Lab 5 Report

思考题

Thinking 5.1

数据一致性问题。

Cache是用于加速内存访问的临时存储，它保存了最近访问的数据副本。如果写入设备的数据仅存在于Cache中而不及时更新到设备本身，可能导致数据的不一致性。当其他部件或设备读取设备数据时，它们可能会读取到过期或不正确的缓存数据，而不是最新写入的数据。

Thinking 5.2

磁盘块的大小: 4KB（与页面大小相同） `#define BY2BLK BY2PG`
文件控制块大小: 256B `#define BY2FILE 256`

因此一个磁盘块中最多能存储16个文件控制块。

```
#define NDIRECT 10
#define NINDIRECT (BY2BLK / 4)
```

直接指针和间接指针**共同指向**1024个磁盘块，因此**目录文件**下最多有1024*16个文件控制块，即16384个文件。

文件大小最大为1024*4KB = 4MB。 `#define MAXFILESIZE (NINDIRECT * BY2BLK)`

Thinking 5.3

2^30B = 1GB

```
/* Maximum disk size we can handle (1GB) */
#define DISKMAX 0x40000000
```

Thinking 5.4

fs/serv.h

- SECT2BLK 4096/512 = 8

```
#define PTE_DIRTY 0x0002 // file system block cache is dirty

/* IDE disk number to look on for our file system */
#define DISKNO 1

#define BY2SECT 512         /* Bytes per disk sector */
#define SECT2BLK (BY2BLK / BY2SECT) /* sectors to a block */

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP 0x10000000

/* Maximum disk size we can handle (1GB) */
#define DISKMAX 0x40000000
```

user/include/fs.h

- BY2BLK 4096
- NINDIRECT 1024
- MAXFILESIZE 1024*4096B = 4MB
- FILE2BLK 4096/256 = 16

```
// Bytes per file system block - same as page size
#define BY2BLK BY2PG
#define BIT2BLK (BY2BLK * 8)

// Maximum size of a filename (a single path component), including
null
#define MAXNAMELEN 128

// Maximum size of a complete pathname, including null
#define MAXPATHLEN 1024

// Number of (direct) block pointers in a File descriptor
#define NDIRECT 10
#define NINDIRECT (BY2BLK / 4)

#define MAXFILESIZE (NINDIRECT * BY2BLK)

#define BY2FILE 256
```

```
#define FILE2BLK (BY2BLK / sizeof(struct File))

// File types
#define FTYPE_REG 0 // Regular file
#define FTYPE_DIR 1 // Directory
```

Thinking 5.5

文件描述符和定位指针均存储在**用户空间**，该部分空间被映射到子进程。

因此fork前后的父子进程**共享**文件描述符和定位指针。

Thinking 5.6

```
// file descriptor
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};

// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

- Fd
  - fd_dev_id: 设备编号
  - fd_offset: 偏移量，用于记录文件**读写位置**
  - fd_omode: 文件打开模式(只读/只写/读写)
- Filefd
  - f_fd: 文件描述符
  - f_fileid: 文件编号
  - f_file: 文件控制块

Thinking 5.7

- 实线闭三角箭头
  用于表示**同步消息**。这种消息是一种阻塞操作，发送方必须等待接收方完成操作后才能继续执行。
- 虚线开三角箭头
  用于表示**返回消息**。这种消息是一种阻塞操作，表示消息的接收者对象向发送者对象返回结果或响应。

```
// Overview:
//  Send an IPC request to the file server, and wait for a reply.
//
// Parameters:
//  @type: request code, passed as the simple integer IPC value.
//  @fsreq: page to send containing additional request data, usually
fsipcbuf.
//          Can be modified by server to return additional response
info.
//  @dstva: virtual address at which to receive reply page, 0 if none.
//  @*perm: permissions of received page.
//
// Returns:
//  0 if successful,
//  < 0 on failure.
static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm) {
    u_int whom;
    // Our file system server must be the 2nd env.
    ipc_send(envs[1].env_id, type, fsreq, PTE_D);
    return ipc_recv(&whom, dstva, perm);
}
```

fsipc()函数中使用了ipc_send()函数，发送方向文件系统服务器发送**同步消息**，并**持续等待**文件系统服务器的响应，直到ipc_recv()函数返回。

难点分析

5.1 sys_write_dev() & sys_read_dev()

将**物理地址**加上kseg1的偏移: 0xA0000000，即可得到kseg1段的**虚拟地址**。

- write: va -> pa + KSEG1
- read: pa + KSEG1 -> va

5.3 ide_write() & ide_read()

- 共同前置操作
  - write: disk number -> 0x13000010
  - write: offset -> 0x13000000
- ide_read()
  - write: operand(0) -> 0x13000020
  - read: 0x13000030: status
  - read: 0x13004000: data
- ide_write()
  - write: data -> 0x13004000
  - write: operand(1) -> 0x13000020
  - read: 0x13000030: status

5.4 free_block()

uint32_t: 无符号**32位**整数类型

因此bitmap数组的每一项可以存储32个磁盘块的**空闲信息**。

`bitmap[blockno / 32] |= 1 << (blockno % 32);`

5.5 create_file()

- 普通文件: 其指向的磁盘块存储**文件内容**
- 目录文件: 其指向的磁盘块存储该目录下各个文件对应的**文件控制块**

```
struct Block {
    uint8_t data[BY2BLK];
    uint32_t type;
} disk[NBLOCK];
```

disk[NBLOCK]磁盘块数组，即为**整个磁盘**。

文件控制块中的指针实际上是**磁盘块编号**bno，通过bno从**磁盘**中找到磁盘块(存储着**文件控制块**): `struct File *blk = (struct File *)(disk[bno].data);`

- 直接指针: `bno = dirf->f_direct[i]`

- 间接指针: `bno = ((int *)(disk[dirf->f_indirect].data))[i]`
  间接指针本身只有一个，它指向一个**间接磁盘块**，其中存储了1024个指针(前10个不可用)。

5.7 map_block() & unmap_block()

使用block_is_mapped()函数获取对应**磁盘块的虚拟地址**。

在系统调用时传入参数envid = 0: 进一步传入envid2env()函数，其在envid为0时返回**当前进程**curenv。

5.8 dir_lookup()

该**目录文件**对应的磁盘块数量: `nblock = dir->f_size / BY2BLK;`

- 外层遍历: 磁盘块 `for (int i = 0; i < nblock; i++)`
- 内层遍历: 文件控制块 `for (struct File *f = files; f < files + FILE2BLK; ++f)`

5.9 open()

进行**强制类型转换**: `ffd = (struct Filefd *)fd;`

```
// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

指针指向的地址不变，保留着原结构体Fd的内容。作用相当于 `ffd->f_fd = *fd;`

5.10 read()

检查文件打开模式: `(fd->fd_omode & O_ACCMODE) == O_WRONLY`，此时权限为**只写**。

5.12 fsipc_remove()

用户进程的文件操作通过IPC与**文件系统进程**通信，fsipc()函数中使用了ipc_send()函数发送给文件系统进程，在serve()函数中通过ipc_recv()函数接收用户进程的请求。

以下是用户态下remove()函数的调用链:

- 用户进程：
  - user/lib/file.c remove(): `fsipc_remove(path);`
  - user/lib/fsipc.c fsipc_remove(): `fsipc(FSREQ_REMOVE, req, 0, 0);`
  - user/lib/fsipc.c fsipc(): `ipc_send(envs[1].env_id, type, fsreq, PTE_D);`
- 文件系统进程：
  - fs/serv.c serve():

```
req = ipc_recv(&whom, (void *)REQVA, &perm);
switch (req) {
    case FSREQ_REMOVE:
        serve_remove(whom, (struct Fsreq_remove *)REQVA);
        break;
    default:
        debugf("Invalid request code %d from %08x\n", whom, req);
}
```

  - fs/serv.c serve_remove(): `file_remove(rq->req_path);`
  - fs/fs.c file_remove(): `walk_path(path, 0, &f, 0);`
  - fs/fs.c walk_path(): `dir_lookup(dir, name, &file);`
  - fs/fs.c dir_lookup()

实验体会

本次实验的难点在于理解文件系统中使用的各种**数据结构**，以及**文件系统进程**与**用户进程**之间的**通信**，多线程之间的**协作**让我对进程与内核的关系有了更深入的认识。本单元涉及的知识点较多，需要多次阅读源码，才能理解其中的逻辑关系，源码中的一些函数由于没有出现在题目中，因此没有全部理解，有所缺憾。