Lab5实验报告

思考题

Thinking 5.1

对于写操作，如果采用写回策略与脏位，只有对应的cache项被换出时，对应的数据才会真正完成写操作。对于串口设备，也就只会看到最后一次的输出，而看不到前面的输出。对于IDE磁盘，则只会根据最后一次的磁盘编号与扇区等信息写入对应的磁盘，显然是不符合对磁盘写入的要求。

对于读操作，由于读操作也需要向对应位置直接写入，如果采用写回策略，则同上所述，也难以真正获取到最新的数据，同样不符合对应操作预期的目的。

Thinking 5.2

一个磁盘块中最多能存储4096/256=16个文件控制块，一个目录下最多能有1024×16个文件。支持单个文件最大为1024（直接加间接指针）×4KB（一个磁盘块大小）=4MB。

Thinking 5.3

$4\times2^{(4*7)}=2^{30}$，即为1GB。

Thinking 5.4

- fs/serv.h

```
#define BY2SECT 512  //扇区大小
#define DISKMAP 0x10000000  //块缓存的起始地址
#define DISKMAX 0x40000000  //块缓存的大小
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);       //对ide磁盘的读
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);      //对ide磁盘的写
```

- user/include/fs.h

```
#define BY2BLK BY2PG //磁盘块对应的字节数（4096）
#define BIT2BLK (BY2BLK * 8) //磁盘块对应的位数
#define NDIRECT 10  //直接的i指针的数目
#define NINDIRECT (BY2BLK / 4)  //间接数目
#define MAXFILESIZE (NINDIRECT * BY2BLK) //最大文件大小(4MB)
#define BY2FILE 256     //一个file结构体（文件控制块）的大小
#define FILE2BLK (BY2BLK / sizeof(struct File))  //一个磁盘块中的文件控制块的最大数目
```

- 其他

```
#define NBLOCK 1024 // The number of blocks in the disk.
uint32_t nbitblock; // the number of bitmap blocks.
uint32_t nextbno;   // next availiable block.
struct Block {
    uint8_t data[BY2BLK];   //磁盘块的数据部分
    uint32_t type;          //该磁盘块的类型
} disk[NBLOCK];             //磁盘块数组
void *diskaddr(u_int blockno) //返回第blockno个磁盘块对应的虚拟地址
```

Thinking 5.5

会共享文件描述符和定位指针。

Thinking 5.6

```
struct File {                                           //一个文件控制块的大小是256B。
    char f_name[MAXNAMELEN]; // filename
    uint32_t f_size;     // file size in bytes
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT];                         //存的是文件所在磁盘块的编号，来代替指针效果，而不是指针！
    uint32_t f_indirect;                                //f_indirect也指的是间接磁盘块的编号，需要用
disk[f_indirect]来访问对应间接磁盘块。
    struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory.
    char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

```
struct Fd {
    u_int fd_dev_id;        //表示设备类型（文件/控制台/lab6中的管道）
    u_int fd_offset;        //读写到的当前位置（会随read等函数实时更新）
    u_int fd_omode;         //文件的打开方式，如只读，只写等，与之相关的宏定义如下
};
// File open modes
#define O_RDONLY 0x0000  /* open for reading only */
#define O_WRONLY 0x0001  /* open for writing only */
#define O_RDWR 0x0002    /* open for reading and writing */
#define O_ACCMODE 0x0003 /* mask for above modes */
```

文件描述符主要用于打开文件后记录文件的状态，不对应物理实体，只是单纯的内存数据。

```
// file descriptor + file
struct Filefd {             //实际可以理解为文件描述符加文件控制块的一个结构体集合
    struct Fd f_fd;         //文件描述符
    u_int f_fileid;         //文件本身的id
    struct File f_file;     //文件控制块
};
```

如同注释中已经给出的，FileFd结构体可以看做文件描述符和文件控制块的一个集合体，包含了文件的状态，文件的描述信息还有指向文件所在磁盘块的指针（在本实验中实际为磁盘块的数组下标）。

Thinking 5.7

带圆点的实线箭头为异步消息，实心箭头为同步消息，虚线箭头为返回消息。

在发送异步消息后，线程继续运行而不进入阻塞状态。而对于同步消息，用户线程发送请求后自身进入阻塞状态，由文件系统进程来完成对应的操作，成功后再schedule至用户进程继续运行。

难点分析&理解

扇区的读写

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs) {
    u_int begin = secno * BY2SECT;                                  //起始扇区
    u_int end = begin + nsecs * BY2SECT;                            //根据要读取的
扇区数目确定终点

    for (u_int off = 0; begin + off < end; off += BY2SECT) {
//begin+off=end时，证明读完
        uint32_t temp = diskno;
        /* Exercise 5.3: Your code here. (1/2) */
        panic_on(syscall_write_dev((void *)&temp,DEV_DISK_ADDRESS+DEV_DISK_ID,4));
//在对应偏移处写入要读的磁盘编号
        temp=begin+off;
        panic_on(syscall_write_dev((void
*)&temp,DEV_DISK_ADDRESS+DEV_DISK_OFFSET,4));
                                                                    //在对应偏移处
写入要读的相对偏移量大小
        temp=DEV_DISK_OPERATION_READ;
        panic_on(syscall_write_dev((void
*)&temp,DEV_DISK_ADDRESS+DEV_DISK_START_OPERATION,4));
                                                                    //在对应偏移处
写入要进行读操作
        int resultOfRead;
        panic_on(syscall_read_dev((void
*)&resultOfRead,DEV_DISK_ADDRESS+DEV_DISK_STATUS,4));
                                                                    //在对应偏移处
获取读写操作的结果
        panic_on(resultOfRead==0);                                  //检测磁盘是否
读取成功
        panic_on(syscall_read_dev((void
*)dst+off,DEV_DISK_ADDRESS+DEV_DISK_BUFFER,DEV_DISK_BUFFER_LEN));
                                                                    //从缓冲区读取
读出的数据
    }
}
```

写操作类似，只需要将读缓冲区改为写缓冲区并放置在最前部分即可。

磁盘空间的整体概念

磁盘与操作系统交互的最小单位为磁盘块(4096B)，整个磁盘空间是由一定数量(1024)的磁盘块构成的。

其中0号块作为引导扇区和分区表，1号块作为超级块存储描述文件系统的基本信息（魔数，磁盘块个数，根目录的File结构体等）。对于块的使用情况在Mos中以位图的形式存储，对应位为1则代表该块空闲可以使用，否则代表对应磁盘块已经存储了内容不可以使用。

磁盘空间的初始化函数(init_disk,tools/fsformat.c)

```
void init_disk() {
    int i, diff;

    // Step 1: Mark boot sector block.
    disk[0].type = BLOCK_BOOT;

    // Step 2: Initialize boundary.
    nbitblock = (NBLOCK + BIT2BLK - 1) / BIT2BLK;        //(NBLOCK/BIT2BLK)的向上对齐，计算出需保存位图需要占用的磁盘块个数
    nextbno = 2 + nbitblock;                             //标记下一个可用的磁盘块

    // Step 2: Initialize bitmap blocks.
    for (i = 0; i < nbitblock; ++i) {
        disk[2 + i].type = BLOCK_BMAP;                   //标记存储微秃的磁盘块的磁盘块类型
    }
    for (i = 0; i < nbitblock; ++i) {
        memset(disk[2 + i].data, 0xff, BY2BLK);          //.data代表对应磁盘块的数据部分，为8*4096,0xff对应8位1，BY2BLK代表4096个8位（1B）
    }
    if (NBLOCK != nbitblock * BIT2BLK) {                 //如果存在位图不满一个磁盘块，要将不满的部分置0，因为不存在这些磁盘块，不能使用。
        diff = NBLOCK % BIT2BLK / 8;
        memset(disk[2 + (nbitblock - 1)].data + diff, 0x00, BY2BLK - diff);
    }

    // Step 3: Initialize super block.
    disk[1].type = BLOCK_SUPER;
    super.s_magic = FS_MAGIC;
    super.s_nblocks = NBLOCK;
    super.s_root.f_type = FTYPE_DIR;
    strcpy(super.s_root.f_name, "/");
}
```

对File结构体的理解

```
struct File {                                                    //一个文件控制块的大小是256B。
    char f_name[MAXNAMELEN]; // filename
    uint32_t f_size;     // file size in bytes
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT];                                  //存的是文件所在磁盘块的编号，来代替指针效果，而不是指针！
    uint32_t f_indirect;                                         //f_indirect也指的是间接磁盘块的编号，需要用
disk[f_indirect]来访问对应间接磁盘块。
    struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory.
    char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

对create_file的理解

create_file实际上是在指定的目录文件中寻找到一个没有使用过的文件控制块，用于满足创建新文件的要求。

```
struct File *create_file(struct File *dirf) {
    int nblk = dirf->f_size / BY2BLK;                            //获取目录文件占有的磁盘块数

    // Step 1: Iterate through all existing blocks in the directory.
    for (int i = 0; i < nblk; ++i) {
        int bno; // the block number                             //目录文件内容所在的磁盘块的序号(下标)
        // If the block number is in the range of direct pointers (NDIRECT), get the 'bno'
        // directly from 'f_direct'. Otherwise, access the indirect block on 'disk' and get
        // the 'bno' at the index.
        if(i < NDIRECT){
            bno = dirf->f_direct[i];
        } else {
            bno = ((uint32_t*)disk[dirf->f_indirect].data)[i];
        }
        // Get the directory block using the block number.
        // 将目标块的data起始地址转化为以文件控制块为单元的的地址，以进行对该磁盘块内文件控制块的遍历。
        struct File *blk = (struct File *)(disk[bno].data);

        // Iterate through all 'File's in the directory block.
        // 遍历其中所有的文件控制块，FILE2BLK代表一个磁盘块能存储的文件控制块数的上限
        for (struct File *f = blk; f < blk + FILE2BLK; ++f) {
            // If the first byte of the file name is null, the 'File' is unused.
            // Return a pointer to the unused 'File'.
            if(f->f_name[0]==NULL) return f;
        }
    }
```

```
    // Step 2: If no unused file is found, allocate a new block using
'make_link_block' function
    // and return a pointer to the new block on 'disk'.
    // 如果该文件控制块已经链接到的磁盘块(dirc&indirc)都没有空闲的文件控制块，就重新申请一个磁盘
块并与其建立链接
    return (struct File *)disk[make_link_block(dirf,nblk)].data;
    return NULL;
}
```

```
int make_link_block(struct File *dirf, int nblk) {
    int bno = next_block(BLOCK_FILE);       //获取到下一个没有使用的磁盘块块号，并设置其类型
为File类型
    save_block_link(dirf, nblk, bno);       //将该块号存到dirf对应的文件控制块的指针（块
号）中
    dirf->f_size += BY2BLK;                 //该文件大小增加一个block
    return bno;
}
void save_block_link(struct File *f, int nblk, int bno) {
    assert(nblk < NINDIRECT); // if not, file is too large !
    if (nblk < NDIRECT) {                   //直接指针区域
        f->f_direct[nblk] = bno;
    } else {
        if (f->f_indirect == 0) {
            // create new indirect block.
            f->f_indirect = next_block(BLOCK_INDEX);    //申请一个存储间接指针的磁盘块
        }
        ((uint32_t *)(disk[f->f_indirect].data))[nblk] = bno;   //将间接磁盘块的对应位置
设置为bno
    }
}
```

块缓存

目的：提前将磁盘内容读入内存中，在用户需要调用其中的内容时直接从内存的缓冲区中读取，减少访问磁盘的次数和时间消耗。（0x10000000-0x4fffffff）在MOS中为真正将磁盘块的内容放入内存中，供用户进程读取与使用。

相关函数如下：

```
//返回对应磁盘块在内存中的虚拟地址(在0x10000000-0x4fffffff中
void *diskaddr(u_int blockno) {
    return (void *)(DISKMAP + blockno * BY2BLK);
}

//  检测block对应的虚拟地址是否已经映射到一个物理页
//  Returns the virtual address of the cache page if mapped, 0 otherwise.
void *block_is_mapped(u_int blockno) {
    void *va = diskaddr(blockno);
    if (va_is_mapped(va)) {
        return va;
    }
```

```
    return NULL;
}

int map_block(u_int blockno) {
    // 如果该block对应的va已经建立了映射，直接返回
    if(block_is_mapped(blockno)!=NULL){
        return 0;
    }
    // 利用系统调用为该va分配一页物理页并建立映射（页表对应位有效）
    syscall_mem_alloc(0,diskaddr(blockno),PTE_D);
}

//  Unmap a disk block in cache.
void unmap_block(u_int blockno) {
    // Step 1: Get the mapped address of the cache page of this block using
'block_is_mapped'.
    void *va;
    //获取目标块对应的虚拟地址并检测是否已经建立物理页的映射
    va=block_is_mapped(blockno);
    if(va==NULL) return;
    // Step 2: If this block is used (not free) and dirty in cache, write it back to
the disk
    // first.
    //如果该block在内存中被写了(对应va留下了脏位)，就要将修改写回磁盘块
    if((!block_is_free(blockno))&&(block_is_dirty(blockno))){
        write_block(blockno);
    }
    // Step 3: Unmap the virtual address via syscall.
    syscall_mem_unmap(0,va);
    user_assert(!block_is_mapped(blockno));
}

//  将内存中的磁盘块(块缓存)写回磁盘中
void write_block(u_int blockno) {
    // Step 1: detect is this block is mapped, if not, can't write it's data to
disk.
    if (!block_is_mapped(blockno)) {
        user_panic("write unmapped block %08x", blockno);
    }

    // Step2: write data to IDE disk. (using ide_write, and the diskno is 0)
    void *va = diskaddr(blockno);
    ide_write(0, blockno * SECT2BLK, va, SECT2BLK); //SECT2BLK为一个block对应几个扇区
(4096/512)
}

// Overview:
//  Make sure a particular disk block is loaded into memory.
//
// Post-Condition:
//  Return 0 on success, or a negative error code on error.
//
```

```
//  If blk!=0, set *blk to the address of the block in memory.
//
//  If isnew!=0, set *isnew to 0 if the block was already in memory, or
//  to 1 if the block was loaded off disk to satisfy this request. (Isnew
//  lets callers like file_get_block clear any memory-only fields
//  from the disk blocks when they come in off disk.)
//
// Hint:
//  use diskaddr, block_is_mapped, syscall_mem_alloc, and ide_read.
int read_block(u_int blockno, void **blk, u_int *isnew) {
    // 读取的是超级块中记录的合法范围内的磁盘块
    if (super && blockno >= super->s_nblocks) {
        user_panic("reading non-existent block %08x\n", blockno);
    }

    // Step 2: 查询位图，该块是否没有用过
    // Hint:
    //  If the bitmap is NULL, indicate that we haven't read bitmap from disk to
memory
    //  until now. So, before we check if a block is free using `block_is_free`, we
must
    //  ensure that the bitmap blocks are already read from the disk to memory.
    if (bitmap && block_is_free(blockno)) {
        user_panic("reading free block %08x\n", blockno);
    }

    // Step 3: transform block number to corresponding virtual address.
    void *va = diskaddr(blockno);

    // Step 4: read disk and set *isnew.
    // Hint:
    //  If this block is already mapped, just set *isnew, else alloc memory and
    //  read data from IDE disk (use `syscall_mem_alloc` and `ide_read`).
    //  We have only one IDE disk, so the diskno of ide_read should be 0.
    if (block_is_mapped(blockno)) { // the block is in memory
        if (isnew) {
            *isnew = 0;
        }
    } else { // the block is not in memory
        if (isnew) {
            *isnew = 1;
        }
        syscall_mem_alloc(0, va, PTE_D);
        ide_read(0, blockno * SECT2BLK, va, SECT2BLK);
    }

    // Step 5: 返回该磁盘块对应的虚拟地址
    if (blk) {
        *blk = va;
    }
    return 0;
}
```

在已经有了块缓存的基础上，就可以通过一些指令，获取某个目录下的文件在内存中的虚拟地址

```
//  Set *blk to point at the filebno'th block in file f.
//  用于将某个指定的文件指向的磁盘块读入内存
// Hint: use file_map_block and read_block.
//
// Post-Condition:
//  return 0 on success, and read the data to `blk`, return <0 on error.
int file_get_block(struct File *f, u_int filebno, void **blk) {
    int r;
    u_int diskbno;
    u_int isnew;

    // Step 1: find the disk block number is `f` using `file_map_block`.
    if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0) {
        return r;
    }

    // Step 2: read the data in this disk to blk.
    if ((r = read_block(diskbno, blk, &isnew)) < 0) {
        return r;
    }
    return 0;
}

//  Find a file named 'name' in the directory 'dir'. If found, set *file to it.
//
// Post-Condition:
//  Return 0 on success, and set the pointer to the target file in `*file`.
//  Return the underlying error if an error occurs.
int dir_lookup(struct File *dir, char *name, struct File **file) {
    int r;
    // 计算该目录下有多少磁盘块
    u_int nblock;
    nblock = dir->f_size / BY2BLK;
    // Step 2: 遍历所有的磁盘块
    for (int i = 0; i < nblock; i++) {
        // Read the i'th block of 'dir' and get its address in 'blk' using
'file_get_block'.
        void *blk;
        /* Exercise 5.8: Your code here. (2/3) */
        try(file_get_block(dir,i,&blk));
        struct File *files = (struct File *)blk;    //获取到对应磁盘块的地址

        // Find the target among all 'File's in this block.
        for (struct File *f = files; f < files + FILE2BLK; ++f) {
            // Compare the file name against 'name' using 'strcmp'.
            // If we find the target file, set '*file' to it and set up its 'f_dir'
            // field.
            /* Exercise 5.8: Your code here. (3/3) */
            if(!strcmp(f->f_name,name)){
                *file = f;
```

```
                f->f_dir = dir;
                return 0;
            }
        }
    }

    return -E_NOT_FOUND;
}
```

文件描述符(fd)，文件服务进程与用户进程的沟通

在每个文件打开时，都要申请一个fd来记录文件相关的状态。同时，file.c中的open，还有fd.c的read/write都是宏观的读/写，真正的读/写还是在文件进程服务中的读/写

类似内核态与用户态，文件服务的操作遵循用户调用函数->fsipcxxx进程对应的进程间通信->文件服务进程调用serv.c的函数完成对应操作并返回的流程。

```
struct Fd {
    u_int fd_dev_id;        //表示设备类型（文件/控制台/lab6中的管道）
    u_int fd_offset;        //读写到的当前位置（会随read等函数实时更新）
    u_int fd_omode;         //文件的打开方式，如只读，只写等，与之相关的宏定义如下
};
// File open modes
#define O_RDONLY 0x0000  /* open for reading only */
#define O_WRONLY 0x0001  /* open for writing only */
#define O_RDWR 0x0002    /* open for reading and writing */
#define O_ACCMODE 0x0003 /* mask for above modes */
```

文件描述符主要用于打开文件后记录文件的状态，不对应物理实体，只是单纯的内存数据。

实验体会

在完成本次实验的过程中，个人最大的体会就是虽然在指导书和注释的提醒之下，完成相关部分的填空较为简单，但倘若细看整个文件系统的相关代码，简直是无底洞一般的存在。尤其是从dir_lookup后，理解文件进程和用户进程的沟通过程是一个相当困难的过程，仍然需要在完成实验后花一定时间去理解。

同时，本单元的实现过程对系统调用的依赖程度较深，这从最近的exam和extra的考试中亦可以窥见端倪。在完成相关题目的过程中，也让我对这部分操作的运用和理解越来越深。