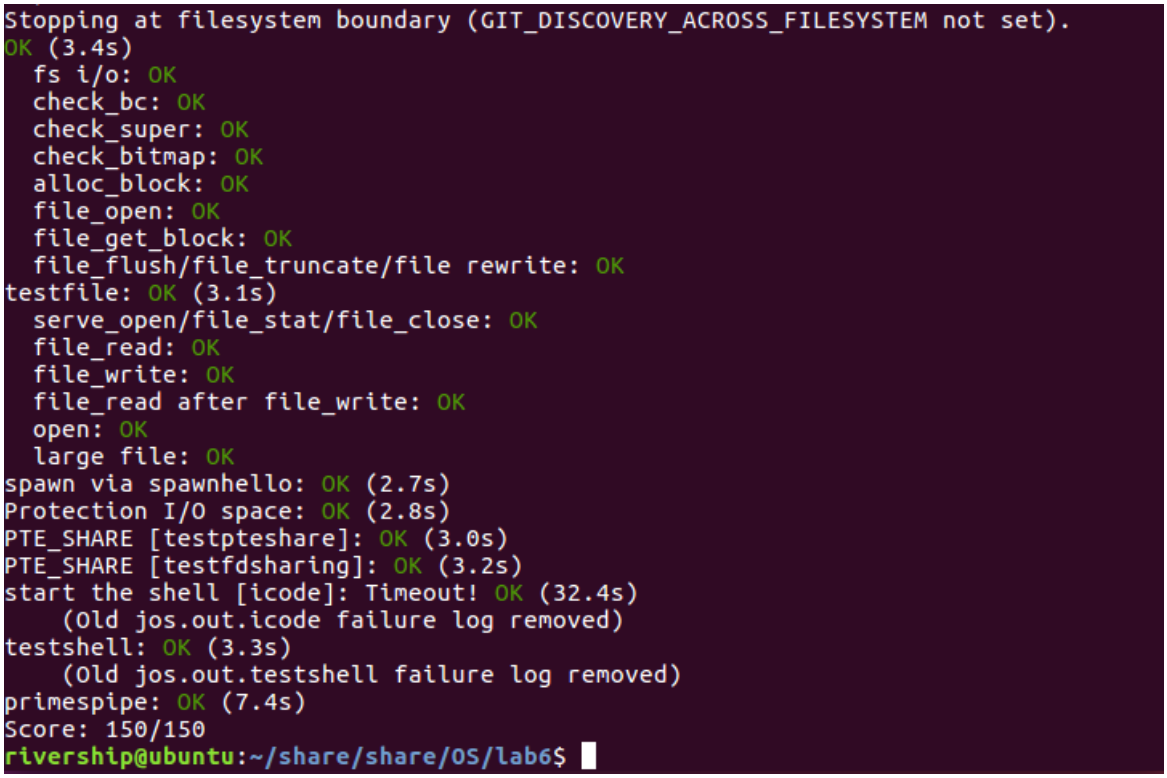
fduos_lab6: File system

18300240005 罗铭源

make grade截图如下



A.代码实现

exercise 1

file system也是一个用户 env，不过其具有较高的IO privilege（IOPL）。IOPL保存在eflags寄存器中，处理器通过读取eflags来确定env是否有IO权限。本exercise要求在创建file system env时，设置其IOPL 。

只需要在 `env_create()` 判断type，并且设置env trapframe中eflags的 IOPL bit 即可

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

exercise 2

file system 将内容视作disk block的缓存，每当读写一个disk block时，file system需要先将其调入自己的地址空间，再进行读写操作。该过程通过缺页处理完成，当file system尝试读/写一个块时，其会访问内存中该块的缓存地址（每个块唯一地缓存在一个内存中的虚拟地址处），若发生缺页，说明该块未被调入内存，于是 file system 调用缺页处理函数，将该块调入内存。

bc_pgfault() 即为完成上述功能的缺页处理函数，本exercise需要完成将block调入内存的功能。当调入一个block到指定的内存页面时，需要：

- 在block对应的内存地址处，申请一个page
- 通过ide_read()将位于磁盘上的block读入指定的内存地址

- ide_read()以sector为单位，这里读取一个block相当于读取 BLKSECTS 个sector
- ide_read()需要传入sector number，这里需要将 block number * BLKSECTS

```
// LAB 5: you code here:
void * aligned_addr = ROUNDDOWN(addr,PGSIZE);
// 申请一个page
if((r=sys_page_alloc(thisenv->env_id,aligned_addr,PTE_W|PTE_P|PTE_U))<0){
    panic("page_alloc in bc_pgfault: %e",r);
}
// 通过ide_read()将位于磁盘上的block读入指定的内存地址
if((r=ide_read(blockno*BLKSECTS,aligned_addr,BLKSECTS))<0)
    panic("in bc_pgfault, ide_read: %e",r);
```

flush_block() 用来将一个block在内存中的缓存写回磁盘。进行写回需要满足两个条件：

- 该内存page缓存了一个block，在file system的地址空间中，可以检查页表项中是否存在映射来确定该点
- block缓存被写过，可以检查页表项中的 PTE_D （dirtiy）。

本excercise要求完成判断是否需要写回，并且进行写回的功能，如下

```
// LAB 5: Your code here.
if(va_is_mapped(va) && va_is_dirty(va)){
    // 满足条件， 通过ide_write()写回
    if((r = ide_write(blockno*BLKSECTS,va,BLKSECTS))<0)
        panic("in flush_block, ide_write: %e",r);
    // 写回后，将block缓存重新设置为未修改过
    if((r=sys_page_map(0,va,0,va,uvpt[PGNUM(addr)] & PTE_SYSCALL))<0)
        panic("in flush_block, sys_page_map: %e",r);
}
```

exercise 3

file system采用bitmap来跟踪block的使用情况，0代表使用中，1代表未使用。alloc_block() 就是通过bitmap找到第一个free block，将该bit置0，并返回找到的块号。

```
// LAB 5: Your code here.
//panic("alloc_block not implemented");
int i,j;
for(i=0; i < (super->s_nblocks); i+=32){
    if(bitmap[i/32]){
        for(j=0;j<32;j++){
            if(bitmap[i/32] & (1<<j)){  // 找到了free block
                // 通过异或置该bit为0
                bitmap[i/32] ^= 1<<j;
                flush_block(bitmap);
                return i+j;
            }
        }
    }
}
return -E_NO_DISK;
```

exercise 4

对于一个文件，存在一个逻辑块号到物理块号的映射（就像虚拟地址到物理地址的映射），访问文件第 i 块时，访问的是逻辑块，需要经过映射转换。映射关系保存在File结构中（通过 f_direct 与 f_indirect），每个逻辑块号对应一个 4bytes 的表项，表项中保存映射的物理块号。

file_block_walk()，在给定文件逻辑块号的情况下，返回映射表项的地址。注意:

- 表项可能位于 f_direct 中，也可能位于 f_indirect 中。对于位于 f_indirect 中的表项，若 f_indirect 块未申请，file_block_walk 需要根据参数决定是否需要申请一个 f_indirect 块。
- file_block_walk 返回的是表项的地址，而不是表项的内容，这样调用者可以通过地址来读写表项

```
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");
    int r=0;
    if(!f)
        panic("in file_block_walk, empty file.");
    // 表项位于 f_direct 中
    if(filebno<NDIRECT){
        *ppdiskbno = &(f->f_direct[filebno]);
    }
    // 表项位于 f_indirect 中
    else if(filebno < NDIRECT+NINDIRECT){
        // f_indirect 存在
        if( f->f_indirect){
            *ppdiskbno = &((uint32_t*)(diskaddr(f->f_indirect)))[filebno-
NDIRECT];
        }
        else{
            // f_indirect 不存在，根据 alloc 决定是否申请一个块作为 f_indirect
            if(alloc){
                if((r = alloc_block())<0)
                    return r;
                f->f_indirect = r;
                // 申请出的block置0，并写回磁盘
                memset(diskaddr(r), 0, BLKSIZE);
                flush_block(diskaddr(r));
                *ppdiskbno = &((uint32_t*)(diskaddr(f->f_indirect)))[filebno-
NDIRECT];
            }
            else{
                return -E_NOT_FOUND;
            }
        }
    }
    else{
        return -E_INVAL;
    }
    return 0;
}
```

file_get_block()，给定文件以及文件的逻辑块号，返回物理块在内存中的地址。实现只需要通过 file_walk_block() 获取映射表项的地址，然后根据表项的内容进行返回，需要:

- 若表项不为空，说明对应的物理block存在，直接返回

- 若表项为空，则需要申请一个物理block，将逻辑块映射到该物理block上（通过修改表项），再返回

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    // LAB 5: Your code here.
    //panic("file_get_block not implemented");
    int r;
    uint32_t* pdiskno=NULL;
    assert(blk);
    if((r=file_block_walk(f,filebno,&pdiskno,1))<0){
        return r;
    }
    // 映射表项存在
    if(*pdiskno){
        *blk = (char*) diskaddr(*pdiskno);
    }
    // 映射表项不存在
    else{
        // 申请一个block
        r = alloc_block();
        if(r<0)
            return r;
        memset(diskaddr(r), 0, BLKSIZE);
        flush_block(diskaddr(r));
        // 设置表项，映射到新申请的block上
        *pdiskno = r;
        // 设置blk指针
        *blk = (char*) diskaddr(r);
    }
    return 0;
}
```

exercise 5

当用户env需要读文件时，其通过RPC向file system发起请求，于是操作系统调用serve_read()，将文件内容写入一个用户env与file system可以共同写的page中（对于用户，该page为fsipcbuf；对于file system, 该page为fsreq）

本exercise要求实现file system读取文件的操作 serve_read()：

- 根据fileid得到目标文件的OpenFile结构，通过调用openfile_lookup()实现
- 随后调用file_read()，向fsreq所在页面写入读取的内容
- 最后更新当前读取位置

```
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:
```

```
    // 从id得到openfile
    struct OpenFile* o;
    int r;
    // 获取OpenFile结构
    if ((r=openfile_lookup(envid,req->req_fileid,&o))<0){
        return r;
    }
    // 读取文件内容，写入buf，该buf为用户与file system共同写
    if((r=file_read(o->o_file, ret->ret_buf, req->req_n, o->o_fd->fd_offset))<0)
{
        return r;
    }
    // 更新当前读取位置
    o->o_fd->fd_offset += r;
    return r;
}
```

exercise 6

本exercise要求实现file system写文件的操作 serve_write() 与用户发出写请求的函数 devfile_write()

serve_write()需要：

- 根据fileid得到目标文件的OpenFile结构，通过调用openfile_lookup()实现
- 随后调用file_write()，向fsreq所在页面写入读取的内容
    - file_write()在需要时会扩展文件
- 更新当前写入读取位置

```
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* o;
    int r;
    // 获取OpenFile
    if((r=openfile_lookup(envid,req->req_fileid,&o))<0){
        return r;
    }
    // 将buf中的内容写入文件
    if((r=file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset))
<0){
        return r;
    }
    // 更新当前写位置
    o->o_fd->fd_offset += r;
    return r;
}
```

devfile_write() 通过fsipc发出RPC请求给file system，写请求的参数保存在 fsipcbuf 中由fsipc发送，此处即设置好写请求的参数以及写入的内容，需要：

- 设置写入的目标文件的file id
- 设置写入的字节数
- 将写入的内容复制到fsipcbuf中

```
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
    // LAB 5: Your code here
    // panic("devfile_write not implemented");
    int r;
    // 设置fsipcbuf
    // 目标文件id
    fsipcbuf.write.req_fileid = fd->fd_file.id;
    // 写入字节数
    fsipcbuf.write.req_n = n;
    // 复制写入内容到 fsipcbuf 中
    memmove(fsipcbuf.write.req_buf, buf, n);
    if((r=fsipc(FSREQ_WRITE,NULL))<0){
        return r;
    }
    assert(r <= n);
    return r;
}
```

exercise 7

为 spawn 添加一个系统调用 sys_env_set_trapframe()，用以设置子进程的进程状态。

```
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!
    // panic("sys_env_set_trapframe not implemented");
    int r;
    struct Env* e;
    if((r=envid2env(envid,&e,1))<0){
        return r;
    };
    if(tf)
        e->env_tf = *tf;
    return 0;
}
```

还要添加 syscall() 中对 sys_env_set_trapframe() 的dispatch

exercise 10

在shell中添加 IO 重定向中的标准输入重定向 '<'，仿造后面的标准输出重定向即可

```
// LAB 5: Your code here.
// 打开重定向的目标文件
if((fd=open(t,O_RDONLY))<0){
    cprintf("open %s for read: %e",t,fd);
    exit();
}
// 重定向到标准输入
if(fd!=0){
    dup(fd,0);
    close(fd);
}
```

B.问题回答

(1) 请回答Exercise 1后的Question 1

不需要。设置之后的eflags保存在 env_tf 的中，每当进程切换时，内核会为用户程序进行上下文切换，于是不同的进程会装载各自的 env_tf 中的 tf_eflags 到 eflags 中，而这个过程中，原进程的eflags 会被保存到 env_tf 中，并且在下一次唤醒该进程前不会被修改。

(2).详细描述 JOS 中文件存储的结构、打开文件的过程以及往文件中写入数据的过程

a.文件存储结构：struct File

```
struct File {
    char f_name[MAXNAMELEN];
    off_t f_size;
    uint32_t f_type;

    uint32_t f_direct[NDIRECT];
    uint32_t f_indirect;

    uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4];
} __attribute__((packed));
```

- f_type指示了该File结构代表的是一个目录还是文件
- f_name为目录或者文件的名字，有一个最大长度限制
- f_size为目录或文件的大小，其中：
  - 文件大小即文件包含的字节数
  - 目录下可以有子目录和文件，每个子目录或文件都对应着一个struct File。目录需要保存这些struct File，于是目录的f_size就是 n*sizeof(struct File)，其中n为子目录和本目录下文件的个数之和
- f_direct[NDIRECT]，目录或文件的数据，即子目录（文件）的File结构或文件的字节数据，需要保存在disk上，保存的单位是block=4096bytes。f_direct保存的是存储用的前 NDIRECT 个block的块号。
- f_indirect，当文件（目录）的数据不能用 NDIRECT 个block存储时，需要更多的块进行存储，于是就需要一个额外的block来保存这些存储块的块号。f_indirect为该额外块的块号。
- f_pad，填充对齐用，使 sizeof struct File == 256

b.文件打开过程

文件系统打开文件时，需要：

- 为打开的文件创建一个OpenFile结构
  - OpenFile结构如下;

    ```
    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
    };
    ```

    包含了打开文件的id，打开文件的File结构，打开模式，文件描述符Fd
  - 文件系统通过OpenFile结构来维护和操作一个打开的文件，如o_fd中就存有当前文件读写的指针，o_file中有该文件对应的block，也就是文件的数据。fileid可以用来一一确定一个OpenFile结构，方便查找。
- 检查打开文件的模式，确定是否需要创建新文件，或者是否需要truncate。这一步保证了后续打开文件时，file_open()能够找到对应的文件，并且有期望的size。
  - 创建新文件可以使用 file_create(path, &f)
    - 该函数在path指定的对应目录下创建一个新的 struct File，并将其指针存放在 f 中
  - 进行truncate可以使用 file_set_size()，即设置File结构中的文件大小
- 打开文件，即找到对应目录下的文件的File结构，并且令OpenFile结构中的o_file指向它。还要设置OpenFile结构中的 o_fd ，即一个Fd结构，来保存打开文件的信息，包括fileid，当前读写指针等
  - 使用 file_open(path,&f)，可以打开path指定的文件，并将其File结构指针存在 f 中。

c.向文件中写入数据的过程：

写入文件时，需要：

- file system获得目标文件的OpenFile结构
  - 通过 openfile_lookup()，根据 fileid 来获得对应的OpenFile结构
- 调用file_write()进行写文件
  - file_write()从当前的写指针开始，以block为单位进行写入。对于每个block，通过file_get_block()获得它在block cache（即内存）中的地址（虚拟地址），随后写入该虚拟地址。若该block不在cache中，file system的缺页处理函数 bc_pgfault() 会将该block调入内存。
  - file_write()在必要时会扩展文件，即为文件申请新的block。当通过file_get_block()访问文件的第 i 个逻辑块、但该逻辑块还未映射一个物理block时，file_get_block()会申请一个block，并将其映射到File的第i个逻辑块上。于是可以利用file_get_block()来扩展文件的block。
- 更新OpenFile结构的相关信息
  - 维护offset，即当前写入的位置

(3) 对于此JOS，一个磁盘有多少个扇区?

```
#define SECTSIZE    512
#define DISKSIZE    0xC0000000
```

在/fs/fs.h中有如上宏，因此扇区数量最多为 0xC0000000 / 512 = 6291456 个

(4) 请详细阐述，JOS中superblock的概念，以及superblock的布局和结构

文件系统需要将一些描述文件系统的元数据保存在容易访问块中，如block size，sector size，根目录，文件系统上次挂载的时间以及上一次检查出错的时间，等等。保存上述信息的块被称为superblock

JOS中只有一块superblock，**位于block1**。JOS中superblock结构如下

```
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
};
```

其中：

- s_magic = 'J\0S'
- s_nblocks表示磁盘上block的总数
- s_root为目录树的根节点的File结构

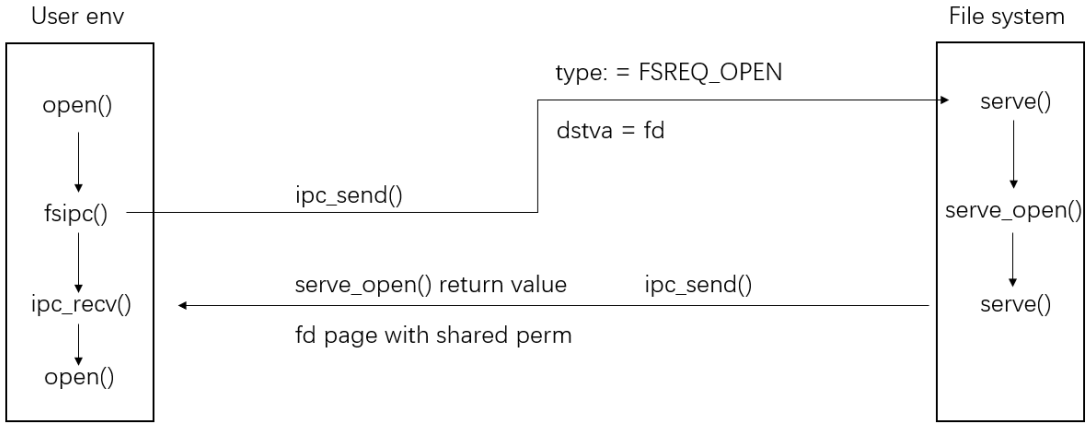
(5) a.以open文件为例，阐述regular环境访问磁盘的流程; b.画出对应的流程图; c.fd page是什么时候设置好的?

a.对于一个用户env来说，需要和File system进程进行通信才能打开一个文件（RPC），流程如下：

- 用户env调用 open() 函数，申请一个fd结构，通过fsipc()函数向file system发出打开文件请求
  - 通过 fd_alloc() 申请 fd ，fd拥有一个独占的page，该page会被设置为 IPC 的接收page
  - 在 fsipc() 完成后，fd page接收了 file system 传输的页面，即其中的fd被file system设置好了。接收时，该页面会被设置为 User env 与 file system 共享，以备后续 read/write 使用
- fsipc()会将请求通过 ipc_send 发送给file_system，并调用 ipc_recv 等待接收响应
- file system本处于serve()的循环之中，收到了来自用户env的open请求，根据请求类别调用 serve_open() 函数
- serve_open() 完成了以下过程
  - 申请一个OpenFile结构
  - 打开文件，获取文件的File结构
  - 设置好之前申请的OpenFile结构
  - 最后，将fd所在的page设置为和用户env共享的

    ```
    *pg_store = o->o_fd;
    *perm_store = PTE_P|PTE_U|PTE_W|PTE_SHARE;
    ```
- serve_open()调用结束，返回到serve()
- serve()通过 ipc_send 将结果返回给用户env

b.流程图如下：



c. fd page何时被设置好

用户用来接收fsipc结果的 fd page 于open()中调用 fd_alloc() 设置好；file system中的fd page在serve_open()中设置好。

在serve_open()中，首先会建立一个OpenFile结构，申请该结构时，会**调用 openfile_alloc()**，在这个函数中，会调用 sys_page_alloc() 为 `o_fd` 分配一个page。serve_open()后续会设置这个page中 fd 结构的字段，并且通过IPC将该fd page发送给 user env。