2024 操作系统 Lab5 串讲

2024 操作系统助教组

Lab5 概述

Lab5 主要包括以下三部分内容：

- 如何控制磁盘以实现数据的持久化？

  （**外设控制**，包括设备读写系统调用、IDE 磁盘驱动）

- 如何组织磁盘中的数据存储？

  （**文件系统**，包括磁盘布局、文件系统数据结构、文件系统服务进程等内容）

- 如何进行文件操作？

  （**文件系统接口**，包括 `open`、`read`、`write` 等一系列用户库函数）

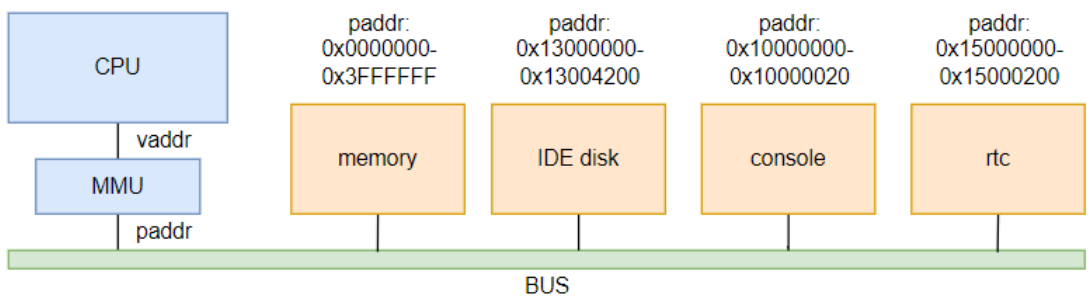
外设控制

设备、MMIO

CPU 通过读写**设备控制器上的寄存器**（注意与 CPU 寄存器区分）实现对设备的控制和通信。

在 MIPS 体系结构下，我们使用 MMIO（内存映射 IO）机制访问设备寄存器。MMIO 使用不同的物理内存地址为设备寄存器编址，将一部分对物理内存的访问“重定向”到设备地址空间中。CPU 对这部分物理内存的访问等同于对相应设备的访问。

对设备物理内存的访问需要通过 kseg1 段。因为通过该段进行的读写不经过 TLB 映射，且不使用高速缓存。不使用缓存的原因是设备物理内存处的数据不只由 CPU 决定，还和对应的外设的行为有关。而缓存只能记录 CPU 的读写结果，无法在外设对数据进行修改时及时调整。



外设控制

操作系统对设备的控制

由于 kseg1 段只能在内核态读写，因此需要操作系统提供系统调用来实现在用户态读写设备。我们引入 `sys_write_dev` 与 `sys_read_dev` 两个系统调用来实现设备读写操作。

```
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);
```

- `sys_write_dev` ：将起始虚拟地址为 `va` ，长度为 `len` 字节的一段数据写到起始物理地址为 `pa` ，长度为 `len` 字节的物理空间上。
- `sys_read_dev` ：将起始物理地址为 `pa` ，长度为 `len` 字节的物理空间中的数据拷贝到起始虚拟地址为 `va` ，长度为 `len` 字节的空间中。

需要注意：

- 在操作前需要检查 `va` 是否合法。同时还需检查物理地址 `pa` 是否位于设备对应的地址范围内。
- `len` 的取值必须为 1、2 或 4，可以使用 `iowrite8` 、 `iowrite16` 和 `iowrite32` 函数实现数据的读取。

外设控制

设备驱动程序

系统调用 `sys_write_dev` 与 `sys_read_dev` 提供了通用的设备控制接口。而针对于 IDE 磁盘这一特定设备，我们还需要为其编写用户态的控制接口，也即**设备驱动程序**。设备驱动为操作系统提供了标准化的设备控制接口。

IDE 外设一般不能立即完成数据操作，需要 CPU 检查 IDE 状态并等待操作完成。为此我们构建了检查 IDE 状态的帮助函数 `wait_ide_ready`，用于等待 IDE 上的操作就绪。

```
static u_int wait_ide_ready() {
    u_int flag;
    while (1) {
        panic_on(syscall_read_dev(&flag, MALTA_IDE_STATUS, 1));
        if ((flag & MALTA_IDE_BUSY) == 0) {
            break;
        }
        syscall_yield();
    }
    return flag;
}
```

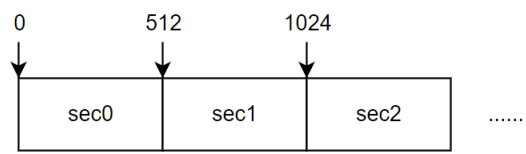
外设控制

IDE 磁盘驱动

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);
```

`ide_read` 中对每一次磁盘读操作（写操作调用 `syscall_write_dev` ，读操作调用 `syscall_read_dev` ）：

- 使用 `wait_ide_ready` 等待 IDE 设备就绪。
- 向物理地址 `MALTA_IDE_NSECT` 处写入单字节 1，表示设置操作扇区数目为 1。
- 分四次设置扇区号，并在最后一次一同设置扇区寻址模式和磁盘编号。
- 向物理地址 `MALTA_IDE_STATUS` 处写入单字节值 `MALTA_IDE_CMD_PIO_READ` ，设置 IDE 设备为读状态。
- 使用 `wait_ide_ready` 等待 IDE 设备就绪。
- 从物理地址 `MALTA_IDE_DATA` 处以四字节为单位读取数据。



文件系统

文件系统概述

所有文件系统相关操作均在用户态实现，主要包括以下三个部分：

- **IDE 设备驱动**：位于 `fs/ide.c` 中，使用 `sys_write_dev` 与 `sys_read_dev` 完成设备读写操作。
- **文件系统服务进程**：代码位于 `fs` 目录下。文件系统服务进程是一个运行在用户态下的进程，使用 IPC 机制与其它请求文件操作的用户进程进行通信，调用 `fs/ide.c` 中的 `ide_write` 与 `ide_read` 读写磁盘
- **文件系统的用户库**：代码包括 `user/lib` 中的 `file.c`、`fd.c`、`fsipc.c`，这部分代码会和用户程序一同编译成可执行文件，用户程序可以调用其中的函数完成文件系统操作，其中
  - `fsipc.c` 实现了与文件系统服务进程基于 IPC 的交互
  - `file.c` 实现了文件系统的用户接口
  - `fd.c` 实现了文件描述符相关操作

除此之外，我们还有一个运行在 Linux 宿主机上的工具程序 `fsformat`，用于帮助我们构建磁盘镜像文件。代码位于 `tools/fsformat.c`。

文件系统

磁盘布局

MOS 中以磁盘块（Block）为单位进行文件系统的管理。系统将相邻 8 个扇区组合在一起形成一个 4096 字节大小的磁盘块。不同于扇区，磁盘块是一个虚拟概念，是操作系统与磁盘交互的最小单位。

MOS 以磁盘最开始的一个磁盘块当作引导扇区和分区表使用。接下来的一个磁盘块作为超级块（Super Block），用来描述文件系统的基本信息。

```
struct Super {
    uint32_t s_magic;   // Magic number: FS_MAGIC
    uint32_t s_nblocks; // Total number of blocks on disk
    struct File s_root; // Root directory node
};
```

- `s_magic`：魔数，为一个常量，用于标识该文件系统。
- `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。
- `s_root`：根目录，其 `f_type` 为 `FTYPE_DIR`，`f_name` 为 `"/"`。

文件系统

文件系统结构

MOS 的文件系统是以根目录为根节点的树状数据结构。节点的数据结构由文件控制块（`File` 结构体）定义。一个 `File` 结构体的大小为固定的 256 字节（`BY2FILE`）。

```
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[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

- `f_name`：文件名。
- `f_size`：文件大小，单位为字节。
- `f_type`：文件类型，分为普通文件 `FTYPE_REG` 和目录 `FTYPE_DIR`。

文件系统

文件系统结构

```
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[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

- `f_direct`：文件的直接“指针”，用于记录文件的数据块在磁盘上的**位置编号**。最多维护 `NDIRECT * 4 KB` 大小的数据。
- `f_indirect`：指向一个间接磁盘块。该磁盘块中存储指向文件内容的磁盘块的“指针”，最多存储 1024 个。我们不使用间接磁盘块的前十个“指针”，因此使用间接磁盘块后，单个文件最大为 `1024 * 4 KB`。
- `f_dir`：指向文件所处的目录对应的文件控制块。
- `f_pad`：用于将 `File` 结构体大小填充到 256 字节。

文件系统

制作磁盘镜像

`fsformat.c` 用于创建磁盘镜像文件。使用 Linux 下的 gcc 编译器编译，运行于 Linux 宿主机而非 QEMU 上。

两个核心函数：

- `write_directory` 用于将宿主机上路径为 `path` 的目录（及其下所有目录和文件）写入磁盘镜像中 `dirf` 所指向的文件控制块所代表的目录下。

```
void write_directory(struct File *dirf, char *path);
```

- `write_file` 用于将宿主机上路径为 `path` 的文件写入磁盘镜像中 `dirf` 所指向的文件控制块所代表的目录下。

```
void write_file(struct File *dirf, const char *path);
```

文件系统

制作磁盘镜像

实验中需要完成 `create_file` 函数，以支持在给定目录下分配新的文件控制块。

```
struct File create_file(struct File *dirf);
```

主要流程：

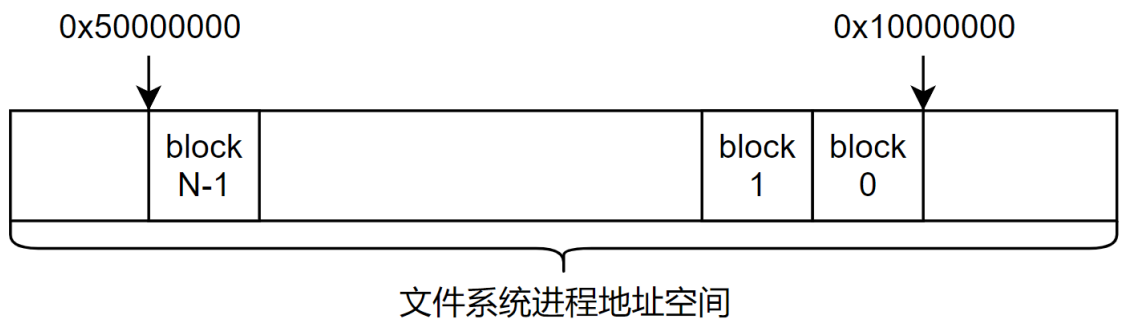
- 遍历 `dirf` 所代表的目录拥有的每一个磁盘块（包括直接和间接指向的磁盘块）。
- 对每一个磁盘块，从头遍历该磁盘块上的所有文件控制块。
- 找到第一个未被使用的文件控制块并返回该文件控制块指针（未被使用的文件控制块中 `f_name` 的第零个字符为 `'\0'`）。

文件系统

块缓存机制

MOS 采用微内核架构，将文件系统的功能从内核中抽离，使用一个文件系统服务进程提供文件系统的相关功能。该文件系统服务进程会将一段虚拟地址空间（0x10000000-0x4fffffff）作为缓冲区，用来映射磁盘数据。编号为 0 的磁盘块映射到虚拟地址空间中 0x10000000 至 0x10000fff 这 4096 字节大小的空间。其他磁盘块则以此类推。

使用块缓存机制，当其他进程想要访问文件数据的时候，只需要将自身虚拟地址空间的一部分映射到文件系统服务进程缓冲区中的相应位置即可。



文件系统

映射磁盘数据

为了在文件系统服务进程中实现磁盘块与内存空间之间的映射。我们需要管理缓冲区内的内存。这需要用到 `diskaddr`、`map_block` 和 `unmap_block` 等函数。

- `diskaddr` 函数用于将数据块编号转换为缓冲区范围内的对应虚拟地址。

```
void *disk_addr(u_int blockno);
```

- `map_block` 函数用于分配映射磁盘块需要的物理页面。

```
int map_block(u_int blockno);
```

  - 先使用 `block_is_mapped` 查看 `blockno` 对应磁盘块是否已被映射到内存，若已经建立了映射，直接返回 0 即可。
  - 若没有建立映射，则使用 `diskaddr` 函数将 `blockno` 转换为对应虚拟地址，并调用 `syscall_mem_alloc` 函数为该虚拟地址分配一页物理页面。

文件系统

映射磁盘数据

- `unmap_block` 用于释放用来映射磁盘块的物理页面。

```
void unmap_block(u_int blockno);
```

  - 传入的参数为磁盘块编号，首先使用 `block_is_mapped` 函数获取对应磁盘块的虚拟起始地址。
  - 如果该磁盘块不是空闲的（`block_is_free` 函数）并且该磁盘块缓存被写过（`block_is_dirty` 函数），则调用 `write_block` 函数将磁盘块缓存数据**写回**磁盘。
  - 最后调用 `syscall_mem_unmap` 函数解除对应磁盘块缓存虚拟页面到物理页面的映射。

和 `unmap_block` 对比，为什么 `map_block` 中没有读取磁盘块数据的操作？因为 `map_block` 所对应的数据块在此前并未使用，其中的数据是无效数据。

文件系统

查找文件

`dir_lookup` 函数用于在 `dir` 指向的文件控制块所代表的目录下寻找名为 `name` 的文件。

```
int dir_lookup(struct File *dir, char *name, struct File **file);
```

该函数的主要流程如下：

- 遍历目录文件 `dir` 中的每一个磁盘数据块。（使用 `file_get_block` 获取第 `i` 个数据块的数据。）
- 遍历单个磁盘块中保存的所有文件控制块。
- 判断要查找的文件名 `name` 与当前文件控制块的 `f_name` 字段是否相同，若相同则返回该文件控制块。
- 若未找到则返回 `-E_NOT_FOUND`。

文件操作

概述

我们前面提到，MOS 的文件系统服务由用户进程提供。因此为了实现创建文件、读写文件等等文件操作，就需要使用 Lab4 中实现的进程间通信机制。

请求文件系统服务进程的接口为 `fsipc` ，其中包括一对 `ipc_send` 和 `ipc_recv` 操作。该函数将进程间通信封装成了函数调用的形式。

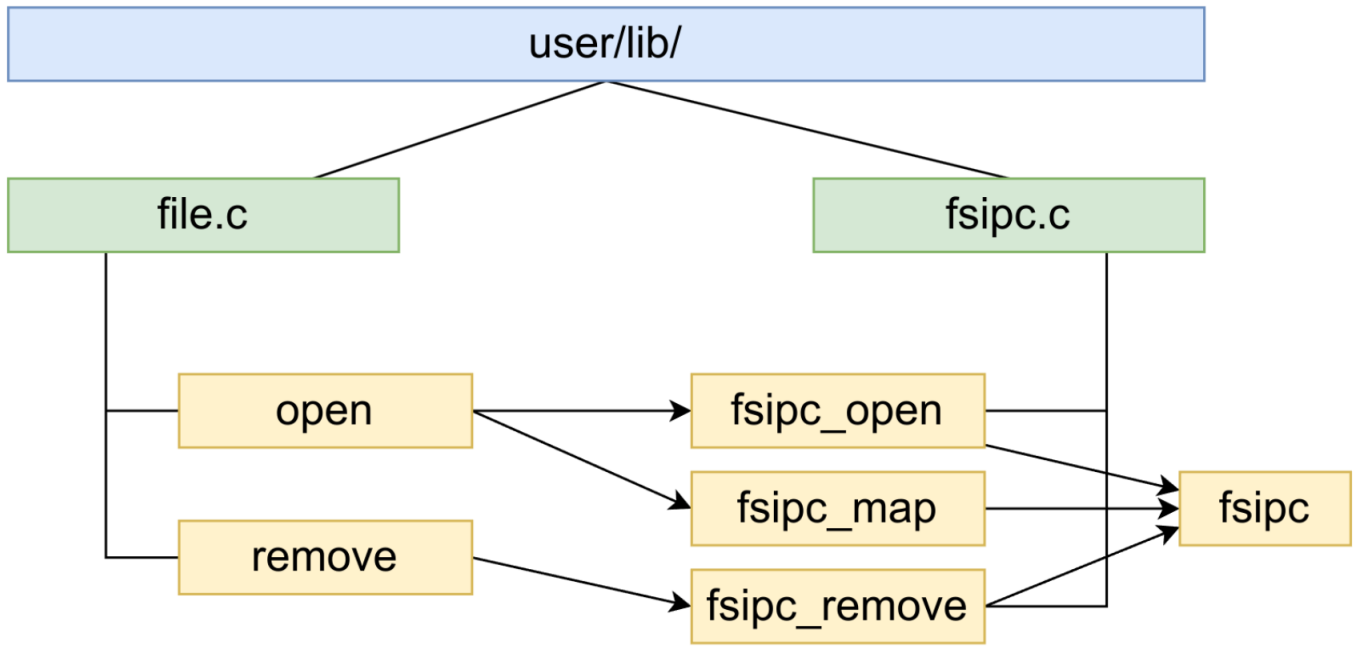
```
static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm);
```

和系统调用类似，不同的操作类型通过参数 `type` 区分。并被进一步封装为不同函数，如 `fsipc_open` 、`fsipc_map` 等等。

而文件系统服务进程的入口可以在 `fs/serv.c` 中找到。程序主体是一个死循环，循环中调用了 `ipc_resv` 不断接收其他进程发来的请求，并通过 `serve_table` 转发到不同处理函数。处理结束后，程序调用 `ipc_send` 返回调用结果。

文件操作

概述


user/lib/
file.c
fsipc.c
open
remove
fsipc_open
fsipc_map
fsipc_remove
fsipc

文件操作

open 函数

在 `fsipc_*` 函数之上，我们继续进行函数的封装，从而提供更加清晰友好的文件操作接口。

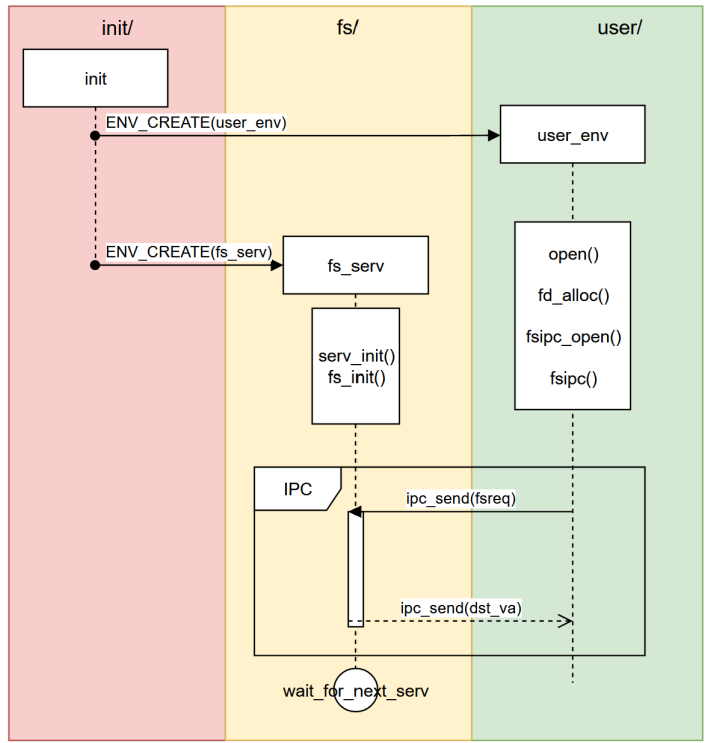
`open` 用于打开指定文件。函数接受文件路径 `path` 和模式 `mode` 作为输入参数，并返回**文件描述符**。

```
int open(const char *path, int mode);
```

- 首先使用 `fd_alloc` 和 `fsipc_open` 函数分配一个文件描述符。
- 之后使用 `fd2data` 函数获取文件描述符对应的数据缓存页地址，并从 `Filefd` 结构体中获取文件的 `size` 和 `fileid` 属性。
- 之后遍历文件内容，使用 `fsipc_map` 函数为文件内容分配页面并映射。
- 最后使用 `fd2num` 函数返回文件描述符。

文件操作

open 函数


init/
fs/
user/
init
ENV_CREATE(user_env)
user_env
ENV_CREATE(fs_serv)
fs_serv
open()
fd_alloc()
fsipc_open()
fsipc()
serv_init()
fs_init()
IPC
ipc_send(fsreq)
ipc_send(dst_va)
wait_for_next_serv

文件操作

文件描述符

`open` 函数的返回值为一整数，称为文件描述符。文件描述符本质是对描述符表的索引，一个文件描述符对应一个 `Filefd` 结构体。该结构体在文件系统服务进程中创建，并通过 `ipc_send` 共享给其他进程。

值得注意，在 `open` 等函数中存在将 `struct Fd*` 型指针强制转换为 `struct Filefd*` 型指针的操作。通过改变指针类型，实现了对相同数据的不同解释。

```
// 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;
};
```

文件操作

read 函数

`read` 用于读取文件内容，该函数接收一个文件描述符编号 `fdnum` ，一个缓冲区指针 `buf` 和一个最大读取字节数 `n` 作为输入参数，返回成功读取的字节数并更新文件描述符的偏移量。

```
int read(int fdnum, void *buf, u_int n);
```

- 首先使用 `fd_lookup` 和 `dev_lookup` 函数获取文件描述符和设备结构体。如果查找失败，则返回错误码。
- 接下来，函数检查文件描述符的打开模式是否允许读取操作。
- 然后，函数调用设备的读取函数 `dev_read` 从文件当前的偏移位置读取数据到缓冲区中。
- 最后，如果读取操作成功，则函数更新文件描述符的偏移量并返回成功读取的字节数。

文件操作

remove 函数

本次实验中需要实现 `remove - fsipc_remove - file_remove` 的整条调用链。

`remove` 函数用于删除给定路径 `path` 所对应的文件。该函数调用了 `fsipc_remove` 函数。

```
int fsipc_remove(const char *path);
```

- `fsipc_remove` 函数首先检查路径字符串的长度是否在 0 到 `MAXPATHLEN` 之间。如果不是，则返回错误码，表示路径字符串不合法。
- 接下来，函数将缓冲区 `fsipcbuf` 视为一个 `Fsreq_remove` 结构体，代表将要发送的是 `remove` 请求，并将路径字符串复制到结构体的路径字段中。
- 随后使用 `fsipc` 函数将删除请求发送至文件系统服务进程，并返回删除操作的结果。

最后，`serve_remove` 则位于文件系统服务进程中，是文件删除请求的处理函数。其中我们调用了 `file_remove` 函数实现了文件的删除。

课下习题提示

- Exercise 5.1

  可以使用 `include/io.h` 中定义的 `iowrite32`、`iowrite16` 和 `iowrite8` 函数向 kseg1 中对应物理地址处写入数据

- Exercise 5.2

  参考 Lab4 中其他系统调用接口的实现

- Exercise 5.3

  参考指导书中内核态设备驱动的写法，将其中对地址的直接写入和读取转换为使用 `syscall_write_dev` 和 `syscall_read_dev` 实现写入和读取

- Exercise 5.4

  对 `bitmap` 的操作可以参考 `alloc_block_num` 函数

课下习题提示

- Exercise 5.5

  1. 对于文件中的第 `i` 个数据块，如果 `i` 小于 `NDIRECT` 则从 `f_direct` 中获取数据块编号；否则从 `f_indirect` 中获取
  2. 通过 `f_name` 的第一个字符是否为 `'\0'` 判断文件控制块是否未被使用

- Exercise 5.6

  数据块编号到缓冲区空间是线性映射的关系，使用 `DISKMAP` 和 `BLOCK_SIZE` 计算得到地址

- Exercise 5.7

  注意 `unmap_block` 中如果该磁盘块不是空闲的且被写入过，则需要使用 `write_block` 将数据写回磁盘

课下习题提示

- Exercise 5.8

  1. 使用 `file_get_block` 获取文件中的第 `i` 个数据块
  2. 使用 `strcmp` 判断 `name` 和 `f_name` 是否一致

- Exercise 5.9

  1. 将 `fd` 强制类型转换为 `struct Filefd *` 并赋给 `ffd`
  2. 使用 `fsipc_map` 将文件服务进程的缓冲区中文件数据映射到当前进程中

- Exercise 5.10

  使用 `dev->dev_read` 将数据读取到缓冲区 `buf` 中

- Exercise 5.11-13

  `fsipc_remove` 的实现参考其他 `fsipc_*` 函数