OS_lab5实验报告

思考题

Thinking 5.1 设备操作与高速缓存

- 只有缓存块将要被新进入的数据取代时，缓存数据才会被写入内存。因此当外部设备更新数据的时候，如果此前cpu写入外设的数据只存在cache的时候，那么这部分数据则无法进行更新，就会发生错误行为。
- 串口设备的读写频率比IDE磁盘更快，相同时间内更容易发生错误。

Thinking 5.2 单个文件的最大体积、一个磁盘块最多存储的文件控制块及一个目录最多子文件

- 根据File结构体中的`char* f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];`所以一个文件控制块大小为256B。
- 一个文件**直接指针+间接指针**最多有1024个，每个磁盘块4KB，所以一个文件最大体积为4MB。
- 一个磁盘块最多储存4KB/256B=16个文件控制块。
- 一个文件**直接指针+间接指针**最多有1024个，对应1024个磁盘块，每个磁盘块有16个文件控制块，一共16K给文件控制块。

Thinking 5.3 磁盘最大容量

缓冲区大小为1GB，那么磁盘的最大容量也是1GB。

Thinking 5.4 文件系统进程中宏定义理解

```
#define SECT_SIZE 512              /* Bytes per disk sector */
#define SECT2BLK (BLOCK_SIZE / SECT_SIZE) /* sectors to a block */

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

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

fs/serv.h文件中大部分都是和磁盘属性相关的宏，用于堆缓存中相应的函数。

```
// Bytes per file system block - same as page size
#define BLOCK_SIZE PAGE_SIZE
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 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 (BLOCK_SIZE / 4)

#define MAXFILESIZE (NINDIRECT * BLOCK_SIZE)

#define FILE_STRUCT_SIZE 256
```

user/include/fs.h中大部分都是关于file结构体的属性，用于文件系统结构中的相应函数。

Thinking 5.5 文件描述符与 fork 函数

```
int main(){
    int r;
    char buf[512];

    r = open("/newmotd", O_RDWR);
    int pid = fork();
    if (pid == 0) {
          read(r, buf, 5);
          debugf("child buf:%s\n",buf);
    } else {
          read(r, buf, 5);
          debugf("parent buf:%s\n",buf);
    }
    return 0;
}
/*
ans:
parent buf:This
child buf:is a
*/
```

通过上面的简单程序可以看出，父子进程之间共享文件描述符和定位指针。

Thinking 5.6 文件系统用户接口中的结构体

```
struct File {
    char f_name[MAXNAMELEN]; // filename
    uint32_t f_size;     // file size in bytes
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT];
    uint32_t f_indirect;

    struct File *f_dir; // the pointer to the dir where this file is in, valid
only in memory.
    char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void
*)];
} __attribute__((aligned(4), packed));

// 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;
};
```

结构体均为内存信息，但是`Fd`和`Filefd`中的`File`结构体对应了磁盘上的block。

Thinking 5.7 解释时序图，思考进程间通信

- `ENV_CREATE(user_env)`和`ENV_CREATE(fs_serv)`消息由init线程发出后，user和fs线程分别进行初始化，消息发出后init即可进行后续的指令，所以是异步消息。
- user线程发出`ipc_send(fsreq)`消息之后，自身会被阻塞，直到fs线程返回`ipc_send(dst_va)`消息，才能继续运行，所以是同步消息。

难点分析

磁盘镜像

```
int main(int argc, char **argv) {
    static_assert(sizeof(struct File) == FILE_STRUCT_SIZE);
    init_disk();

    if (argc < 3) {
        fprintf(stderr, "Usage: fsformat <img-file> [files or
directories]...\n");
        exit(1);
    }

    for (int i = 2; i < argc; i++) {
        char *name = argv[i];
        struct stat stat_buf;
        int r = stat(name, &stat_buf);
        assert(r == 0);
        if (S_ISDIR(stat_buf.st_mode)) {
            printf("writing directory '%s' recursively into disk\n", name);
            write_directory(&super.s_root, name);
        } else if (S_ISREG(stat_buf.st_mode)) {
            printf("writing regular file '%s' into disk\n", name);
            write_file(&super.s_root, name);
        } else {
            fprintf(stderr, "'%s' has illegal file mode %o\n", name,
stat_buf.st_mode);
            exit(2);
        }
    }

    flush_bitmap();
    finish_fs(argv[1]);

    return 0;
```

```
}
```

tools/fsformat.c 文件中存储了磁盘镜像制作工具的源代码。在main函数中，我们首先调用了 init_disk 函数对磁盘进行初始化，在这个函数中，我们首先将第一个block设置成主引导块，然后空出第二个 block（超级块），从第三个block开始，使用位图匹配机制来存放存储位图，最后再设置超级块。

初始化之后，我们再不断读取命令行参数，调用 write_directory 和 write_file 将文件内容写入磁盘镜像中。在这两个函数中，我们都需要使用 create_file 函数在指定目录下创建新的文件。在 create_file 函数中，遍历了 `dirf` 文件下用于保存内容（对于目录来说，内容就是文件控制块）的所有磁盘块。查找是否有未使用的文件控制块，如果有，就返回；否则，调用 make_link_block 新申请一个磁盘块，该磁盘块中第一个文件控制块的位置就代表了新创建的文件。

```
struct File *create_file(struct File *dirf) {
    int nblk = dirf->f_size / BLOCK_SIZE;
    for (int i = 0; i < nblk; ++i) {
        int bno;
        if (i < NDIRECT) {
            bno = dirf->f_direct[i];
        } else {
            bno = disk[dirf->f_indirect].data[i];
        }
        struct File *blk = (struct File *)(disk[bno].data);
        for (struct File *f = blk; f < blk + FILE2BLK; ++f) {
            if (f->f_name[0] == '\0') {
                return f;
            }
        }
    }
    u_int32_t new_bno = make_link_block(dirf, nblk);
    return (File *)(disk[new_bno].data);
}
```

write_directory 函数创建好文件之后，会递归调用路径下的所有文件或者目录，递归调用 write_directory 和 write_file 函数。

写入文件之后， mian 函数还调用 flush_bitmap() 完成位图的设置。这样，磁盘镜像就设置完成了。

文件系统

文件操作库函数

以 open 函数为例，open 函数 等一系列库函数位于 user/lib/file.c 文件中。

首先，open 函数调用了 fd_alloc 来申请一个文件描述符，它的功能就是遍历并找到没被使用的最小的文件描述符，返回其地址。然后，调用 fsipc_open 来获取文件描述符。fsipc_open 中，我们将一块缓冲区 fsipcbuf 视为 struct Fsreq_open，向其中写入了请求打开的文件路径 req_path 和打开方式 req_omode。并调用 fsipc 进行发送。fsipc 函数就是简单的向服务进程发送消息，并接收服务进程返回的消息。

设置好fd之后，我们通过fd2data获取文件内容应该映射到的地址，接着我们将文件所有的内容都从磁盘中映射到内存。使用的函数为 fsipc_map。

文件系统服务进程的初始化

文件系统服务进程是一个完整的进程，有自己的 main 函数。该进程的代码都位于 fs 文件夹下。main 函数位于 fs/serv.c 中。

首先，调用 serve_init 对程序进行初始化。函数中，实际进行初始化的只有 opentab，这个数组用于记录整个操作系统中所有处于打开状态的文件。

之后，我们又调用 fs_init 函数，完成文件系统的初始化，这其中包含了三个函数。

```
void fs_init(void) {
    read_super();
    check_write_block();
    read_bitmap();
}
```

文件系统服务进程的服务

完成初始化之后，我们调用 serve 开启服务。serve 是一个死循环，不断地调用 ipc_recv 以接收其他进程发来的请求，根据请求类型的不同分发给不同的处理函数进行处理，并进行回复。

```
void *serve_table[MAX_FSREQNO] = {
    [FSREQ_OPEN] = serve_open,   [FSREQ_MAP] = serve_map,     [FSREQ_SET_SIZE] =
serve_set_size,
    [FSREQ_CLOSE] = serve_close, [FSREQ_DIRTY] = serve_dirty, [FSREQ_REMOVE] =
serve_remove,
    [FSREQ_SYNC] = serve_sync,
};
```

心得体会

lab5是操作系统最后一次上机了，本次实验我感觉还是有一点困难的，主要在于lab5涉及到的文件与函数非常多，我们需要理清楚每个文件下的函数都负责了什么功能，除此之外，我感觉指导书写的也不够详细，当时完成课下实验的时候花费了很大的功夫去理解指导书和代码。总之，顺利完成最后一次实验还是非常欣慰的，在os这门课的实验中，我对操作系统的架构有了进一步的认识，也学到了许多linux的使用技巧，收获丰富。