lab 6 文件系统

宁晨然 17307130178

1.实验目的

- 文件系统是操作系统用于明确存储设备或分区上的文件的方法和数据结构，由文件系统的接口，对对象操纵和管理的软件集合，对象及属性组成。
- `xv6`使用的文件系统比大多数"真实"文件系统（包括`xv6 UNIX`）要简单得多，它可以提供基本功能：创建、读取、写入和删除按层次目录结构组织的文件。
- 您将负责将块读入块缓存并将其刷新回磁盘；分配磁盘块；将文件偏移映射到磁盘块；以及在`IPC`接口中实现读、写和打开。

2.实验过程

2.1 系统环境I/O权限

修改`env.c`中的`env_create()`函数，当`i386_init()`最后调用`ENV_CREATE(fs_fs, ENV_TYPE_FS);`启动文件系统的时候，这个宏定义在`env.h`中；其实不用关注其中的宏定义细节，重要的就是`env_create()`函数，当接收`EnvType`为`ENV_TYPE_FS`，意思就是设置文件系统的环境。这个时候赋予系统IO操作权限。

```
#define ENV_CREATE(x, type)                     \
    do {                                \
        extern uint8_t ENV_PASTE3(_binary_obj_, x, _start)[];   \
        env_create(ENV_PASTE3(_binary_obj_, x, _start),     \
               type);                   \
    } while (0)
```

赋予IO操作权限的方式是通过`EFLAGS`寄存器的`IOPL`位来控制。可知`env_create()`中创建了一个环境，修改其中的`trapframe`中的`eflags`值，`e->env_tf.tf_eflags |= FL_IOPL_MASK;`就可以将`IOPL`位置1。

```
void env_create(uint8_t *binary, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *e;
    int r;
    if ((r = env_alloc(&e, 0) != 0))
    {
        panic("create env failed\n");
    }

    // 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)
        e->env_tf.tf_eflags |= FL_IOPL_MASK;

    load_icode(e, binary);
    e->env_type = type;
```

```
}
```

Question: 当您从一个环境切换到另一个环境时，是否必须执行其他操作以确保正确保存和还原此I/O权限设置？为什么？

不需要执行。因为环境切换的时候，调用了`env_pop_tf(&e->env_tf);`函数，这个函数会根据环境保存或恢复上下文，包括`trapframe`中的`eflags`，就包括了IO权限，所以不需要执行其他操作。

2.2 实现`bc_pgfault`和`flush_block`函数

- **在`fs / bc.c`中实现`bc_pgfault`和`flush_block`函数。`bc_pgfault`是一个页面错误处理程序，`bc_pgfault`的工作是从磁盘加载页面。编写此代码时，请记住（1）`addr`可能未与块边界对齐（2）`ide_read`在扇区而不是块中操作。**
- **必要时，`flush_block`函数应将一个块写出到磁盘。如果该块甚至不在块缓存中或它不脏，则`flush_block`不应执行任何操作。我们将使用`VM`硬件来跟踪自从磁盘上一次读取或写入磁盘以来是否已修改了磁盘块。要查看某个块是否需要写入，我们可以看看是否在`uvpt`条目中设置了`PTE_D“ dirty”`位。 将块写入磁盘后，`flush_block`应使用`sys_page_map`清除`PTE_D`位。**

2.2.1 `bc_pgfault`

作用：页错误处理程序，从磁盘上装载页。

步骤：

- 检查错误发生在块缓存区
- 检查块号
- 分配磁盘映射区中一页，从磁盘中读取内容到该页（待实现）
- 清理磁盘块页的脏位
- 检查读取的块已经定位

因为需要读取内容，可以从`fs/ide.c`中找到读写磁盘的函数：

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

`int ide_write(uint32_t secno, void *dst, size_t nsecs)`

由于需要分配磁盘映射区中的一页，所以也需要`kern/syscall.c`中的`SYS_page_alloc`函数：

`static int sys_page_alloc(envid_t envid, void *va, int perm)`

根据上函数定义，可知需要传参三个参数：

- `envid`：`sys_page_alloc`函数中调用的了`int envid2env(envid_t envid, struct Env **env_store, bool checkperm)`函数获取当前`env`，而当`envid==0`表示当前环境，所以传值为0。
- `va`：需要分配页的虚拟地址，注意需要跟页地址边界对齐，所以需要先调用`ROUNDDOWN`函数。`addr = ROUNDDOWN(addr, PGSIZE);`，此时传入`addr`才可以分配页。
- `perm`：权限要求用户、现在环境、可读可写，所以传入`PTE_U | PTE_P | PTE_W`。

```
addr = ROUNDDOWN(addr, PGSIZE);
r = sys_page_alloc(0, addr, PTE_U | PTE_P | PTE_W)
```

接下来是把磁盘中块中的内容读入到页中：

- `secno`：传入扇区号，已知`blockno`，乘上每块扇区数就是扇区号，`BLKSECTS=8`。所以传入`blockno * BLKSECTS`。

- `dst`：读入到的地址就是`addr`。
- `nsecs`：内容大小（以扇区大小为单位），一块大小就是`BLKSECTS`这么大。

```
r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)
```

还因为上下文可知，模仿写`panic`语句即可：

```
static void
bc_pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
    int r;
    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);
    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n", blockno);

    // LAB 5: you code here:
    addr = ROUNDDOWN(addr, PGSIZE);
    if ((r = sys_page_alloc(0, addr, PTE_U | PTE_P | PTE_W)) < 0)
        panic("in bc_pgfault, sys_page_alloc: %e", r);

    if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
        panic("in bc_pgfault, ide_read: %e", r);

    if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) <
0)
        panic("in bc_pgfault, sys_page_map: %e", r);
    if (bitmap && block_is_free(blockno))
        panic("reading free block %08x\n", blockno);
}
```

2.2.2 `flush_block`

作用：将一个块写出到磁盘（如果不在块缓存中或者不脏，则不做操作）

步骤：

- 检查该地址是否在块缓存中
- 对齐到块边界
- 如果该地址有映射且为脏块，则写入到磁盘中，且清除`PTE_D`位。

步骤一已经写好，需要实现后面的。

首先还是用`ROUNDDOWN`将`addr`对齐到块边界。使用`va_is_mapped`函数可以判断虚拟地址是否已经映射，使用`va_is_dirty`可以判断虚拟地址是否为脏。

写入到磁盘可以使用`ide_write`，`int ide_write(uint32_t secno, void *dst, size_t nsecs)`，参数可以参考`bc_pgfault`。`ide_write(blockno * BLKSECTS, addr, BLKSECTS)`即可。

重点是清除脏位，同理可以参考`bc_pgfault`中写好的：

```
    if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) <
0)
        panic("in bc_pgfault, sys_page_map: %e", r);
```

可以使用`sys_page_map`和`uvpt`和`PTE_SYSCALL`清理脏位。结合起来就写成`flush_block`函数。

```
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);

    int r;
    addr = ROUNDDOWN(addr, PGSIZE);
    if (va_is_mapped(addr) && va_is_dirty(addr))
    {
        if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
            panic("in flush_block, ide_write: %e", r);
        if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
PTE_SYSCALL)) < 0)
            panic("in flush_block, sys_page_map: %e", r);
    }
}
```

2.3 `alloc_block`

- **使用`free_block`作为模板在`fs / fs.c`中实现`alloc_block`，后者应在`bitmap`中找到可用的磁盘块，将其标记为已使用，然后返回该块的编号。 分配该块时，应立即使用`flush_block`将更改后的`bitmap`块刷新到磁盘，以帮助文件系统保持一致。**

作用：分配块

步骤：

- 在`bitmap`中找到可用磁盘块，标记已使用
- 分配块后立即更新`bitmap`并刷新回磁盘
- 返回块编号

这个函数实现较简单，因为有`free_block`做参考：

```
// Mark a block free in the bitmap
void free_block(uint32_t blockno)
{
    // Blockno zero is the null pointer of block numbers.
    if (blockno == 0)
        panic("attempt to free zero block");
    bitmap[blockno / 32] |= 1 << (blockno % 32);
}
```

位图`bitmap`是一个`uint32_t`的数组，每个向量是`01`串，表示32个块的空闲情况。如果对应位为1则空闲。可以发现释放块的操作中，寻找位图的操作就是`bitmap[blockno/32]`，因为每个位向量是32个块的空闲信息。同理，如果找到某个空闲位信息，将该位占用，则可以将该位改为0，可以使用异或运算。

遍历所有块，块总数有`super->s_nblocks`，如果该块空闲，则返回该块。

每次找到空闲块都会更新`bitmap`，更新完了之后调用`flush_block(bitmap)`刷新磁盘中的`bitmap`。

如果没有找到空闲块，则返回`-E_NO_DISK`。

```
int alloc_block(void)
{
    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;
}
```

2.4 实现`file_block_walk`和`file_get_block`

- **实现`file_block_walk`和`file_get_block`。`file_block_walk`从文件中的块偏移量映射到`struct File`或间接块中的指针，非常类似于`pgdir_walk`对页表所做的操作。**
  `file_get_block`更进一步，并映射到实际的磁盘块，并在必要时分配一个新的磁盘块。

2.4.1 `file_block_walk`

作用：根据块偏移量映射到`file`或间接块的指针

步骤：

- 检查块偏移量是否在范围内
- 若属于直接块范围：将`*ppdiskbno`置为`f->f_direct[]`
- 若属于间接块范围：
  - 如果间接块为空，判断`alloc`，为假则返回`E_NOT_FOUND`；为真则分配间接块地址，并初始化后刷新到磁盘
  - 如果有间接块，则将`*ppdiskbno`置为间接块中指针。

```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool
alloc)
{
    int r;
    if (filebno < NDIRECT)
    {
        if (ppdiskbno)
            *ppdiskbno = f->f_direct + filebno;
        return 0;
    }
    else if (filebno < NDIRECT + NINDIRECT)
    {
        if (!f->f_indirect && !alloc)
            return -E_NOT_FOUND;
```

```
        if (!f->f_indirect && alloc)
        {
            if ((r = alloc_block()) < 0)
                return -E_NO_DISK;
            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;
    }
    else
        return -E_INVAL;
}
```

2.4.2 `file_get_block`

作用：根据块偏移量映射到指针，必要时分配块。

步骤：

- 查找该指针
- 如果该块未分配，则分配
- `blk`指向该块

首先使用刚才写好的`file_block_walk`函数查找到对应的块偏移量的指针，并且对于间接块采用需要分配的方法。但是注意这样获取的指针，可能指向的块是已经分配的间接块、直接块，直接块也有可能没有分配块，所以需要检查`*ppdiskbno`槽中是否有块号，就是判断是否为0，为0时还需要再次分配块空间，方式同上一个函数一样。

最后将`*blk`指向这个块的地址。

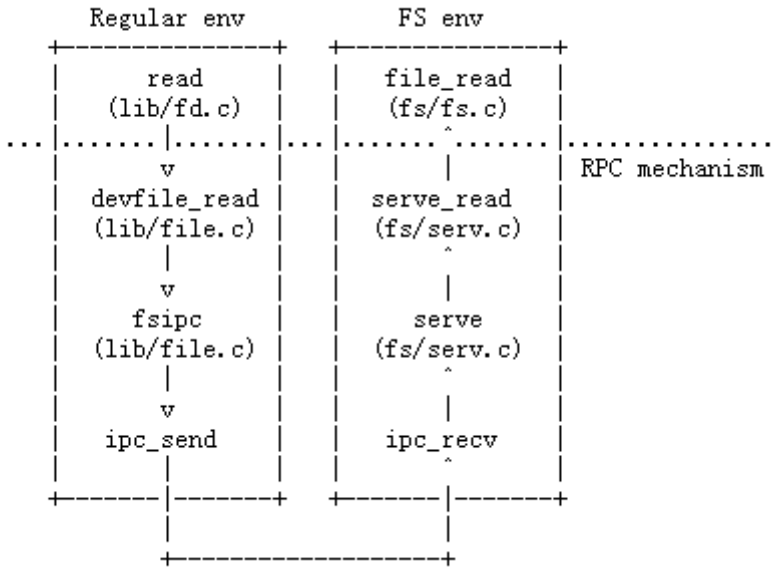
```
int file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    int r;
    uint32_t *ppdiskbno;

    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;
}
```

2.5 实现`serve_read`

- **在`fs/serv.c`中实现`serve_read`。**

- **`serve_read`的大量操作将由`fs / fs.c`中已经实现的`file_read`来完成。 `serve_read`只需提供`RPC`接口即可读取文件。 查看`serve_set_size`中的注释和代码，以大致了解服务器功能的结构。**

上面已经实现了很多文件系统环境下，进程进行的文件操作。而实际情况下，其他环境如果要进行文件操作，是通过`IPC`方式，即进程间通信的方式来进行多任务操作系统。`IPC`方式分为本地过程调用`LPC`和远程过程调用`RPC`。一个正常运行的环境，如果需要文件系统操作，是通过下图的方式，与文件系统环境进行交互，从而进行文件操作。



一个普通进程需要读文件，发送一个读请求到文件系统服务进程。

> 首先read操作文件描述符，分发给合适的设备读函数`devfile_read`。`devfile_read`函数实现读取磁盘文件，作为客户端文件操作函数。然后建立请求结构的参数，调用`fsipc`函数来发送`IPC`请求并解析返回的结果。
>
> 文件系统服务端的代码在`fs/serv.c`中，服务进程循环等待直到收到1个`IPC`请求。然后分发给合适的处理函数，最后通过`IPC`发回结果。对于读请求，服务端会分发给`serve_read`函数
>
> 在`JOS`实现的`IPC`机制中，允许进程发送1个32位数和1个页。为了实现发送1个请求从客户端到服务端，我们使用32位数来表示请求类型，存储参数在联合`Fsipc`位于共享页中。在客户端我们一直共享`fsipcbuf`所在页，在服务端我们映射请求页到`fsreq`地址(`0x0ffff000`)。
>
> 服务端也会通过`IPC`发送结果。我们使用32位数作为函数的返回码。`FSREQ_READ`和`FSREQ_STAT`函数也会返回数据，它们将数据写入共享页返回给客户端。

`Fsipc`是一个联合体，`serv`会循环接收`IPC`请求，所以当有一个请求是`serve_read`时，分配好`file_read`所需要的接口即可。

`serve_read`函数：

作用：提供`file_read`接口，读取文件

步骤：

- 读取命令`read`和返回`readret`
- 寻找相关需要打开的文件`openfile_lookup`
- 读取文件

`*ret = &ipc->readRet`是返回的部分，其中包括了读取文件的缓存`ret_buf`，将文件读入到这个部分。

首先利用`openfile_lookup(envid, req->req_fileid, &o)`，根据`request`中的请求的`file_id`，在文件系统中找到该文件，并放入`openfile`中。

利用`ssize_t file_read(struct File *f, void *buf, size_t count, off_t offset)`读出文件，因为文件大小可能超过`pgsize`，所以需要先设置`req_n`为`min(pgsize, req->req_n)`。其中的`buf`就是`ret`中的`buf`。

读完文件后，需要将偏移量加上读取文件的大小。

```
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);

    struct OpenFile *o;
    int r, req_n;

    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
    req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;
    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 0;
}
```

2.6 实现`serve-write`和`devfile-write`

- **在`fs/serv.c`中实现`serve-write`，在`lib/file.c`中实现`devfile-write`。**

2.6.1 `serve-write`

上面已经实现了`serve-read`函数，所以这个就很简单了，基本一样。

作用：文件系统中的写操作

步骤：

- 寻找需要写操作的文件，并打开
- 使用`file_write`写文件

这个函数跟`serve-read`函数过于类似，不同之处就是写操作中，`request`请求中的`buf`就是需要写进文件的内容。同理需要注意：

- `req_n`不能超过`pgsize`
- 文件写完之后需要加上`offset`

```
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);
```

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

    req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;
    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;
}
```

2.6.2 `devfile_write`

作用：实现磁盘写入文件操作，作为客户端写入文件操作函数

步骤：

- 判断是否大于`pgsize`，如果大于则赋值`pgsize`
- 利用`fsipbuf.write`建立`IPC`请求

这个函数的实现参考`devfile_read`，基本都是一样的。首先赋值`fsipcbuf.write`中的值，利用`fsipc`建立`IPC`请求，然后将`fsipcbuf.write`中的缓存清空。

```
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.
    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;
    if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
        return r;
    assert(r <= n);
    assert(r <= PGSIZE);
    memmove(fsipcbuf.write.req_buf, buf, n);
    return r;
}
```

3.实验结果

实验中`test.c`文件中有差错，`msg`和文件`newmotd`本身就不一样，应该改`msg`末尾改成`\r\n`。原因是`windows`和`unix`系统中行末的标识符不一样，`windows`中是`\r\n`表示换行，`unix`中是`\n`表示换行。

`make qemu`运行结果中查看文件系统操作的正确性：

```
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
[00000000] new env 00001000
[00000000] new env 00001001
FS is running
FS can do I/O
Device 1 presence: 1
icode startup
icode: open /motd
block cache is good
superblock is good
bitmap is good
alloc_block is good
file_open is good
file_get_block is good
file_flush is good
file_truncate is good
file_rewrite is good
```

4.实验感想

- 本次实验实现了文件系统中几个重要的函数，深刻理解了文件系统的操作方式，了解了位图、文件读写、`IPC`方式等文件系统的构架。
- 提升了阅读代码能力。本身函数实现不需要几行代码，但是要读懂原有框架已经使用的函数，需要理解别人写的代码的意思，并且调用接口函数完成自己需要写的东西。

5.参考资料

[1] https://blog.csdn.net/bysui/article/details/51868917

[2] https://blog.csdn.net/a747979985/article/details/99551678

[3] https://www.jianshu.com/p/e894b3660d75