文件系统

文件系统，指的是操作系统中管理（硬盘上）持久存储数据的模块。

为什么需要文件系统?

能够"持久存储数据的设备", 可能包括: 机械硬盘, 固态硬盘, 光盘（加上它的驱动器），软盘，U盘，甚至磁带或纸带等等。一般来说，每个设备上都会连出很多针脚（这些针脚很可能符合某个特定协议，如 USB 协议， SATA 协议），它们都可以按照事先约定的接口/协议把数据从设备中读到内存里，或者把内存里的数据按照一定的格式储存到设备里。
我们希望做到的第一件事情，就是把以上种种存储设备当作"同样的设备"进行使用。不管机械硬盘的扇区有多少个，或者一块固态硬盘里有多少存储单元，我们希望都能用同样的接口进行使用，提供"把硬盘的第 a 到第 b 个字节读出来"和"把内存里这些内容写到硬盘的从第 a 个字节开始的位置"的接口，在使用时只会感受到不同设备速度的不同。否则，我们还需要自己处理什么 SATA 协议， NVMe 协议啥的。处理具体设备，具体协议并向上提供简单接口的软件，我们叫做设备驱动(device driver)，简称驱动。
文件的概念在我们脑子里根深蒂固，但"文件"其实也是一种抽象。理论上，只要提供了上面提到的读写两个接口，我们就可以进行编程，而并不需要什么文件的概念。只要你自己在小本本上记住，你的某些数据存储在硬盘上从某个地址开始的多么长的位置，以及硬盘上哪些位置是没有被占用的。编程的时候，如果用到/修改硬盘上的数据， 就把小本本上记录的位置给硬编码到程序里。
但这很不灵活！如果你的小本本丢了，你就再也无法使用硬盘里的数据了，因为你不知道那些数据都是谁跟谁。另外，你也可能一不小心修改到无关的数据。最后，这个小本本经常需要修改，你很容易出错。
显然，我们应该把这个小本本交给计算机来保存和自动维护。这就是文件系统。
我们把一段逻辑上相关联的数据看作一个整体，叫做文件。
除了硬盘，我们还可以把其他设备（如标准输入 stdin ， 标准输出 stdout ）看成是文件，由文件系统进行统一的管理。

虚拟文件系统 VFS

虚拟文件系统VFS可以理解为是"面向用户的"文件系统，其承担OS和更具体、更详细文件系统（偏系统向）之间的接口（过渡）。
例如:
电脑本地可以访问自己的硬盘同时可以访问云端的云盘，但在自己电脑上基本操作逻辑是一样的，这就可以理解为OS给我们提供了一个虚拟文件系统，允许我们用相同的操作去访问两个地方的文件（一段数据）。

对于不同文件系统的目录，我们可以使用相同的 open ， read ， write ， close 接口，好像它们在同一个文件系统里一样。这就是虚拟文件系统VFS的功能。

拓展（可略）： linux 的 VFS 支持虚拟块设备 （ virtual block devices), 可以把当前文件系统中的某个文件，认为是"一块具备虚拟文件系统的磁盘"进行挂载，也就是说这个文件里包含一个完整的文件系统，如同"果壳里的宇宙"。
这种设备也叫做 loop device ， 可以用来简易地在现有文件系统上进行磁盘分区。

Ucore 文件系统（自己做的就是Ucore）

其和传统 UNIX 文件系统类似。

UNIX 里可以把"文件"理解成一串**按顺序排好的字节。**这些字节要落到磁盘上存起来，但磁盘是按"块"分配空间的，所以文件在磁盘上占的块可能**连续**也可能**不连续**；而且因为按块分配、还要存一些管理信息，所以磁盘上实际占用的空间往往会比你真实写进去的数据多一点。

每个文件对人来说有一个好记的名字（路径 `path`），但对文件系统内部来说还有一个"底层编号"（用户一般看不到）。你对文件做读写，本质上就是对一串字节在不同位置进行读/改；如果你越写越多，文件系统会自动再给它分配更多磁盘块，让它"变大"。文件也可以被创建、删除。

目录（directory）本质上也是文件，只不过它的内容不是普通数据，而是"一个列表"：列出它里面有哪些名字，每个名字对应哪个子目录/文件。

挂载点（mount point）可以理解成：把一个新的文件系统"接"到当前这棵目录树的某个位置上。从用户视角看，系统只有一棵以 `/` 为根的目录树；但这棵树的不同分支可能来自不同的文件系统。比如插 U 盘后，系统会把 U 盘的文件系统挂到某个路径（比如 `/mnt/usb`）。对原来的系统来说，这个路径原本只是树上的一个"叶子位置"；挂载后，这个位置就变成了 **U 盘文件系统的入口**，同时也仍然是整棵树中的一个节点。

为了让各种文件系统（Ext4、ZFS、FAT……）都能被内核用同一套方式操作，UNIX 内核会定义一个"通用文件模型"（Common File Model）：不管底层是什么文件系统，只要你实现了这套模型规定的接口，上层就能统一地 `open/read/write/close`。可以把它理解成"面向对象的接口规范"：有一些对象，每个对象有属性和方法。因为内核大多用 C 写，所以"方法"通常用**函数指针**来实现（类似虚函数表）。

通用模型里常见的几个对象，用大白话说就是：

- 超级块（superblock）：整个文件系统的"总说明书/总目录"。里面记录这个文件系统整体信息，比如块大小、总块数、空闲块怎么管理、inode 表位置等。对应磁盘上的文件系统控制块。
- inode（索引节点）：每个文件/目录的"身份证 + 档案"。它记录元数据：权限、大小、属主、时间戳，以及"数据块在哪里"等信息。inode 还有一个唯一编号（在这个文件系统内部唯一），文件系统用它来定位一个文件。对应磁盘上的文件控制块。
- file（文件对象）：注意这不是磁盘上的"文件内容"，而是**某个进程打开某个文件之后**，内核在内存里创建的一份"打开记录"。里面会保存比如"当前读到哪了（偏移量）""以什么方式打开的（只读/可写）"等状态。只有进程 `open` 之后才存在，`close` 后就可能释放。
- dentry（目录项）：这是"名字到文件"的中间桥梁。用户访问文件靠路径名（比如 `/tmp/test`），但文件系统底层识别文件靠 inode。dentry 就是在内存中缓存这种对应关系：某个目录下的某个名字（比如 `tmp`、`test`）对应哪个 inode。

这些抽象概念最终要落到"磁盘怎么摆放数据"这件事上：具体文件系统会设计一种磁盘布局，把 superblock/inode/数据块等放在磁盘的某些位置。

- 比如 inode 的信息实际存放在磁盘的某些块里；当需要访问某个文件时，内核会把磁盘里的 inode 读到内存里，构造出内存中的 inode 对象来操作。
- 再比如 dentry 这个东西一般**不直接存到磁盘上**。当你解析路径 `/tmp/test` 时，内核会一段一段地找：先从根目录 `/` 开始，找到名字为 `tmp` 的那一项，于是内存里会形成一个"`/` 的 dentry"和一个"`tmp` 的 dentry"；接着在 `/tmp` 里再找 `test`，又形成一个对应的 dentry。这样下次再访问相同路径时，就不必每次都从头做完整查找，速度会更快。

我们这次会在Ucore内用虚拟文件系统VFS管理三类设备：

1. 硬盘，我们管理硬盘的具体文件系统是 Simple File System （地位和 Ext2 等文件系统相等）
2. 标准输出（控制台输出），只能写不能读；
3. 标准输入（键盘输入），只能读不能写。

硬盘用一块内存来模拟。

lab8 的镜像构建为什么要分三段

- sfs.img 是“文件系统数据盘”。里面已经提前放好了一个能被 ucore 识别的 SFS（Simple FS）文件系统结构，并且把编译好的用户程序（例如 sh、hello、其他 user/ 下的可执行文件）作为“文件”写进了这个文件系统的目录和数据块里。运行时内核从这块盘里查目录、读 inode、读数据块，才能把用户程序读出来。
- swap.img 是“交换分区盘”。它不是普通文件系统，而是一段全 0 的磁盘区域，专门用来做 swap（把暂时不用的页换出到磁盘、需要时再换入）。在 lab8 里它更多是为后续/整体设计预留：让内核有一个稳定的“交换空间设备”，便于页面置换或内存压力时使用。
- kernel objects 是“内核本体”。就是你编译出来的 ucore 内核目标文件（.o）以及链接后形成的内核镜像主体代码与数据段。

三者共同组成 ucore.img 的含义是：QEMU 启动时加载的这个大镜像里，同时包含了“内核代码”和“两段模拟磁盘的数据”。这样做的关键收益是：不需要真的给 QEMU 挂载真实硬盘文件或复杂设备，内核在运行时就能在内存中把这两段区域当作磁盘来读写。

ucore 如何在运行时找到 swap.img 和 sfs.img

- 这两段“硬盘”在最终镜像里只是连续的二进制数据区域，本质上会被加载到内存里。
- 链接时会人为在链接脚本或构建流程里插入一些“首尾符号”（类似 _binary_xxx_start / _binary_xxx_end 这种），内核代码就能通过这些符号拿到这段数据在内存中的起始地址和结束地址。
- 有了起止地址，内核就能把它注册成一个“磁盘设备”（例如 disk0），之后通过统一的块设备读写接口（读写若干个 block）来访问它。

为什么必须在内核启动前先构造 sfs.img（mksfs.c 的作用）

- 内核运行时并不会“凭空生成一个文件系统”，它期望磁盘上已经有合法的 SFS 布局：超级块、空闲块位图、根目录 inode、目录项、普通文件 inode、文件数据块等。
- tools/mksfs.c 就是把“一个目录树 + 若干可执行文件”打包成“符合 SFS 格式的磁盘镜像”。它做的事类似现实中的 mkfs + 把文件拷进去，只不过是针对 ucore 的 SFS 定制格式。
- 所以流程是：先在宿主机上编译用户程序 → 用 mksfs 把这些程序写入 sfs.img → 再把 sfs.img 链到最终 ucore.img 里 → QEMU 启动后内核才能通过文件系统把这些程序读出来执行。

ucore 文件系统四层架构分别解决什么问题

通用文件系统访问接口层（面向用户态）

- 这一层直接接住用户程序发起的文件相关系统调用，比如 open/read/write/close/dup/seek 等（在 ucore 里通常体现在 sysfile_xxx 这一类函数）。
- 它的核心职责是“把用户态请求安全地转成内核态操作”：做参数拷贝与校验（用户指针合法性、长度检查）、把“fd/路径字符串”等用户态概念转成内核里的 file/inode 等对象，然后把请求交给 VFS。

文件系统抽象层 VFS（Virtual File System，面向内核其他模块）

- VFS 提供统一的抽象接口：无论底层是 SFS、设备文件系统、未来可能加的其他 FS，上层都只用一套接口（vfs_open / vop_read / vop_write / vop_lookup 等）。
- 它通过“函数指针表 + 统一的数据结构”来屏蔽差异：inode 里挂着 inode_ops（vop_open/vop_read/...），不同文件系统只要实现各自的 inode_ops，就能被统一调用。
- 这一层也是路径解析与命名空间管理的关键：把 “/a/b/c” 这种路径逐级解析为目录项/ inode，并处理设备名前缀（如 “stdin:” “disk0:”）或根目录/当前目录的起点选择。

Simple FS 文件系统层（SFS 的具体实现）

- 这一层才真正理解“SFS 的磁盘格式”：超级块长什么样、inode 如何组织、direct/indirect 如何映射数据块、目录项如何存储等。
- 当上层请求 read/write 时，SFS 把“按字节读写”转成“按块定位并读写”：通过 bmap（block map）从文件的逻辑块号定位到磁盘块号，再调用块设备接口去读写对应 block。
- 这一层还负责一致性与元数据更新：写入导致文件变大时要分配新块、更新 inode size/blocks、置 dirty 并在 close/fsync 时回写。

外设接口层（device 层，面向具体硬件/模拟硬件）

- 这一层提供统一的 device 抽象：d_open/d_close/d_io/d_ioctl。
- 对上层来说，不关心是“真实磁盘驱动、ramdisk、串口、键盘”，反正都用 d_io 读写；对下层来说，它实现具体策略：比如 disk0 把 block 读写转成对内存模拟磁盘区域的拷贝或对 IDE/ramdisk 的访问；stdin/stdout 把读写转成控制台缓冲区/字符输出。

把一次文件操作放进四层里看（以 write 为例的完整含义）

- 用户程序调用 write(fd, buf, len)。
- 通用接口层接收系统调用，检查 fd、把用户 buf 安全拷贝/映射到内核可访问区域，调用 file_write 或 vfs 相关接口。
- VFS 通过 fd 找到 file 对象，再找到其 inode；调用 inode->in_ops->vop_write（对不同对象分派到不同实现：SFS 普通文件是 sfs_write，stdout 设备文件是 dev_write）。
- 如果是普通文件：进入 SFS 层，把“字节偏移 + 长度”切分成“若干个逻辑块 + 块内偏移”，做 bmap，得到磁盘块号，再向下调用 device 的块读写把数据写入。
- 最终由具体 device（disk0）完成“块级读写”，实现对模拟硬盘区域的更新。

四类关键数据结构分别是什么、作用域在哪里

超级块（SuperBlock，文件系统全局视角，OS 全局范围）

- 保存一个文件系统整体的元信息：魔数、总块数、空闲块数量、块大小、版本/描述等。
- 一旦挂载（mount）成功，这些信息会长期驻留在内存中供整个内核使用。
- 作用域是“整个文件系统/整个 OS”，不是某个进程私有。

索引节点 inode（单个文件视角，OS 全局范围）

- 描述一个文件/目录的元数据与数据位置映射：文件大小、类型、链接数、占用块数、direct/indirect 指针等。
- inode 是 VFS 的核心对象：所有“对文件的操作”最终都会落到某个 inode 的 inode_ops 上。
- 作用域是 OS 全局：不同进程打开同一个文件，通常会共享同一个 inode（引用计数不同）。

目录项 dentry（路径视角，OS 全局范围）

- 描述“路径中的一段名字如何对应到 inode”。它把“父目录 + 名称”映射到“目标 inode”。
- 在 ucore 的 SFS 描述里，磁盘上的目录项结构是 sfs_disk_entry（包含 name 和 ino）；而 VFS 语义里的 dentry 更多是“路径解析时在内存中建立的关联/缓存”。
- 作用域也是 OS 全局：路径解析是系统范围的行为，不属于某个单独进程。

文件对象 file（进程视角，进程私有范围）

- 表示“某个进程打开某个文件的状态”：可读/可写、当前偏移 pos、引用次数 open_count、指向 inode 的 node 指针等。

- 同一 inode 可以对应多个 file：例如两个进程分别 open 同一文件，会有各自的 file 对象，pos 独立；但 inode 是共享的。
- 作用域是“单个进程（或其线程组）”：通常挂在 proc_struct 的 files_struct->fd_array 里，由 fd 索引。

“用户进程打开一个文件”时这些结构之间的关系应该怎么理解

- 进程维度：进程的 fd（整数）指向它自己的 fd_array 里的某个 file 结构，这个 file 记录该进程对文件的打开模式与当前位置 pos。
- OS 全局维度：file->node 指向一个 inode；inode 代表系统里这个文件的统一身份与元数据入口。
- 路径维度：open(path) 时，VFS 需要从某个起点目录 inode 出发，沿着 path 的每一段名称做 lookup，等价于逐级查目录项（dentry/目录项记录），最后解析到目标 inode。
- 设备一致性：如果打开的是 “stdin:” “stdout:” 这种设备路径，最终解析得到的 inode 类型是 device，inode_ops 会分派到 dev_read/dev_write；如果打开的是 “/hello”，最终解析得到的是 SFS inode，inode_ops 会分派到 sfs_read/sfs_write。上层对两者调用 read/write 的方式完全一样。

文件系统的抽象层——VFS

VFS（文件系统抽象层）到底在抽象什么

- 目的很直接：让“上层代码”只认识一套统一的文件接口，不需要知道底层到底是 SFS、设备文件系统、还是以后新增的别的文件系统。
- 做法也很直接：把“各种文件系统都会有的操作”（open/close/read/write/lookup/create...）整理成一组函数指针（inode_ops）。不同文件系统各自把这些函数指针填成自己的实现，上层只通过指针调用。
- 结果是：通用系统调用层（sysfile_open/sysfile_read...）只需要和 VFS 打交道，不需要写一堆 if/else 去区分“这是 SFS 还是设备文件”。

file & dir 接口：进程视角的“打开文件状态”

file 结构体解决的是：进程打开文件后，必须记住“我以什么方式打开的、读到哪里了、对应的是哪个 inode”。

```
// kern/fs/file.h
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;                          // 访问文件的执行状态
    bool readable;                     // 文件是否可读
    bool writable;                     // 文件是否可写
    int fd;                            // 文件在 filemap(fd_array) 中的索引值
    off_t pos;                         // 访问文件的当前位置（文件偏移）
    struct inode *node;                // 该文件对应的内存 inode 指针（VFS inode）
    int open_count;                    // 打开此文件的次数（引用/共享计数相关）
};
```

file 结构体在系统里的位置与作用

- 它描述的不是“磁盘上的那个文件本体”，而是“某个进程打开一个文件之后，在内核里为这次打开建立的一份状态”。

- 进程拿到的 fd（一个整数）本质上就是指向 fd_array 的下标；fd_array 里每个槽位就是一个 struct file。
- 所以 file 结构体解决的问题是：进程打开文件后，内核需要记住哪些运行时信息，才能正确实现 read/write/seek/close/dup 等行为。

status：file 槽位的生命周期状态

- FD_NONE
  这个槽位完全空闲，没有被分配过，或者已经被彻底释放回空闲态。
  分配 fd 时会优先找这种槽位。
- FD_INIT
  槽位已被"预占用"，但 open 的完整流程还没走完。
  这样做的意义是避免并发情况下两个线程同时拿到同一槽位：一旦标记为 INIT，别的线程就不会再分配它。
- FD_OPENED
  open 成功完成后进入该状态。
  只有在 OPENED 状态下，pos/readable/writable/node 等字段才可被正常使用。
- FD_CLOSED
  close 之后的状态。
  在很多实现里 close 会把槽位直接回收为 NONE；但保留 CLOSED 也有价值：便于引用计数/延迟释放、调试定位"重复 close"、以及与 dup/共享语义配合。

readable / writable：对本次打开的访问权限结果

- 这两个字段不是文件系统权限（比如 UNIX 的 rwx 位）本身，而是"本次 open_flags 解析后的读写能力"。
- open_flags & O_ACCMODE 可能是：
  - O_RDONLY → readable=1 writable=0
  - O_WRONLY → readable=0 writable=1
  - O_RDWR → readable=1 writable=1
- 之后 sys_read/file_read 只要检查 file->readable 就能快速拒绝非法读；sys_write/file_write 同理检查 writable。
- 这保证了"权限检查不需要每次都重新解析 flags"，也避免上层误用。

fd：把"用户态 fd"与"内核槽位"固定绑定

- fd 字段通常等于它在当前进程 fd_array 里的下标。
- 之所以还存一份在 struct file 里，是为了：
  - 调试与断言（验证一致性）
  - 在一些路径里需要快速拿到 fd 值（比如日志、错误处理）
- 用户态看到的 fd（int）就是由 file_open 返回的 file->fd。

pos：文件偏移（file offset），决定下一次读写从哪开始

- pos 是"进程视角的文件指针"。read/write 都以 pos 为起点进行，并在成功后推进 pos。
- 典型行为：
  - read(fd, buf, n)：从 pos 开始读 n 字节（或读到 EOF），然后 pos += 实际读到的字节数
  - write(fd, buf, n)：从 pos 开始写 n 字节，然后 pos += 实际写入字节数

- 与 lseek 的关系：
    - sysfile_seek 会改 pos，从而实现随机访问
- 与 O_APPEND 的关系：
    - open 时如果带 O_APPEND，通常会把 pos 初始化到文件末尾，保证每次写都是追加（或者每次 write 前都强制定位到末尾，取决于实现）。

node：把本次打开关联到"VFS inode"（统一对象）

- node 指向的是 VFS 层的 struct inode，而不是 SFS 的磁盘 inode。
- 这点非常关键：
    - file 层不需要知道底层是 SFS 还是设备文件（stdin/stdout/disk0）
    - read/write/lookup 等操作通过 node->in_ops（函数指针表）自动分派到正确实现
- 所以 file->node 是"统一入口"：上层 file_read 只需要调用 vop_read(file->node, iob)，不用写任何分支判断具体文件系统类型。

open_count：打开/共享计数，用于 dup/fork/并发场景的正确回收

- open_count 用来描述"这份 file 状态当前被多少个引用持有"。
- 会影响 close 的语义：
    - 如果一个 fd 关闭，但该 file 仍被其他 fd（dup/dup2）共享，那么不能立刻释放 file 结构与 inode 引用
    - 只有当 open_count 降到 0，才能真正回收 file 槽位、对 inode 做最后一次 vop_close、释放缓存等
- 与 inode 的 open_count/ref_count 的区别：
    - file->open_count：描述"这份打开状态被共享了几份"
    - inode->open_count：描述"这个 inode 被打开了多少次"（跨进程也可能累计）
    - inode->ref_count：更宽泛，只要有人持有 inode 指针就算引用（不一定是打开的文件）

```
// kern/fs/fs.h
struct files_struct {
    struct inode *pwd;                // 进程当前工作目录的内存 inode 指针
    struct file *fd_array;            // 进程打开文件的数组（fd -> file 槽位）
    atomic_t files_count;             // 共享这份 files_struct 的线程/上下文数量
    semaphore_t files_sem;            // 互斥访问 files_struct（尤其是 fd_array）
};
```

files_struct 解决的问题：把"一个进程的所有文件状态"集中管理

- files_struct 挂在 proc_struct 上，是进程文件相关状态的总入口。
- 它负责两类核心信息：
    - 路径解析需要的上下文（pwd）
    - fd 管理需要的表结构（fd_array + 并发保护）

pwd：当前工作目录（相对路径解析的起点）

- 当用户传入相对路径（不以 / 开头）时，VFS lookup 从 pwd 对应的目录 inode 开始逐级查找。
- 当用户传入绝对路径（以 / 开头）时，从根目录开始查找，pwd 不参与。

- cd 命令的本质就是修改当前进程的 pwd。

fd_array：进程的“打开文件表”

- fd_array 是一个数组，每个元素是 struct file。
- open 时：
  - 找空槽位 → 填入 file 的字段 → 返回槽位下标作为 fd
- read/write/close 时：
  - 通过 fd 下标定位到 file → 检查状态/权限 → 调 inode_ops 完成实际读写
- 这个结构也是“一切皆文件”的基础：stdin/stdout 只是特殊 inode，对应的 fd 仍然存在于同一张 fd_array 里。

files_count：共享计数（多线程/复制语义的基础）

- 在有线程或共享文件表的设计里，多个执行流可能共享同一个 files_struct。
- files_count 用来决定何时释放这份 files_struct：
  - fork/clone 时可能增加计数
  - exit/线程结束时减少计数
  - 减到 0 才释放 fd_array/pwd 等资源
- 这能避免“一个线程退出把整个进程共享的文件表释放掉”。

files_sem：保护 files_struct 的互斥锁（用 semaphore 实现）

- 主要保护对象是 fd_array 和 pwd 的修改。
- 典型需要加锁的操作：
  - open：分配槽位 + 写入 file 字段（必须原子，否则会重复分配/覆盖）
  - close：清理槽位 + 更新计数 + 可能触发回收
  - dup/dup2：共享/复制 file 槽位，涉及多个槽位的更新
  - 改 pwd：cd/exec 相关路径处理可能修改当前目录
- 不加锁会出现的问题：
  - 两个线程同时 open，拿到同一个空槽位
  - 一个线程 close 正在被另一个线程 read 的 fd
  - dup2 覆盖目标 fd 时与并发 close 冲突

---

inode 结构（key）

```
// kern/vfs/inode.h
struct inode {
    union {                                     // 不同文件系统的私有 inode 信息
        struct device __device_info;            // 设备文件系统的 inode 信息
        struct sfs_inode __sfs_inode_info;      // SFS 的内存 inode 信息
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_sfs_inode_info,
    } in_type;                                  // inode 所属类型（对应 union 当前有效分支）
    atomic_t ref_count;                         // 引用计数（有人持有 inode 指针就算引用）
```

```
    atomic_t open_count;              // 打开计数（有多少 file 处于打开使用状态）
    struct fs *in_fs;                 // 抽象文件系统对象（文件系统级操作与私有数据）
    const struct inode_ops *in_ops;   // inode 操作表（函数指针：
open/read/write/...）
};
```

inode 在 VFS 里的定位：统一“文件节点对象”，屏蔽具体文件系统差异

- inode 是 VFS 的核心抽象：
    - 上层只认 inode 和 inode_ops
    - 具体文件系统把自己的实现“塞进” in_info，并把操作函数填进 in_ops
- 这使得：
    - 普通文件（SFS）与设备文件（stdin/stdout/disk0）对上层表现一致
    - 系统调用层不需要知道底层类型

in_info + in_type：把具体实现藏起来，但还能正确访问

- union 的意义：同一个 inode 只会属于一种具体类型（设备或 SFS），用 union 能共享内存布局。
- in_type 的意义：告诉你当前 union 哪个分支有效，防止访问错误。
- 典型用法：
    - vop_info(node, sfs_inode) 这种宏会在内部根据 in_type 做断言/转换
    - 设备 inode 走 device 分支；SFS inode 走 sfs_inode 分支

ref_count：保证 inode 生命周期安全

- 只要有人持有 inode 指针（例如：路径查找返回 node_store、file->node 指向它、目录项缓存里挂着它），ref_count 就不能为 0。
- ref_count 归零后才能回收 inode 对象，否则会出现悬空指针导致崩溃。
- 所以 lookup/open/close 等路径里经常出现：
    - vop_ref_inc(node) / vop_ref_dec(node)
      用来平衡引用。

open_count：区分“被引用”与“被打开”

- inode 可能被引用但不一定被打开：例如目录遍历时缓存了 inode；或者路径解析拿到 inode 但 open 失败。
- open_count 专门用来统计“真正处于打开状态的数量”。
- 最后一次 close（open_count 从 1→0）往往需要触发：
    - 文件系统把脏数据写回磁盘
    - 释放与打开状态绑定的资源（缓存块、锁、设备句柄等）
- 这就是为什么 inode 同时维护 ref_count 与 open_count：语义不同。

in_fs：inode 属于哪个文件系统实例

- 这个指针把 inode 和“文件系统对象”关联起来。
- 文件系统对象（struct fs）一般保存：
    - 文件系统级别的操作（挂载/卸载、获取根目录、同步等）
    - 文件系统实例的私有信息（比如 SFS 的 superblock、freemap 缓存等）

- 这样同一种文件系统也可以挂载多个实例（多个磁盘/镜像），inode 通过 in_fs 区分自己属于哪一个。

(重要)in_ops：核心中的核心——统一接口 + 动态分派

- inode_ops 是一组函数指针：vop_open/vop_read/vop_write/vop_lookup/...
- 对上层来说：
    - 只调用 VOP_READ(node, iob) 这类宏
    - 宏最终会跳到 node->in_ops->vop_read(node, iob)
- 对底层来说：
    - SFS 的 inode 把 in_ops 指向 sfs_node_fileops 或 sfs_node_dirops
    - 设备文件 inode 把 in_ops 指向 dev_node_ops
- 所以"同一个 read 系统调用"，遇到不同 inode 会走不同实现：
    - SFS 文件：读磁盘块、处理 bmap、更新 iobuf
    - stdin：读键盘缓冲区、可能阻塞等待
    - stdout：写控制台输出、禁止读

```
// kern/vfs/inode.h (节选)
struct inode_ops {
    unsigned long vop_magic;
    int (*vop_open)(struct inode *node, uint32_t open_flags);
    int (*vop_close)(struct inode *node);
    int (*vop_read)(struct inode *node, struct iobuf *iob);
    int (*vop_write)(struct inode *node, struct iobuf *iob);
    int (*vop_getdirentry)(struct inode *node, struct iobuf *iob);
    int (*vop_create)(struct inode *node, const char *name, bool excl, struct
inode **node_store);
    int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
    /* ... 省略若干接口 ... */
};
```

inode_ops 的抽象边界：上层只看语义，不看实现细节

- vop_open / vop_close
  语义：打开/关闭一个 inode 对应的对象。
  上层用它维护 open_count，并在最后一次 close 时触发文件系统的收尾工作。
- vop_read / vop_write
  语义：从 inode 对应对象读/写数据，数据通过 iobuf 描述（包含缓冲区地址、偏移、剩余长度等）。
  上层不关心底层是文件还是设备，底层自己决定怎么实现。
- vop_lookup
  语义：在目录 inode 下按名字查找子项，返回子项 inode。
  路径解析就是不断调用 lookup，把 "/a/b/c" 分解成逐级查找。
- vop_create
  语义：在目录 inode 下创建新文件（或节点），处理 excl 等语义，并返回新 inode。
  具体文件系统负责更新目录内容与分配 inode。

- vop_getdirentry
  语义：枚举目录内容（按 offset 逐项读出名字/ino）。
  目录不是按字节流读取，而是按“目录项”读取，所以需要专门接口。

最终串起来：为什么这套设计能让“上层不关心底层文件系统”

- 进程通过 fd 找到 struct file（进程打开状态）
- struct file 里持有 VFS inode（统一节点对象）
- VFS inode 里持有 inode_ops（统一接口的函数指针表）
- 调用 read/write/open/lookup 时，上层只调用 vop_xxx，真正实现由 inode_ops 自动分派到 SFS 或设备文件实现
- 这样新增文件系统时，只要实现一套 inode_ops 并把它挂到 inode 上，上层系统调用层完全不用改

结论：为什么进程“不需要关心具体文件系统”

- 进程层拿到的是 fd → file → inode。
- inode 的 in_ops 已经把“具体实现”隐藏了：同样的 vop_read，在 SFS 上走磁盘块映射与读写；在 stdout 上走字符输出；在 stdin 上走输入缓冲与阻塞等待。
- 所以上层逻辑只写一次：open/read/write/close 的通用路径，具体差异靠函数指针分派完成。

硬盘文件系统 SFS

```
// kern/fs/sfs/sfs.h
struct sfs_super {
    uint32_t magic;                                  /* magic number, should be
SFS_MAGIC */
    uint32_t blocks;                                 /* # of blocks in fs */
    uint32_t unused_blocks;                          /* # of unused blocks in fs
*/
    char info[SFS_MAX_INFO_LEN + 1];                 /* infomation for sfs */
};
```

SFS 用 block 作为磁盘管理单位

SFS 不按扇区做分配与寻址，而是用 block 作为最小单位，一个 block 是 4KB，并且与内存 page 大小一致。这样做直接带来的效果是：分配、回收、读写、映射都可以用“块号”来描述，减少换算与边界情况处理。代价是内部碎片更明显，例如只写入几十字节也至少占用一个 block。

SFS 的磁盘布局从块号顺序理解

块 0：superblock
块 1：root-dir inode
块 2 开始：freemap（位图，记录哪些块已用/空闲，占若干块）
后续块：目录/文件的 inode 与数据块（注意 inode 也按“一 inode 一整块”存放）

这意味着只要知道文件系统从哪里开始（镜像起始地址或设备起始块），就能固定地把 0 号块当超级块读出来，再根据超级块的参数定位 freemap 的范围，再进入其余数据区。

SFS block 0 Superblock

superblock 的字段含义与加载时机

magic(魔数)是文件系统格式标识。内核 mount 的时候会先读块 0，把它当 struct sfs_super 解析，然后先检查 magic 是否等于 SFS_MAGIC（0x2f8dbe2a）。不匹配就说明这段镜像不是 SFS，或者镜像损坏，mount 直接失败。
blocks 是整个文件系统总块数，也就是镜像总大小 / 4KB。后续所有对"块号是否合法"的判断都靠它，比如访问块号必须满足 0 <= blk < blocks。
unused_blocks 是空闲块数量统计。它不是唯一可信来源，真正的空闲信息在 freemap 位图里；unused_blocks 更像一个快速统计值，用于显示或快速判断空间是否足够。任何分配/释放块都会同时修改 freemap 的 bit 与 unused_blocks。
info 是一个字符串，用于描述文件系统（如 "simple file system"），对功能正确性影响不大，更多用于调试与识别。

SFS block 1 root-dir inode

root-dir inode 的固定位置

块 1 放根目录 inode。根目录就是整个 SFS 的路径解析起点。只要拿到根 inode，就能读取根目录的数据块，从而得到根目录下的条目（文件名 → inode 号）。
例如访问路径 `/bin/sh`，路径解析的第一步就是从根目录 inode 开始，在根目录里查找名为 `bin` 的目录项，找到它对应的 inode，再进入 `bin` 目录继续查找 `sh`。

SFS block 2 freemap

freemap 的位图表示与定位方式

从块 2 起是 freemap。它用 1 bit 对应 1 个 block，bit=1 表示该块已占用，bit=0 表示空闲。
因为 freemap 自己也占用 block，所以 freemap 对应的那些 block 在 freemap 里也会被标记为已用。
SFS 提供 `kern/fs/sfs/bitmap.[ch]` 来把 "块号 ↔ 位图中的 bit 位置" 这件事封装好。常见接口行为是：给一个 blockno，计算它落在 freemap 的第几个 block、那个 block 内的第几个 bit，然后读取/设置这个 bit。

剩余空间如何存 inode 和 data

superblock、root inode、freemap 之后的区域存放所有其他文件/目录的 inode 与数据块。SFS 的一个简化点是：一个磁盘 inode（sfs_disk_inode）不管实际结构大小多少，都独占一个 4KB block。这样 inode 的定位非常直接：inode 号就是它所在的块号，读该块即可得到 inode 内容。代价是 inode 区浪费空间，但实现更简单。

在 sfs_fs.c 文件中的 sfs_do_mount 函数中，完成了加载位于硬盘上的 SFS 文件系统的超级块 superblock 和 freemap 的工作。这样，在内存中就有了 SFS 文件系统的全局信息。

```
// kern/fs/sfs/sfs.h
struct sfs_disk_inode {
    uint32_t size;                          // regular file: file size
    uint16_t type;                          // file type
    uint16_t nlinks;                        // hard link count
    uint32_t blocks;                        // number of data blocks
    uint32_t direct[SFS_NDIRECT];           // direct block numbers
    uint32_t indirect;                      // indirect block number
};
```

sfs_disk_inode 在磁盘上的意义

它是“磁盘上文件/目录的元数据 + 数据块索引表”。文件内容不直接放在 inode 里，而是放在若干数据块里，inode 里记录这些数据块的块号。
这个结构体以连续字节的形式存到磁盘 block 中，读入内存后把这段字节按该结构解释；写回磁盘时反向把结构体内容按字节写回 block。

size、type、nlinks、blocks 的含义

size 表示文件大小（字节数）。对普通文件，它决定读到哪里算 EOF；对目录，也可以用它表示目录数据的逻辑大小（依实现），但目录通常按 entry 槽位理解更直接。
type 表示 inode 类型，例如普通文件或目录（具体取值在 sfs.h）。VFS 层会用它决定允许哪些操作：目录允许 lookup/getdirentry，不允许按普通文件方式 read/write（或语义不同）。
nlinks 是硬链接数。SFS 即使简化，也要支持“一个 inode 可以被多个目录项引用”这一语义，删除文件时只有 nlinks 降到 0 才能回收 inode 与数据块。
blocks 表示该文件/目录当前拥有多少个数据块（不是 inode 本身占用的块数），也就是 direct/indirect 这套索引指向的数据块数量。它直接影响 bmap（按索引取块号）时 index 的合法范围。

direct 与 indirect 的寻址逻辑

direct 是直接块号数组，默认 SFS_NDIRECT=12。也就是文件的前 12 个数据块，inode 里直接存块号。
indirect 是一级间接索引块的块号。当文件超过 12 个数据块时，indirect 指向一个“间接块”，这个间接块里不存文件数据，而是存一整块的块号数组（每项 4 字节）。由于一个 block 是 4096 字节，所以能存 4096/4 = 1024 个块号，因此间接部分最多再引用 1024 个数据块。

最大文件大小怎么得到

直接部分最多 12 个块，每块 4KB，总计 48KB。
间接部分最多 1024 个块，每块 4KB，总计 4096KB = 4MB。
所以最大约为 48KB + 4MB（忽略元数据开销）。这解释了为什么 SFS 只实现一级间接：容量足够实验用，逻辑简单。

0 作为无效索引为什么可行

块号 0 固定是 superblock，不可能被分配给普通文件数据块或 inode 块，所以用 0 作为“无效块号/未分配”不会与合法数据块冲突。
因此 direct[i]==0 或 indirect==0 都可以表示“还没有分配对应块”。

具体例子：一个 10KB 的文件如何映射

10KB 约等于 3 个 4KB 块（最后一块有内部碎片）。
它的 inode 会满足 blocks=3，direct[0..2] 分别是三个数据块的块号，direct[3..] 为 0，indirect 为 0。
读取时，如果要读第 0 个块的数据，就取 direct[0]；要读第 2 个块，就取 direct[2]。写入扩展到第 13 个块时才会触发 indirect 分配。

具体例子：一个 80KB 的文件如何映射

80KB 需要 20 个 4KB 块。
前 12 个块：direct[0..11] 存块号。
剩余 8 个块：需要 indirect。此时 indirect 指向一个间接块，这个间接块里第 0..7 项存对应的数据块号。
inode 的 blocks=20，read/write 第 15 个块时会走 indirect 路径读取间接块，再取第 3 项块号。

假设根目录下有一个文件 `a.txt`，它的 inode 在磁盘 block 100。

1. 内核 mount(mount:把一个文件系统接到当前系统的目录树上，让它可以通过路径访问) 时：

- 读 **block 0** 得到 `superblock`（检查 magic，知道总块数，知道 freemap 在哪）
- 读 **block 1** 得到“根目录 inode”（这是一个 `sfs_disk_inode`）

2. 访问 `/a.txt` 时：

- 用 **根目录 inode** 去读它的目录数据块（这些块号来自根目录 inode 的 `direct[]/indirect`）
- 在目录内容里找到一个 `sfs_disk_entry { name="a.txt", ino=100 }`
- 于是知道 `a.txt` 的 inode 在 **block 100**，再读 block 100 得到 `a.txt` 的 `sfs_disk_inode`

3. 读 `a.txt` 内容时：

- 看 `a.txt` inode 的 `direct[0]`，比如它等于 300
- 那么 `a.txt` 的第一个数据块就在 **磁盘 block 300**
- 继续读 `direct[1]`、`direct[2]`……

你会看到：**superblock 从来不通过 `direct[0]` 找到**，它是“约定死在 block 0”；`direct[0]` 是“文件自己的第一个数据块”。

```
// kern/fs/sfs/sfs.h
struct sfs_disk_entry {
    uint32_t ino;                                   // inode number == disk block number
    char name[SFS_MAX_FNAME_LEN + 1];               // filename
};
```

sfs_disk_entry 的作用：目录内容的存储格式

目录本质上是“名字到 inode 的映射表”。SFS 把目录的数据块内容解释为若干个 sfs_disk_entry 组成的序列。

name 是目录项名字。ino 是该名字对应的 inode 编号。由于 SFS 让 inode 编号等于其所在磁盘块号，所以 ino 也是一个块号。比如，root block 的 inode 编号为 1

ino 为 0 表示无效项

目录项被删除时，SFS 不一定立即压缩目录内容，而是把该 entry 的 ino 置 0，表示这个槽位空了。之后创建新文件时可以复用这些 ino==0 的槽位，避免目录不断增长。

此外，和 inode 相似，每个 sfs_disk_entry 也占用一个 block。

具体例子：根目录下有 `bin` 与 `readme.txt`

根目录 inode 的数据块里会有两条 entry：
一条 name="bin"，ino=某个块号（这个块号是 `bin` 目录 inode 所在的块）
一条 name="readme.txt"，ino=另一个块号（这个块号是 `readme.txt` 文件 inode 所在的块）
路径解析 `/bin/ls` 时先在根目录 entry 里找 "bin" 得到 ino，读该 ino 对应块拿到 `bin` 目录 inode，再在 `bin` 目录里找 "ls"。

```c
// kern/fs/sfs/sfs.h
struct sfs_inode {
    struct sfs_disk_inode *din;               /* on-disk inode */
    uint32_t ino;                             /* inode number */
    uint32_t flags;                           /* inode flags */
    bool dirty;                               /* true if inode modified */
    int reclaim_count;                        /* kill inode if it hits zero */
    semaphore_t sem;                          /* semaphore for din */
    list_entry_t inode_link;                  /* list in sfs_fs */
    list_entry_t hash_link;                   /* hash list in sfs_fs */
};
```

内存 inode 为什么要在磁盘 inode 之上再包一层

磁盘 inode（sfs_disk_inode）只负责持久化的元数据与索引信息。内核运行时还需要额外信息来支持并发、缓存、回写与快速查找，所以在内存里用 sfs_inode 作为运行时对象。
打开文件时会把磁盘 inode 读入内存（或通过缓存命中已有对象），形成 sfs_inode；关机或卸载时内存对象消失，但磁盘上的 sfs_disk_inode 仍在镜像里。

din 指向磁盘 inode 的内存副本

din 是一段内存，内容来自磁盘读出的 inode 块。对文件大小、direct/indirect 的修改最终都会反映到 din 里。
写回磁盘时把 din 的内容写回对应 inode 块。

ino 仍然是 inode 编号（块号）

因为 SFS 的 inode 编号就是块号，所以 ino 同时也是“去磁盘读写这个 inode”的定位依据。

dirty 表示是否需要回写

只要修改了 din（例如扩展文件导致 blocks 增加、direct/indirect 更新、size 更新），就必须把 dirty 置 true。
在 close 或 fsync、或者 inode 被回收时，若 dirty==true，就把 din 写回磁盘，保证持久化一致。

sem 保护 inode 的并发访问

多个线程可能同时读写同一个文件或目录。对 din、blocks 映射、目录 entry 的增删查都必须在持有 sem 的情况下进行，否则会出现：
一个线程扩展文件同时另一个线程读 indirect，读到不一致的索引表；
一个线程删除目录项同时另一个线程创建文件复用槽位，导致目录内容破坏。
所以你看到很多函数名带 _nolock，含义是调用者必须先拿到 sem。

inode_link / hash_link 用于缓存管理

SFS 需要快速从 ino 找到对应的内存 inode，因此会维护链表与哈希桶。
hash_link 挂入哈希表，支持“给定 ino 快速定位 sfs_inode”。
inode_link 可能用于全局链表管理，便于遍历、回收、刷盘等。

---

sfs_do_mount 在 mount 阶段做什么

mount 的目标是把磁盘上的文件系统全局结构读入内存，至少包括 superblock 与 freemap，这样后续才能分配/回收块、查找根目录 inode、进行路径解析。
sfs_do_mount 的典型流程是：读取块 0 得到 sfs_super 并校验 magic；根据 blocks 计算 freemap 占用多少块并读入；把这些信息挂到 sfs_fs（文件系统实例）对象里，形成可用的运行时状态。

"魔数"在 SFS 里怎么用

SFS 的 superblock.magic 是格式识别的第一道检查。mount 的时候读块 0 后立即比对 magic，失败就拒绝继续解析。
这样避免把"不是 SFS 的镜像"按 SFS 的结构去读，导致后续 freemap、inode、目录项的解析全错。

具体例子：把一个随机文件当作 sfs.img

如果这个随机文件前 4 字节不是 0x2f8dbe2a，mount 直接失败。
如果没有 magic 检查，内核可能会把随机字节当 blocks 数，算出一个不合理的 freemap 长度，再去读超出镜像范围的数据，最终触发错误或破坏内存。

```
// kern/fs/sfs/sfs_inode.c
static const struct inode_ops sfs_node_fileops = {
    .vop_magic                      = VOP_MAGIC,
    .vop_open                       = sfs_openfile,
    .vop_close                      = sfs_close,
    .vop_read                       = sfs_read,
    .vop_write                      = sfs_write,
    /* ... */
};
```

文件 inode 的 inode_ops 表示"普通文件支持哪些操作"

VFS 层不会直接调用 sfs_read/sfs_write，而是调用 VOP_READ(node, iob) 这类宏，最终跳到 node->in_ops->vop_read。
当一个 inode 代表普通文件时，它的 in_ops 指向 sfs_node_fileops，于是 vop_read 就是 sfs_read，vop_write 就是 sfs_write。
sfs_openfile 可能只做参数检查或空实现；sfs_close 会在需要时把 dirty 的 inode 与相关缓存写回磁盘；sfs_read/sfs_write 通常调用更底层的 sfs_io，通过块设备接口把数据读写到对应的数据块。

用户态具体例子（Keykeykey）

具体例子：用户态调用 write(fd, buf, 100)

fd → file → inode（VFS inode）
VFS inode 识别这是 SFS 普通文件 → inode->in_ops = sfs_node_fileops
VFS 层调用 vop_write → 实际进入 sfs_write → 再进入 sfs_io
sfs_io 根据 file->pos 计算当前写入落在哪个数据块 index，调用 bmap 获取对应块号，不存在则分配新块并标 dirty，最后通过磁盘驱动把 100 字节写入正确 block 的正确偏移位置，并推进 pos 与更新 size。

详细解释：

```
1) fd → file：fd 只是"索引号"
```

- 进程里有 `files_struct.fd_array`（打开文件表）
- `fd` 就是这个数组下标
- 所以 `write(fd, ...)` 的第一步是：
  - 找到 `current->files->fd_array[fd]` 得到 `struct file *f`

  - 里面有 `f->pos`（当前偏移）和 `f->node`（指向 VFS inode）

你可以把它理解成：**fd 不是文件本体，只是"我这进程打开的第几个文件"的编号**。

```
2) file → inode：inode 是"文件本体的代表"
```

`f->node` 指向 `struct inode`（VFS inode），它是 VFS 统一包装的 inode：

- `inode->in_type` 表示底下是哪种文件系统（device / sfs）
- `inode->in_ops` 是一组函数指针（open/read/write/...）

当 VFS 发现这个 inode 属于 SFS 且是普通文件时，就会让：

- `inode->in_ops = sfs_node_fileops`
- 这样 `VOP_WRITE(inode, iob)` 实际就是调 `sfs_write(inode, iob)`

关键点：**VFS 并不自己写磁盘，它只负责把请求"派发"给具体文件系统实现。**

```
3) pos → (block index, block offset)：把"字节偏移"换成"落在哪个块"
```

SFS 的 block 大小是 4KB = 4096 字节。

假设：

- `pos = f->pos`
- 写入长度 `len = 100`

那么：

- `block_index = pos / 4096`
- `block_offset = pos % 4096`

写入可能跨块，所以通常会循环写：

- 第一次最多能写 `min(len, 4096 - block_offset)` 字节到当前块
- 写完更新 `pos += wrote`，`len -= wrote`，继续下一块

举一个具体数值让你看清：

例 1：pos=0 写 100 字节

- block_index = 0 / 4096 = 0
- block_offset = 0
- 本次写不会跨块，直接写到第 0 个数据块的偏移 0..99

例 2：pos=4080 写 100 字节（会跨块）

- block_index = 4080/4096 = 0
- block_offset = 4080
- 当前块剩余空间 = 4096-4080 = 16
- 先写 16 字节到块0末尾，再写剩余 84 字节到块1开头

这就是 “pos 计算落在哪个数据块 index” 的真正含义。

```
4) bmap 是什么：把“第 block_index 个逻辑数据块”映射到“磁盘块号”
```

文件在 inode 里记录的是“数据块号”，但你写的时候说的是“第几个块”（逻辑块序号）。

- 逻辑块序号：block_index（0,1,2,...）
- 物理块号：磁盘上的 block number（例如 57, 103, ...）

`bmap` 做的是：

- 给你一个 inode 和 `block_index`
- 返回“这个逻辑块对应的物理块号”
- 如果这个逻辑块还不存在：
  - 写操作允许扩展文件，就分配一个新空闲块（从 freemap 找）
  - 把新块号写进 inode 的 direct[] 或 indirect 表
  - inode 标记 dirty（表示 inode 元数据被改了，之后要写回磁盘）

所以 bmap 是**映射 + 必要时分配。**

```
5) direct / indirect 怎么参与 bmap
```

SFS 里 `SFS_NDIRECT = 12`：

- block_index = 0..11 走 direct：
  - 物理块号 = din->direct[block_index]
  - 如果是 0，说明还没分配 → 分配新块并填进去
- block_index >= 12 走 indirect：
  - 先看 din->indirect 指向的“间接块”是否存在
    - 不存在则先分配一个间接块，并把 din->indirect 设为它的块号
  - 间接块里存了很多个“数据块号”
  - 用 `(block_index - 12)` 作为下标去查/填

你之前问过“superblock 不是在 direct[0] 吗？”——不是。

- **superblock 是整个文件系统的 block 0**（全局固定位置）
- `direct[0]` 是“这个文件的第 0 个数据块”的块号
- 也就是说：superblock 属于“文件系统元数据区域”，不属于任何普通文件的数据块，更不可能放进某个文件的 direct[] 里

```
6) 真正写盘：磁盘块号 + 块内偏移 → 发给设备层
```

当 `sfs_io` 算出：

- 物理块号 `blkno`
- 块内偏移 `block_offset`

- 这次写多少字节 `n`

它会构造一次对“disk device”的写请求：

- 写到 `blkno` 对应的 4KB block
- 从 `block_offset` 开始覆盖 `n` 字节

然后：

- `f->pos += n`
- 如果 `f->pos` 超过原来的文件大小，就更新 inode->size
- inode/din 被改了就 dirty，必要时 `close` 或 `fsync` 把 inode 写回

```
把你那段话翻译成“发生了什么”（一句一句对应）
```

- “fd → file → inode”
  fd 找到进程打开文件表里的 `struct file`，里面的 `node` 指向 VFS inode。
- “识别这是 SFS 普通文件 → inode->in_ops = sfs_node_fileops”
  inode 的类型决定用哪套函数指针；SFS 普通文件用 `sfs_node_fileops`。
- “VFS 调 vop_write → sfs_write → sfs_io”
  `vop_write` 是抽象接口，最终落到 SFS 的实现。
- “sfs_io 根据 pos 计算 index”
  `index = pos/4096`，`offset = pos%4096`，跨块就循环。
- “bmap 获取块号，不存在则分配并 dirty”
  direct/indirect 找到物理块号；没有就从 freemap 分配新块并更新 inode 元数据。
- “通过磁盘驱动把 100 字节写入正确 block 的正确偏移”
  有了物理块号 + 块内偏移，就能发起设备写。
- “推进 pos 与更新 size”
  写了多少就 pos 加多少；写过文件尾就 size 变大。

```c
// kern/fs/sfs/sfs_inode.c
static const struct inode_ops sfs_node_dirops = {
    .vop_magic                  = VOP_MAGIC,
    .vop_open                   = sfs_opendir,
    .vop_close                  = sfs_close,
    .vop_getdirentry            = sfs_getdirentry,
    .vop_lookup                 = sfs_lookup,
    /* ... */
};
```

目录 inode 的 inode_ops 表示“目录支持路径查找与枚举”

目录也是 inode，但它与普通文件的差别在于：目录的内容数据不是任意字节流，而是由 sfs_disk_entry 组成的“名字表”。
因此目录的读取不是用 vop_read，而是用 vop_getdirentry 来按条目读出目录项；路径解析需要用 vop_lookup 在目录下按名字查找子项。
当 inode 代表目录时，in_ops 指向 sfs_node_dirops，于是 lookup/getdirentry 会进入 sfs_lookup/sfs_getdirentry 等实现。

具体例子：用户态调用 open("/tmp/a.txt", O_RDONLY)

VFS 从根目录 inode 开始，调用目录 inode 的 vop_lookup 查找 "tmp"，得到 tmp 目录 inode；再对 tmp 目录 inode 调用 vop_lookup 查找 "a.txt"，得到文件 inode。
最后对文件 inode 调用 vop_open（文件 ops 表）完成打开流程。整个过程上层只看到 vop_lookup/vop_open，不需要写任何 “如果是 SFS 就调用 sfs_lookup” 的分支。

```
// kern/fs/vfs/inode.h
struct inode_ops {
    unsigned long vop_magic;
    int (*vop_open)(struct inode *node, uint32_t open_flags);
    int (*vop_close)(struct inode *node);
    int (*vop_read)(struct inode *node, struct iobuf *iob);
    int (*vop_write)(struct inode *node, struct iobuf *iob);
    int (*vop_fstat)(struct inode *node, struct stat *stat);
    int (*vop_fsync)(struct inode *node);
    int (*vop_namefile)(struct inode *node, struct iobuf *iob);
    int (*vop_getdirentry)(struct inode *node, struct iobuf *iob);
    int (*vop_reclaim)(struct inode *node);
    int (*vop_gettype)(struct inode *node, uint32_t *type_store);
    int (*vop_tryseek)(struct inode *node, off_t pos);
    int (*vop_truncate)(struct inode *node, off_t len);
    int (*vop_create)(struct inode *node, const char *name, bool excl, struct
inode **node_store);
    int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
    int (*vop_ioctl)(struct inode *node, int op, void *data);
};
```

inode_ops 作为抽象层接口怎么把 SFS“接到”VFS 上

VFS 只认 inode_ops 里的这些统一动作：open/close/read/write/lookup/create 等。每个具体文件系统把自己的实现函数填进去。
对 SFS 来说，普通文件填 sfs_node_fileops，目录填 sfs_node_dirops；如果还有设备文件系统，则会填另外一套 ops。
因此系统调用层实现 open/read/write 时不需要写 “SFS 用 sfs_write，设备用 dev_write” 的判断，它只通过 inode->in_ops 进行分派。

具体例子：同样是 write，写到普通文件与写到串口

写普通文件：inode->in_ops = sfs_node_fileops → vop_write = sfs_write → 写磁盘块
写串口设备：inode->in_ops = dev_node_ops → vop_write = dev_write → 写串口寄存器/缓冲区
系统调用层完全不改，调用路径只在函数指针分派处发生分岔。

辅助函数（_nolock）在数据块与目录项管理中的角色

sfs_bmap_load_nolock 的工作是“给定文件的第 index 个数据块，得到它对应的块号；如果 index 等于当前 blocks，就扩展文件分配新数据块”。它会在需要时分配 direct 或 indirect 所需的结构，并把 inode 标 dirty。
具体例子：向一个 48KB 的文件再写入 1KB
原本 blocks=12，写入会触发需要第 12 号数据块（从 0 开始计），index==blocks，于是分配第 13 个数据块，同时若 indirect 还没分配，会先分配 indirect block，再把新数据块号写入 indirect[0]，然后 blocks 变为 13，dirty=true。

sfs_bmap_truncate_nolock 是释放最后一个数据块的逆操作，常用于 truncate 或删除文件时回收空间。它会修改 direct/indirect 索引，并在必要时释放 indirect block 自身，最后更新 blocks，并置 dirty。
具体例子：删除一个有 20 个数据块的文件
会循环释放最后一个块：先释放 indirect 的第 7 个索引块号指向的数据块，再释放第 6 个......直到释放完 indirect 部分；当 blocks 回到 12 时，若间接块已空，会释放 indirect block 并把 inode->indirect 置 0；再继续释放 direct[11] 到 direct[0]。

sfs_dirent_read_nolock 用于"从目录里按 slot 读出一个 entry"。它会根据 slot 计算应该读目录的第几个数据块，再读取该块中对应位置的 sfs_disk_entry。
具体例子：列出目录内容时，slot 从 0 增长
每次读取一个 entry，如果 entry.ino==0 就跳过；否则输出 entry.name，并可进一步读取 ino 对应 inode。

sfs_dirent_search_nolock 用于"在目录下按 name 查找 entry"。它会遍历目录的 entry，找到 name 匹配且 ino!=0 的项就返回对应 ino；同时也会记录第一个空闲槽位（ino==0），便于 create 时复用。
具体例子：在目录里创建 "a.txt"
先 search，如果找到同名且 excl=true 就失败；如果没找到，就看 search 返回的 free_slot 是否存在，存在则把该槽位写成新 entry（ino=新分配 inode 块号，name="a.txt"），否则就在目录末尾追加一个 entry（可能需要给目录扩展新的数据块）。

设备即文件

这次实验，我们可以把设备也当成一个文件去访问。。目前实现了 stdin 设备文件文件、 stdout 设备文件、 disk0 设备。 stdin 设备就是键盘， stdout 设备就是控制台终端的文本显示，而 disk0 设备是承载 SFS 文件系统的磁盘设备。

为了表示一个设备，需要有对应的数据结构，ucore 为此定义了 struct device，如下：

可以认为 struct device 是一个比较抽象的"设备"的定义。一个具体设备，只要实现了 d_open() 打开设备， d_close() 关闭设备， d_io() (读写该设备，write参数是true/false决定是读还是写)， d_ioctl() (input/output control)四个函数接口，就可以被文件系统使用了。

```
// kern/fs/devs/dev.h
/*
 * Filesystem-namespace-accessible device.
 * d_io is for both reads and writes; the iobuf will indicates the direction.
 */
struct
 device {
    size_t d_blocks;
    size_t d_blocksize;
    int (*d_open)(struct device *dev, uint32_t open_flags);
    int (*d_close)(struct device *dev);
    int (*d_io)(struct device *dev, struct iobuf *iob, bool write);
    int (*d_ioctl)(struct device *dev, int op, void *data);
};

#define dop_open(dev, open_flags)           ((dev)->d_open(dev, open_flags))
#define dop_close(dev)                      ((dev)->d_close(dev))
#define dop_io(dev, iob, write)             ((dev)->d_io(dev, iob, write))
#define dop_ioctl(dev, op, data)            ((dev)->d_ioctl(dev, op, data))
```

但这个设备描述没有与文件系统以及表示一个文件的 inode 数据结构建立关系，为此，还需要另外一个数据结构把 device 和 inode 联通起来，这就是 vfs_dev_t 数据结构。

利用 vfs_dev_t 数据结构，就可以让文件系统通过一个链接 vfs_dev_t 结构的双向链表找到 device 对应的 inode 数据结构，一个 inode 节点的成员变量 in_type 的值是 0x1234，则此 inode 的成员变量 in_info 将成为一个 device 结构。这样 inode 就和一个设备建立了联系，这个 inode 就是一个设备文件。

```
// kern/fs/vfs/vfsdev.c
// device info entry in vdev_list
typedef struct
 {
    const char *devname;
    struct inode *devnode;
    struct fs *fs;
    bool mountable;
    list_entry_t vdev_link;
} vfs_dev_t;
#define le2vdev(le, member)                         \
    to_struct((le), vfs_dev_t, member)
//为了使用链表定义的宏，做到现在应该对它很熟悉了

static list_entry_t vdev_list;     // device info list in vfs layer
static semaphore_t vdev_list_sem; // 互斥访问的semaphore
static void lock_vdev_list(void)
{
    down(&vdev_list_sem);
}
static void unlock_vdev_list(void)
{
    up(&vdev_list_sem);
}
```

为了把这些设备链接在一起，还定义了一个设备链表，即双向链表 vdev_list，这样通过访问此链表，可以找到 ucore 能够访问的所有设备文件。

注意这里的 vdev_list 对应一个 vdev_list_sem 。在文件系统中，互斥访问非常重要，所以我们将看到很多的 semaphore 。

我们使用 iobuf 结构体传递一个IO请求（要写入设备的数据当前所在内存的位置和长度/从设备读取的数据需要存储到的位置

注意设备文件的inode也有一个 inode_ops 成员, 提供设备文件应具备的接口。

```
// kern/fs/devs/dev.c
/*
 * Function table for device inodes.
 */
static const struct inode_ops dev_node_ops = {
    .vop_magic                      = VOP_MAGIC,
    .vop_open                       = dev_open,
    .vop_close                      = dev_close,
    .vop_read                       = dev_read,
    .vop_write                      = dev_write,
```

```
    .vop_fstat                      = dev_fstat,
    .vop_ioctl                      = dev_ioctl,
    .vop_gettype                    = dev_gettype,
    .vop_tryseek                    = dev_tryseek,
    .vop_lookup                     = dev_lookup,
};
```

stdin 设备

代码主要拿来看文件位置，免得答辩找不到：

```
// user/libs/file.c
int read(int fd, void *base, size_t len) {
    return sys_read(fd, base, len);
}

// user/libs/umain.c
static int initfd(int fd2, const char *path, uint32_t open_flags) {
    int fd1, ret;
    if ((fd1 = open(path, open_flags)) < 0) {
        return fd1;
    }
    if (fd1 != fd2) {
        close(fd2);
        ret = dup2(fd1, fd2);
        close(fd1);
    }
    return ret;
}

void umain(int argc, char *argv[]) {
    int fd;
    if ((fd = initfd(0, "stdin:", O_RDONLY)) < 0) { ... }
    if ((fd = initfd(1, "stdout:", O_WRONLY)) < 0) { ... }
    int ret = main(argc, argv);
    exit(ret);
}
```

```
// kern/trap/trap.c (timer interrupt path)
case IRQ_S_TIMER:
    clock_set_next_event();
    ...
    run_timer_list();
    dev_stdin_write(cons_getc());
    break;
```

```
// kern/driver/console.c
static struct {
    uint8_t buf[CONSBUFSIZE];
    uint32_t rpos;
    uint32_t wpos;
} cons;

int cons_getc(void) {
```

```
    int c = 0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        serial_intr(); // poll serial and push into cons.buf
        if (cons.rpos != cons.wpos) {
            c = cons.buf[cons.rpos++];
            if (cons.rpos == CONSBUFSIZE) cons.rpos = 0;
        }
    }
    local_intr_restore(intr_flag);
    return c;
}
```

```
// kern/fs/devs/dev_stdin.c
#define STDIN_BUFSIZE 4096
static char stdin_buffer[STDIN_BUFSIZE];
static off_t p_rpos, p_wpos;
static wait_queue_t __wait_queue, *wait_queue = &__wait_queue;

void dev_stdin_write(char c) {
    bool intr_flag;
    if (c != '\0') {
        local_intr_save(intr_flag);
        {
            stdin_buffer[p_wpos % STDIN_BUFSIZE] = c;
            if (p_wpos - p_rpos < STDIN_BUFSIZE) {
                p_wpos++;
            }
            if (!wait_queue_empty(wait_queue)) {
                wakeup_queue(wait_queue, WT_KBD, 1);
            }
        }
        local_intr_restore(intr_flag);
    }
}

static int dev_stdin_read(char *buf, size_t len) {
    int ret = 0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        for (; ret < len; ret++, p_rpos++) {
        try_again:
            if (p_rpos < p_wpos) {
                *buf++ = stdin_buffer[p_rpos % STDIN_BUFSIZE];
            } else {
                wait_t __wait, *wait = &__wait;
                wait_current_set(wait_queue, wait, WT_KBD);
                local_intr_restore(intr_flag);

                schedule(); // sleep until woken

                local_intr_save(intr_flag);
                wait_current_del(wait_queue, wait);
```

```
                if (wait->wakeup_flags == WT_KBD) {
                    goto try_again;
                }
                break;
            }
        }
    }
    local_intr_restore(intr_flag);
    return ret;
}
```

stdin 的整体工作流程（按时间顺序）

用户态先把 stdin “打开成文件”

- 用户程序真正跑起来之前，会先跑 `umain()`。
- `initfd(0, "stdin:", O_RDONLY)` 的目的：让“文件描述符 0”绑定到一个叫 `"stdin:"` 的设备文件。
- 之后用户态调用 `read(0, buf, len)`，就等价于 `sys_read(0, buf, len)`，也就是“从 fd=0 对应的那个设备读数据”。

内核里用“时钟中断轮询”把键盘/串口字符拿进来

- QEMU 环境里收不到标准的键盘中断，于是内核在 **每次时钟中断**里做一次检查：
    - `cons_getc()`：去 console 子系统里看看有没有新字符
    - 如果有就返回一个字符；没有就返回 0
    - 然后把这个字符交给 `dev_stdin_write(c)`，等于“把输入喂给 stdin 设备”

为什么QEMU收不到键盘中断?

在 ucore 的这个实验环境里，“收不到键盘中断”通常不是说 QEMU 完全不能产生键盘事件，而是 **当前这套输入链路没有把键盘事件以“外设中断”的方式送到内核。** 常见原因有三类：

- **输入设备根本没走“键盘控制器 → 中断”的模型**
  ucore 在 QEMU 里通常是通过 **SBI 控制台**读字符（你代码里就是 `sbi_console_getchar()`），这更像“固件提供的轮询接口”。它返回字符，但并不等价于“有一个键盘控制器给 CPU 拉中断线”。
- **缺少对应硬件/驱动的中断接入**
  要想有“键盘中断”，需要：
    - QEMU 里提供一个会产生中断的键盘设备（例如 PS/2 控制器或 virtio-input 等）
    - 设备树/平台把它挂到某个中断号
    - 内核里有该设备的驱动，并把中断处理函数注册到 PLIC/中断控制器
      ucore 实验一般没有实现完整的键盘控制器驱动与中断注册路径，所以即便 QEMU 有键盘输入，也只被当成“控制台输入”，不会触发你期待的 IRQ。
- **实验选择了最简实现：用时钟中断“补一个轮询点”**
  时钟中断是肯定存在的（调度/定时器必须用），所以在 timer IRQ 里调用 `cons_getc()` 轮询一次，能保证最终把字符搬进 stdin 缓冲区。这是“没有键盘中断链路时”的替代方案。

一句话：**输入走的是 SBI/串口控制台的“轮询读字符”，而不是键盘控制器的“中断驱动输入”，所以内核看不到键盘 IRQ，只能借助时钟中断周期性去 poll。**

---

console 缓冲区做“硬件输入采集”

- `cons_getc()` 内部会先调用 `serial_intr()`，它本质是在轮询串口：把硬件来的字符尽量塞进 `cons.buf`（512 环形队列）。
- `cons_getc()` 再从 `cons.buf` 里取出 **一个** 字符返回给调用者。
- 所以 console 这一层更像"把硬件输入先收进来，别丢"。

stdin 设备缓冲区做"给 read 的稳定队列 + 阻塞唤醒"

- `dev_stdin_write(c)` 把字符写入 `stdin_buffer`（4096 环形队列），并且：
  - 如果有进程正在等待读（wait_queue 里有人），就 `wakeup_queue` 唤醒它。
- 用户进程调用 `sys_read(0, ...)` 最终会走到 `dev_stdin_read(buf, len)`：
  - 如果 `stdin_buffer` 有数据：拷贝字符到用户缓冲区，返回读取字节数
  - 如果没有数据：把当前进程挂到 `wait_queue`，`schedule()` 睡眠
  - 之后时钟中断把新字符写进来并唤醒，进程再回来继续读

一句话总结路径

- 输入路径：硬件/串口 → console(512 缓冲) → cons_getc() → dev_stdin_write() → stdin(4096 缓冲 + 唤醒)
- 读取路径：user read(0) → sys_read → 设备 stdin 的 d_io → dev_stdin_read → 从 stdin_buffer 拿数据（没数据就睡）

能不能把 console 的缓冲区和 stdin 的缓冲区合并？为什么？

不建议直接合并，原因是两者职责不同、接口形态不同：

- console 缓冲区服务的是"底层采集/轮询"：`cons_getc()` 需要在很多场景可用（包括内核监控、可能中断关闭时），它倾向于"尽快把硬件字符收进来"，逻辑是"取一个就返回一个"，并不负责阻塞等待某个进程。
- stdin 缓冲区服务的是"文件语义的 read"：它必须支持 `read()` 的阻塞/唤醒（wait_queue）、一次读多个字节、与进程调度配合。这个语义 console 层没有。

如果强行合并，你要么：

- 让 console 缓冲区也承担 wait_queue/阻塞语义（console 变复杂，耦合到进程调度）
- 要么让 stdin read 直接读 console 缓冲区（会遇到"没数据时怎么睡、谁来唤醒、在中断关闭/不同调用路径下是否安全"等问题）

可以做的"半合并"是：去掉 `dev_stdin_write(cons_getc())` 这种"搬运"，改成 stdin 直接从 console 的"字符源"取数据，但仍然需要在 stdin 层保留自己的队列与 wait_queue 语义。换句话说：**可以共享"取字符的入口"，但不太适合共享同一个缓冲区实现。**

---

stdout

```
// kern/fs/devs/dev_stdout.c
static int stdout_io(struct device *dev, struct iobuf *iob, bool write)
{
    //对应struct device 的d_io()
    if (write) {
        char *data = iob->io_base;
        for (; iob->io_resid != 0; iob->io_resid--) {
            cputchar(*data++);
```

```
        }
        return 0;
    }
    //if !write:
    return -E_INVAL;//对stdout执行读操作会报错
}
```

stdout 的工作逻辑（大白话）

stdout 被当成“只能写的设备文件”。VFS/设备层最终会调用它的 `stdout_io()`。

- `write == true` 表示有人在对 stdout 做写操作（比如用户态 `write(1, buf, len)`）。
- `iob->io_base` 指向要输出的内存缓冲区（要打印的字符串/字节）。
- `iob->io_resid` 表示还剩多少字节需要处理。
- 循环里每次取一个字符 `*data`，调用 `cputchar()` 把这个字符送到控制台输出；同时 `data++` 前进，`io_resid--` 递减，直到输出完。

stdout 注释里的问题：为什么读会报错？

- 因为 stdout 这个设备的语义在这里被设计成“输出口”，只提供写能力。
- 如果有人调用 `read(1, ...)`，`write` 参数就是 false，代码直接返回 `-E_INVAL`，表示“这个设备不支持读”。
- 这属于接口层的约束：让错误尽早暴露，避免把“读 stdout”这种不合理操作悄悄当成读空数据。

---

disk0

```c
// kern/fs/devs/dev_disk0.c
static char *disk0_buffer;
static semaphore_t disk0_sem;

static void lock_disk0(void)  { down(&(disk0_sem)); }
static void unlock_disk0(void){ up(&(disk0_sem)); }

static void disk0_read_blks_nolock(uint32_t blkno, uint32_t nblks)
{
    int ret;
    uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
    if ((ret = ide_read_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0) {
        panic("disk0: read ... ret");
    }
}

static void disk0_write_blks_nolock(uint32_t blkno, uint32_t nblks)
{
    int ret;
    uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
    if ((ret = ide_write_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0) {
        panic("disk0: write ... ret");
    }
}

static int disk0_io(struct device *dev, struct iobuf *iob, bool write)
```

```
{
    off_t  offset = iob->io_offset;
    size_t resid  = iob->io_resid;
    uint32_t blkno = offset / DISK0_BLKSIZE;
    uint32_t nblks = resid  / DISK0_BLKSIZE;

    /* don't allow I/O that isn't block-aligned */
    if ((offset % DISK0_BLKSIZE) != 0 || (resid % DISK0_BLKSIZE) != 0) {
        return -E_INVAL;
    }

    /* don't allow I/O past the end of disk0 */
    if (blkno + nblks > dev->d_blocks) {
        return -E_INVAL;
    }

    /* read/write nothing ? */
    if (nblks == 0) {
        return 0;
    }

    lock_disk0();
    while (resid != 0) {
        size_t copied, alen = DISK0_BUFSIZE;
        if (write) {
            iobuf_move(iob, disk0_buffer, alen, 0, &copied);
            assert(copied != 0 && copied <= resid && copied % DISK0_BLKSIZE ==
0);
            nblks = copied / DISK0_BLKSIZE;
            disk0_write_blks_nolock(blkno, nblks);
        } else {
            if (alen > resid) {
                alen = resid;
            }
            nblks = alen / DISK0_BLKSIZE;
            disk0_read_blks_nolock(blkno, nblks);
            iobuf_move(iob, disk0_buffer, alen, 1, &copied);
            assert(copied == alen && copied % DISK0_BLKSIZE == 0);
        }
        resid -= copied, blkno += nblks;
    }
    unlock_disk0();
    return 0;
}
```

disk0 的工作逻辑（大白话）

disk0 是一个"块设备"：它只按 **整块**（block）读写，不支持像文件那样随便从任意字节偏移读一点写一点。它底下实际调用的是 ide 的"按扇区读写"接口，但对上层表现为"按 block 读写"。

整体流程按 `disk0_io()` 看：

- 上层传入一个 `iobuf`，里面包含三件关键事
  - `io_offset`：从磁盘的哪个偏移开始 I/O
  - `io_resid`：要读/写多少字节

    - `io_base`：内存缓冲区地址（读：写进这里；写：从这里取数据）
- 先把“字节偏移/长度”换算成块号/块数
    - `blkno = offset / DISK0_BLKSIZE`：从第几个块开始
    - `nblks = resid / DISK0_BLKSIZE`：涉及多少块
- 两个硬性检查（不满足直接报错 `-E_INVAL`）
    - 必须块对齐：`offset` 和 `resid` 都要是 `DISK0_BLKSIZE` 的整数倍
    - 不能越界：`blkno + nblks` 不能超过设备总块数 `dev->d_blocks`
- 用信号量做互斥
    - `lock_disk0()`：保证同一时刻只有一个线程在操作 disk0（避免多个线程共用 `disk0_buffer` 时互相覆盖数据，也避免读写交错导致错误）
- 循环处理（分批次），原因是 `disk0_buffer` 大小有限（`DISK0_BUFSIZE`）
    - 每次最多处理 `alen = DISK0_BUFSIZE` 字节（也就是一批若干个块）

写路径（write=true）做的事

- 先把用户/上层提供的内存数据，从 `iobuf` 搬到 `disk0_buffer`（`iobuf_move(..., 0, &copied)` 这里的 0 表示“从 iobuf → buffer”）
- `copied` 必须是块大小的整数倍（保证整块写）
- 计算这批对应的块数 `nblks = copied / DISK0_BLKSIZE`
- 调 `disk0_write_blks_nolock(blkno, nblks)`，把 `disk0_buffer` 里的数据写到磁盘对应块号的位置

读路径（write=false）做的事

- 先决定这一批读多少：`alen = min(DISK0_BUFSIZE, resid)`
- 计算这批要读的块数 `nblks`
- 调 `disk0_read_blks_nolock(blkno, nblks)`，把磁盘数据读进 `disk0_buffer`
- 再把 `disk0_buffer` 的数据搬回 `iobuf` 指向的内存（`iobuf_move(..., 1, &copied)` 这里的 1 表示“从 buffer → iobuf”）
- 每一批结束：`resid -= copied`，`blkno += nblks`，继续下一批直到读完/写完

为什么有 `disk0_buffer` 这种中间缓冲？

- 因为底层 ide 接口一次读写是“给定内存地址 + 扇区数量”，这里统一用 `disk0_buffer` 做一个固定大小的中转区，方便把上层任意的 iobuf（可能跨页、分段）整理成连续的一段再交给 ide。
- 同时也让分批次（`DISK0_BUFSIZE`）逻辑变简单。

open系统调用的执行过程

```c
// kern/fs/sysfile.c
/* sysfile_open - open file */
int sysfile_open(const char *__path, uint32_t open_flags)
{
    int ret;
    char *path;
    if ((ret = copy_path(&path, __path)) != 0) {
        return ret;
    }
    ret = file_open(path, open_flags);
    kfree(path);
    return ret;
}
```

```c
// kern/fs/file.c
// open file
int file_open(char *path, uint32_t open_flags)
{
    bool readable = 0, writable = 0;
    switch (open_flags & O_ACCMODE) {
    case O_RDONLY: readable = 1; break;
    case O_WRONLY: writable = 1; break;
    case O_RDWR:
        readable = writable = 1;
        break;
    default:
        return -E_INVAL;
    }
    int ret;
    struct file *file;
    if ((ret = fd_array_alloc(NO_FD, &file)) != 0) {
        return ret;
    }
    struct inode *node;
    if ((ret = vfs_open(path, open_flags, &node)) != 0) {
        fd_array_free(file);
        return ret;
    }
    file->pos = 0;
    if (open_flags & O_APPEND) {
        struct stat __stat, *stat = &__stat;
        if ((ret = vop_fstat(node, stat)) != 0) {
            vfs_close(node);
            fd_array_free(file);
            return ret;
        }
        file->pos = stat->st_size;
    }
    file->node = node;
    file->readable = readable;
    file->writable = writable;
    fd_array_open(file);
```

```
    return file->fd;
}
```

```
// kern/fs/vfs/vfsfile.c
int vfs_open(char *path, uint32_t open_flags, struct inode **node_store)
{
    bool can_write = 0;
    switch (open_flags & O_ACCMODE) {
    case O_RDONLY:
        break;
    case O_WRONLY:
    case O_RDWR:
        can_write = 1;
        break;
    default:
        return -E_INVAL;
    }

    if (open_flags & O_TRUNC) {
        if (!can_write) {
            return -E_INVAL;
        }
    }

    int ret;
    struct inode *node;
    bool excl   = (open_flags & O_EXCL)  != 0;
    bool create = (open_flags & O_CREAT) != 0;

    ret = vfs_lookup(path, &node);

    if (ret != 0) {
        if (ret == -16 && (create)) {
            char *name;
            struct inode *dir;
            if ((ret = vfs_lookup_parent(path, &dir, &name)) != 0) {
                return ret;
            }
            ret = vop_create(dir, name, excl, &node);
        }
        else return ret;
    }
    else if (excl && create) {
        return -E_EXISTS;
    }
    assert(node != NULL);

    if ((ret = vop_open(node, open_flags)) != 0) {
        vop_ref_dec(node);
        return ret;
    }

    vop_open_inc(node);
    if (open_flags & O_TRUNC || create) {
        if ((ret = vop_truncate(node, 0)) != 0) {
```

```
            vop_open_dec(node);
            vop_ref_dec(node);
            return ret;
        }
    }
    *node_store = node;
    return 0;
}
```

```
// kern/fs/vfs/vfslookup.c（你给的片段里有这些）
static int get_device(char *path, char **subpath, struct inode **node_store)
{
    int i, slash = -1, colon = -1;
    for (i = 0; path[i] != '\0'; i ++) {
        if (path[i] == ':') { colon = i; break; }
        if (path[i] == '/') { slash = i; break; }
    }
    if (colon < 0 && slash != 0) {
        *subpath = path;
        return vfs_get_curdir(node_store);
    }
    if (colon > 0) {
        path[colon] = '\0';
        while (path[++ colon] == '/');
        *subpath = path + colon;
        return vfs_get_root(path, node_store);
    }

    int ret;
    if (*path == '/') {
        if ((ret = vfs_get_bootfs(node_store)) != 0) {
            return ret;
        }
    }
    else {
        assert(*path == ':');
        struct inode *node;
        if ((ret = vfs_get_curdir(&node)) != 0) {
            return ret;
        }
        assert(node->in_fs != NULL);
        *node_store = fsop_get_root(node->in_fs);
        vop_ref_dec(node);
    }

    while (*(++ path) == '/');
    *subpath = path;
    return 0;
}

int vfs_lookup(char *path, struct inode **node_store)
{
    int ret;
    struct inode *node;
    if ((ret = get_device(path, &path, &node)) != 0) {
```

```c
        return ret;
    }
    if (*path != '\0') {
        ret = vop_lookup(node, path, node_store);
        vop_ref_dec(node);
        return ret;
    }
    *node_store = node;
    return 0;
}

int vfs_lookup_parent(char *path, struct inode **node_store, char **endp)
{
    int ret;
    struct inode *node;
    if ((ret = get_device(path, &path, &node)) != 0) {
        return ret;
    }
    *endp = path;
    *node_store = node;
    return 0;
}
```

```c
// kern/fs/sfs/sfs_inode.c（你给的片段）
static int sfs_lookup(struct inode *node, char *path, struct inode **node_store)
{
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    assert(*path != '\0' && *path != '/');
    vop_ref_inc(node);
    struct sfs_inode *sin = vop_info(node, sfs_inode);
    if (sin->din->type != SFS_TYPE_DIR) {
        vop_ref_dec(node);
        return -E_NOTDIR;
    }
    struct inode *subnode;
    int ret = sfs_lookup_once(sfs, sin, path, &subnode, NULL);

    vop_ref_dec(node);
    if (ret != 0) {
        return ret;
    }
    *node_store = subnode;
    return 0;
}

static int sfs_lookup_once(struct sfs_fs *sfs, struct sfs_inode *sin, const char
*name,
                           struct inode **node_store, int *slot)
{
    int ret;
    uint32_t ino;
    lock_sin(sin);
    {
        ret = sfs_dirent_search_nolock(sfs, sin, name, &ino, slot, NULL);
    }
```

```c
    unlock_sin(sin);
    if (ret == 0) {
        ret = sfs_load_inode(sfs, node_store, ino);
    }
    return ret;
}
```

open 的完整调用链（从用户态到 SFS）

假设用户程序执行：

- `fd = open("/sfs_filetest1", O_RDONLY);`

内核里大致按下面这条链走（你给的代码对应的）：

- `sysfile_open(__path, flags)`
  → `file_open(path, flags)`
  → `vfs_open(path, flags, &node)`
  → `vfs_lookup(path, &node)`
  → `get_device(path, &subpath, &start_inode)`
  → `vop_lookup(start_inode, subpath, &node)`
  → （如果 start_inode 是 SFS 目录，就会落到）`sfs_lookup(start_inode, subpath, &node)`
  → `sfs_lookup_once(sfs, dir_sin, name, &node, ...)`
  → `sfs_dirent_search_nolock(...)` 找到 `ino` （块号/ inode号）
  → `sfs_load_inode(..., ino)` 把磁盘 inode 读进内存，构造 VFS inode 返回
  → 回到 `vfs_open()`：`vop_open(node, flags)` （SFS 普通文件一般不做太多事）
  → 回到 `file_open()`：填好 `struct file`，返回 `fd`

下面逐个函数说明"它负责干什么"。

---

sysfile_open：系统调用入口，先把用户地址变成内核可用

- 输入：用户态给的 `__path` 指针（在用户地址空间里），以及 `open_flags`。
- 做的事：
  - `copy_path(&path, __path)`：把用户态字符串复制到内核里，避免内核直接读用户内存出问题。
  - `file_open(path, open_flags)`：真正开始"打开文件"的流程。
  - `kfree(path)`：释放内核里临时拷贝的字符串。
- 输出：成功返回 `fd`，失败返回错误码（负数）。

---

file_open：给"当前进程"分配一个 fd，并把它和 inode 绑定起来

这个函数做两件关键事：

- 第一件：在当前进程的"打开文件表"里找一个空位
  - `fd_array_alloc(NO_FD, &file)`：找到一个空闲 `struct file` 格子
  - 这个格子的下标就是未来返回给用户的 `fd`
- 第二件：让这个 file 指向真正的文件 inode

    - `vfs_open(path, open_flags, &node)`：让 VFS 去找到/创建对应文件，拿到 `inode *node`
    - 然后把 `file->node = node`，并填好读写权限、位置 `pos` 等
    - 最后 `fd_array_open(file)` 标记这个 `file` 状态为已打开
- 返回：`file->fd`

你可以把它理解成：
“我先给你一个号码 fd（进程私有），然后把这个号码关联到一个真正的文件 inode（全局对象）。”

---

vfs_open：VFS 的总控，负责“找 inode + 必要时创建 + 调用具体文件系统的 open”

它主要按这个顺序做：

1. 检查 flags 合不合法

- 例如：`O_TRUNC` 必须允许写，否则返回 `-E_INVAL`

2. `vfs_lookup(path, &node)`：尝试找到这个路径对应的 inode

- 找到了就得到 `node`

3. 如果没找到且 `O_CREAT`：创建

- `vfs_lookup_parent(path, &dir, &name)`：先找到“父目录 inode”和“最后一级名字”
- `vop_create(dir, name, excl, &node)`：让具体文件系统去创建文件并返回新 inode

4. 打开这个 inode

- `vop_open(node, open_flags)`：让具体文件系统处理 open（比如检查类型、权限等）

5. 需要截断就截断

- `vop_truncate(node, 0)`：把文件长度变成 0（当 `O_TRUNC` 或刚创建时）

6. `*node_store = node`：把 inode 交回上层

---

vfs_lookup：把“路径字符串”解析成“从哪个起点 inode 开始查 + 具体查哪个子路径”

它做两步：

1. `get_device(path, &subpath, &node)`

- 选择“查找起点 inode”，并得到真正要查的 `subpath`

2. 如果 `subpath` 不是空串：

- `vop_lookup(node, subpath, node_store)`
- 也就是：让“起点 inode 所属的文件系统”去完成路径查找

---

get_device：决定“从哪里开始找”（当前目录？根目录？某个设备的根？）

它根据 path 的形式分情况：

- 没有 `:` 且不是以 `/` 开头（例如 `"a/b"` 或 `"file"`）
  → 从当前目录开始：`vfs_get_curdir(node_store)`
  → `subpath = 原路径`

- 有 `device:path`（例如 `"disk0:/a/b"` 或 `"stdin:xxx"` 这种风格）
  → 取出 device 名字
  → `vfs_get_root(device, node_store)`：拿到这个设备对应文件系统的根 inode
  → `subpath` 指向 device 后面的那段
- 以 `/` 开头（例如 `"/sfs_filetest1"`）
  → 从 bootfs 根开始：`vfs_get_bootfs(node_store)`
  → `subpath` 是去掉前导 `/` 的剩余部分

最终效果：它告诉 vfs_lookup：
“你应该从哪个 inode 开始查，接下来要查的子路径是什么。”

---

vop_lookup：真正进入“具体文件系统”的查找逻辑

`vop_lookup` 不是一个真实函数，它是通过 inode 的 `in_ops->vop_lookup` 这个函数指针调用的。

- 如果这个 inode 属于 SFS 目录
  - `inode->in_ops = sfs_node_dirops`
  - 所以 `vop_lookup(...)` 实际会跳进 `sfs_lookup(...)`

---

sfs_lookup：在一个目录 inode 里，根据名字找到目标文件 inode（lab8 简化为单层）

它做的事很直白：

- 确认传入的 `node` 必须是目录
  - 不是目录就 `-E_NOTDIR`
- 调 `sfs_lookup_once(...)`
  - 在这个目录里找一次 `name`
- 找到后把结果 inode 放到 `*node_store`

注意：你这里的代码注释也说了，lab8 的 `sfs_lookup` 没做多级目录循环，所以它一般处理的是“目录里的一层名字”。

---

sfs_lookup_once：在某个目录里查一个名字，找到对应 inode 号，再加载 inode

它分两步：

1. `sfs_dirent_search_nolock(...)`

- 作用：在目录的数据里查找 `name` 对应的目录项
- 找到则返回 `ino`（在 SFS 里就是块号/ inode号）

2. `sfs_load_inode(sfs, node_store, ino)`

- 作用：根据 `ino` 去磁盘读出 `sfs_disk_inode`，再在内存里构造 `sfs_inode`，并封装成 VFS 的 `struct inode` 返回给上层

同时它用 `lock_sin / unlock_sin` 保护目录 inode，避免并发下目录内容被同时改动导致查找不一致。

把流程串成一句话

open 做的事就是：

- 先进入内核，把路径字符串安全地拷贝进来
- 给当前进程分配一个新的 fd 位置并且把它和inode（inode可以立即为一个真正的文件）绑定
- 让 VFS 根据路径，从正确的起点目录开始，（lab8 简化为一层）去目录里找名字
- 找到目录项后拿到 inode 号（块号），把磁盘上的 inode 读进内存，变成内核能用的 inode 对象
- 调用具体文件系统的 open（必要时创建/截断）
- 最后把"fd ↔ inode"的关系建立好，把 fd 返回给用户程序

具体一点：
open 做的事就是：

- 先进入内核，把用户态的路径字符串安全拷贝到内核态缓冲区
  （kern/fs/sysfile.c：sysfile_open() → copy_path()）
- 给当前进程分配一个空闲的 file 结构和对应的 fd（fd 就是进程打开文件表的下标），先把"fd 位置"占住
  （kern/fs/file.c：file_open() → fd_array_alloc()）
- 交给 VFS 去真正"打开/找到/必要时创建"这个路径对应的 inode
  （kern/fs/file.c：file_open() → vfs_open()）
  （kern/fs/vfs/vfsfile.c：vfs_open()）
- VFS 先根据路径格式决定从哪个目录 inode 开始查（当前目录 / bootfs 根 / 某个 device 的根），并得到要查的子路径
  （kern/fs/vfs/vfslookup.c：vfs_lookup() → get_device() → vfs_get_curdir()/vfs_get_bootfs()/vfs_get_root()）
- 让"起点目录 inode 所属的具体文件系统"来查找子路径对应的目标 inode
  （kern/fs/vfs/vfslookup.c：vfs_lookup() → vop_lookup() → inode->in_ops->vop_lookup）
- 如果起点 inode 属于 SFS 目录，就会进入 SFS 的查找实现：在该目录的数据里找名字，找到目录项后拿到 ino（SFS 里 ino 就是块号/ inode 号），再把磁盘 inode 读进内存并构造出对应的 VFS inode 返回
  （kern/fs/sfs/sfs_inode.c：sfs_lookup() → sfs_lookup_once() → sfs_dirent_search_nolock() → sfs_load_inode()）
- 回到 VFS：对拿到的 inode 执行"打开动作"，并维护 inode 的打开计数；如带 O_CREAT/O_TRUNC 则可能创建/截断文件
  （kern/fs/vfs/vfsfile.c：vfs_open() → vop_open() → vop_open_inc() → vop_truncate()）
- 回到 file_open：把 file->node 绑定到刚得到的 inode，设置 readable/writable、pos（如 O_APPEND 则先取 size 决定 pos），把 file 状态标为已打开，最终把 fd 返回给用户程序
  （kern/fs/file.c：file_open() → vop_fstat()(可选) → fd_array_open() → return file->fd）

read系统调用的执行过程

```
read(fd, data, len) 的完整执行链条（带文件路径 + 函数名 + 每步作用）

用户态 → 内核态 → VFS → SFS → 磁盘设备（disk0）→ 再一路返回，把数据放进 data。

------------------------------------------------------------------
1) 用户态发起 read
------------------------------------------------------------------
user/libs/file.c: read(int fd, void *base, size_t len)
```

```
作用：用户程序调用的库函数外壳，不做真正读，只是触发系统调用。
做的事：直接 return sys_read(fd, base, len)。

---------------------------------------------------------------
2) 进入系统调用通道，切到内核态
---------------------------------------------------------------
（系统调用分发，通常在 kern/syscall/syscall.c / kern/syscall.c 之类）
kern/...: sys_read(uint64_t arg[])
作用：把系统调用参数从寄存器/陷入帧里取出来，转交给文件系统的 sysfile 层。
做的事：
- fd = arg[0], base = arg[1], len = arg[2]
- return sysfile_read(fd, base, len)

---------------------------------------------------------------
3) 通用文件访问接口层：sysfile_read
---------------------------------------------------------------
kern/fs/sysfile.c: sysfile_read(int fd, void *base, size_t len)
作用：这是“内核里处理 read 系统调用”的入口，负责做安全检查、分配临时缓冲区、循环搬运数据到用户
态。
它做的事分成三块：

(1) 参数与权限检查
- len == 0 → 直接返回 0
- file_testfd(fd, readable=1, writable=0) → 检查这个 fd 是否有效、对应文件是否可读
  不可读/无效 → 返回 -E_INVAL

(2) 分配一个内核缓冲区 buffer（大小 IOBUF_SIZE，一般 4096）
- buffer = kmalloc(IOBUF_SIZE)
  分配失败 → 返回 -E_NO_MEM
为什么要有这个 buffer：
- 内核更容易控制“先从文件系统读到内核，再拷贝到用户”，避免直接让底层文件系统写用户地址带来的安全/
复杂性问题。

(3) 循环读取（一次最多 4096 字节）
while (len != 0):
- alen = min(IOBUF_SIZE, len)
- ret = file_read(fd, buffer, alen, &alen)
  这里 alen 会被改成“实际读到的字节数”
- 如果 alen > 0:
  - copy_to_user(mm, base, buffer, alen)
    把内核 buffer 里的内容拷贝到用户 data 指向的内存
  - base += alen, len -= alen, copied += alen
- 终止条件：
  - ret != 0：出错，跳出
  - alen == 0：读到文件末尾（EOF），跳出
最后：
- kfree(buffer)
- 如果 copied != 0：返回 copied（读到多少就返回多少）
- 否则返回 ret（比如直接出错）

---------------------------------------------------------------
4) 文件系统抽象层：file_read（真正“读这个 fd 代表的文件”）
---------------------------------------------------------------
kern/fs/file.c: file_read(int fd, void *base, size_t len, size_t *copied_store)
作用：把 “fd” 变成 “file 对象”，再用 file->node（VFS inode）调用 vop_read，让具体文件系统
去读。
```

```
关键步骤：

(1) fd → struct file*
- fd2file(fd, &file)
  作用：在当前进程的 fd_array 里找到这个 fd 对应的 file 结构
  找不到 → 返回错误码

(2) 检查可读
- if (!file->readable) return -E_INVAL

(3) 增加引用/并发保护
- fd_array_acquire(file)
  作用：防止读的过程中这个 file 被别的线程 close 掉或回收

(4) 构造 iobuf（描述“读到哪、从哪读、读多少”）
- iobuf_init(&iob, base, len, file->pos)
  iob 里包含：
  - io_base：目标缓冲区（这里是内核的 buffer）
  - io_resid：还剩多少字节要读
  - io_offset：文件偏移（file->pos）

(5) 调用 vop_read（关键：进入具体文件系统）
- ret = vop_read(file->node, iob)

(6) 更新文件当前位置 file->pos
- copied = iobuf_used(iob)   （本次实际读了多少）
- file->pos += copied
- *copied_store = copied

(7) 释放引用
- fd_array_release(file)
- return ret

到这一步为止，file_read 已经把数据读到了“它传入的 base”（即 sysfile_read 里的内核
buffer）。

----------------------------------------------------------------
5) VFS 层：vop_read 通过 inode_ops 分发到具体文件系统
----------------------------------------------------------------
宏展开逻辑（概念）：
kern/fs/vfs/inode.h：VOP_READ(inode, iob)
≈ inode->in_ops->vop_read(inode, iob)

如果这个 inode 是 SFS 普通文件：
- inode->in_ops 指向 SFS 的文件操作表
  kern/fs/sfs/sfs_inode.c：sfs_node_fileops
  其中 .vop_read = sfs_read

于是 vop_read 实际进入：

----------------------------------------------------------------
6) SFS 层：sfs_read → sfs_io → sfs_io_nolock
----------------------------------------------------------------
kern/fs/sfs/sfs_inode.c：sfs_read(struct inode *node, struct iobuf *iob)
作用：读文件的入口（SFS 实现），只是把读操作交给统一的 sfs_io。
做的事：return sfs_io(node, iob, write=0)
```

```
--------------------------------
kern/fs/sfs/sfs_inode.c：sfs_io(struct inode *node, struct iobuf *iob, bool write)
作用：做锁保护，然后调用无锁版本 sfs_io_nolock 执行真实 I/O。
做的事：
- sfs = fsop_info(vop_fs(node), sfs)        （取到这个文件系统实例）
- sin = vop_info(node, sfs_inode)           （取到 SFS 的内存 inode）
- lock_sin(sin)                             （给这个文件 inode 加锁，防止并发读写破坏一致性）
- alen = iob->io_resid
- ret = sfs_io_nolock(sfs, sin, iob->io_base, iob->io_offset, &alen, write)
- iobuf_skip(iob, alen)                     （更新 iobuf：已经完成了 alen 字节）
- unlock_sin(sin)
- return ret

--------------------------------
kern/fs/sfs/sfs_inode.c：sfs_io_nolock(...)
作用：把“文件偏移 offset + 长度”拆成若干块访问：
- 计算会涉及哪些 4KB block
- 每个 block 通过 bmap 找到对应的“磁盘块号”
- 调用 sfs_rbuf/sfs_rblock 读出数据
- 把数据写入 buf（这里 buf 就是 sysfile_read 里分配的内核 buffer）
它主要做的事：

(0) 边界与读文件的特殊规则
- offset < 0 / offset 太大 / endpos 溢出 → -E_INVAL
- 如果是读（write=0）：
  - offset >= din->size → 返回 0（已经在 EOF 之后了）
  - endpos > din->size → endpos 截断到 din->size（读不能超过文件大小）

(1) 选择读函数指针
- 读：sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock
- 写：对应 wbuf/wblock（这里是读，所以用 rbuf/rblock）

(2) 计算从哪个文件块开始
- blkno = offset / 4096
- blkoff = offset % 4096
- endpos = min(offset+len, din->size)

(3) 分三段读（这是你代码注释里提示的三段）
A. 起始不对齐部分（如果 offset 不是块边界）
- size = min(4096 - blkoff, endpos - offset)
- sfs_bmap_load_nolock(sfs, sin, blkno, &ino)  得到“第 blkno 个文件块”映射到的磁盘块号 ino
- sfs_rbuf(sfs, buf, size, ino, blkoff)       从该磁盘块的 blkoff 开始读 size 字节到 buf
- buf += size, offset += size, alen += size, blkno++

B. 中间整块部分（一次读多个完整 4KB 块）
- 对每个整块：
  - sfs_bmap_load_nolock(..., blkno, &ino)
  - sfs_rblock(sfs, buf, ino, 1) 或批量读多个块（实现可能会循环/分批）
  - buf += 4096, offset += 4096, alen += 4096, blkno++

C. 末尾不对齐部分（如果最后剩下不到 4KB）
```

```
- size = endpos - offset
- sfs_bmap_load_nolock(..., blkno, &ino)
- sfs_rbuf(sfs, buf, size, ino, 0)
- alen += size

(4) 返回实际读到的字节数
- *alenp = alen
- 读操作不会扩大文件大小，所以不会触发 “更新 din->size + dirty”
- return ret

----------------------------------------------------------------
7) SFS 的块映射：sfs_bmap_load_nolock（文件块号 → 磁盘块号）
----------------------------------------------------------------
kern/fs/sfs/sfs_inode.c：sfs_bmap_load_nolock(struct sfs_fs *sfs, struct
sfs_inode *sin, uint32_t index, uint32_t *ino_store)
作用：给定“文件内第 index 个数据块”，找出它对应的“磁盘块号 ino”。
关键点：
- index 是“文件逻辑块序号”（0,1,2,...），不是磁盘块号
- 返回的 ino 才是磁盘块号（在 SFS 里很多时候也被当作 inode/块号使用）

它做的事：
- assert(index <= din->blocks)
- create = (index == din->blocks)
  含义：
  - 如果 index < din->blocks：这是读已有块
  - 如果 index == din->blocks：这在写扩展文件时表示“需要新分配一个块”
  读文件时一般不会走 create=true（因为读被 endpos 截断到 size 以内）
- 调 sfs_bmap_get_nolock(sfs, sin, index, create, &ino)
  作用：真正实现 direct/indirect 查表或分配
- 如果 create：din->blocks++，并标记 inode dirty（写扩展才会）
- *ino_store = ino

----------------------------------------------------------------
8) 真正的磁盘 I/O：sfs_rbuf / sfs_rblock → disk0_io → ide_read_secs
----------------------------------------------------------------
整体链条（读）：
kern/fs/sfs/sfs_io.c（或相关文件）：sfs_rbuf / sfs_rblock
→ sfs_rwblock_nolock（统一读写块）
→ 通过设备层的操作函数（dop_io）
→ kern/fs/devs/dev_disk0.c：disk0_io(...)
→ disk0_read_blks_nolock(...)
→ ide_read_secs(...)

关键含义：
- SFS 不自己直接碰硬件，它只会说“我要读某个磁盘块”
- 设备层 disk0 把“块号”换算成“扇区号”，调用 ide_read_secs 完成真正读取

disk0_io 做的事（读）：
- 检查 offset/resid 必须块对齐（否则 -E_INVAL）
- 检查不能越界（blkno + nblks <= dev->d_blocks）
- 加锁 lock_disk0，防止并发读写把 disk0_buffer 搞乱
- 循环：
  - disk0_read_blks_nolock(blkno, nblks) 把磁盘内容读到 disk0_buffer
  - iobuf_move(iob, disk0_buffer, alen, direction=1, &copied)
    把 disk0_buffer 复制到上层提供的 buf（也就是 SFS 传下来的目标内存）
- 解锁 unlock_disk0
```

```
----------------------------------------------------------------
9) 数据如何回到用户 data?
----------------------------------------------------------------
到这里，数据已经被读到：
- sysfile_read 分配的内核 buffer

然后 sysfile_read 做：
- copy_to_user(mm, data, buffer, alen)
把这段内容拷贝进用户态 data 指针指向的内存。

最后 sysfile_read 返回 copied（实际读到的字节数）
用户态 read(...) 收到返回值，并且 data 里已经有内容。

----------------------------------------------------------------
一句话把 read 的流程串起来（对应上面链条）
----------------------------------------------------------------
user/libs/file.c:read → sys_read → kern/fs/sysfile.c:sysfile_read（检查 + 分配内核 buffer + 循环）→
kern/fs/file.c:file_read（fd→file→iobuf→vop_read）→
kern/fs/sfs/sfs_inode.c:sfs_read/sfs_io/sfs_io_nolock（按偏移拆块 + bmap 找块号）→
kern/fs/sfs/sfs_inode.c:sfs_bmap_load_nolock（逻辑块→磁盘块）→
kern/fs/devs/dev_disk0.c:disk0_io → disk0_read_blks_nolock → ide_read_secs（真读磁盘）→
回到 sysfile_read 用 copy_to_user 把数据放进用户 data，返回实际字节数。
```

用户程序加载（key）

```
do_execve + load_icode：用户程序是怎么从“文件”里被加载起来并跑起来的（lab8）

这段代码想解决的问题很简单：用户在终端里输入一个程序名（比如 ls），系统要去文件系统里找到这个程序文件，把它读出来，放进这个进程的用户态内存，然后让 CPU 从用户态跳到这个程序的入口开始执行。

================================================================
一、do_execve 做了什么（kern/process/proc.c：do_execve）
================================================================

do_execve(name, argc, argv) 可以理解为“把当前进程换成另一个程序来运行”。

它做的事按顺序是：

1) 参数基本合法性检查
- 位置：kern/process/proc.c：do_execve
- 检查 argc 是否在允许范围内（至少 1，因为 argv[0] 必须是程序路径）

2) 把用户传进来的 name / argv 安全地拷到内核里
- 位置：kern/process/proc.c：do_execve
- lock_mm(mm) 后：
  - copy_string(mm, local_name, name, ...)：把进程名复制到内核缓冲 local_name
  - copy_kargv(mm, argc, kargv, argv)：把用户态 argv 指针数组复制到内核态 kargv
- 目的：内核后面要用这些字符串/参数，不能直接一直依赖用户态地址（可能不安全、也可能马上就要清空用户地址空间）

3) 关闭当前进程打开的所有文件
```

```
- 位置：kern/process/proc.c：do_execve
- files_closeall(current->filesp)
- 目的：exec 之后“换了程序”，通常不希望旧程序留下的一堆 fd 继续占资源（这里按实验实现选择全部关闭）

4) 用文件系统打开 argv[0] 指定的可执行文件
- 位置：kern/process/proc.c：do_execve
- path = argv[0]
- fd = sysfile_open(path, O_RDONLY)
- 目的：拿到一个文件描述符 fd，后面用它去读 ELF 文件内容
- 注意注释说“必须用用户态指针的 path”：因为 sysfile_open 内部会做路径检查/拷贝，要求传入的指针形式符合它的处理逻辑（实验实现细节）

5) 清空/回收旧的用户态内存（如果当前进程原来就有用户空间）
- 位置：kern/process/proc.c：do_execve
- if (mm != NULL)：
  - lcr3(boot_cr3)：把页表切回内核页表（先保证下面释放内存时访问地址安全）
  - mm_count_dec(mm) == 0 时：
    - exit_mmap(mm)：按 mm 里记录的 vma，把用户空间映射关系撤销、释放物理页
    - put_pgdir(mm)：释放页表页
    - mm_destroy(mm)：释放 mm 结构
  - current->mm = NULL
- 目的：旧程序的“用户虚拟内存空间”必须清掉，否则新程序没法按自己的布局建立映射
- 文中提到 initproc 是内核线程 mm 可能为 NULL：那表示它之前没有用户态地址空间，就不用释放

6) 把新程序真正装进内存并建立新用户环境
- 位置：kern/process/proc.c：do_execve
- ret = load_icode(fd, argc, kargv)
- 目的：这是最关键的一步：建立新的用户地址空间，把 ELF 里的段放进去，准备用户栈，把入口地址写进 trapframe

7) 清理内核态的参数拷贝，设置进程名，返回成功
- put_kargv(argc, kargv)
- set_proc_name(current, local_name)
- return 0

如果中途任何一步失败：
- execve_exit 分支：put_kargv，然后 do_exit(ret) 直接结束当前进程（实验实现选择）

================================================================
二、load_icode 做了什么（建立“能跑的用户环境”）
================================================================

load_icode(fd, argc, kargv) 的目标是：让当前进程具备一个完整的用户态运行现场：
- 有自己的 mm（内存管理结构）
- 有自己的页表（pgdir）
- 用户地址空间里有程序的代码段/数据段/BSS
- 有用户栈（stack）
- trapframe 配好：将来从内核返回用户态时，CPU 知道从哪条指令开始执行、用哪个用户栈

文中列的 6 步可以按“做什么/为什么”理解：

------------------------------------------------------------
1) 创建 mm 和页表（pgdir），并让它能映射内核空间
------------------------------------------------------------
- mm_create：创建并初始化 mm_struct
```

```
- setup_pgdir：分配一页作为新的页目录，并把 boot_pgdir（内核映射）拷贝进去
- mm->pgdir 指向新的页表
为什么要拷贝内核映射：
- 用户进程运行时也需要能进入内核（系统调用/中断），内核空间映射必须一直可用

------------------------------------------------------------
2) 读取并解析 ELF 头和 Program Header，建立 vma
------------------------------------------------------------
- 通过 fd 读取 ELF 文件结构（不是靠“内存里已有一整块 ELF”）
- 对 ELF 的每个段（代码/数据/BSS）：
  - mm_map：在 mm 里创建 vma，记录这段在用户虚拟地址空间的范围、权限等
为什么要先建 vma：
- vma 是“这段地址是合法的吗、权限是什么”的依据，后面缺页/访问检查都用它

------------------------------------------------------------
3) 给每个段分配物理页、建页表映射、把段内容拷贝进去
------------------------------------------------------------
- 按段的大小，逐页分配物理内存
- 在页表里建立：用户虚拟地址 → 物理页
- 把 ELF 段里的内容复制到对应内存位置
- BSS 段通常需要清零（ELF 里不一定有实际数据）
结果：
- 程序的代码和数据已经出现在“用户虚拟地址空间”里了

------------------------------------------------------------
4) 建用户栈（顶端 1MB）
------------------------------------------------------------
- mm_mmap：给栈区域建立 vma（用户空间高地址处）
- 分配一定数量物理页并映射
为什么必须有栈：
- 用户程序要用栈保存返回地址、局部变量、函数调用参数等
- 还需要在栈上放 argc/argv 这些启动参数

------------------------------------------------------------
5) 切换到新页表（写 cr3），并把 argc/argv 放到用户栈中
------------------------------------------------------------
- 把 mm->pgdir 装入 cr3：CPU 以后做地址翻译就用新的页表了
- 在用户栈里布置 uargc / uargv（用户态可见的参数）
注意：
- 此时进程的“内容”已经变了，但 CPU 还没真正跳到用户态入口执行

------------------------------------------------------------
6) 设置 trapframe，让 iret/eret 后进入用户态入口
------------------------------------------------------------
- 清空 trapframe
- 设置：
  - 用户态代码段/数据段选择子（或等价字段）
  - 用户栈指针 esp/rsp（或等价字段）
  - 指令指针 eip/rip（入口地址 entry）
  - 用户态标志位（允许中断等）
结果：
- 下一次从内核“返回到用户态”时，CPU 会从 entry 开始执行新程序

================================================================
三、lab8 用文件描述符读 ELF：load_icode_read 的作用
================================================================
```

```
lab8 不再把 ELF 当作“已经在内存里的一段二进制”，而是当作“文件系统里的一个普通文件”。
因此需要一个小工具函数：从 fd 指向的文件里，按偏移读取指定长度到 buf。

代码：kern/process/proc.c（或相邻文件）：load_icode_read(int fd, void *buf, size_t
len, off_t offset)

它做的事只有两步：

1) 定位到文件 offset
- sysfile_seek(fd, offset, LSEEK_SET)
- 作用：把这个文件的“读指针”移动到指定位置

2) 读取 len 字节到 buf
- ret = sysfile_read(fd, buf, len)
- 要求：必须读满 len
  - 如果 ret < 0：读失败，返回错误码
  - 如果 ret != len：说明文件不够长或中途异常，返回 -1
- 成功返回 0

这就是“ELF 解析器想读哪一段，就从文件里读哪一段”。

================================================================
四、lab5 vs lab8 的对比（改了什么、为什么改）
================================================================

------------------------------------------------------------
lab5 的做法（内存方式）
------------------------------------------------------------
do_execve 的原型更像：
do_execve(name, len, binary_ptr, size)
含义：
- 可执行文件的全部内容（binary）已经在内存里
- load_icode 解析 ELF 时直接在内存里按指针读数据，不需要 seek/read

优点：
- 实现简单，少一层文件系统依赖

缺点：
- 程序来源固定：必须事先把程序镜像放到内存某个位置
- 不像真正操作系统：用户无法“在文件系统里放一个新程序然后运行”

------------------------------------------------------------
lab8 的做法（文件方式）
------------------------------------------------------------
do_execve 改为：
do_execve(name, argc, argv)
关键变化：
- argv[0] 是路径
- do_execve 先 sysfile_open(path) 得到 fd
- load_icode 通过 load_icode_read(fd, ..., offset) 读取 ELF

为什么必须这样改：
1) 因为 lab8 引入了文件系统（SFS）和“终端 shell”
- 用户输入的是“程序名/路径”
- 程序文件存放在 sfs.img 里
```

```
- 内核必须通过 VFS/SFS 把文件读出来

2) 这更接近真实 OS 的 exec
- 真实系统里 execve 打开的就是文件系统里的可执行文件
- 解析 ELF 需要随机读取（读头、读 program header、读各段），所以需要 seek + read

3) 让 fork/exec 组合成为可能
- shell fork 出子进程
- 子进程 exec 读取文件系统里的程序并替换自身
- 父进程继续做终端交互

总结成一句话：
lab5 是“程序已经在内存里，exec 直接从内存解析 ELF”；lab8 是“程序在文件系统里，exec 先 open 得到 fd，再用 seek+read 按需读取 ELF”，这样终端才能按名字加载并运行任意文件系统中的程序。
```

终端执行

user/sh.c
终端程序需要对命令进行词法和语法分析。

Challenge 1

pipe（管道）机制是什么

pipe 是一种 **进程间通信（IPC）** 手段：内核在内存里维护一段“字节队列缓冲区”，进程把数据写入 pipe 的写端，另一个进程从读端把字节按顺序读出来。pipe 的核心特性是：

- **单向**：一条 pipe 默认是“写端 → 读端”（需要双向通信就建两条）。
- **字节流**：不保留“消息边界”，读出来就是连续字节。
- **有容量限制**：缓冲区满了，写端通常会阻塞；缓冲区空了，读端通常会阻塞。
- **关闭语义很重要**：
  - 所有写端都关闭后，读端再读会读到 **EOF**（read 返回 0）。
  - 所有读端都关闭后，写端再写会失败（POSIX 里是触发 SIGPIPE/返回 EPIPE）。
- **并发规则**：POSIX 要求对 pipe 的写入在不超过 PIPE_BUF 时具备原子性（不会和别人的小写入交错）

在 ucore 里要加入 pipe，至少需要哪些数据结构

目标：让 pipe 在 VFS 里看起来“像文件一样”，能被 `read(fd,...) / write(fd,...) / close(fd)` 使用；同时要处理并发读写、阻塞唤醒。

下面给一个**够用的最小设计**（偏 ucore 风格：semaphore + wait_queue + schedule）。

1. 内核里的 pipe 对象（缓冲区 + 同步信息）

```
// 一个 pipe 的内核对象：被读端/写端共同引用
#define PIPE_BUF_SIZE 4096    // 可以先做成 4KB 或 16KB，后续可扩展

typedef struct pipe_info {
    // 环形缓冲区（字节队列）
    uint8_t buf[PIPE_BUF_SIZE];
```

```
    uint32_t rpos;          // 读指针（消费位置）
    uint32_t wpos;          // 写指针（生产位置）
    uint32_t used;          // 当前缓冲区已用字节数（用它更直观）

    // 引用计数（决定 EOF / EPIPE 语义）
    uint32_t readers;       // 当前还打开着的读端数量
    uint32_t writers;       // 当前还打开着的写端数量

    // 同步互斥
    semaphore_t sem;        // 互斥保护：rpos/wpos/used/readers/writers
    wait_queue_t r_wait;    // 读等待队列：缓冲区空时，读者睡这里
    wait_queue_t w_wait;    // 写等待队列：缓冲区满时，写者睡这里

    // 标志位（可选：支持 O_NONBLOCK 等）
    bool nonblock_read;
    bool nonblock_write;
} pipe_info_t;
```

为什么要这些字段

- `buf/rpos/wpos/used`：实现“先写先读”的字节队列。
- `readers/writers`：实现“写端全关→读到 EOF”，“读端全关→写失败”的行为。(man7.org)
- `sem + 两个 wait_queue`：处理并发与阻塞：
    - 读：缓冲区空 → 把当前进程挂到 `r_wait`，`schedule()` 睡眠；写端写入后唤醒。
    - 写：缓冲区满 → 挂到 `w_wait`，睡眠；读端读走后唤醒。
- Linux 里也有类似的“pipe 对象”，包含互斥锁、读写等待队列、head/tail、readers/writers 等字段。(anquanke.com)

1. 每个 fd 对应的“端点对象”（区分读端/写端）

在 ucore 里 fd 对应 `struct file`，你需要把它和 `pipe_info` 关联起来，并且区分这个 fd 是读端还是写端：

```
typedef enum { PIPE_END_READ = 1, PIPE_END_WRITE = 2 } pipe_end_t;

typedef struct pipe_endpoint {
    pipe_info_t *pipe;
    pipe_end_t end;         // 读端 or 写端
    uint32_t flags;         // O_NONBLOCK 等
} pipe_endpoint_t;
```

然后把 `pipe_endpoint_t*` 放到某个位置：

- 要么扩展 `struct file` 加 `void *private_data;`
- 要么在 `inode` 的 union 里加一个 pipe 类型（更接近 VFS 思路）

在 ucore 里要定义哪些接口（语义即可）

最少要两层：系统调用层 + VFS/inode 操作层。

系统调用层（用户态能看到的）

- `int sys_pipe(int pipefd[2]);`
  语义：创建一条管道，返回两个 fd：`pipefd[0]` 读端，`pipefd[1]` 写端（和 UNIX/Linux 一样）。(man7.org)
- 可选：`int sys_pipe2(int pipefd[2], int flags);`
  语义：同上，但能传 `O_NONBLOCK`、`O_CLOEXEC` 等（先不实现也可以）。

VFS/inode 操作层（让它"像文件一样"）

给 pipe 做一个 `inode_ops`（或 device_ops）实现这些回调：

```
struct inode_ops pipe_iops = {
    .vop_open   = pipe_open,      // 可选：主要做权限/标志检查
    .vop_close  = pipe_close,     // 关键：减少 readers/writers，必要时唤醒对端
    .vop_read   = pipe_read,      // 关键：从环形缓冲区读；空则阻塞/返回 E_AGAIN
    .vop_write  = pipe_write,     // 关键：写入环形缓冲区；满则阻塞/返回 E_AGAIN
    .vop_fstat  = pipe_fstat,     // 可选：类型、大小等
    .vop_tryseek = pipe_tryseek // 一般 pipe 不支持 seek，返回 -E_INVAL
};
```

读写语义（必须讲清楚的行为）

- `pipe_read(fd=读端)`
  - 若 `used > 0`：拷贝 min(len, used) 字节给用户，推进 `rpos/used`，并唤醒 `w_wait`。
  - 若 `used == 0` 且 `writers > 0`：
    - 阻塞模式：睡在 `r_wait`；被唤醒后重试
    - 非阻塞：返回 `-E_AGAIN`
  - 若 `used == 0` 且 `writers == 0`：返回 **0（EOF）**。
- `pipe_write(fd=写端)`
  - 若 `readers == 0`：写失败（POSIX 语义是 SIGPIPE/EPIPE；ucore 没信号也至少要返回错误码如 `-E_PIPE`）.
  - 若缓冲区未满：写入尽可能多的字节，推进 `wpos/used`，并唤醒 `r_wait`。
  - 若缓冲区满且 `readers > 0`：
    - 阻塞模式：睡在 `w_wait`；被唤醒后重试
    - 非阻塞：返回 `-E_AGAIN`

原子性（建议写进设计方案）

- 至少满足：**小于等于 PIPE_BUF 的一次 write 不和别人的小写入交错**（简单做法：对写入路径用同一把互斥锁保护，并且把一次 write 当作一个整体写入/要么全写入要么阻塞等待足够空间）。POSIX 对 PIPE_BUF 有原子性要求。(pubs.opengroup.org)

同步互斥问题：你的设计里要体现什么

必须把下面几类竞态讲清楚，否则 pipe 很容易"卡死"或"丢唤醒"：

- 多个读者/写者并发更新 `rpos/wpos/used`：需要互斥（`semaphore_t sem`）。
- 阻塞与唤醒：
  - 读者发现空 → **在持锁状态下**把自己加入 `r_wait`，再释放锁并 `schedule()`；

    - 写者写入后 **在持锁状态下**检查队列并唤醒，避免“刚睡下就错过唤醒”的窗口。
- close 触发的唤醒：
    - 写端最后一个关闭时，要唤醒所有在 `r_wait` 上睡眠的读者，让它们读到 EOF。(man7.org)
    - 读端最后一个关闭时，要唤醒 `w_wait` 上的写者，让它们立刻返回错误（否则会永远等空间）。(pubs.opengroup.org)

一句话总结你在报告里可以写的“概要方案”

在 ucore 的 VFS 中新增一种 pipe inode/设备类型；`sys_pipe` 创建一个 `pipe_info`（环形缓冲区 + readers/writers + 信号量 + 读写等待队列），再创建两个 `file`/fd：一个只读端一个只写端；读写通过 `vop_read/vop_write` 进入 `pipe_read/pipe_write`，在缓冲区空/满时用等待队列阻塞并由对端唤醒，close 时用 readers/writers 计数实现 EOF/EPIPE 语义并唤醒对端，必要时按 PIPE_BUF 保证小写入原子性。