操作系统 Lab5实验报告

实验思考题

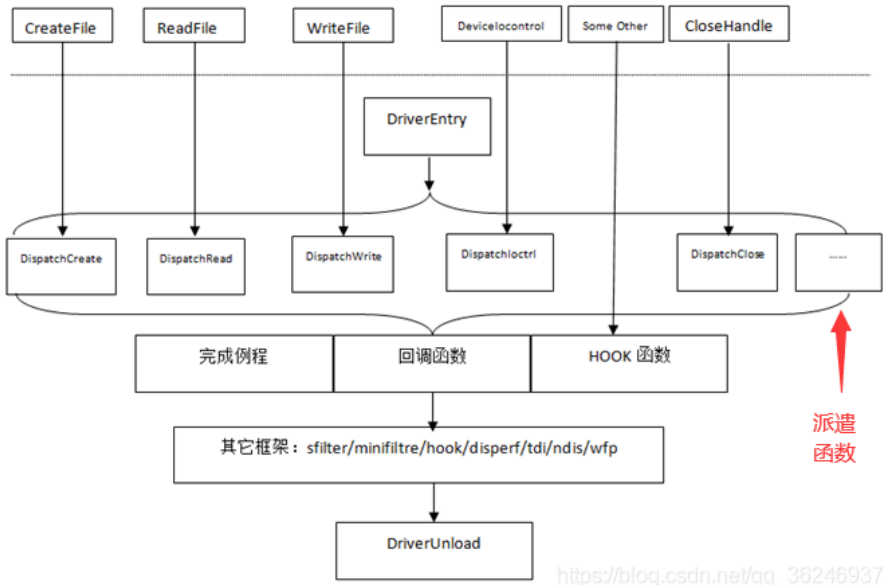
Thinking 5.1

Linux的proc文件系统

- proc文件系统是一个虚拟的文件系统，通过proc文件系统，可以在Linux内核空间与用户空间进行通信。
- proc文件系统的思路为：在内核中构建一个虚拟文件系统proc，内核运行时，将内核中一些关键的数据结构以文件的方式呈现在/proc目录中的一些特定的文件中。这样，相当于将不可见的内核数据结构以可视化的方式呈现给用户与内核开发者。proc文件系统提供给内核开发者一种调试内核的方法，即可以通过实时的观察目录/proc下的文件，来查看内核中特定的数据结构的值。
- /proc目录下的文件大小都是0，因为这些文件本身并不存在于硬盘中，且文件本身并不是一个真实的文件而只是一个接口。当读取该文件时，其实内核并不是去硬盘上读取这些文件，而是映射为内核内部的一个数据结构被读取，并且格式化为字符串返回。因此尽管感觉上proc文件与普通的文件一样，但是实际上文件的内容是实时从内核中数据结构中返回而来的。
- proc文件系统优点：
  - 将系统调用和内核接口抽象为文件，方便用户进程进行访问。
- proc文件系统缺点：
  - 占用内存空间。

Windows的IRP数据结构

- 在Windows内核中，有一种数据结构IRP（I/O Request Package，输入输出请求包），用于上层应用程序与底层驱动程序进行通信。在通信时应用程序会发出I/O请求，操作系统将I/O请求转化为相应的IRP数据，不同类型的IRP会根据类型传递到不同的派遣函数内。IRP两个基本类型MajorFunction和MinorFunction分别记录IRP的主类型和子类型，操作系统根据MajorFunction将IRP“派遣”到不同的派遣函数中，在派遣函数中还可以继续判断这个IRP属于哪种MinorFunction。



Thinking 5.2

- `kseg0` 段用于存放内核代码与数据结构，如果也通过这部分读写外部设备，则会在cache中与内核代码产生冲突，导致访问内核代码的效率降低。同时，由于 `kseg0` 段通过cache访存，当外部设备产生中断信号或更新数据时，可能与cache中的数据产生冲突，导致外设访问和读写产生错误。
- 串口设备相较于IDE磁盘读写更加频繁，如果通过 `kseg0` 段访问串口则会频繁访问cache，进而导致发生上述错误的几率大大增加。

Thinking 5.3

- MOS操作系统的文件控制块存储的信息如下：

```
struct File {
    u_char f_name[MAXNAMELEN];  // 文件名
    u_int f_size;               // 文件大小
    u_int f_type;               // 文件类型
    u_int f_direct[NDIRECT];    // 10个磁盘块直接指针
    u_int f_indirect;           // 磁盘块间接指针
    struct File *f_dir;         // 目录指针
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
// 用于填充磁盘块，无实际意义
};
```

- Linux操作系统中的inode存储的部分信息如下：

```
struct inode {
    umode_t             i_mode;     // 类型及权限
    unsigned short      i_opflags;
    kuid_t              i_uid;      // 文件拥有者的用户id
    kgid_t              i_gid;      // 文件拥有者所在的组id
    unsigned int        i_flags;    // 独立于文件系统的标志位

    const struct inode_operations   *i_op;      // inode相关操作集
    struct super_block              *i_sb;      // 相关联的super block
    struct address_space            *i_mapping; // imap

#ifdef CONFIG_SECURITY
    void                *i_security;
#endif

    unsigned long       i_ino;      // 节点号

    union {
        const unsigned int i_nlink;
        unsigned int __i_nlink;
    };                              // 硬连接数
    dev_t               i_rdev;     // 设备标识符
    loff_t              i_size;     // 以字节为单位的文件大小
    /* 文件时间戳 */
    struct timespec64   i_atime;    // 文件上一次访问的时间
    struct timespec64   i_mtime;    // 文件上一次数据修改的时间
    struct timespec64   i_ctime;    // 文件上一次状态变动的时间
    spinlock_t          i_lock;     // 互斥锁
    unsigned short      i_bytes;    // 文件在最后一个块中所使用字节数
    u8                  i_blkbits;  // 以bit为单位的文件大小
    u8                  i_write_hint;
```

```
    blkcnt_t            i_blocks;   // 文件占用的块数

#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t      i_size_seqcount;
#endif

    /* Misc */
    unsigned long       i_state;    // 状态位
    struct rw_semaphore i_rwsem;

    /* 脏数据信息 */
    unsigned long       dirtied_when;   /* jiffies of first dirtying */
    unsigned long       dirtied_time_when;

    struct hlist_node   i_hash;     // inode hash链表
    struct list_head    i_io_list;  // backing dev IO list

#ifdef CONFIG_CGROUP_WRITEBACK
    struct bdi_writeback    *i_wb;      /* the associated cgroup wb */
    /* foreign inode detection, see wbc_detach_inode() */
    int                 i_wb_frn_winner;
    u16                 i_wb_frn_avg_time;
    u16                 i_wb_frn_history;
#endif

    struct list_head    i_lru;      // inode LRU链表
    struct list_head    i_sb_list;  // superblock的inode链表
    struct list_head    i_wb_list;  // backing dev writeback list
    union {
        struct hlist_head   i_dentry;   // 目录项
        struct rcu_head     i_rcu;
    };
    atomic64_t      i_version;
    atomic64_t      i_sequence; /* see futex */
    atomic_t        i_count;
    atomic_t        i_dio_count;
    atomic_t        i_writecount;

#if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
    atomic_t        i_readcount; /* struct files open RO */
#endif

    union {
        const struct file_operations    *i_fop; // file相关操作集
        void (*free_inode)(struct inode *);
    };
    struct file_lock_context    *i_flctx;
    struct address_space        i_data;         // 数据块
    struct list_head            i_devices;      // 设备链表
    union {
        struct pipe_inode_info  *i_pipe;
        struct cdev             *i_cdev;
        char                    *i_link;
        unsigned                i_dir_seq;
    };
    __u32                       i_generation;   // 版本信息

    ......
```

```
90 }
```

- 相较于MOS系统的文件控制块，Linux系统的inode所存储的信息更加丰富、全面。但可以发现，Linux系统的inode并未存储文件名，这是因为在Linux系统中是通过inode号码，即 `i_ino` 来识别文件的。

Thinking 5.4

- 由 `include/fs.h` 中的定义

```
1 #define BY2FILE     256
```

  文件控制块的大小为 $256B$，则一块磁盘最多存储 $\frac{4KB}{256B} = 16$ 个文件控制块。
- 一个文件控制块最多有 1024 个指针指向 1024 个磁盘块，因此一个目录下最多有 $1024 \times 16 = 16384$ 个文件。
- 一个文件控制块最多有 1024 个指针指向 1024 个磁盘块，因此单个文件最大为 $1024 \times 4KB = 4096KB = 4MB$。

Thinking 5.5

根据 `fs/fs.h` 中的定义和注释：

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

实验使用的内核支持的最大磁盘大小为 $1GB$。

Thinking 5.6

如果将 `DISKMAX` 改成 `0xC0000000`，超过用户空间，文件系统将不能继续工作，因为在MOS操作系统中，文件系统服务是一个用户进程。

Thinking 5.7

- `fs/fs.h` 中部分宏定义如下：

```
1 #define BY2SECT     512 /* Bytes per disk sector */
2 #define SECT2BLK    (BY2BLK/BY2SECT)    /* sectors to a block */
3
4 /* Disk block n, when in memory, is mapped into the file system
5  * server's address space at DISKMAP+(n*BY2BLK). */
6 #define DISKMAP     0x10000000
7
8 /* Maximum disk size we can handle (1GB) */
9 #define DISKMAX     0x40000000
```

  - `BY2SECT`：每个磁盘扇区的字节数，用于接下来定义 `SECT2BLK`。
  - `SECT2BLK`：每个磁盘块的扇区数，用于在函数中计算扇区数进行磁盘读写。
  - `DISKMAP`：磁盘块缓存的起始地址。
  - `DISKMAX`：磁盘块缓存的最大空间。
- `include/fs.h` 中部分宏定义如下：

```
1 // 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
```

- `BY2BLK`：每个磁盘块的字节数，用于与磁盘块大小相关的计算和定义中。
- `BIT2BLK`：每个磁盘块的位数，主要用于磁盘管理位图的相关计算的定义中。
- `MAXNAMELEN`：文件名的最大长度，用于与文件名相关字符数组的长度定义中。
- `MAXPATHLEN`：文件路径的最大长度，用于与文件路径相关字符数组的长度定义中。
- `NDIRECT`：文件控制块中直接指针的最大数量，用于文件控制块直接指针的定义和遍历中。
- `NINDERICT`：文件控制块中间接指针的最大数量，用于文件控制块间接指针的遍历中。
- `MAXFILESIZE`：文件最大大小，用于与文件大小相关的判断中。
- `BY2FILE`：文件控制块大小，用于与文件控制块大小相关的定义和判断中。
- `FILE2BLK`：每个磁盘块中存储最大文件控制块数量，用于对文件控制块的遍历中。
- `FTYPE_REG`：文件类型，代表普通文件。
- `FTYPE_DIR`：文件类型，代表目录文件。

Thinking 5.8

- `struct Fd` 定义如下：

```
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};
```

- `struct Filefd` 定义如下：

```
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

- `struct Filefd` 结构体中第一个成员为 `struct Fd`，相当于每个 `struct Fd` 对应一个 `struct Filefd`，而 `struct Filefd` 的其他成员变量在内存上紧跟在 `struct Fd f_fd` 之后，因此 `struct Fd *` 型的指针转换为 `struct Filefd *` 型指针是可行的。

Thinking 5.9

- `fork` 前后的父子进程会共享文件描述符和定位指针，但文件应在 `fork` 之前打开。
- 测试函数如下：

```c
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

void printFd(int fd);

int main(void) {
    int fd = open("./test.txt", O_RDWR);

    printf("-----fd before fork begin-----\n");
    printFd(fd);
    printf("-----fd before fork end-----\n");

    int pid = fork();
    if (pid == 0) { /* child */
        sleep(2);
        printf("-----child begin from here-----\n");
        printFd(fd);
        printf("child seek %ld\n", lseek(fd, 3, SEEK_CUR));
        printf("-----child ends in here-----\n");
    } else { /* parent */
        printf("-----pre_parent begin from here-----\n");
        printFd(fd);
        printf("parent seek %ld\n", lseek(fd, 0, SEEK_CUR));
        printf("-----pre_parent ends in here-----\n");
        sleep(4);
        printf("-----pos_parent begin from here-----\n");
        printFd(fd);
        printf("parent seek %ld\n", lseek(fd, 0, SEEK_CUR));
        printf("-----pos_parent ends in here-----\n");
    }
    return 0;
}

void printFd(int fd) {
    struct stat sb;
    fstat(fd, &sb);
    printf("I-node number: %ld\n", (long)sb.st_ino);
    printf("Ownership: UID=%ld, GID=%ld\n", (long)sb.st_uid,
(long)sb.st_gid);
    return;
}
```

- 测试输出结果如下：

```
git@20373785:~/20373785$ gcc test_fork_fd.c -o test_fork_fd
git@20373785:~/20373785$ ./test_fork_fd
-----fd before fork begin-----
I-node number: 4725391
Ownership: UID=997, GID=997
-----fd before fork end-----
-----pre_parent begin from here-----
I-node number: 4725391
Ownership: UID=997, GID=997
parent seek 0
-----pre_parent ends in here-----
-----child begin from here-----
I-node number: 4725391
Ownership: UID=997, GID=997
child seek 3
-----child ends in here-----
-----pos_parent begin from here-----
I-node number: 4725391
Ownership: UID=997, GID=997
parent seek 3
-----pos_parent ends in here-----
git@20373785:~/20373785$
```

- 通过父子进程分别输出的文件描述符信息可以判断他们共享了文件描述符，而子进程更改定位指针后父进程中指针也发生了变化，因此可以判断父子进程也共享了定位指针。

Thinking 5.10

- `struct Fd` 是文件描述符结构，为内存数据，定义如下：

```
struct Fd {
    u_int fd_dev_id;    // 文件对应的设备id
    u_int fd_offset;    // 当前读写偏移量
    u_int fd_omode;     // 文件打开模式（只读/只写/读写）
};
```

- `struct Filefd` 是文件描述符，文件id，文件控制块的共同结构，为内存数据，定义如下：

```
struct Filefd {
    struct Fd f_fd;     // 文件描述符
    u_int f_fileid;     // 全局文件编号
    struct File f_file; // 文件控制块
};
```

- `struct Open` 是用于保存系统中已打开文件的结构，为内存数据，定义如下：

```
struct Open {
    struct File *o_file;    // 文件控制块
    u_int o_fileid;         // 全局文件编号
    int o_mode;             // 文件打开模式（只读/只写/读写）
    struct Filefd *o_ff;    // 文件描述符
};
```

- 其中，`struct Filefd` 和 `struct Open` 中指向文件控制块的 `struct File` 成员记录了对应磁盘上物理实体的指针。

Thinking 5.11

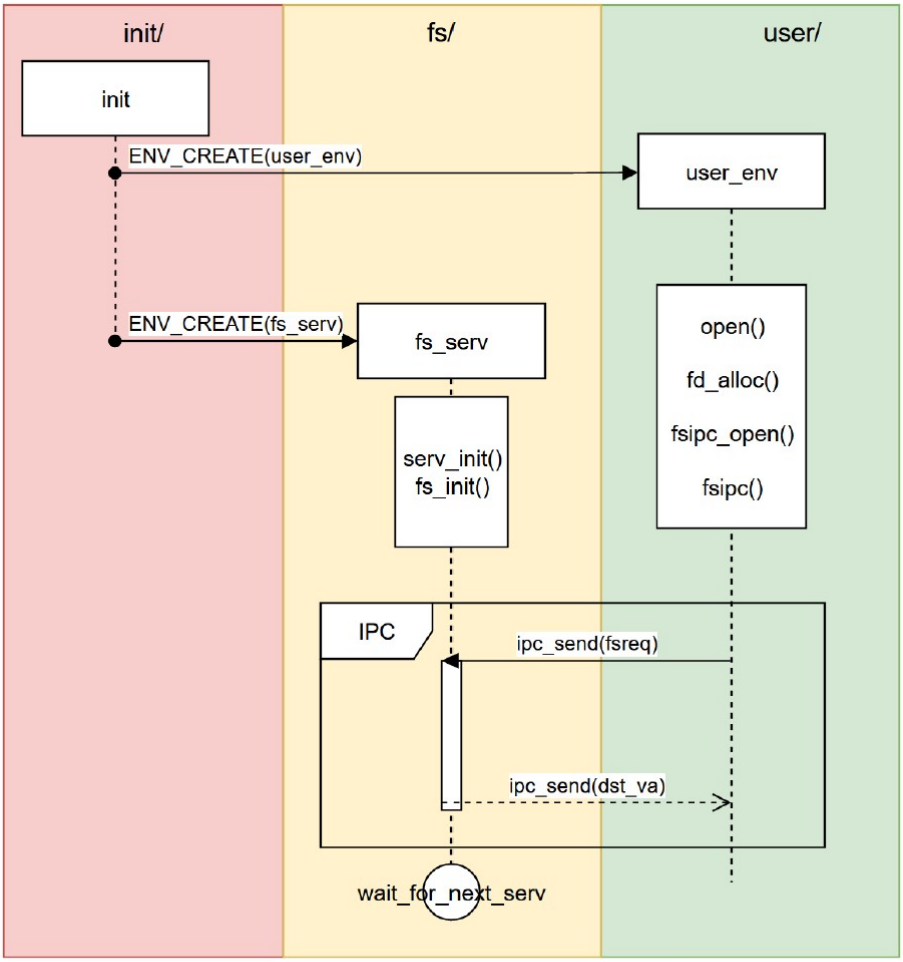


图 5.6: 文件系统服务时序图

- 在上方UML顺序图中，箭头实线代表简单消息，箭头虚线代表返回消息，矩形框代表循环片段。
- 文件系统服务时序流程如下：
  - `init/init.c` 中创建 `user_env` 用户进程，并进入 `user/file.c` 中通过用户接口进行文件操作；
  - `init/init.c` 中创建 `fs_serv` 文件系统服务进程，并进入 `fs/serv.c` 中进行文件系统初始化；
  - 用户进程通过IPC机制向文件系统服务进程发送请求，进行相应的文件操作；
  - 文件系统服务进程接收用户进程的请求，并执行对应的文件操作，完成后将结果通过IPC机制返回给用户进程；
  - 文件系统服务进程完成用户请求后将进入等待状态，等待下一次文件请求。

Thinking 5.12

`serve` 函数在执行过程中每次循环都调用了 `ipc_send` 在进程间发送信息，如果目标进程此时不在接收状态，当前进程将进入 `NOT_RUNNABLE` 状态并被调度，而不会一直占用处理器资源。故 `serve` 函数中的 `for( ; ; )` 循环不会一直处于运行状态，内核也就不会陷入 `panic` 。

实验难点

Gxemul外设

Gxemul提供了若干地址对于外设的映射，可以通过对特定地址的读写来模拟访问外设的过程。需要注意的是，Gxemul文档中提供的地址是物理地址，而在进程中访问外设时需要使用对应的虚拟地址。在MOS系统中，我们通过虚拟地址的`kseg1`段来访问外设，由于该段地址实现了物理地址和虚拟地址硬件级别的直接映射，因此只需要在物理地址加上偏移值`0xA0000000`即可得到对应在`kseg1`段的物理地址。

cons

模拟控制台，主要实现从控制台读写字符的功能。

address: `0x10000000`

Offset	Effect
`0x00`	Read: getchar() (non-blocking; returns 0 if no char was available) Write: putchar(ch)
`0x10`	Read or write: halt() (Useful for exiting the emulator.)

disk

模拟磁盘，主要实现对IDE磁盘的访问和读写功能。

address: `0x13000000`

Offset	Effect
`0x0000`	Write: Set the offset (in bytes) from the beginning of the disk image. This offset will be used for the next read/write operation.
`0x0008`	Write: Set the high 32 bits of the offset (in bytes). (*)
`0x0010`	Write: Select the IDE ID to be used in the next read/write operation.
`0x0020`	Write: Start a read or write operation. (Writing `0` means a Read operation, a `1` means a Write operation.)
`0x0030`	Read: Get status of the last operation. (Status 0 means failure, non-zero means success.)
`0x4000 - 0x41ff`	Read/Write: 512 bytes data buffer.

rtc

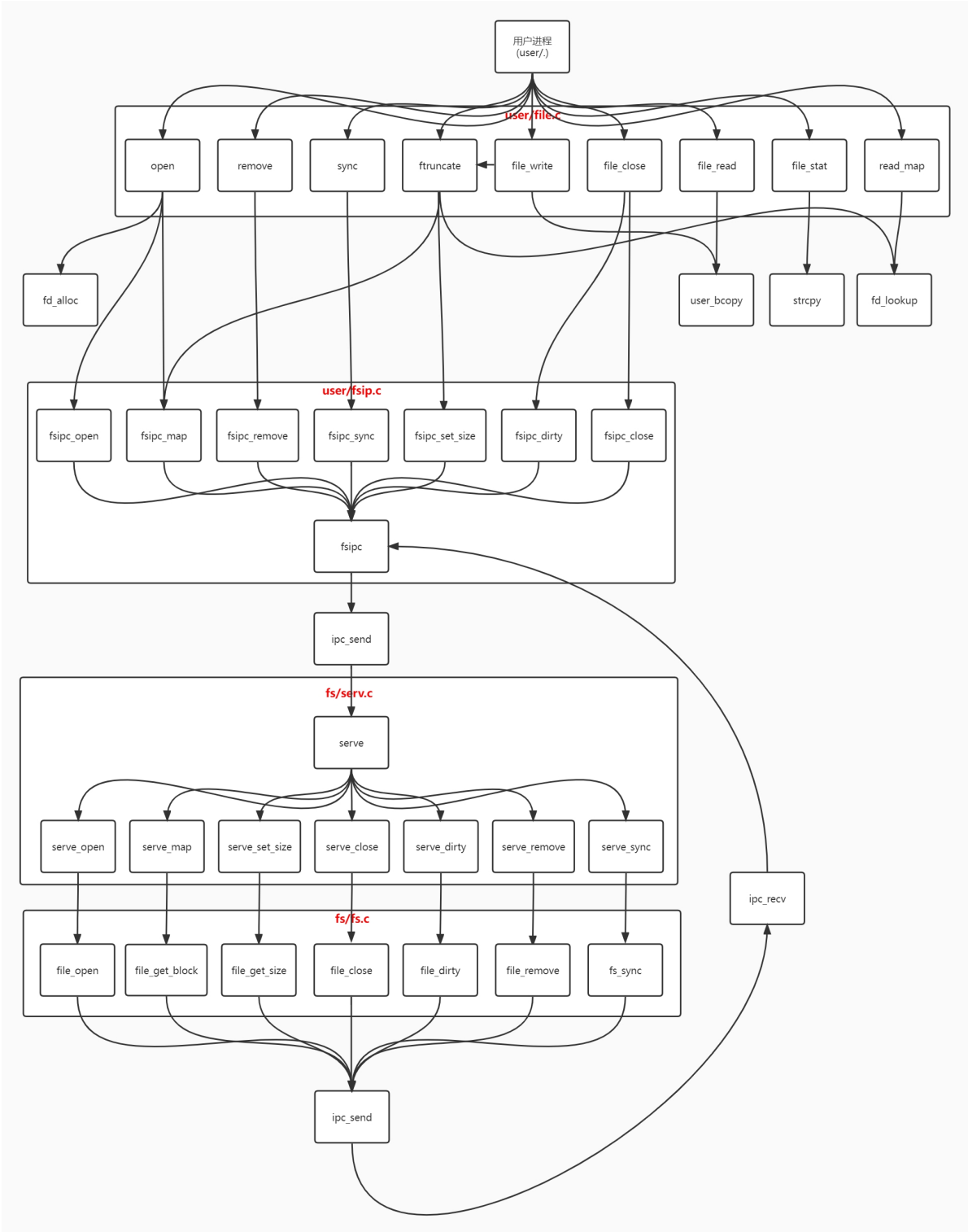
模拟时钟，主要实现对时钟信号的访问。

address: `0x15000000`

Offest	Effect
0x0000	Read or Write: Trigger a clock update (a gettimeofday() on the host).
0x0010	Read: Seconds since 1st January 1970
0x0020	Read: Microseconds
0x0100	Read: Get the current timer interrupt frequency. Write: Set the timer interrupt frequency. (Writing 0 disables the timer.)
0x0110	Read or Write: Acknowledge one timer interrupt. (Note that if multiple interrupts are pending, only one is acknowledged.)

文件系统结构

文件系统的实现流程如下：



用户进程通过 `user/file.c` 中的用户接口调用相关的文件系统服务，部分函数调用 `user/fsipc.c` 内的通信函数与文件服务进程进行通信。文件服务进程运行在 `fs.serv.c` 上，在接收到用户请求后分发至具体的文件服务函数进行处理，之后将处理结果再通过 `IPC` 系统发送回用户进程。

体会感想

为完整实现MOS操作系统中的文件系统，lab5的代码相较于lab4添加了许多新内容，但指导书中只介绍了其中的一部分代码，仍有许多函数不在课下理解考察范围内，但却是一个完整的文件系统所必须的内容。同时，考虑到操作系统的安全性和稳定性，MOS操作系统的文件系统运行在用户态，因此涉及到的函数调用关系也比较复杂，需要频繁与请求文件服务的用户进程和内核进程进行交互，在一定程度上提高了代码的理解难度。

宏观来看，lab5中所实现的文件系统可以归纳为几层抽象，首先是 `fs/ide.c` 中对于磁盘块的抽象，`fs/serv.c` 中对于文件进程的抽象， `user/fsipc.c` 中对于文件系统进程通信的抽象，`user/file.c` 中面向用户的文件操作的抽象。在分别理解每个层级所实现的功能以及他们之间的关系之后，便能较为清晰地理解文件系统了。

在完成操作系统实验的过程中，不仅需要理解每个具体函数的功用以及函数之间的调用关系，还需要从宏观层面归纳并理解操作系统实现功能的不同层级，只有时刻抱有这种结构化、系统化的思维，才能更加深刻地理解操作系统的各项实现原理。