SYSCALL constant when calling sys_page_map.
// Hint: Don't forget to round addr down.
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);
    if (va_is_mapped(addr) && va_is_dirty(addr))
    {
        int r;
        if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
            panic("flush_block: Not ready!\n");
        if ((r = sys_page_map(0, addr, 0, addr, PTE_SYSCALL)) < 0)
            panic("flush_block: %e!\n", r);
    }
}
```

同样是做一些检查，然后调用`ide_write()`来写回磁盘。其中`va_is_mapped()`和`va_is_dirty()`分别检查`PTE_P`和`PTE_D`位。我们也可以看一看`ide.c`中底层的读入写回：

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

    assert(nsecs <= 256);

    ide_wait_ready(0);

    outb(0x1F2, nsecs);
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, 0xE0 | ((diskno&1)<<4) | ((secno>>24)&0x0F));
    outb(0x1F7, 0x20);  // CMD 0x20 means read sector

    for (; nsecs > 0; nsecs--, dst += SECTSIZE) {
        if ((r = ide_wait_ready(1)) < 0)
            return r;
```

```
        insl(0x1F0, dst, SECTSIZE/4);
    }

    return 0;
}

int
ide_write(uint32_t secno, const void *src, size_t nsecs)
{
    int r;

    assert(nsecs <= 256);

    ide_wait_ready(0);

    outb(0x1F2, nsecs);
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, 0xE0 | ((diskno&1)<<4) | ((secno>>24)&0x0F));
    outb(0x1F7, 0x30);  // CMD 0x30 means write sector

    for (; nsecs > 0; nsecs--, src += SECTSIZE) {
        if ((r = ide_wait_ready(1)) < 0)
            return r;
        outsl(0x1F0, src, SECTSIZE/4);
    }

    return 0;
}
```

可见，本质就是在`0x1F2`到`0x1F7`这些端口中写入一些信息，等待磁盘就绪，然后用`insl` `outsl`指令完成读写。

在`fs_init`设置了`bitmap`指针后，我们可以把它当作一个位数组，每一个位代表磁盘上的一个块。比如`block_is_free`可以检查`bitmap`中对应的块是否被标记为空闲。

> Exercise 3. 在`fs/fs.c`中以`free_block`为模板实现`alloc_block`，在`bitmap`中找到一个空闲磁盘块，标记为已用，返回其编号。分配完磁盘块后需要立即写回被改变的`bitmap`，以维护文件系统的连贯性。

实现如下：

```
// Search the bitmap for a free block and allocate it.  When you
// allocate a block, immediately flush the changed bitmap block
// to disk.
//
// Return block number allocated on success,
// -E_NO_DISK if we are out of blocks.
//
// Hint: use free_block as an example for manipulating the bitmap.
int

alloc_block(void)
```

```
{
    // The bitmap consists of one or more blocks.  A single bitmap block
    // contains the in-use bits for BLKBITSIZE blocks.  There are
    // super->s_nblocks blocks in the disk altogether.

    // LAB 5: Your code here.
    int blockno, i, j;
    for (i = 0; i < (super->s_nblocks + 31) / 32; ++i)
        if (bitmap[i])
            for (j = 0; j < 32; ++j)
                if (bitmap[i] & (1 << j))
                {
                    blockno = i * 32 + j;
                    if (blockno >= super->s_nblocks)
                        return -E_NO_DISK;
                    bitmap[i] ^= (1 << j);
                    flush_block(bitmap);
                    return blockno;
                }
    return -E_NO_DISK;
}
```

从前向后遍历，通过一些位运算找到并分配一个可用的空闲块。这里一开始比较坑的点是`1`代表可用，`0`代表不可用。这个一开始是如何初始化的呢？可以打开`obj/fs/fs.img`，用十六进制观察。`bitmap`对应的是磁盘块2，从地址`0x00002000`开始：

```
00002000: 0000 0000 0000 0000 0000 0000 00f0 ffff
00002010: ffff ffff ffff ffff ffff ffff ffff ffff
...
```

可见，一开始除了开头的一些块被设为不可用之外其它的块都是空闲的。这个是在编译时由`clean-fs.img`中的内容决定的。

之前的`bc_pgfault()`代码注释中提了一个问题：为什么在读入磁盘内容之后再检查该块是否已被分配？检查是否分配是通过`bitmap`这个数组来完成的，假如触发缺页的正是尝试读这个数组的操作，那么我们就必须在真正读取磁盘内容之后才能完成检查，不然就会陷入无限循环的缺页了。

在`fs/fs.c`中已经实现了一些管理`File`的基本功能。其中重要的函数主要功能如下：

- `block_is_free()`：通过`bitmap`查询某一块是否空闲
- `free_block()`：标记`bitmap`某一块为空闲
- `alloc_block()`：遍历`bitmap`找到并分配一个块
- `fs_init()`：初始化文件系统，包括查询磁盘，初始化块缓存，初始化`super` `bitmap`
- `file_block_walk()`：找到文件内容中的某一磁盘块的指针
- `file_get_block()`：与上类似，进一步找到实际的磁盘块对应的虚拟地址
- `dir_lookup()`：根据文件名在目录中找到文件信息
- `dir_alloc_file()`：在目录下分配一个空闲的文件信息项
- `walk_path()`：从根目录开始，根据路径找到对应的文件和所在目录
- `file_create()`：根据路径创建文件
- `file_open()`：打开文件
- `file_read()`：从文件中读入，可指定`offset` `count`

- `file_write()`：向文件中写入，同上
- `file_free_block()`：将文件中的一块设为空闲
- `file_truncate_blocks()`：截断文件到指定大小
- `file_set_size()`：重新指定文件大小
- `file_flush()`：将文件内容写回磁盘

> Exercise 4. 实现`file_block_walk` `file_get_block`，其功能如上所述。

理解了注释中的提示，实现并不困难：

```
// 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;
   if (filebno < NDIRECT)
   {
       if (ppdiskbno)
           *ppdiskbno = f->f_direct + filebno;
       return 0;
   }
   if (!f->f_indirect)
   {
       if (!alloc)
           return -E_NOT_FOUND;
       int r;
       if ((r = alloc_block()) < 0)
           return r;
       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;
}
```

```
// 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;
    void *addr;
    if ((r = file_block_walk(f, filebno, &pdiskbno, 1)) < 0)
        return r;
    if (!*pdiskbno)
    {
        if ((r = alloc_block()) < 0)
            return r;
        *pdiskbno = r;
        addr = diskaddr(r);
        memset(addr, 0, BLKSIZE);
        flush_block(addr);
    }
    else
        addr = diskaddr(*pdiskbno);
    *blk = (char*)addr;
    return 0;
}
```

更详细准确地解释，在`file_block_walk()`中，要根据`struct File *f`提供的文件信息和`uint32_t filebno`找到指向所需块的那个指针的位置。根据`filebno`的大小，该指针可能位于`f->f_direct`数组中或`f->f_indirect`指向的块中。对于后一种情况，在找不到`f->f_indirect`块且`alloc`为真时需要分配该块。在`file_get_block()`中，先通过`file_block_walk()`找到对应指针，如果还没有实际分配磁盘块则用`alloc_block()`分配并清空磁盘块，然后转成一个虚拟地址。

有了这些必要的文件系统环境功能后，我们就可以让其它环境来通过该环境使用文件系统了。由于权限的设置，其它环境需要利用IPC间接地通过文件系统环境进行操作，以读取文件为例：

```
read -> devfile_read -> fsipc -> ipc_send -> ipc_recv -> serve -> serve_read ->
file_read
|<------------- Regular env ------------>||<------------------ FS env --------------
->|
```

我们重点关注文件系统环境中的内容。文件系统服务器的代码位于`fs/serv.c`，在`serve`函中无限循环地等待接收IPC，然后分发给对应的处理函数，再通过IPC将结果传回。

```
void
serve(void)
```

```
{
    uint32_t req, whom;
    int perm, r;
    void *pg;

    while (1) {
        perm = 0;
        req = ipc_recv((int32_t *) &whom, fsreq, &perm);
        if (debug)
            cprintf("fs req %d from %08x [page %08x: %s]\n" ,
                req, whom, uvpt[PGNUM(fsreq)], fsreq);

        // All requests must contain an argument page
        if (!(perm & PTE_P)) {
            cprintf("Invalid request from %08x: no argument page\n" ,
                whom);
            continue; // just leave it hanging...
        }

        pg = NULL;
        if (req == FSREQ_OPEN) {
            r = serve_open(whom, (struct Fsreq_open*)fsreq, &pg, &perm);
        } else if (req < ARRAY_SIZE(handlers) && handlers[req]) {
            r = handlers[req](whom, fsreq);
        } else {
            cprintf("Invalid request code %d from %08x\n" , req, whom);
            r = -E_INVAL;
        }
        ipc_send(whom, r, pg, perm);
        sys_page_unmap(0, fsreq);
    }
}
```

客户端通过IPC发起请求时，先通过一个整数表明请求编号（类似于系统调用中的调用号），再把请求的参数信息置于共享页中的`union Fsipc`结构中。对于用户端，这个页的地址在`fsipcbuf`，对于服务器则是`fsreq`。在服务器返回结果时也是利用IPC，用整数来表示函数的返回值。对于`FSREQ_READ` `FSREQ_STAT`，还需要返回一些数据，这些数据存储在客户端共享给服务器的那个页中。对于`FSREQ_OPEN`，则要给客户端新共享一个页，表示着该文件描述符。

> Exercise 5. 实现`fs/serv.c`中的`serve_read`。其中主要的工作已经由`fs/fs.c`中的`file_read`完成，`serve_read`需要提供一个接口，可以参考`serve_set_size`。

> Exercise 6. 实现`fs/serv.c`中的`serve_write`以及`lib/file.c`中的`devfile_write`。

首先参看一下已经提供的其它函数的一些代码。`fs/serv.c`中的函数大多涉及到`struct Fsreq_set_size *req`，先看看这个结构体的定义，在`inc/fs.h`中：

```
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;

    // Ensure Fsipc is one page
    char _pad[PGSIZE];
};
```

该结构体占用正好一个页的大小。它是由很多种`struct Fsreq_*`联合形成的，代表着不同的请求类型，但可以共用同样的内存，实际使用时根据请求类型进行选择即可。`fs/serv.c`中的样例`serve_set_size()`中：

```
// Set the size of req->req_fileid to req->req_size bytes, truncating
// or extending the file as necessary.
int
serve_set_size(envid_t envid, struct Fsreq_set_size *req)
{
    struct OpenFile *o;
    int r;

    if (debug)
        cprintf("serve_set_size %08x %08x %08x\n", envid, req->req_fileid, req->req_size);

    // Every file system IPC call has the same general structure.
    // Here's how it goes.

    // First, use openfile_lookup to find the relevant open file.
```

```
    // On failure, return the error code to the client with ipc_send.
    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;

    // Second, call the relevant file system function (from fs/fs.c).
    // On failure, return the error code to the client.
    return file_set_size(o->o_file, req->req_size);
}
```

注释中解释了我们应该如何解释IPC调用，首先使用`openfile_lookup()`来找到相关的文件，然后再调用`fs/fs.c`中具体的文件系统函数来处理请求，错误码将会被返回。然后我们就可以自己写出`serve_read()`和`serve_write()`了：

```
// Read at most ipc->read.req_n bytes from the current seek position
// in ipc->read.req_fileid.  Return the bytes read from the file to
// the caller in ipc->readRet, then update the seek position.  Returns
// the number of bytes successfully read, or < 0 on error.
int
serve_read(envid_t envid, union Fsipc *ipc)
{
    struct Fsreq_read *req = &ipc->read;
    struct Fsret_read *ret = &ipc->readRet;

    if (debug)
        cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid, req->req_n);

    // Lab 5: Your code here:
    struct OpenFile *o;
    int r;
    size_t req_n = req->req_n;
    if (req_n > PGSIZE)
        req_n = PGSIZE;
    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
    if ((r = file_read(o->o_file, ret->ret_buf, req_n, o->o_fd->fd_offset)) < 0)
        return r;
    o->o_fd->fd_offset += r;
    return r;
}


// Write req->req_n bytes from req->req_buf to req_fileid, starting at
// the current seek position, and update the seek position
// accordingly.  Extend the file if necessary.  Returns the number of
// bytes written, or < 0 on error.
int
serve_write(envid_t envid, struct Fsreq_write *req)
{
    if (debug)
        cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid, req->req_n);


    // LAB 5: Your code here.
```

```
    struct OpenFile *o;
    int r;
    size_t req_n = req->req_n;
    if (req_n > PGSIZE - sizeof(int) - sizeof(size_t))
        req_n = PGSIZE - sizeof(int) - sizeof(size_t);
    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
    if ((r = file_write(o->o_file, req->req_buf, req_n, o->o_fd->fd_offset)) < 0)
        return r;
    o->o_fd->fd_offset += r;
    return r;
}
```

需要注意的是我们对于读写长度的限制。这是根据`union Fsipc`中的定义来的，保证不会超出一个页的大小。对于读文件请求，`struct Fsret_read`中只有一个一页大小的`char ret_buf[PGSIZE]`，所以读一次文件最大大小为`PGSIZE`。对于写文件请求，`struct Fsreq_write`中有两个整数，再是`char req_buf[PGSIZE - (sizeof(int) + sizeof(size_t))]`，所以写一次文件最大大小为`PGSIZE - (sizeof(int) + sizeof(size_t))`。我们也可以进一步看看较底层的`file_read()`：

```
// Read count bytes from f into buf, starting from seek position
// offset.  This meant to mimic the standard pread function.
// Returns the number of bytes read, < 0 on error.
ssize_t
file_read(struct File *f, void *buf, size_t count, off_t offset)
{
    int r, bn;
    off_t pos;
    char *blk;

    if (offset >= f->f_size)
        return 0;

    count = MIN(count, f->f_size - offset);

    for (pos = offset; pos < offset + count; ) {
        if ((r = file_get_block(f, pos / BLKSIZE, &blk)) < 0)
            return r;
        bn = MIN(BLKSIZE - pos % BLKSIZE, offset + count - pos);
        memmove(buf, blk + pos % BLKSIZE, bn);
        pos += bn;
        buf += bn;
    }

    return count;
}
```

这是通过我们之前所写的`file_get_block()`来找到对应的磁盘块，然后进行内存内容的复制。这里的`offset`会在外面的包装函数中被更新。在客户端，模仿`lib/file.c`中的`devfile_read()`我们也可以写出`devfile_write()`：

```
// 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;

    if (n > sizeof(fsipcbuf.write.req_buf))
        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;
}
```

保证写的长度不超过要求的最大大小，然后调用`fsipc()`即可。

接下来我们开始考虑如何`spawn`新的进程。在`lib/spawn.c`中已有`spawn()`的代码：

```
// Spawn a child process from a program image loaded from the file system.
// prog: the pathname of the program to run.
// argv: pointer to null-terminated array of pointers to strings,
//   which will be passed to the child as its command-line arguments.
// Returns child envid on success, < 0 on failure.
int
spawn(const char *prog, const char **argv)
{
    unsigned char elf_buf[512];
    struct Trapframe child_tf;
    envid_t child;

    int fd, i, r;
    struct Elf *elf;
    struct Proghdr *ph;
    int perm;

    // This code follows this procedure:
    //
    // ...

    if ((r = open(prog, O_RDONLY)) < 0)

        return r;
```

```
    fd = r;

    // Read elf header
    elf = (struct Elf*) elf_buf;
    if (readn(fd, elf_buf, sizeof(elf_buf)) != sizeof(elf_buf)
        || elf->e_magic != ELF_MAGIC) {
        close(fd);
        cprintf("elf magic %08x want %08x\n", elf->e_magic, ELF_MAGIC);
        return -E_NOT_EXEC;
    }

    // Create new child environment
    if ((r = sys_exofork()) < 0)
        return r;
    child = r;

    // Set up trap frame, including initial stack.
    child_tf = envs[ENVX(child)].env_tf;
    child_tf.tf_eip = elf->e_entry;

    if ((r = init_stack(child, argv, &child_tf.tf_esp)) < 0)
        return r;

    // Set up program segments as defined in ELF header.
    ph = (struct Proghdr*) (elf_buf + elf->e_phoff);
    for (i = 0; i < elf->e_phnum; i++, ph++) {
        if (ph->p_type != ELF_PROG_LOAD)
            continue;
        perm = PTE_P | PTE_U;
        if (ph->p_flags & ELF_PROG_FLAG_WRITE)
            perm |= PTE_W;
        if ((r = map_segment(child, ph->p_va, ph->p_memsz,
                    fd, ph->p_filesz, ph->p_offset, perm)) < 0)
            goto error;
    }
    close(fd);
    fd = -1;

    // Copy shared library state.
    if ((r = copy_shared_pages(child)) < 0)
        panic("copy_shared_pages: %e", r);

    // child_tf.tf_eflags |= FL_IOPL_3;   // devious: see user/faultio.c
    if ((r = sys_env_set_trapframe(child, &child_tf)) < 0)
        panic("sys_env_set_trapframe: %e", r);

    if ((r = sys_env_set_status(child, ENV_RUNNABLE)) < 0)
        panic("sys_env_set_status: %e", r);

    return child;

error:

    sys_env_destroy(child);
```

```
    close(fd);
    return r;
}
```

其过程主要如下：

- 调用`open()`来打开目标程序文件
- 读取ELF文件头并检查
- 调用`sys_exofork()`创建新的空白环境
- 为子环境设置`struct Trapframe child_tf`
- 调用`init_stack()`，为子环境初始化栈
- 在子环境地址空间中映射程序中所有`ELF_PROG_LOAD`类型的段，需要根据段指定的类型来完成映射的权限等设置
- 调用`sys_env_set_trapframe()`设置子环境初始的`eip` `esp`值
- 调用`sys_env_set_status()`允许子环境的运行

其中的`sys_env_set_trapframe()`需要我们来实现。

> Exercise 7. `spawn`需要`sys_env_set_trapframe`来初始化子环境的状态。在`kern/syscall.c`中实现它。

和另一个系统调用`sys_env_set_status()`类似，这也是在`struct Env`中进行修改：

```
// Set envid's trap frame to 'tf'.
// tf is modified to make sure that user environments always run at code
// protection level 3 (CPL 3) with interrupts enabled.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
    // LAB 5: Your code here.
    // Remember to check whether the user has supplied us with a good
    // address!
    struct Env *e = NULL;
    int r = envid2env(envid, &e, 1);
    if (r < 0) return r;
    user_mem_assert(e, tf, sizeof(struct Trapframe), PTE_U);
    e->env_tf = *tf;
    e->env_tf.tf_cs = GD_UT | 3;
    e->env_tf.tf_eflags |= FL_IF;
    return 0;
}
```

需要注意的是利用`user_mem_assert()`检查该地址是否有效，然后要保证用户环境中的代码运行于用户态，中断已打开。

JOS中的文件描述符可能代表着真实文件、管道、控制台I/O等，每种类型都有一个对应的`struct Dev`，其中有着指向实现具体读写操作函数的指针，在`inc/fd.h`中：

```
// Per-device-class file descriptor operations
struct Dev {
    int dev_id;
    const char *dev_name;
    ssize_t (*dev_read)(struct Fd *fd, void *buf, size_t len);
    ssize_t (*dev_write)(struct Fd *fd, const void *buf, size_t len);
    int (*dev_close)(struct Fd *fd);
    int (*dev_stat)(struct Fd *fd, struct Stat *stat);
    int (*dev_trunc)(struct Fd *fd, off_t length);
};
```

在`lib/fd.c`中，定义了三种类型的文件：

```
static struct Dev *devtab[] =
{
    &devfile,
    &devpipe,
    &devcons,
    0
};
```

其余对于文件的操作，基本上都只是根据`struct Dev`来选择对应的更低一层的函数。另一方面，`lib/fd.c`中在每个非文件系统的用户环境中维护了一个文件描述符表，地址从`FDTABLE = 0xD0000000`开始，最多支持`MAXFD = 32`个文件描述符，每个文件描述符可能占用一页大小的内存。不过根据`struct Fd`的定义，其实一页可以存下更多的文件描述符。每个文件描述符还可以有一个数据页，其地址起始于`FILEDATA = FDTABLE + MAXFD*PGSIZE`。文件描述符的定义如下：

```
struct FdFile {
    int id;
};

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

包括一个表示类型的数字，当前读写偏移量，打开方式等。

我们希望在`fork`、`spawn`之间共享文件描述符状态，但是这些内容是存放在用户内存空间中的。在COW机制中，这些状态将会被复制而不是共享。我们将修改`fork`来实现不同环境之间对于操作系统库所使用内存的共享，具体而言，对于标有`PTE_SHARE`位的内存内容，它们的PTE应当直接被复制，这样父子环境就可以共享这部分内存了。

> Exercise 8. 修改`lib/fork.c`中的`duppage`。如果PTE中`PTE_SHARE`位被设置，则直接复制映射。类似地，实现`lib/spawn.c`中的`copy_shared_pages`，遍历所有PTE，复制共享内存的映射。

我们修改后的`duppage()`如下：

```
//
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    void *va = (void*)(pn * PGSIZE);
    int perm = uvpt[pn] & PTE_SYSCALL;
    if (perm & PTE_SHARE)
    {
        r = sys_page_map(0, va, envid, va, perm);
        if (r < 0) panic("duppage: %e!\n", r);
        return 0;
    }
    if (perm & (PTE_W | PTE_COW))
    {
        perm &= ~PTE_W;
        perm |= PTE_COW;
    }
    r = sys_page_map(0, va, envid, va, perm);
    if (r < 0) panic("duppage: %e!\n", r);
    r = sys_page_map(0, va, 0, va, perm);
    if (r < 0) panic("duppage: %e!\n", r);
    return 0;
}
```

如果`perm & PTE_SHARE`成立，则直接复制该映射。Exercise中提示了需要用`PTE_SYSCALL`来设置权限位。`copy_shared_pages()`也是一样：

```
// Copy the mappings for shared pages into the child address space.
static int
copy_shared_pages(envid_t child)
{
    // LAB 5: Your code here.
    int r;
    uint32_t va;
    for (va = 0; va < USTACKTOP; va += PGSIZE)
        if ((uvpd[PDX(va)] & PTE_P) && (uvpt[PGNUM(va)] & PTE_P) &&
            (uvpt[PGNUM(va)] & PTE_SHARE))
```

```
        {
            r = sys_page_map(0, (void*)va, child, (void*)va, uvpt[PGNUM(va)] &
PTE_SYSCALL);
            if (r < 0) panic("copy_shared_pages: %e!\n", r);
        }
    return 0;
}
```

最后我们来看JOS中的shell。我们在JOS中输入有两种方式，一个是QEMU的CGA图形窗口，一个是控制台，对于两者的输入应该是等同的。

> Exercise 9. 在`kern/trap.c`中调用`kbd_intr`处理`IRQ_OFFSET+IRQ_KBD`，调用`serial_intr`处理`IRQ_OFFSET+IRQ_SERIAL`。

在`trap_dispatch()`中加上两个情况：

```
    // Handle keyboard and serial interrupts.
    // LAB 5: Your code here.
    case IRQ_OFFSET + IRQ_KBD:
        kbd_intr();
        break;

    case IRQ_OFFSET + IRQ_SERIAL:
        serial_intr();
        break;
```

此时已经可以运行用户程序`icode`，其中会`spawn()`出程序`init`，对控制台I/O进行初始化，然后`spawn()`得到shell程序`sh`。可以运行`echo` `cat`等基本的指令。

> Exercise 10. 目前shell还不支持I/O重定向，在`user/sh.c`中为`<`添加实现。

`sh.c`中已经实现了输出重定向，模仿它可以写出：

```
        case '<':   // Input redirection
            // Grab the filename from the argument list
            if (gettoken(0, &t) != 'w') {
                cprintf("syntax error: < not followed by word\n");
                exit();
            }
            // Open 't' for reading as file descriptor 0
            // (which environments use as standard input).
            // We can't open a file onto a particular descriptor,
            // so open the file as 'fd',
            // then check whether 'fd' is 0.
            // If not, dup 'fd' onto file descriptor 0,
            // then close the original 'fd'.

            // LAB 5: Your code here.
            if ((fd = open(t, O_RDONLY)) < 0) {
                cprintf("open %s for read: %e", t, fd);
                exit();
            }
            if (fd != 0) {
```

```
                dup(fd, 0);
                close(fd);
            }
            break;
```

打开文件，用`dup`将其复制给标准输入，再关闭即可。

最后在初次`make grade`时发现无法通过`faultio`这个测试，究其原因是创建非文件系统的子环境时赋予了`FL_IOPL_3`的权限，再仔细探寻一番发现是在原已实现的`spawn()`中有一句：

```
    child_tf.tf_eflags |= FL_IOPL_3;   // devious: see user/faultio.c
```

这显然没有道理，注释掉之后即可通过该测试。此外还有运行shell的测试总是会超时，不影响分数但是原因不明，猜测可能是输出较长匹配太慢。

> Challenge! 目前的块缓存没有替换策略。一旦磁盘中的某个块进入了内存，就不会被移出。为我们的块缓存添加换出机制。使用由硬件自动设置的页表中的`PTE_A`访问位，无需修改每一处访问磁盘映射区域的代码就可以大概确定磁盘块的使用情况。要小心被写过的块。

在不修改整个内存管理大框架的前提下，我们可以为文件系统环境设置一个可使用内存上限，代表它在任意时刻所能实际占用的内存大小。尽管文件系统环境实际上可以访问至多3G大的整个磁盘，但它并不需要占用这么大的内存。比如我们可以设置同时在内存中的磁盘块数量不超过某个值，达到上限后若要再将块加入缓存，则要先将之前存在的一些块换出。

我们使用时钟替换策略：循环遍历所有已在内存中的块，检查`PTE_A`位判断最近是否有过访问。如果有过访问，将`PTE_A`位清除；如果没有访问，则将该块换出。换出时，先检查`PTE_D`位判断该块是否被写过，如果是的话调用`flush_block()`写回磁盘，然后调用`sys_page_unmap()`解除该页映射。

具体的实现如下。首先，我们在`fs/fs.h`中补充以下声明：

```
#define NBC         1024
uint32_t bcmap[NBC];
int nbc, ibc;
```

我们允许通过`bc_pgfault()`进入块缓存的磁盘块最大同时只能有`NBC`个，在这里设置为1K，其总大小为4M，块缓存中的块所对应的虚拟地址存在数组`bcmap`中，数组元素数量当前为`nbc`，上限为`NBC`，这里恰好占用一页。`cbc`将在后面的寻找换出块中被用到。

在`fs/bc.c`中的`bc_pgfault()`中，正式分配内存页之前我们先检查当前的`nbc`，如果还未达到上限则将其加一，否则寻找一个最近没有被访问过的块换出，具体情况已在前面叙述过：

```
    if (nbc < NBC)
        bcmap[nbc++] = (uint32_t)addr;
    else
    {
        void *eaddr = NULL;
        int r;
        while (1)
        {
            eaddr = (void*)(bcmap[ibc]);

            if (uvpt[PGNUM(eaddr)] & PTE_A)
```

```
            {
                if ((r = sys_page_map(0, eaddr, 0, eaddr, uvpt[PGNUM(eaddr)]
                                      & PTE_SYSCALL)) < 0)
                    panic("bc_pgfault: %e!\n", r);
                if (++ibc == NBC) ibc = 0;
            }
            else
            {
                if (uvpt[PGNUM(eaddr)] & PTE_D)
                    flush_block(eaddr);
                sys_page_unmap(0, eaddr);
                bcmap[ibc] = (uint32_t)addr;
                break;
            }
        }
    }
```

为了保证`bcmap`正常工作，要注释掉`bc_init()`中的`check_bc()`，里面出现了对超级块取消映射的操作。测试时，我们把`NBC`设置为一个较小的值（比如8）以便观察到频繁的替换。在`bc_pgfault()`中的上述过程之后，我们输出一下当前块缓存的调试信息：

```
static void
print_bcmap_info()
{
    cprintf("\n");
    cprintf("nbc: %d, ibc: %d\n", nbc, ibc);
    int i;
    for (i = 0; i < nbc; ++i)
        cprintf("%p ", bcmap[i]);
    cprintf("\n");
}
```

然后用用户程序`icode`来测试，结果如下：

```
...

FS is running
FS can do I/O
Device 1 presence: 1

nbc: 1, ibc: 0
0x10001000
superblock is good

nbc: 2, ibc: 0
0x10001000 0x10002000
bitmap is good
alloc_block is good

nbc: 3, ibc: 0
icode startup
icode: open /motd
```

```
0x10001000 0x10002000 0x1006a000

nbc: 4, ibc: 0
0x10001000 0x10002000 0x1006a000 0x1006b000
file_open is good

nbc: 5, ibc: 0
0x10001000 0x10002000 0x1006a000 0x1006b000 0x10003000
file_get_block is good
file_flush is good
file_truncate is good
file rewrite is good
icode: read /motd

nbc: 6, ibc: 0
0x10001000 0x10002000 0x1006a000 0x1006b000 0x10003000 0x10004000
This is /motd, the message of the day.

Welcome to the JOS kernel, now with a file system!

icode: close /motd
icode: spawn /init

nbc: 7, ibc: 0
0x10001000 0x10002000 0x1006a000 0x1006b000 0x10003000 0x10004000 0x10009000

nbc: 8, ibc: 0
0x10001000 0x10002000 0x1006a000 0x1006b000 0x10003000 0x10004000 0x10009000
0x1000a000

nbc: 8, ibc: 4
0x10001000 0x10002000 0x1006a000 0x1006b000 0x1000b000 0x10004000 0x10009000
0x1000a000

nbc: 8, ibc: 3
0x10001000 0x10002000 0x1006a000 0x1000c000 0x1000b000 0x10004000 0x10009000
0x1000a000

nbc: 8, ibc: 4
0x10001000 0x10002000 0x1006a000 0x1000c000 0x1000d000 0x10004000 0x10009000
0x1000a000

nbc: 8, ibc: 5
0x10001000 0x10002000 0x1006a000 0x1000c000 0x1000d000 0x1000e000 0x10009000
0x1000a000
icode: exiting
init: running
init: data seems okay
init: bss seems okay
init: args: 'init' 'initarg1' 'initarg2'
init: running sh
init: starting sh
```

```
nbc: 8, ibc: 6
0x10001000 0x10002000 0x1006a000 0x1000c000 0x1000d000 0x1000e000 0x1006b000
0x1000a000

nbc: 8, ibc: 7
0x10001000 0x10002000 0x1006a000 0x1000c000 0x1000d000 0x1000e000 0x1006b000
0x1003f000

nbc: 8, ibc: 3
0x10001000 0x10002000 0x1006a000 0x10040000 0x1000d000 0x1000e000 0x1006b000
0x1003f000

nbc: 8, ibc: 4
0x10001000 0x10002000 0x1006a000 0x10040000 0x10041000 0x1000e000 0x1006b000
0x1003f000

nbc: 8, ibc: 5
0x10001000 0x10002000 0x1006a000 0x10040000 0x10041000 0x10042000 0x1006b000
0x1003f000

nbc: 8, ibc: 7
0x10001000 0x10002000 0x1006a000 0x10040000 0x10041000 0x10042000 0x1006b000
0x10043000

nbc: 8, ibc: 2
0x10001000 0x10002000 0x10044000 0x10040000 0x10041000 0x10042000 0x1006b000
0x10043000
$
```

可见，`0x10001000` `0x10002000`所代表的`super` `bitmap`重要的磁盘块是没有被换出的，而其它的磁盘块如果没有被用到就换出了，的确是有效的替换策略。