title: Jos-Lab5
categories: Jos-Lab

Introduction

Lab5要求我们实现一个基于磁盘的简单文件系统。它将拥有一个自己的environment，其它用户environment通过发起对这个environment的IPC请求来访问文件系统。我们不需要实现整个文件系统(lab5已经实现了很大一部分)，只需要实现一些相关调用：从磁盘中读写块、分配磁盘块、实现文件读写的IPC调用等。本文档主要介绍exercise以及challenge的代码思路，question的回答请见`answers-lab5.txt`,challenge请见本文档最后。

Disk Access

Lab5文件系统的实现不同于以往的将文件驱动程序和相关系统调用添加到内核，而是通过生成一个特殊的environment来控制文件生成、读取等，其他用户environment通过对这个environment发起IPC通信来间接访问文件系统，因此我们要为这个特殊的environment提供访问磁盘的权限。

Exercise 1

Exercise1要求我们修改`kern/env.c`中的`env_create`函数，使其能够生成一个特殊的environment(type=ENV_TYPE_FS),并为其提供更高的IO权限。
在进行exercise之前，我发现lab4的代码无法通过检测，少了包含`new env`和`free env`的两句输出，`grep -r "new env" *`全局搜索后发现`kern/env.c`里面为了提高代码运行速度将和这两个输出有关的`cprintf`注释掉了，奇怪的是lab4当时测试是能通过的，现在必须解除注释才能通过。

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    //cprintf("env_create here!\n");
    struct Env* env1;
    if(env_alloc(&env1,0)==0)
    {
        load_icode(env1,binary,size);
        env1->env_type=type;
    }

    // If this is the file server (type == ENV_TYPE_FS) give it I/O privileges.
    // LAB 5: Your code here.
    if(type==ENV_TYPE_FS)
    {
        env1->env_tf.tf_eflags|=FL_IOPL_MASK;
    }
}
```

The Block Cache

Exercise 2

Lab5要为文件系统提供一个缓存，exercise2要求我们实现`fs/bc.c`中的`bc_pgfault`和`flush_block`函数。`flush_block`负责在必要的时候将缓存中的文件写入磁盘。
`bc_pgfault`是page fault的handler函数，在page fault发生后需要调用它来将需要的page从磁盘中加载出来。因为提示中说addr并不一定是PGSIZE对齐，所以使用前需要先对齐，然后分配一个page作为cache，然后调用`ide_read`将磁盘中的内容读取到page中，`ide_read`以sector为单位读取磁盘，根据addr到DISKMAP地址的距离，读取的内容是从第`blockno*BLKSECTS`个sector开始的`BLKSECTS`个sector(一个BLOCK)。

```
    // LAB 5: Your code here
    //panic("bc_pgfault not implemented");
    addr=ROUNDDOWN(addr,PGSIZE);
    r=sys_page_alloc(0,addr,PTE_P|PTE_U|PTE_W);
    if(r<0)
    {
        panic("fs/fs.c:bc_pgfault:sys_page_alloc failed");
    }
```

```
    r=ide_read(blockno*BLKSECTS,addr,BLKSECTS);
    if(r<0)
    {
        panic("fs/fs.c:bc_pgfault:ide_read failed");
    }
```

`flush_block`函数需要判断addr所在的磁盘是否被映射而且已被修改，是则要将这个block写回磁盘，然后通过`sys_page_map`函数将PTE_D标记清除掉。如果addr所折算出来的磁盘地址仍没被映射或者并不是dirty数据，则不做处理。同样，addr还是可能是未对齐的。

```
    // LAB 5: Your code here.
    //panic("flush_block not implemented");
    addr=ROUNDDOWN(addr,PGSIZE);
    if(!va_is_mapped(addr)||!va_is_dirty(addr))
    {
        return;
    }
    int r;
    if((r=ide_write(blockno*BLKSECTS,addr,BLKSECTS))<0)
    {
        panic("fs/bc.c:flush_block: ide_write failed");
    }
    if((r=sys_page_map(0,addr,0,addr,PTE_SYSCALL))<0)
    {
        panic("fs/bc.c:flush_block: sys_page_map failed");
    }
```

The Block Bitmap

Exercise 3

Exercise3要求我们实现`fs/fs.c`中的`alloc_block`函数，它在bitmap中找到一个还没被分配的block，将其标记为已分配(反转对应的bit)，并返回这个block的块号。为了保证一致性，我们还要在改变了bitmap后立即将其flush到磁盘中。

```
    // LAB 5: Your code here.
    //panic("alloc_block not implemented");
    uint32_t 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;
```

File Operations

Exercise 4

Exercise4要求我们实现`fs/fs.c`中的`file_block_walk`函数及`file_get_block`函数。
`file_block_walk`函数根据参数f和filebno，找到文件f的第filebno个block在磁盘中对应的block，并设置相应指针。同时，如果参数alloc被设置，则要为文件alloc一个间接块。

```
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(fileno>=NDIRECT+NINDIRECT)
    {
```

```
        return -E_INVAL;
`    }
    if(fileno<NDIRECT)
    {
        *ppdiskbno=&(f->f_direct[filebno]);
        return 0;
    }
    int r;
    if(f->f_indirect==0)
    {
        if(alloc==0)
        {
            return -E_NOT_FOUND;
        }
        if((r=alloc_block())<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);
    *ppdiskno=&(indirect[filebno-NDIRECT]);
    return 0;
}
```

`file_get_block`在上一个函数的基础上将fileno对应的block在map到的地址存在*blk中。

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    // LAB 5: Your code here.
    //panic("file_get_block not implemented");
    if(filebno>=NDIRECT+NINDIRECT)
    {
        return -E_INVAL;
    }
    int r;
    uint32_t* ppdiskbno=NULL;
    if((r=file_block_walk(f,filebno,&ppdiskbno,1))<0)
    {
        return r;
    }
    if(*ppdiskbno==0)
    {
        if((r=alloc_block())<0)
        {
            return -E_NO_DISK;
        }
        *ppdiskbno=r;
        memset(diskaddr(r),0,BLKSIZE);
        flush_block(diskaddr(r));
    }
    *blk=diskaddr(*ppdiskbno);
    return 0;
}
```

Client/Server File System Access

这一part我们要让负责文件系统调用的environment能够被其它的environment访问。

Exercise 5

Exercise5要求我们实现`fs/serv.c`中的`serve_read`和`lib/file.c`中的`devfile_read`函数。

`serve_read`函数负责从给定的fileid对应的文件的特定offset处读取一定量的数据(但最多只有一个BLOCKSIZE)，并更新当前fd的offset。

```
    int r;
    struct OpenFile *openfile1;
    if ((r = openfile_lookup(envid, req->req_fileid, &openfile1)) < 0)
    {
        return r;
    }

    if ((r = file_read(openfile1->o_file, ret->ret_buf, MIN(req->req_n, PGSIZE), openfile1->o_fd->fd_offset)) < 0)
    {
        return r;
    }

    openfile1->o_fd->fd_offset += r;
    return r;
```

`devfile_read`函数通过发起FSREQ_READ请求来让文件系统server environment读取fd对应的文件的当前offset处的n个byte并返回。

```
    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;
```

Exercise 6

Exercise6和5类似，要求我们实现`fs/serv.c`中的`serve_write`函数以及`lib/file.c`中的`devfile_write`函数。
`serve_write`函数负责向给定的fileid对应的文件的特定offset处写入一定量的数据，并更新当前fd的offset。

```
    int r;
    struct OpenFile *openfile1;
    if ((r = openfile_lookup(envid, req->req_fileid, &openfile1)) < 0)
    {
        return r;
    }

    if ((r = file_write(openfile1->o_file, req->req_buf, req->req_n, openfile1->o_fd->fd_offset)) < 0)
    {
        return r;
    }

    openfile1->o_fd->fd_offset += r;
    return r;
```

`devfile_write`函数通过发起FSREQ_WRITE请求来让文件系统server environment向fd对应的文件的当前offset处写入数据。

```
    int r;
    fsipcbuf.write.req_fileid = fd->fd_file.id;
    void *p = (void*)buf;
    while (n>0)
    {
        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 (ssize_t)(p-buf);
```

Client-Side File Operations

Exercise 7

Exercise7要求我们实现`lib/file.c`中的`open`函数，为用户environment提供打开文件的接口。`open`函数使用`fd_alloc`函数找到一个未使用的文件描述符，然后向管理文件系统的environment发送IPC请求(FSREQ_OPEN)打开文件，并返回文件描述符的编号。如果参数path的长度过长，要返回`-E_BAD_PATH`错误，另外如果已打开的文件数目已超过最大数量、IPC请求失败时均需返回错误。错误退出时还应注意关闭已打开的描述符。

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

Spawning Processes

Exercise 8

Exercise8要求我们实现`kern/syscall.c`中的`sys_env_set_trapframe`函数，它将envid对应的environment的trap frame设置成参数tf，并将代码的级别设置为3以允许interrupt。如果参数envid对应的environment尚未存在就返回`-E_BAD_ENV`，一切顺利就返回0.

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

然后还需要在`kern/syscall.c`中的`syscall`函数中注册一下。

```
case SYS_env_set_trapframe:
            return sys_env_set_trapframe((envid_t)a1, (struct Trapframe*)a2);
```

这样，lab5就算完成了，单就exercise来说，这好像是几次lab中最友善的一个lab了。

Challenge

这次的Challenge感觉都好难。。。

```
Challenge! The file system is likely to be corrupted if it gets interrupted in
the middle of an operation (for example, by a crash or a reboot). Implement soft
```

```
updates or journalling to make the file system crash-resilient and demonstrate
some situation where the old file system would get corrupted, but yours doesn't.
```

这个challenge的实现是按照之前CSE的lab上要求的用log提高文件系统的容错性的方式来实现的，主要就是对文件做出的所有操作均要记录在log上面，比如文件操作是a=1 ==> a=2,则在log中记录这一操作，记录下a的原始值1和修改后的值2，然后记录下当前操作是否已finish(已写入硬盘)。这里对log的所有操作都是即时写入硬盘的，避免引入log的容错性问题。在系统重启后，先检查log中的上一操作是否已经finish，已finish则说明现在硬盘中存储的就是a=2，不需要任何恢复操作，如果没有finish则需要将上一finish后的所有操作均重现，即将a=1 ==> a=2操作再重复一遍。这样就避免了系统突然重启的情况下在缓存中的文件数据来不及写入硬盘而用户却误认为操作已完成的情况。