LAB 5

5120809022

1. 主要思路

这个 lab 主要是建立一个简单的文件系统。首先把文件系统当作一个 environment，实现文件系统对 disk 的操作。然后通过 ipc 共享内存的机制让其他进程发送请求给 fs，fs 实现相应的操作，这样就是先其他进程访问 disk 的操作，另外实现了一写额外的功能，比如 cache，spawn 等

2. Exercise 1

Create environment 的时候加入比普通进程高一等的权限，设置一下 bit 位就可以了

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

3. Exercise 2

这两个函数，一个是从硬盘读一个 sector 的数据到内存，另一个数 flush 写回硬盘，当然实际操作都是操作整个 sector 所在的 block。

bc_pgfalut():

```
    addr = ROUNDDOWN(addr, PGSIZE);
    r = sys_page_alloc(0, addr, PTE_W | PTE_U | PTE_P);
    if (r < 0)
        panic("pc_pgfault: can't alloc page\n");
    r = ide_read(blockno*BLKSECTS, addr, BLKSECTS);
```

Flush_block():

```
addr = ROUNDDOWN(addr, PGSIZE);
if (va_is_mapped(addr) && va_is_dirty(addr)) {
        r=ide_write(blockno*BLKSECTS, addr, BLKSECTS);
        if ( r < 0)
                panic("flush_block: ide_write error");
        r=sys_page_map(0, addr, 0, addr, PTE_SYSCALL);
        if ( r < 0)
                panic("flush_block: sys_page_map error");
}
```

4. Exercise 3

很简单，按照 hint 来。

```
int  i=0;
 for (; i < super->s_nblocks; i++) {
        if (block_is_free(i)) {
                bitmap[i/32] &= ~((int)1<<(i%32));
                flush_block(bitmap);
                return i;
        }
 }
 return -E_NO_DISK;
```

5. Exercise 4

感觉没必要把这两个函数分开，不过注释写的很详细，基本按照注释来。

首先是 file_block_walk，根据给定的 blockno 返回一个文件对应的 blockno 的位置，如果在 indirect 的位置并且没有初始化，初始化位置了的话就 alloc 一个给 indirect 然后返回.

```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool
alloc)
{
	// LAB 5: Your code here.
	//panic("file_block_walk not implemented");
	if(filebno >= NDIRECT + NINDIRECT)
		return -E_INVAL;
	if (filebno < NDIRECT)
	{
		*ppdiskbno = &(f->f_direct[filebno]);
		return 0;
	}
	if(f->f_indirect == 0)
	{
		if(alloc == 0)
			return -E_NOT_FOUND;
		int r = alloc_block();
		if(r < 0)
			return -E_NO_DISK;
		memset(diskaddr(r), 0, BLKSIZE);
		f->f_indirect = r;
		flush_block(diskaddr(r));
```

```
        }
        uint32_t* indirect = diskaddr(f->f_indirect);
        *ppdiskbno = &(indirect[filebno-NDIRECT]);
        return 0;
}
```

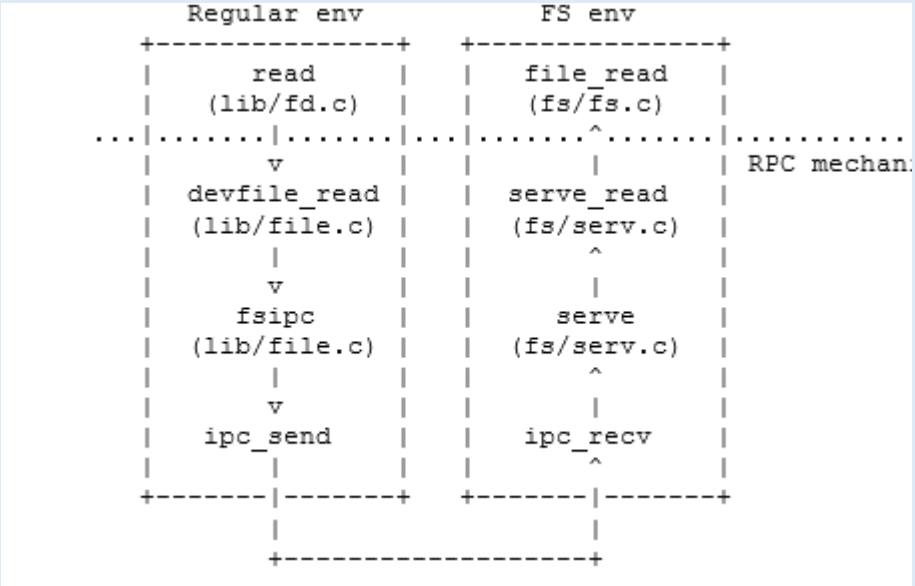
file_get_block 这个函数把上面的函数包装下，把位置传给 char **blk

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
        // LAB 5: Your code here.
        //panic("file_get_block not implemented");
        uint32_t * ppdiskbno = NULL;
        if(filebno >= NDIRECT + NINDIRECT)
                return -E_INVAL;

        int r = file_block_walk(f, filebno, &ppdiskbno, 1);
        if(r < 0)
                return r;

        // need alloc block point's block
        if(*ppdiskbno == 0)
        {
```

```
                r = alloc_block();
                if(r < 0)
                        return -E_NO_DISK;
                *ppdiskbno = r;
                memset(diskaddr(r), 0, BLKSIZE);
                flush_block(diskaddr(r));
        }
        *blk = diskaddr(*ppdiskbno);
        return 0;
}
```

6. Exercise 5-6

IPC 让其他进程调用 fs 进程，共享 memory，写到 memory 交流就可以了。机制如下：



代码如下

```
//server 端
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);

        // Look up the file id, read the bytes into 'ret', and update
        // the seek position.  Be careful if req->req_n > PGSIZE
        // (remember that read is always allowed to return fewer bytes
        // than requested).  Also, be careful because ipc is a union,
        // so filling in ret will overwrite req.
        //
        // Hint: Use file_read.
        // Hint: The seek position is stored in the struct Fd.
        // LAB 5: Your code here
        //panic("serve_read not implemented");
        struct OpenFile *f;
        int r;
```

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

	r = file_read(f->o_file, ret->ret_buf, MIN(req->req_n, PGSIZE),
		f->o_fd->fd_offset);
	if (r < 0)
		return r;
	f->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.
	//panic("serve_write not implemented");

	struct OpenFile *f;
	int r;
	r = openfile_lookup(envid, req->req_fileid, &f);
	if (r < 0)
		return r;
	r = file_write(f->o_file, req->req_buf, req->req_n,
		f->o_fd->fd_offset);
	if (r < 0)
		return r;
	f->o_fd->fd_offset += r;
	return r;
}
//client 端
static ssize_t
devfile_read(struct Fd *fd, void *buf, size_t n)
{
	// Make an FSREQ_READ request to the file system server after
	// filling fsipcbuf.read with the request arguments.  The
	// bytes read will be written back to fsipcbuf by the file
	// system server.
```

```
	// LAB 5: Your code here
	//panic("devfile_read not implemented");
	int r;
	fsipcbuf.read.req_fileid = fd->fd_file.id;
	fsipcbuf.read.req_n = n;
	if ((r = fsipc(FSREQ_READ, NULL)) < 0)
		return r;
	memmove(buf, &fsipcbuf, r);
	return r;
}


// 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
        //panic("devfile_write not implemented");
        int r;
        fsipcbuf.write.req_fileid = fd->fd_file.id;
        void *p =(void*) buf;
        while (n) {
                fsipcbuf.write.req_n =
                MIN(n, sizeof(fsipcbuf.write.req_buf));
                memmove(fsipcbuf.write.req_buf, buf, fsipcbuf.write.req_n);
                r = fsipc(FSREQ_WRITE, NULL);
                if (r < 0)
                        return r;
                n -= r;
                p += r;
        }
        return (p-buf);
}
```

7. Exercise 7

打开一个文件，逻辑上和上面一样，只是用户进程要获取 fd.

```
struct Fd *fd;
int r;
if (strlen(path) >= MAXPATHLEN)
        return -E_BAD_PATH;
r = fd_alloc(&fd);
if (r < 0)
        return r;
strcpy(fsipcbuf.open.req_path, path);
fsipcbuf.open.req_omode = mode;
r = fsipc(FSREQ_OPEN, fd);
if (r < 0) {
        fd_close(fd, 0);
        return r;
}
return fd2num(fd);
```

8. Exercise 8

系统调用。先设置一下入口，再实现响应函数。

```
struct Env* e;
int r = envid2env(envid, &e, 1);
if (r < 0)
        return -E_BAD_ENV;
e->env_tf = *tf;
e->env_tf.tf_cs |= 3;
e->env_tf.tf_eflags |= FL_IF;
return 0;
```