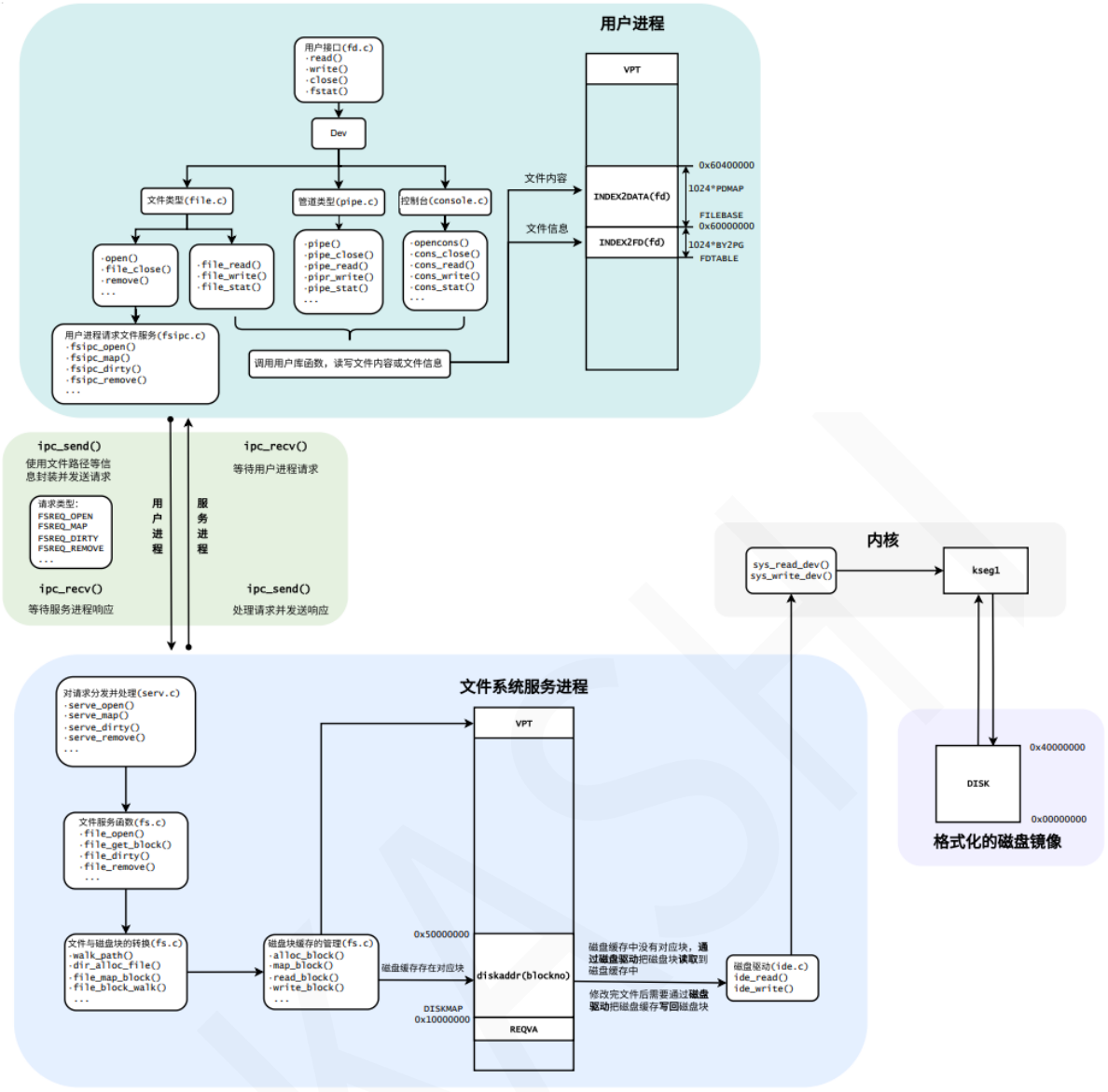
文件系统



内存映射 I/O

外设地址宏定义

```
// include/drivers/dev_cons.h
#define DEV_CONS_ADDRESS 0x10000000
#define DEV_CONS_LENGTH 0x00000020
#define DEV_CONS_PUTGETCHAR 0x0000
#define DEV_CONS_HALT 0x0010


// include/drivers/dev_disk.h
/* Definitions used by the "disk" device in GXemul.
 * This file is in the public domain.*/
#define DEV_DISK_ADDRESS 0x13000000
#define DEV_DISK_OFFSET 0x0000 // 写：设置下一次读写操作时的磁盘镜像偏移的字节数（4 字节）
#define DEV_DISK_OFFSET_HIGH32 0x0008 // 写：设置高 32 位的偏移的字节数（4 字节）
#define DEV_DISK_ID 0x0010 // 写：设置下一次读写操作的磁盘编号（4 字节）
```

```
#define DEV_DISK_START_OPERATION 0x0020 // 写：开始一次读写操作（写 0 表示读操作，1 表示写操作）（4 字节）
#define DEV_DISK_STATUS 0x0030 // 读：获取上一次操作的状态返回值（读 0 表示失败，非 0 则表示成功）（4 字节）
#define DEV_DISK_BUFFER 0x4000 // 读/写：512 字节的读写缓存
#define DEV_DISK_BUFFER_LEN 0x200

/*  Operations:  */
#define DEV_DISK_OPERATION_READ 0
#define DEV_DISK_OPERATION_WRITE 1


// include/drivers/dev_rtc.h
/* Definitions used by the "rtc" device in GXemul.
 * This file is in the public domain.*/
#define DEV_RTC_ADDRESS 0x15000000
#define DEV_RTC_LENGTH 0x00000200

#define DEV_RTC_TRIGGER_READ 0x0000
#define DEV_RTC_SEC 0x0010
#define DEV_RTC_USEC 0x0020

#define DEV_RTC_HZ 0x0100
#define DEV_RTC_INTERRUPT_ACK 0x0110
```

读写外设系统调用

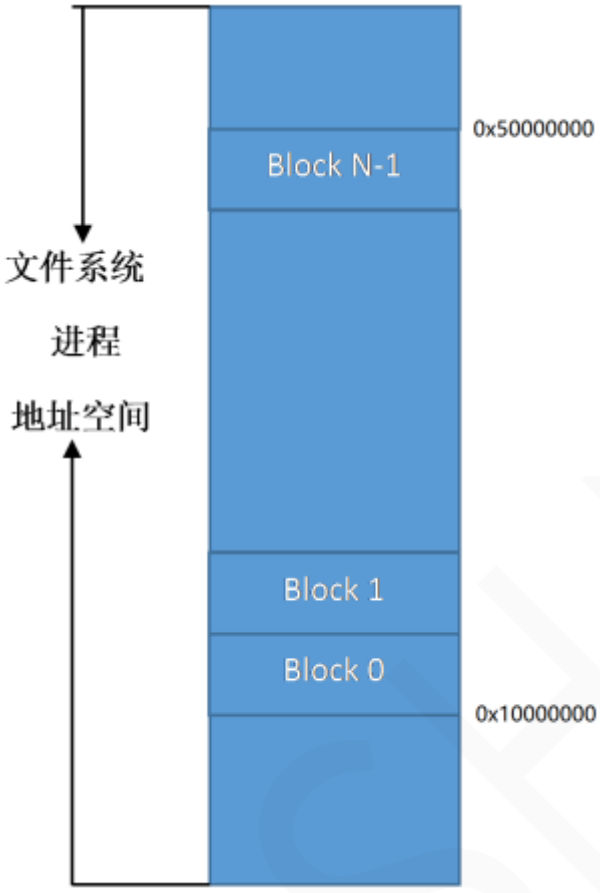
```
// kern/syscall_all.c

/* 读写外设
使用 is_illegal_va_range 检查 va 是否在 UTEMP 到 UTOP 之间，不是返回 -E_INVAL
检查长度是否不小于0，不是返回 -E_INVAL
检查物理地址范围是否在相应外设的范围中，不在返回 -E_INVAL
使用 memcpy 拷贝数据
*/
int sys_write_dev(u_int va, u_int pa, u_int len);
int sys_read_dev(u_int va, u_int pa, u_int len);
```

相应的系统调用

```
// user/lib/syscall_lib.c
int syscall_write_dev(void *va, u_int dev, u_int len);
int syscall_read_dev(void *va, u_int dev, u_int len);
```

块缓存



磁盘相关宏定义

```
// fs/serv.h
#define PTE_DIRTY 0x0002 // 在页表项权限中表示该页对应的磁盘块已被写

#define DISKNO 1 // 我们使用的磁盘号，用于磁盘读写

#define BY2SECT 512 // 磁盘扇区大小，用于磁盘映射与读写
#define SECT2BLK (BY2BLK / BY2SECT) // 一块多少扇区

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP 0x10000000 // 块缓存映射起始地址，用于块映射，计算相应盘块映射到的位置

/* Maximum disk size we can handle (1GB) */
#define DISKMAX 0x40000000 // 最大允许磁盘空间
```

全局变量

```
// fs/fs.c
struct Super *super; // 超级块数据
uint32_t *bitmap; // 磁盘块空闲位图，为1表示空闲
```

磁盘相关函数

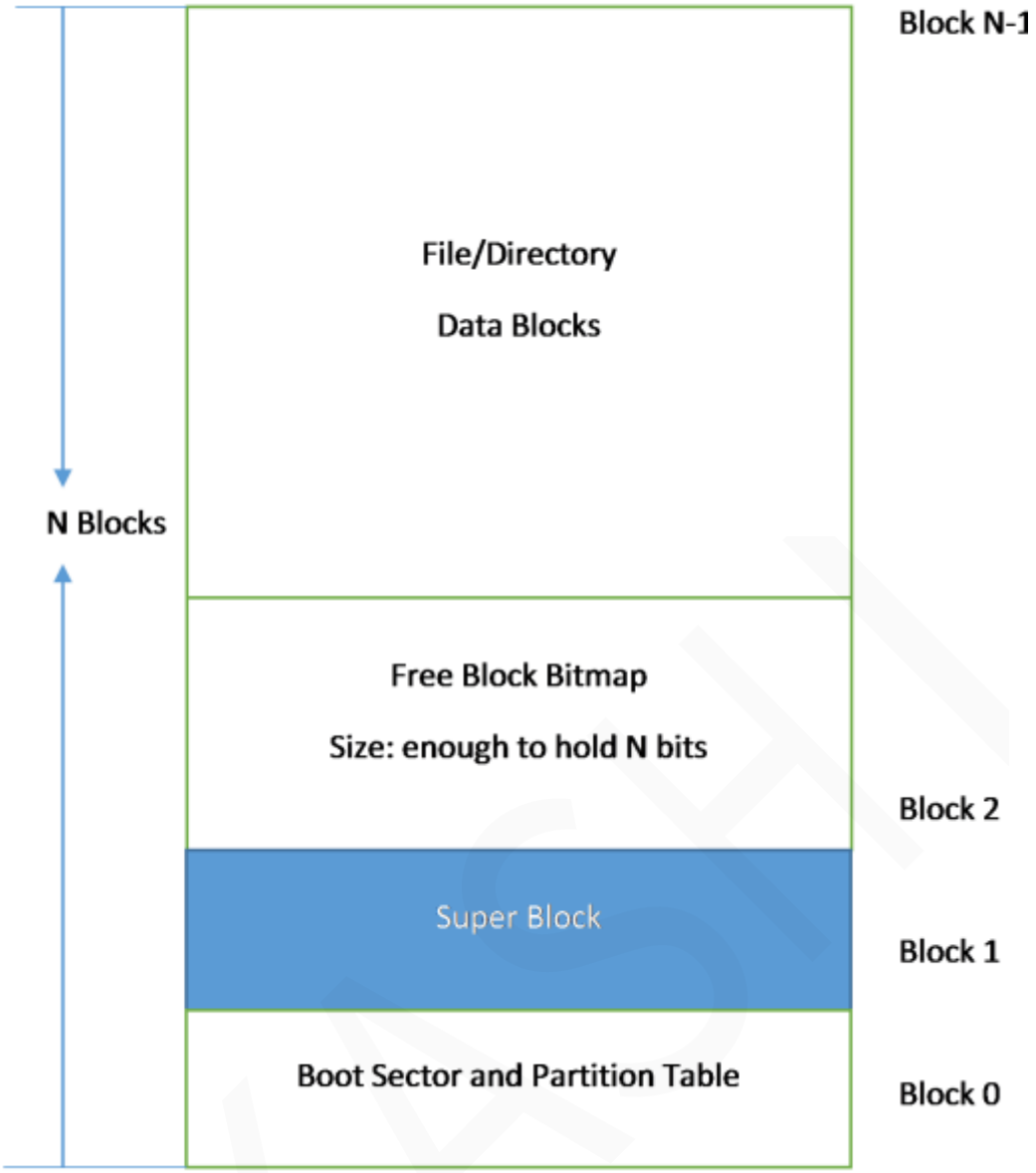
```
// fs/ide.c

/* 从 diskno 的从第 secno 扇区开始的共 nsecs 个扇区读取数据到 dst 指向的位置
计算起始地址和中止地址
进入循环处理每个扇区
将磁盘号写入 DEV_DISK_ADDRESS + DEV_DISK_ID
将读偏移写入 DEV_DISK_ADDRESS + DEV_DISK_OFFSET
将读操作码写入 DEV_DISK_ADDRESS + DEV_DISK_START_OPERATION
从 DEV_DISK_ADDRESS + DEV_DISK_BUFFER 读取数据到 dst + off
从 DEV_DISK_ADDRESS + DEV_DISK_STATUS 读取状态码，为 0 则报错
*/
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);

/* 向 diskno 的从第 secno 扇区开始的共 nsecs 个扇区写入来自 src 的数据
计算起始地址和中止地址
进入循环处理每个扇区
将磁盘号写入 DEV_DISK_ADDRESS + DEV_DISK_ID
将写偏移写入 DEV_DISK_ADDRESS + DEV_DISK_OFFSET
将要写入的数据从 src + off 写入 DEV_DISK_ADDRESS + DEV_DISK_BUFFER
将写操作码写入 DEV_DISK_ADDRESS + DEV_DISK_START_OPERATION
从 DEV_DISK_ADDRESS + DEV_DISK_STATUS 读取状态码，为 0 则报错
*/
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);
```

文件系统概述

磁盘文件系统布局



文件相关宏定义

```
// user/include/fs.h
// Bytes per file system block - same as page size
#define BY2BLK BY2PG // 一个盘块的大小，与页大小相同，用于块读写与映射等等
#define BIT2BLK (BY2BLK * 8)

// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN 128 // 文件名最长大小，在文件创建、查找时用到

// Maximum size of a complete pathname, including null
#define MAXPATHLEN 1024 // 文件路径最大长度，根据路径查找文件时用到

// Number of (direct) block pointers in a File descriptor
#define NDIRECT 10 // 直接指针数量，建立/访问文件到磁盘块的链接时用到
#define NINDIRECT (BY2BLK / 4) // 总共允许的指针数量，后面的都在间接指针指向的块中存储

#define MAXFILESIZE (NINDIRECT * BY2BLK) // 文件的最大大小，用于在文件映射等等时候检查是否越界

#define BY2FILE 256 // 一个File结构体的大小，用于在块中定位结构体
```

部分结构体定义

```
// user/include/fs.h

// 磁盘超级块
struct Super {
    uint32_t s_magic;   // Magic number: 魔数，为一个常量，用于标识该文件系统
    uint32_t s_nblocks; // 记录本文件系统有多少个磁盘块，在本文件系统中为 1024
    struct File s_root; // 根目录文件控制块，其 f_type 为 FTYPE_DIR，f_name 为“/”，注意是控制块结构体不是指针
};

// 文件控制块，用于保存文件静态信息，在打开、编辑、删除等操作时用到
struct File {
    char f_name[MAXNAMELEN]; // 文件名，最长128字节
    uint32_t f_size;     // file size in bytes
    /*
    FTYPE_REG : 普通文件
    FTYPE_DIR : 目录
    */
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT]; // 磁盘块号，文件的直接指针，每个文件控制块有10个，用来记录文件的数据块在磁盘上的位置，每个磁盘块4KB，也就是说文件最大40KB；大于40KB使用下面的间接指针
    uint32_t f_indirect; // 指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针

    struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory. 指向文件所属的目录
    char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed)); // 让整数个文件结构体占满一整个磁盘块，填充剩下的字节
```

注意相应的函数在什么时候能够调用，函数中是否检查 bitmap 或 super 等全局变量的有效性。

创建磁盘镜像（文件系统的生成）

改部分不被编译进内核，在 Linux 环境下作为工具被使用，而不是运行在虚拟机中。

```
// tools/fsformat.c

#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];
    uint32_t type;
} disk[NBLOCK]; // 磁盘数据


// 检查参数正确性，根据参数路径指向的是文件还是目录，选择调用 write_file 或 write_directory 在根目录下添加文件
int main(int argc, char **argv);
```

```
// 初始化磁盘镜像，设置 0 块为启动引导块，1 块为超级块，计算位图块数量并设置，初始化位图，设置超级块魔数、块数
void init_disk();

// 在传入的目录下挂载新的文件，使用 create_file 获得属于该目录的文件控制块，在 f_direct 或 f_indirect 相应区域中，设置新文件类型、名称，读取文件数据，使用 save_block_link 和 next_block 保存并链接
void write_file(struct File *dirf, const char *path);

// 在传入的目录下挂载新的目录，使用 create_file 获得属于该目录的文件控制块，在 f_direct 或 f_indirect 相应区域中，设置新目录类型、名称，递归添加子目录内容
void write_directory(struct File *dirf, char *path);

// 在当前目录下新建一个子目录/文件，返回对应的文件结构体。现在已经拥有的物理盘块中寻找空闲位置，如果没有则调用 make_link_block 分配新的块
struct File *create_file(struct File *dirf);

// 使用 next_block 申请新的物理块，调用 save_block_link
int make_link_block(struct File *dirf, int nblk);

// 检查 nblk，判断文件是否过大，在 f 指向的文件控制块中，nblk 代表的第多少块的位置，建立指向 bno 物理块的链接，该函数不维护文件大小
void save_block_link(struct File *f, int nblk, int bno);

// 将下一个可用的物理块的类型设置为 type，并返回其块号
int next_block(int type)
```

文件系统的用户接口

相关宏定义

```
// user/include/fd.h

#define MAXFD 32 // 最大文件描述符数，即系统中最多可以打开32个文件
#define FILEBASE 0x60000000 // 文件系统的基地址，即文件系统占用的内存空间的起始地址
#define FDTABLE (FILEBASE - PDMAP) // 文件描述符表的地址，在文件系统之前，一个页表页所管辖的 4M 空间

#define INDEX2FD(i) (FDTABLE + (i)*BY2PG) // 给定文件描述符的索引i，计算该文件描述符在文件描述符表中的地址，每个文件描述符占 4K（1页）地址空间
#define INDEX2DATA(i) (FILEBASE + (i)*PDMAP) // 给定文件描述符的索引i，计算该文件描述符指向的文件数据的地址，每个文件对应的数据占 4M 地址空间


// Device struct:
// It is used to read and write data from corresponding device.
// We can use the five functions to handle data.
// There are three devices in this OS: file, console and pipe.
struct Dev {
    int dev_id;
    char *dev_name;
    int (*dev_read)(struct Fd *, void *, u_int, u_int);
    int (*dev_write)(struct Fd *, const void *, u_int, u_int);
```

```
    int (*dev_close)(struct Fd *);
    int (*dev_stat)(struct Fd *, struct Stat *);
    int (*dev_seek)(struct Fd *, u_int);
};

// file descriptor
struct Fd {
    u_int fd_dev_id; // 设备号
    u_int fd_offset; // 读写偏移
    u_int fd_omode; // 打开模式
};

// State
struct Stat {
    char st_name[MAXNAMELEN];
    u_int st_size;
    u_int st_isdir;
    struct Dev *st_dev;
};

// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file; // 注意这个不是指针
};
```

```
// user/include/lib.h

// 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 */

// Unimplemented open modes
#define O_CREAT 0x0100 /* create if nonexistent */
#define O_TRUNC 0x0200 /* truncate to zero length */
#define O_EXCL 0x0400  /* error if already exists */
#define O_MKDIR 0x0800 /* create directory, not regular file */
```

用户全局变量

```
// user/lib/fd.c

// 静态的，仅在该文件中有效
// 这是一个数组变量 devtab 的定义，其中存储了指向各个设备结构体的指针。
static struct Dev *devtab[] = {&devfile, &devcons,
#if !defined(LAB) || LAB >= 6
                   &devpipe,
#endif
                   0};
```

```
// user/lib/file.c

// 定义文件设备
struct Dev devfile = {
    .dev_id = 'f',
    .dev_name = "file",
    .dev_read = file_read,
    .dev_write = file_write,
    .dev_close = file_close,
    .dev_stat = file_stat,
};
```

```
// user/lib/console.c

// 定义控制台
struct Dev devcons = {
    .dev_id = 'c',
    .dev_name = "cons",
    .dev_read = cons_read,
    .dev_write = cons_write,
    .dev_close = cons_close,
    .dev_stat = cons_stat,
};
```

文件描述符

```
// user/lib/fd.c

// 根据相应设备 id，例如‘f’，在 devtab 中寻找相应设备，并回传
int dev_lookup(int dev_id, struct Dev **dev);

// 申请新的文件描述符，通过检查描述符表中相应位置的页目录和页表项是否有效，检查相应描述符是否空闲，返回第一个空闲的描述符表位置，成功返回 0，失败返回 -E_MAX_OPEN
int fd_alloc(struct Fd **fd);

// 通过简简单单解除描述符所指页的映射，标记描述符空闲
void fd_close(struct Fd *fd);

// 通过文件描述符号查找相应文件描述符结构体并回传，过大返回 -E_INVAL，相应页无效返回 -E_INVAL
int fd_lookup(int fdnum, struct Fd **fd);

// 通过文件描述符结构体指针返回文件数据指针，调用 fd2num
void *fd2data(struct Fd *fd);

// 返回文件描述符结构体指针对应的描述符号
int fd2num(struct Fd *fd);

// 返回描述符号对应的文件描述符结构体指针
int num2fd(int fd); // 为什么不返回 void *

// 首先调用 fd_lookup 和 dev_lookup 寻找文件描述符和设备，没找到返回错误码，找到则调用设备结构体中的 dev_close 函数关闭设备，然后调用 fd_close 关闭文件描述符
int close(int fdnum);
```

```
// 关闭所有文件描述符
void close_all(void);

// 将相应文件描述符复制，使用 syscall_mem_map 将文件描述符和文件数据映射 (PTE_D | PTE_LIBRARY)，如果成功返回新文件描述符，失败则错误码
int dup(int oldfdnum, int newfdnum);

// 从文件描述符所指文件中，将最多 n 个字节读入到缓冲区
int read(int fdnum, void *buf, u_int n);

// 忙等读入，直到读入 n 个字节后返回
int readn(int fdnum, void *buf, u_int n);

// 如果文件描述符不存在，文件设备不存在或者文件只读（-E_INVAL）则返回错误码，否则调用 dev_write(fd, buf, n, fd->fd_offset) 写入数据
int write(int fdnum, const void *buf, u_int n);

// 如果文件描述符不存在则返回相应错误码，否则修改其 offset
int seek(int fdnum, u_int offset);

// 如果文件描述符不存在或文件设备不存在则返回相应错误码，否则初始化 stat 结构体并传给 dev_stat 进行状态查询并返回其结果
int fstat(int fdnum, struct Stat *stat);

// 以只读模式打开文件，调用 fstat(fd, stat) 并关闭，返回其返回值
int stat(const char *path, struct Stat *stat);
```

文件服务调用

```
// user/include/fsreq.h

#define FSREQ_OPEN 1
#define FSREQ_MAP 2
#define FSREQ_SET_SIZE 3
#define FSREQ_CLOSE 4
#define FSREQ_DIRTY 5
#define FSREQ_REMOVE 6
#define FSREQ_SYNC 7

struct Fsreq_open {
    char req_path[MAXPATHLEN];
    u_int req_omode;
};

struct Fsreq_map {
    int req_fileid;
    u_int req_offset; // 以字节为单位
};

struct Fsreq_set_size {
    int req_fileid;
    u_int req_size;
};

struct Fsreq_close {
```

```
    int req_fileid;
};

struct Fsreq_dirty {
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_remove {
    char req_path[MAXPATHLEN];
};
```

必须是第二个进程，即 envs[1] 所代表的进程。由于 ipc 接收到的数据是错误码，即代表是否成功，故以下函数返回值应该都代表是否成功。

```
// user/lib/fsipc.c

u_char fsipcbuf[BY2PG] __attribute__((aligned(BY2PG)));

// 向文件服务程序发送消息，并等待回复。调用 ipc_send 向 envs[1] 发送，以 type 为数据值，
fsreq 为源地址，dstva 为目标地址，并传送相应权限，并返回 ipc_recv(&whom, dstva, perm) 的
结果（env->env_ipc_value），注释说返回的是成功与否。以回调方式传回 perm，但 whom 似乎没有用
到，是否可以 NULL？
static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm);

// 在缓冲页上建立 Fsreq_open 结构体，保存相关数据，调用 fsipc 并返回其结果，其中源地址为缓冲
页，目标地址为 fd，权限似乎没有用到，是否可以用 NULL？
int fsipc_open(const char *path, u_int omode, struct Fd *fd);

// 以同样的方式通过 Fsreq_map 传送文件 id 和偏移，接收被映射的页（dstva）。检查回传的权限除了
PTE_D 和 PTE_LIBRARY 是否只有 PTE_V，不是则崩溃
int fsipc_map(u_int fileid, u_int offset, void *dstva);

// 以同样的方式通过 Fsreq_set_size 传送文件 id 和新大小，但不设置权限回传以及 dstva
int fsipc_set_size(u_int fileid, u_int size);

// 以同样的方式通过 Fsreq_close 传送文件 id，不设置回传权限以及 dstva
int fsipc_close(u_int fileid);

// 以同样的方式通过 Fsreq_dirty 传送文件 id 和偏移，不设置回传权限以及 dstva
int fsipc_dirty(u_int fileid, u_int offset);

// 以同样的方式通过 Fsreq_remove 传送文件路径，
int fsipc_remove(const char *path); // 学生完成

// 直接 return fsipc(FSREQ_SYNC, fsipcbuf, 0, 0);
int fsipc_sync(void);
```

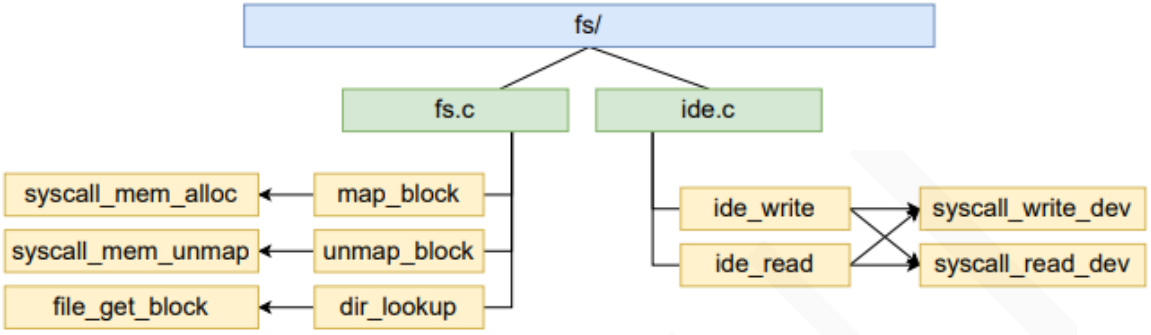
瞎给的权限的问题，但应该问题不大。虽然页面发送实际上是共享，但是文件服务进程只会串行为其他进程提供服务，不会产生交叉，因为当一个进程向其发送数据后，如果文件服务进程正在服务其它进程，则它就还没有准备好接收数据，则调用者会在 ipc_send 中不断尝试发送。

文件

```
// user/lib/file.c

// 懒了，暂略
```

文件服务程序



应该不会被编译进内核，而是独立的，作为用户进程运行，在用户态为上述用户接口提供服务。

相关函数

这部分有返回值的基本上都是成功返回 0，失败返回错误码。

```
// fs/fs.c

// 根据全局位图，判断传入的块号对应的物理盘块是否被使用，不检查全局位图有效性
int block_is_free(u_int);

// 返回物理盘块号对应的盘块在用户虚地址上的映射
void *diskaddr(u_int blockno);

// 检查 va 对应的页目录项和页表项是否有效
int va_is_mapped(void *va);

// 调用 diskaddr 和 va_is_mapped，如果盘块号对应的 va 已经有映射，则返回 va，否则返回空
void *block_is_mapped(u_int blockno);

// 判断 va 对应的页表项是否置脏位，不检查有效性
int va_is_dirty(void *va);

// 检查盘块号对应的 va 是由有效且脏
int block_is_dirty(u_int blockno);

// 使用 syscall_mem_map 为相应盘块号对应的 va 置脏位，如果没有映射则返回 -E_NOT_FOUND
int dirty_block(u_int blockno);

// 调用 ide_write 将相应页面写回磁盘，如果没有映射则引发用户态崩溃
void write_block(u_int blockno);

/* 读入盘块号对应的数据到内存，如果没有映射则建立映射
如果盘块号超过了全局变量超级块中保存的数量，则引发用户进程崩溃
```

```
如果全局变量 bitmap 本身为空（还没读进来）或者相应块空闲，则引发用户进程崩溃
换算得到 va
如果盘块已经加载到内存，且传入 isnew，则将其置零；如果没有，则同样试图修改 isnew，并使用
syscall_mem_alloc 为相应的 va 申请物理页并使用 ide_read 将其读入
如果 blk 非空，则将其指向的位置设置为 va
*/
int read_block(u_int blockno, void **blk, u_int *isnew)

// 申请物理页在相应的 va 建立磁盘块到块缓存的映射，不读入
int map_block(u_int blockno);

// 解除在内存块缓存上的映射，需要时写回
void unmap_block(u_int blockno);

// 将对应磁盘块标记为空闲（修改内存中的全局变量 bitmap ）
void free_block(u_int blockno);

// 从盘块号 3 开始遍历 bitmap，寻找空闲的盘块对应的盘块号并返回，如果没有空闲则返回 -
E_NO_DISK
int alloc_block_num(void);

// 尝试使用 alloc_block_num 申请新盘块号，尝试使用 map_block 建立映射，如果失败，使用
free_block 交还盘块号，返回 -E_NO_MEM。最后返回盘块号
int alloc_block(void);

// 调用 read_block 读入超级块，如果读入失败，魔数不同或是物理盘块数多余盘块缓存，则崩溃
void read_super(void);

// 读入位图，检查引导块和超级块不空闲，确认位图块有效
void read_bitmap(void);

// 在超级块读入后进行乱七八糟的测试
void check_write_block(void);

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

/* 查找文件中保存指向块的链接的位置，并传回，如果使用间接指针，则可能涉及间接块的读入
参数：
    f：一个指向 struct File 类型的指针，表示要操作的文件。
    filebno：表示文件中块的编号，从0开始计数。
    ppdiskbno：一个指向 uint32_t * 类型的指针，表示指向目标块的磁盘块号的指针。
    alloc：表示在确实间接块时是否进行分配。如果设置为0，则函数不会分配新块。
返回值：
    -E_NOT_FOUND：需要申请间接块但 alloc 没有置位
    -E_NO_DISK：申请间接块时磁盘没有空间
    -E_NO_MEM：申请间接块后建立映射时没有足够物理内存
    -E_INVAL：文件块号大于最大块号
如果需要使用间接指针，且当前没有间接块，则申请间接块并建立映射
*/
int file_block_walk(struct File *f, u_int filebno, uint32_t **ppdiskbno, u_int
alloc);
```

```
// 查找相应文件块号对应的磁盘块号（回调传回），不读入到内存。会调用 file_block_walk 寻找记录盘块号的位置。alloc 在该函数体内表示是否在文件相应位置创建新块，同时也传给 file_block_walk 表示是否创建间接块
int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc);

// 清除对应文件中到对应盘块的链接，使用 file_block_walk 寻找保存链接的位置，使用 free_block 释放磁盘块，然后将链接值置0，不解除映射
int file_clear_block(struct File *f, u_int filebno);

// 使用 file_map_block 获取保存文件块号对应的磁盘块号，然后使用 read_block 建立块到块缓存的映射并读入数据，传回指向盘块在内存缓存中位置的指针
int file_get_block(struct File *f, u_int filebno, void **blk);

// 使用 file_map_block 不分配新块，仅获取并检查相应块是否建立从文件到盘块号的链接，是否映射到内存缓存；如果有效，则调用 dirty_block 标记脏
int file_dirty(struct File *f, u_int offset);

// 在目录中寻找文件名对应的文件
int dir_lookup(struct File *dir, char *name, struct File **file);

// 在指定目录下寻找空闲的位置，并通过 **file 将指针传回
int dir_alloc_file(struct File *dir, struct File **file);

/* 这个函数的主要用途是在文件系统中查找特定的文件或目录，或者确定一个路径名对应的文件或目录是否存在。在处理诸如打开文件、创建文件、删除文件等文件操作时，这个函数通常会被调用来获取相关文件和目录的信息。

    char *path：输入参数，表示需要解析的路径。
    struct File **pdir：输出参数，指向找到的文件所在的目录。如果找不到文件，但找到了它应该所在的目录，则设置 *pdir。
    struct File **pfile：输出参数，指向找到的文件。如果找不到文件，则设置 *pfile 为0。
    char *lastelem：输出参数，当找不到文件，但找到了它应该所在的目录时，将路径中的最后一个元素（文件名）复制到 lastelem

初始化：从根目录开始解析路径，并初始化局部变量。

迭代遍历路径中的每个元素：
    如果当前路径元素超过允许的最大长度（MAXNAMELEN），返回错误 -E_BAD_PATH。
    如果当前路径元素所在的目录不是一个目录类型（FTYPE_DIR），返回错误 -E_NOT_FOUND。
    在当前目录中查找与路径元素名称匹配的文件，使用 dir_lookup 函数。如果找到匹配的文件，继续处理下一个路径元素；如果找不到匹配的文件，但已经到达路径的末尾，则设置 *pdir、*pfile 和 lastelem 的值，并返回错误代码；如果找不到匹配的文件，且未到达路径末尾，返回 dir_lookup 的错误代码。

如果遍历完整个路径并找到了目标文件，则设置 *pdir 和 *pfile 的值，并返回成功代码 0。
*/
int walk_path(char *path, struct File **pdir, struct File **pfile, char *lastelem);

// 直接返回 walk_path(path, 0, file, 0) 的结果
int file_open(char *path, struct File **pfile);

// 创建路径对应的文件，并将文件控制块通过 file 返回，应该只能在已有的目录下创建
int file_create(char *path, struct File **file);

// 减少文件大小到 newsize，使用 file_clear_block 抛弃掉后面的块，但不解除映射
```

```
void file_truncate(struct File *f, u_int newsize);

// 如果文件当前更大，调用 file_truncate 减小文件大小，调用 file_flush 将其父目录，没有控制 newsize 对齐到块
int file_set_size(struct File *f, u_int newsize);

// 将对应文件所有磁盘缓冲写回磁盘
void file_flush(struct File *f);

// 将所有脏的磁盘缓冲全部写回磁盘
void fs_sync(void);

// 将文件所有磁盘缓冲及父目录磁盘缓冲写回磁盘
void file_close(struct File *f);

// 调用 walk_path 找到文件，将文件大小设置为0，文件名设置为空。没看懂这里为什么要对文件本身进行 flush
int file_remove(char *path);
```

```
// user/lib/pageref.c

// 给定虚地址，返回页引用次数
int pageref(void *v);
```

文件服务程序

这里的 o_fileid，即文件 id，不同于文件描述符，前者是在全局唯一的 opentab 中的索引，后者是每个进程自己的文件描述符表中的索引。

```
// fs/serv.c

struct Open {
    struct File *o_file; // mapped descriptor for open file
    u_int o_fileid;      // file id
    int o_mode;      // open mode
    struct Filefd *o_ff; // va of filefd page
};

// Max number of open files in the file system at once
#define MAXOPEN 1024
#define FILEVA 0x60000000

// initialize to force into data section
struct Open opentab[MAXOPEN] = {{0, 0, 1}};

// Virtual address at which to receive page mappings containing client requests.
#define REQVA 0x0ffff000
```

```
// fs/serv.c

// 初始化服务，设置 opentab 中每一项的 o_fileid 为当前下标，o_ff 为 FILEVA 开始的相应页
void serve_init(void);
```

```
// 申请新的 Open 块，通过检查页引用数量是否小于2判断是否空闲，为 0 则 syscall_mem_map，为 1 则直接 fileid += MAXOPEN，清理内存，返回相应的 fileid 代表成功，没有空闲块返回 -E_MAX_OPEN
int open_alloc(struct Open **o);

// 根据 fileid 查找 Open 结构体并返回，如果 pageref(o->o_ff) == 1 || o->o_fileid != fileid，即代表着此时这个 Open 块已经被释放或者在释放后分配给了别的进程，返回 -E_INVAL
int open_lookup(u_int envid, u_int fileid, struct Open **po);

// 处理打开文件请求。首先申请一个文件 id，申请失败则发送失败码，但为什么不返回？然后使用 file_open 打开相应文件，失败则发送失败码并返回；然后将文件 id 相应 Open 结构体的 o_file 设置为打开的 f；设置相应 Filefd 结构体（例如设备id）并发回
void serve_open(u_int envid, struct Fsreq_open *rq);

// 处理建立页映射请求。使用 open_lookup 查找相应 Open 结构体，计算块偏移，使用 file_get_block 获取文件块，最后用 ipc_send 发回，具有 PTE_D | PTE_LIBRARY 权限
void serve_map(u_int envid, struct Fsreq_map *rq);

// 设置文件大小。调用 open_lookup 和 file_set_size，ipc 发回 0 代表成功
void serve_set_size(u_int envid, struct Fsreq_set_size *rq);

// 关闭相应文件。调用 open_lookup 和 file_close，ipc 发回 0 代表成功
void serve_close(u_int envid, struct Fsreq_close *rq);

// 移除相应文件。调用 open_lookup 和 file_remove，ipc 发回 0 代表成功
void serve_remove(u_int envid, struct Fsreq_remove *rq);

// 标记相应偏移脏。调用 open_lookup 和 file_dirty，ipc 发回 0 代表成功
void serve_dirty(u_int envid, struct Fsreq_dirty *rq);

// 同步文件。直接 fs_sync(); 同步整个文件系统，然后 ipc 发回 0 表示成功
void serve_sync(u_int envid);

// 死循环，进行服务分发
void serve(void);

int main() {
    user_assert(sizeof(struct File) == BY2FILE);
    debugf("FS is running\n");
    serve_init();
    fs_init();
    serve();
    return 0;
}
```

自己的认识与思考

fs/fs.c 相关问题

在 File 结构体中使用 f_size 表示文件大小，据我观察，对于目录文件，我们保证它的 f_size 对齐到盘块，这也是在 dir_alloc_file 函数中使用 `nblock = dir->f_size / BY2BLK;` 计算其使用的盘块数的基础；而对于普通文件，我们允许它的 f_size 不对齐到盘块，我这样认为的原因是在 fsformat.c 中，我们确定普通文件大小时使用的是 `lseek(fd, 0, SEEK_END);`，并且在更改文件大小时，从 fsipc_set_size 到 serve_set_size 到 file_set_size 再到 file_truncate 均未对 f_size 进行对齐检查。

同时，在 file_truncate 函数中，我们使用如下方式计算文件原本使用的盘块数和修改后使用的盘块数：

```
    old_nblocks = f->f_size / BY2BLK + 1;
    new_nblocks = newsize / BY2BLK + 1;
```

我认为这是由于我们仅能显示修改普通文件的大小，而不能显示修改目录文件的大小，所以 f->f_size 恰好对齐到盘块的几率很低，因此只有很低的概率多计算一个盘块，并且即使多计算了一个盘块也问题不大，如果尝试清除这个盘块但文件又没有这个盘块的话，也只会在 file_clear_block 错误返回，并且这个错误码被忽略了。此外，在 file_flush 函数中，我们也使用了`nblocks = f->f_size / BY2BLK + 1;`的方式计算使用的盘块数。如果我的理解正确的话，使用诸如`(f->f_size + BY2BLK - 1) / BY2BLK`的方式计算是否是更严谨的方案，还是我们当前的方式另有考虑?

到以上部分差不多没有问题，但是比较不理解的地方是在 file_remove 中，我们调用了`file_truncate(f, 0);`这会导致处理过后文件大小为 0，但是依旧有一个数据块仍未被解除，因为 new_nblocks 计算结果为 1，因此并不会对第 0 块使用 free_block。同时我们通过将文件名置空表示当前文件控制块空闲，但事实上它似乎还拥有一个数据块。是我的理解存在问题，我们另有设计呢?

fs/serv.c

在 serve_open 中有以下代码段:

```
    // Find a file id.
    if ((r = open_alloc(&o)) < 0) {
        ipc_send(envid, r, 0, 0);
    }
```

这里发生错误在 ipc_send 之后为什么不返回呢?

对整体流程的思考

文件服务进程使用 0x10000000 到 0x4fffffff 的地址空间作为磁盘块映射，使用 0x60000000 开始的地址作为所有进程的文件描述符表，用户进程的文件描述符是对这里的表项的共享。而用户进程使用 0x60000000 之前的 4M 空间作为文件描述符表，之后的 32（最大文件数目）* 4M 映射文件数据。

当在用户进程下打开文件时，分配 Fd 并传给文件服务进程，后者读取文件，创建 Open 结构体，复制一份文件的 File 结构体拼接成 FileFd 结构体，放在自己的全局文件描述符表中，并将这个共享给用户。用户进程收到之后将让文件服务进程映射全部数据块，并读入数据。最后的结果是文件服务进程，将分散在自己 0x10000000 到 0x4fffffff 的空间的文件的各个数据块，共享到用户进程 0x60000000 之上的属于这个文件的连续的 4M 空间之中。

关闭时，会直接将全部设置为脏，然后全部回写。

如果对于打开的文件进行修改大小操作，用户进程会先改文件描述符中 File 结构体的副本，然后让文件服务进程改相应块缓存上的数据。

仍待研究的问题

首先是关于不同进程打开的文件的隔离问题，曾经看明白了，但是没记下来，之后又忘了。

思考题部分

5-1

可能会引发外设不能即使得到数据，或是CPU不能接收到来自外设的最新的数据的问题。以控制台为例，如果通过 Cache 访问，则有可能导致希望输出到控制台的数据被写入 Cache，但是由于高速缓存更新机制，这个数据很可能不会被立刻写入相应物理地址，即不会被输送到控制台相关的寄存器中，而是在此 Cache 块被淘汰时才将相应数据写回，输送给控制台。控制台不能读取 Cache 中的数据，进而只能拿到 Cache 淘汰时写回的数据，这显然与期望不符。另外，对于键盘等输入设备，如果通过 Cache 访问，则当 Cache 相应块有效时，CPU会从 Cache 中取数据，而不是通过相应的内存访问从键盘的寄存器中获取数据。这将导致CPU无法获取正确的来自键盘的数据。

5-2

一个磁盘块中最多 16 个文件控制块，一个目录下最多有 16K 个文件，最大支持 4M 的文件。

5-3

在虚拟地址空间中预留了 1GB 的块缓存区域，故最大支持 1GB 磁盘。

5-4

我认为以下宏定义比较重要：

```
// user/include/fs.h
// Bytes per file system block - same as page size
#define BY2BLK BY2PG // 一个盘块的大小，与页大小相同，用于块读写与映射等等
#define BIT2BLK (BY2BLK * 8)

// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN 128 // 文件名最长大小，在文件创建、查找时用到

// Maximum size of a complete pathname, including null
#define MAXPATHLEN 1024 // 文件路径最大长度，根据路径查找文件时用到

// Number of (direct) block pointers in a File descriptor
#define NDIRECT 10 // 直接指针数量，建立/访问文件到磁盘块的链接时用到
#define NINDIRECT (BY2BLK / 4) // 总共允许的指针数量，后面的都在间接指针指向的块中存储

#define MAXFILESIZE (NINDIRECT * BY2BLK) // 文件的最大大小，用于在文件映射等等时候检查是否越界

#define BY2FILE 256 // 一个File结构体的大小，用于在块中定位结构体
```

```
// fs/serv.h
#define PTE_DIRTY 0x0002 // 在页表项权限中表示该页对应的磁盘块已被写

#define DISKNO 1 // 我们使用的磁盘号，用于磁盘读写

#define BY2SECT 512 // 磁盘扇区大小，用于磁盘映射与读写
#define SECT2BLK (BY2BLK / BY2SECT) // 一块多少扇区

/* Disk block n, when in memory, is mapped into the file system
```

```
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP 0x10000000 // 块缓存映射起始地址，用于块映射，计算相应盘块映射到的位置

/* Maximum disk size we can handle (1GB) */
#define DISKMAX 0x40000000 // 最大允许磁盘空间
```

5-5

fork 前后的父子进程会共享文件描述符和定位指针， 因为 fork 时将用户空间的全部有效内容均进行映射，文件描述符表也包含在其中。由于文件描述符表具有 PTE_D | PTE_LIBRARY 权限，在 duppage 时进行简单的映射而并不是写时复制，故 fork 前后父子进程共享文件描述符和定位指针。

5-6

```
// 文件控制块，用于保存文件静态信息，在打开、编辑、删除等操作时用到
struct File {
    char f_name[MAXNAMELEN]; // 文件名，最长128字节
    uint32_t f_size;     // file size in bytes
    /*
    FTYPE_REG : 普通文件
    FTYPE_DIR : 目录
    */
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT]; // 磁盘块号，文件的直接指针，每个文件控制块有10个，用来记录文件的数据块在磁盘上的位置，每个磁盘块4KB，也就是说文件最大40KB；大于40KB使用下面的间接指针
    uint32_t f_indirect; // 指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针

    struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory. 指向文件所属的目录
    char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed)); // 让整数个文件结构体占满一整个磁盘块，填充剩下的字节
```

```
// file descriptor
struct Fd {
    u_int fd_dev_id; // 设备号
    u_int fd_offset; // 读写偏移
    u_int fd_omode; // 打开模式
};

// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file; // 注意这个不是指针
};
```

```
struct Open {
    struct File *o_file; // 指向磁盘块映射中的 File 结构体
    u_int o_fileid;      // file id
    int o_mode;      // open mode
    struct Filefd *o_ff; // va of filefd page
};
```

File 结构体主要用于记录文件的各种静态的信息，具体说明见上。

Fd 结构体为各个进程记录打开的文件的动态信息，其中 dev_id 表明这个文件使用的是什么设备，告诉用户程序应该使用什么函数去操作相应的问文件；同时记录读写偏移，表明当前读写到文件的什么位置，omode 则表示文件以什么权限打开。

Filefd 结构体记录文件打开后的信息，首先为一个 Fd 结构体，作用见上，其次是与文件服务进程 Open 结构体数组中的数据相对应的 f_fileid，用于让文件服务进程找到相应的 Open 结构体，并进行操作；f_file 结构体表明文件的各种静态信息。

5-7

从圆点开始的箭头表示进程的创建；细箭头表示进程间通信。我们的操作系统让用户进程发送相应的文件系统请求给文件系统服务进程，后者处理完毕后再使用进程间通信将相应消息返回来实现的。