Lab5_作业报告

1. 简介

在这个lab中，我们要实现一个简单的基于磁盘的文件系统。

在进行exercise 1之前，我将`kern/init.c`里面的`ENV_CREATE(fs_fs, ENV_TYPE_FS);`与`lib/exit.c`里面的`close_all()`注释掉,然后跑pingpong,primes,forktree三个测试可行。

2.文件系统

本实验的目标不是让您实施整个文件系统，而是让您只实现某些关键组件。 特别是，您将负责将块读入块缓存并将其刷新到磁盘; 分配磁盘块; 将文件偏移映射到磁盘块; 并在IPC接口中实现读取，写入和打开。 因为你不会自己实现所有的文件系统，所以熟悉提供的代码和各种文件系统接口是非常重要的。

2.1 磁盘访问

我们操作系统中的文件系统环境需要能够访问磁盘，但我们还没有在内核中实现任何磁盘访问功能。与其采用传统的“单片”操作系统策略，即将IDE磁盘驱动程序添加到内核以及允许文件系统访问它的必要系统调用中，我们将实现IDE磁盘驱动程序作为用户级别的一部分文件系统环境。我们仍然需要稍微修改内核，以便进行设置，以便文件系统环境具有实现本身磁盘访问所需的权限。

只要我们依靠轮询，“基于程序I / O”（PIO）的磁盘访问并且不使用磁盘中断，用户空间中的磁盘访问就很容易实现。也可以在用户模式下实现中断驱动的设备驱动程序（例如，L3和L4内核就是这样做的），但是由于内核必须将现场设备中断并将其分派到正确的用户模式环境。

x86处理器使用EFLAGS寄存器中的IOPL位来确定是否允许保护模式代码执行特殊设备I / O指令，例如IN和OUT指令。由于我们需要访问的所有IDE磁盘寄存器都位于x86的I / O空间中，而不是内存映射，因此为文件系统环境提供“I / O特权”是我们唯一需要做的事情允许文件系统访问这些寄存器。实际上，EFLAGS寄存器中的IOPL位为内核提供了一个简单的“全有或全无”方法来控制用户模式代码是否可以访问I / O空间。在我们的例子中，我们希望文件系统环境能够访问I / O空间，但我们不希望任何其他环境能够访问I / O空间。

在接下来的测试中，如果测试失败，obj / fs / fs.img可能会保持不一致。运行make等级或制作qemu之前一定要将其删除。

Exercise 1

修改kern/env.c里面的env_create函数来给文件系统（ENV_TYPE_FS）I/O特权。

根据提示，增加代码如下

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

2.2 块缓存

在我们的文件系统中，我们将借助处理器的虚拟内存系统实现一个简单的“缓冲区缓存”（实际上就是一个块缓存）。块缓存的代码位于fs / bc.c中。

我们的文件系统将限于处理3GB或更小的磁盘。我们将文件系统环境的地址空间从0x10000000（DISKMAP）直到0xD0000000（DISKMAP + DISKMAX）保留一个固定的大型3GB区域作为磁盘的“内存映射”版本。例如，磁盘块0映射到虚拟地址0x10000000，磁盘块1映射到虚拟地址0x10001000，依此类推。 fs / bc.c中的diskaddr函数实现了从磁盘块编号到虚拟地址的转换（以及一些理智检查）。

由于我们的文件系统环境具有独立于系统中所有其他环境的虚拟地址空间的独立虚拟地址空间，并且文件系统环境需要做的唯一事情就是实现文件访问，所以保留大部分文件系统环境的地址空间。由于现代磁盘大于3GB，因此在32位机器上实际执行文件系统会很麻烦。这种缓冲区高速缓存管理方法在64位地址空间的机器上可能仍然合理。

当然，将整个磁盘读入内存是不合理的，相反，我们将实现一种请求分页的形式，其中我们只分配磁盘映射区域中的页面，并响应于页面从磁盘中读取相应的块在这个地区的错误。这样，我们可以假装整个磁盘都在内存中。

Exercise 2

在fs/bc.c里面实现bc_pgfault跟flush_block功能。

根据提示，bc_pgfault实现如下：

```
if((r = sys_page_alloc((envid_t)0, ROUNDDOWN(addr, PGSIZE), PTE_P | PTE_U | PTE_W)) < 0 )
            panic("sys_page_alloc: %e",r);
        ide_read(blockno * BLKSECTS, ROUNDDOWN(addr, PGSIZE), BLKSECTS);
```

flush_block实现如下：

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

2.3 Block Bitmap

在fs_init设置位图指针之后，我们可以将位图视为一个打包的位数组，每个磁盘块对应一个位。

Exercise 3

修改fs.c里面的alloc_block()，遍历所有的blockno,找到一个free的就改bitmap，然后flush,实现如下：

```
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[blockno/32]);
              return blockno;
           }
        }
```

2.4 文件操作

我们在fs / fs.c中提供了各种函数来实现您将需要解释和管理文件结构，扫描和管理目录文件条目所需的基本设施，并从根目录文件系统解析绝对路径路径名。

Exercise 4

实现fs.c里面的file_block_walk()与file_get_block()

根据注释，实现如下：

```
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;
        else if(filebno < NDIRECT){
           *ppdiskbno = &(f->f_direct[filebno]);
           return 0;
        }
        else{
           if(!f->f_indirect){
              if(!alloc)
                  return -E_NOT_FOUND;
              int r;
              if((r = alloc_block()) < 0)
                  return -E_NO_DISK;
              memset(diskaddr(r), 0, BLKSIZE);
              f->f_indirect = r;
              flush_block(diskaddr(r));
           }
           *ppdiskbno = &(((uint32_t *)(diskaddr(f->f_indirect)))[filebno-NDIRECT]);
         }
       return 0;
}
```

file_get_block()

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

```
    // LAB 5: Your code here.
    uint32_t * pdiskno;
        int r;
        if((r = file_block_walk(f,filebno,&pdiskno,1)) < 0)
            return r;
        if(!*pdiskno){
            if((r = alloc_block()) < 0)
                return r;
            *pdiskno = r;
        }
        *blk = diskaddr(*pdiskno);
        return 0;
}
```

2.5 Client/Server File System Access

实现RPC。

Exercise 5&Exercise6

实现fs/serv.c的serve_read,serve_write与lib/file.c的devfile_read(),devfile_write().

serve_read，serve_write调用了file_read()来实现.

devfile_read(),devfile_write()修改fsipcbuf，调用fsipc来实现。

Exercise 7

实现open函数 需要用`fd_alloc()`函数(已提供)找到一个未使用的文件描述符, 发出一个IPC请求到文件系统环境中来打开文件.. 确保你的代码能优雅的退出,如果 超过了文件打开的上限, 或者任何IPC到文件系统环境请求失败了.

```
int
open(const char *path, int mode)
{
    // Find an unused file descriptor page using fd_alloc.
    // Then send a file-open request to the file server.
    // Include 'path' and 'omode' in request,
    // and map the returned file descriptor page
    // at the appropriate fd address.
    // FSREQ_OPEN returns 0 on success, < 0 on failure.
    //
    // (fd_alloc does not allocate a page, it just returns an
    // unused fd address.  Do you need to allocate a page?)
    //
    // Return the file descriptor index.
    // If any step after fd_alloc fails, use fd_close to free the
    // file descriptor.

    // LAB 5: Your code here.
        struct Fd *fd;
```

```
        int r;
        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);
}
```

Exercise 8

spawn依靠新的系统调用sys_env_set_trapframe来初始化新创建的环境的状态。 实现sys_env_set_trapframe。

在kern/syscall.c里面补全sys_env_set_trapframe如下

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

在syscall添加路由即可。

3 Challenge

Implement Unix-style exec

我的想法（代码没有实现）如下：

先把对应的文件各部分映射到一块空闲的区域（加一个offset），然后调用sys_exec函数，在kernel中进行page_insert,并且修改tf中的esp和eip，这样，就达到了exec的目的。同时，在原先的spawn函数中，还调用了init_stack函数，在USTACKTOP-PGSIZE这块区域，初始化了stack。同样，将这部分映射到一个区域，在kernel中重新映射回USTACKTOP-PGSIZE。之后，在sys_exec修改了curenv的地址空间以及初始化stack后，修改eip和esp，运行env_run，就实现了一次调用，永不返回的exec。