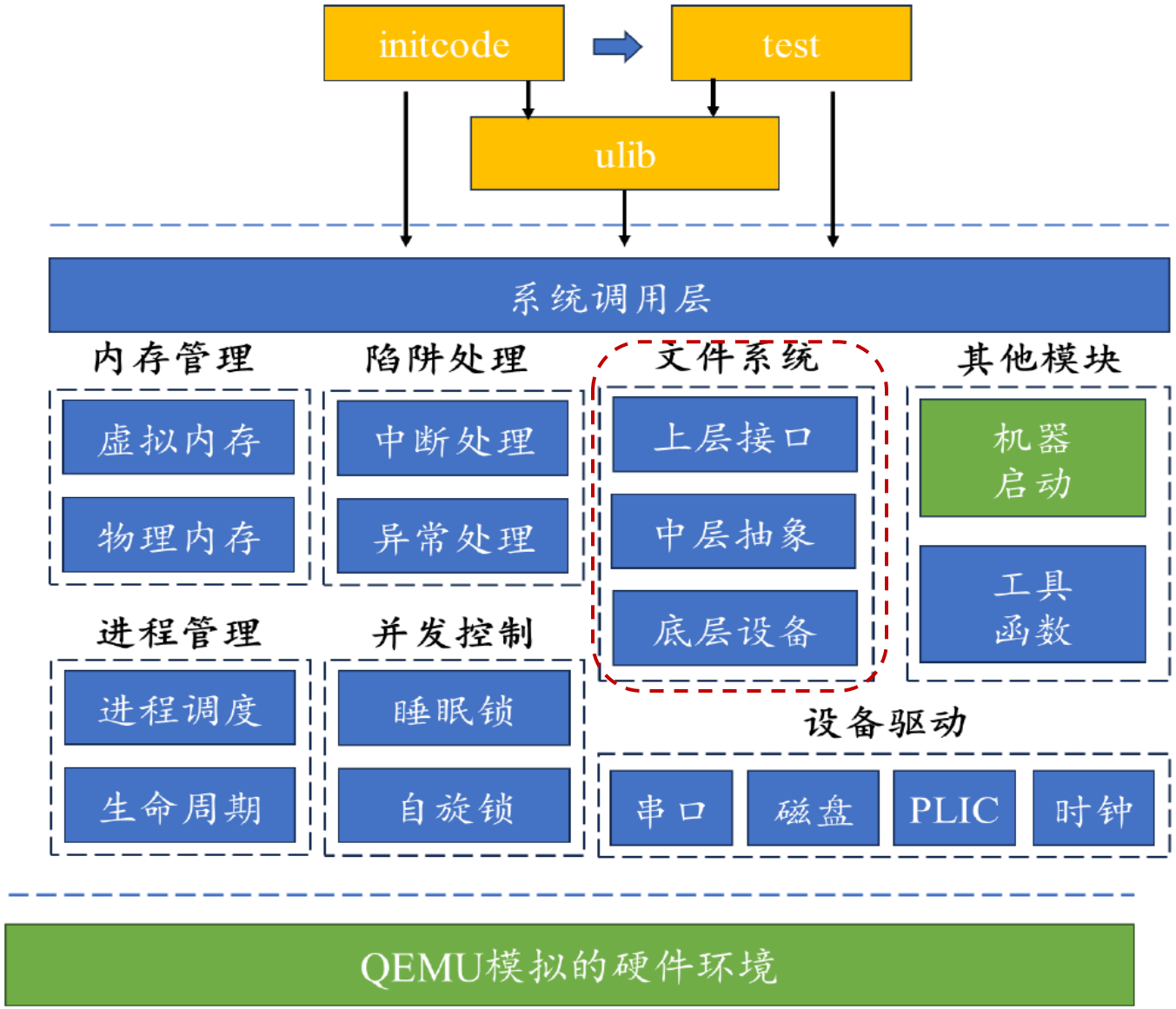
实验七：文件系统



- RISC-V体系结构规定了三种特权模式：用户模式（User mode，U-mode）、监管者模式（Supervisor mode，S-mode）、机器模式（Machine mode，M-mode）。

实验七：文件系统

实验目标：理解现代文件系统的核心概念和实现原理，实现一个简单的文件系统。

1. 理解文件系统的磁盘布局

- 学习如何将磁盘划分为：引导块、超级块、inode位图、数据位图、inode区、数据区
- 掌握磁盘块分配和回收的机制
- 理解元数据（如超级块）的作用

2. 掌握文件系统的基本抽象

- 文件：作为字节序列的抽象
- 目录：作为文件名到inode编号映射的特殊文件
- inode：理解其作为文件元数据核心载体的作用（权限、大小、数据块指针等）

实验七：文件系统

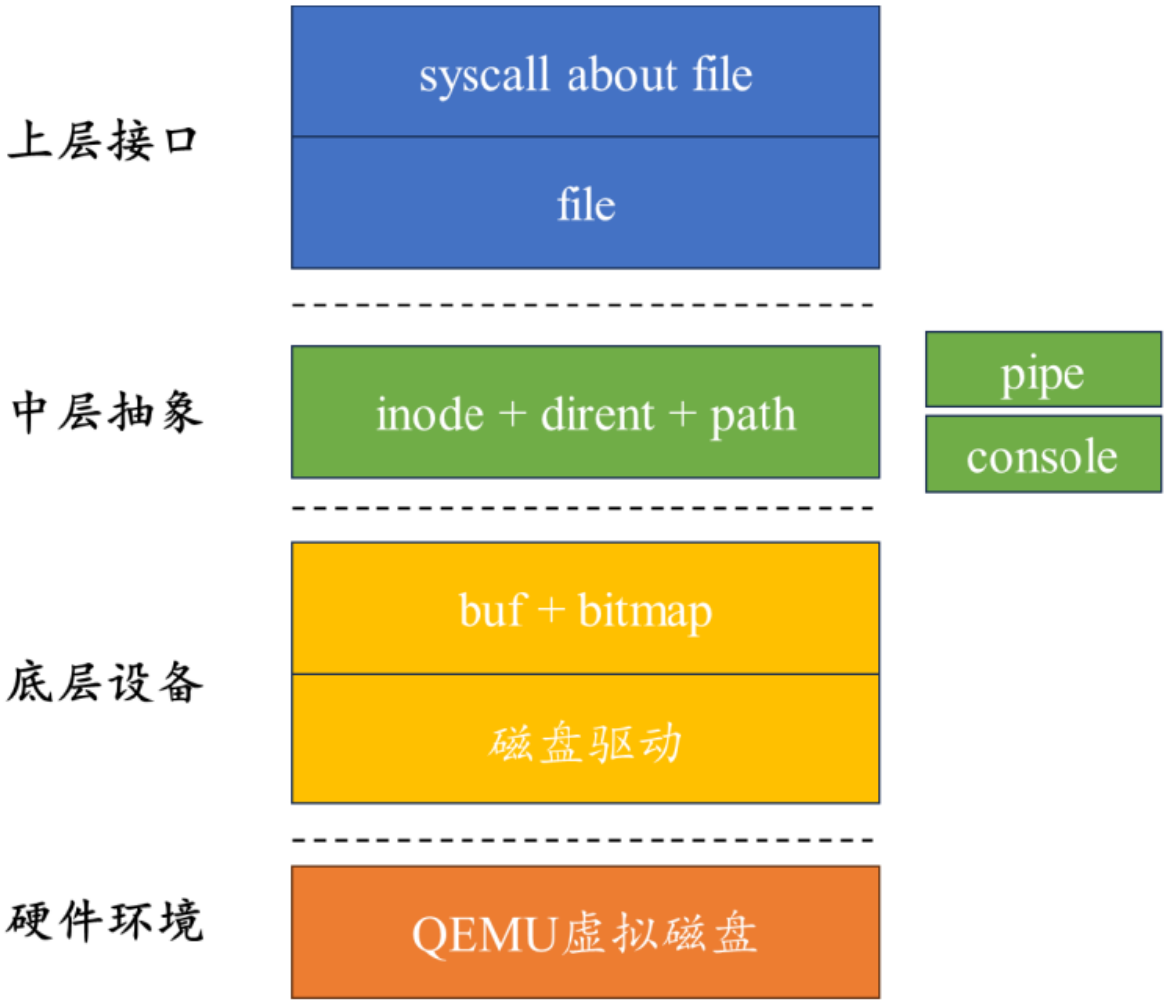
3. 实现关键系统调用

- 文件操作：open, read, write, close
- 目录操作：mkdir, link, unlink
- 文件描述符管理

4. 理解路径解析机制

- 实现从路径名到inode的查找过程
- 处理绝对路径和相对路径
- 理解当前工作目录的概念

实验七：文件系统-底层设备



实验七：文件系统

- 从本次实验开始，将引入外存的概念。 QEMU为操作系统内核提供了虚拟磁盘，通过读写特定寄存器，磁盘驱动程序可以为上层提供两个重要的函数接口：
  - （1）以block为单位将磁盘里的一个区域读取到内存或将内存的一个区域写入磁盘，这个接口提供给文件系统。
  - （2）以磁盘中断的方式将读写操作传递至虚拟磁盘，这个接口提供给中断异常模块。

实验五：文件系统

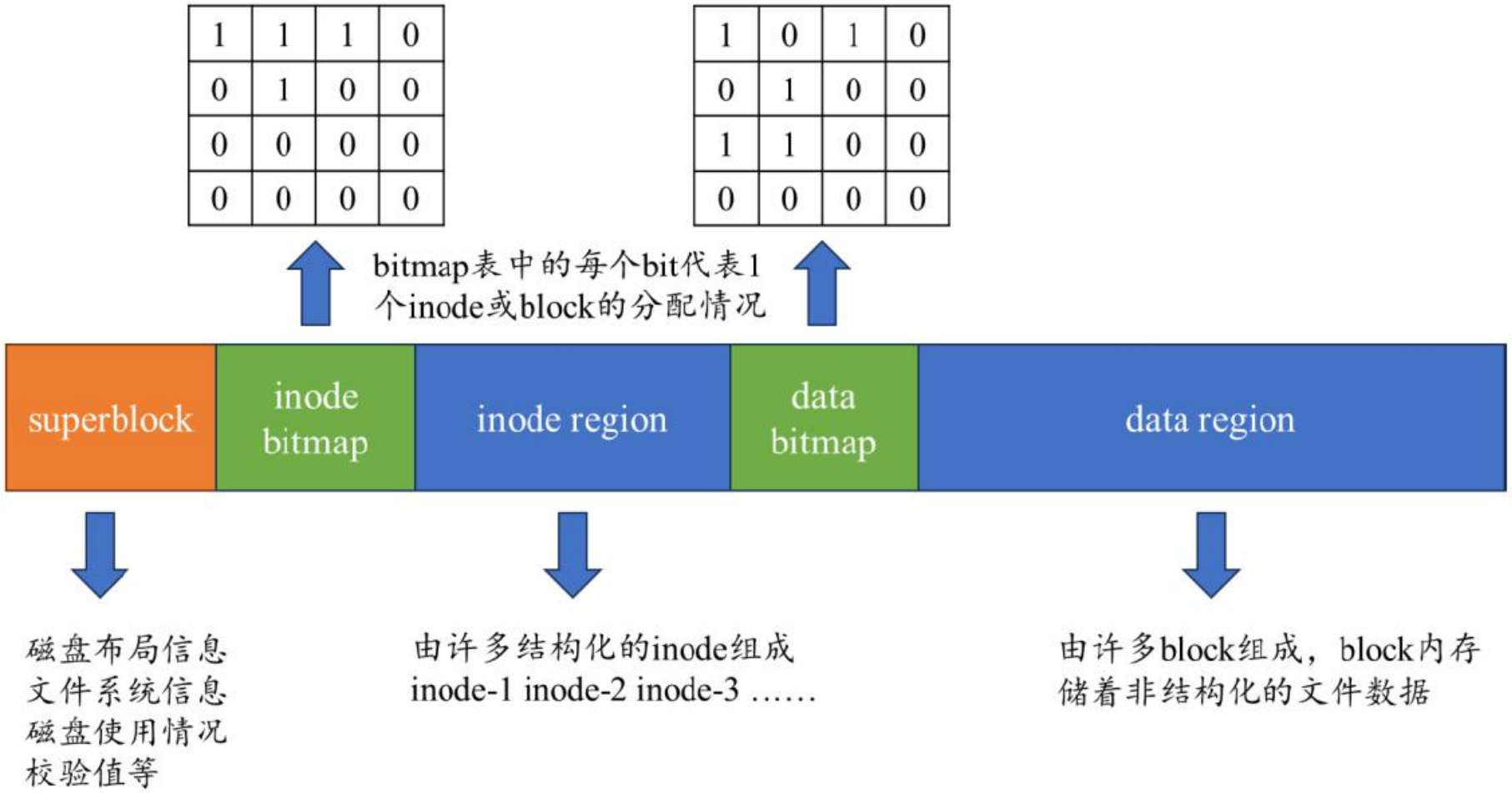
首先要在 根目录下makefile文件***QEMUOPTS*** 中添加文件系统映像和磁盘块设备的选项。

```
QEMUOPTS += -drive file=$(FS_IMG),if=none,format=raw,id=x0

QEMUOPTS += -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
```

它会把一个名为 ***$(FS_IMG)*** 的文件作为操作系统的文件系统映像装入QEMU模拟的虚拟磁盘。

实验七：文件系统-底层设备



实验七：文件系统-底层设备

任务一：理解磁盘映像的制作过程

磁盘在文件系统角度可以理解为一个以 ***block*** 为读写单位的大数组

```
disk layout: [ super block | inode bitmap | inode blocks | data bitmap | data blocks ]
```

- ***super block*** 包括磁盘和磁盘上文件系统的重要元数据
- ***inode bitmap*** 标记 inode blocks 区域里各个 inode 的分配情况 (0:可分配 1:已分配)
- ***inode blocks*** 这个区域由若干连续的 inode 组成
- ***data bitmap*** 标记 data blocks 区域里各个 block 的分配情况 (0:可分配 1:已分配)
- ***data blocks*** 这个区域由若干连续的 block 组成

- kfs.c 会填写 super block 中的信息, 其他四个区域会全部填写为 0

这个非常简单的磁盘映像就制作完毕了

实验七：文件系统-底层设备

任务二：磁盘驱动

在 QEMU 为我们提供了一套读写虚拟磁盘的方法，磁盘驱动的实现涉及三个部分:

- ***virtio.c*** 这个文件定义了磁盘的数据结构, 由于非常复杂且涉及硬件规范, 已经给出完整代码，只需要了解它为上层提供的接口即可。
- ***kvm.c*** 和 PLIC、CLINT、UART一样, 由于使用了内存映射寄存器, 需要在内核页表中加入virtio的映射。
- ***trap_kernel.c*** 和时钟中断、UART中断一样, virtio 也需要在中断发生时调用中断响应函数。

实验七：文件系统-底层设备

需要修改的文件：

memlayout.h中增加定义：

```
// virtio 相关
#define VIRTIO_BASE  0x10001000ul
#define VIRTIO_IRQ   1
```

common.h中增加定义：

```
#define BLOCK_SIZE 1024      // 磁盘的block大小
```

实验七：文件系统-底层设备

- **缓冲区（buffer）**

读写磁盘的过程：

- 第一步应该是主机向磁盘发送一个读请求，这个请求应该包括两个信息：要读取的block序号 + 读到的内容放到内存中何处。

- 第二步应该是磁盘响应请求并完成数据传递任务。

- 第三步应该是主机查看内存中读到的数据，可能做一些修改。

- 如果做了修改，第四步应该要写回磁盘。

实验七：文件系统-底层设备

缓冲区（buffer）（buf.h）

- 从内存的角度考虑，需要在内存里开辟一段空间(BLOCK_SIZE大小)用于暂存磁盘里读入的block，同时还需要记录这块空间对应哪个磁盘里的block。
- 确保多个进程同时只能一个进程进行操作。

```
typedef struct buf {
    /*
        自旋锁: 保护 data[BLOCK_SIZE] + disk
        block_num + buf_ref 由 lk_buf_cache保护
    */
    spinlock_t slk;

    uint32 block_num; // 对应的磁盘block编号
    uint8  data[BLOCK_SIZE]; // block数据的缓存

    uint32 buf_ref; // 还有多少处引用没有释放
    bool disk;      // 在磁盘驱动中使用
} buf_t;
```

实验七：文件系统-底层设备

缓冲区（buffer）(buf.c)

buf的组织结构采用双向循环链表, 以 **head_buf** 为链表的头节点, **buf_cache** 里的节点作为可分配回收的资源节点。

```
typedef struct buf_node {
    buf_t buf;
    struct buf_node* next;
    struct buf_node* prev;
} buf_node_t;

// buf cache
static buf_node_t buf_cache[N_BLOCK_BUF];
static buf_node_t head_buf; // ->next 已分配 ->prev 可分配
static spinlock_t lk_buf_cache; // 这个锁负责保护 链式结构 + buf_ref + block_num
```

实验七：文件系统-底层设备

- **双向循环链表**将buffer的申请和释放转换成链表的插入和删除操作
- 采用LRU策略和懒惰写回策略优化buffer与磁盘的协作：
  - 将最近经常访问的数据留在内存，而不是立刻更新至磁盘；
  - 在buffer不足时将长时间不访问的数据写回磁盘。

实验七：文件系统-底层设备

- **super block 的读取及文件系统初始化 (fs.c)**

这部分的工作分为两个步骤：

- 使用刚刚完成的 buf 层函数, 在 **fs.c** 中读取磁盘里的 super block 并填写到内存里的 **super_block_t** 数据结构中。
- 由于 buf 层的操作涉及锁, 所以文件系统的初始化建立在进程的基础上, 第一个进程的 fork_return() 是一个很好的时机。

实验七：文件系统-底层设备

- **bitmap 的管理（bitmap.c）**

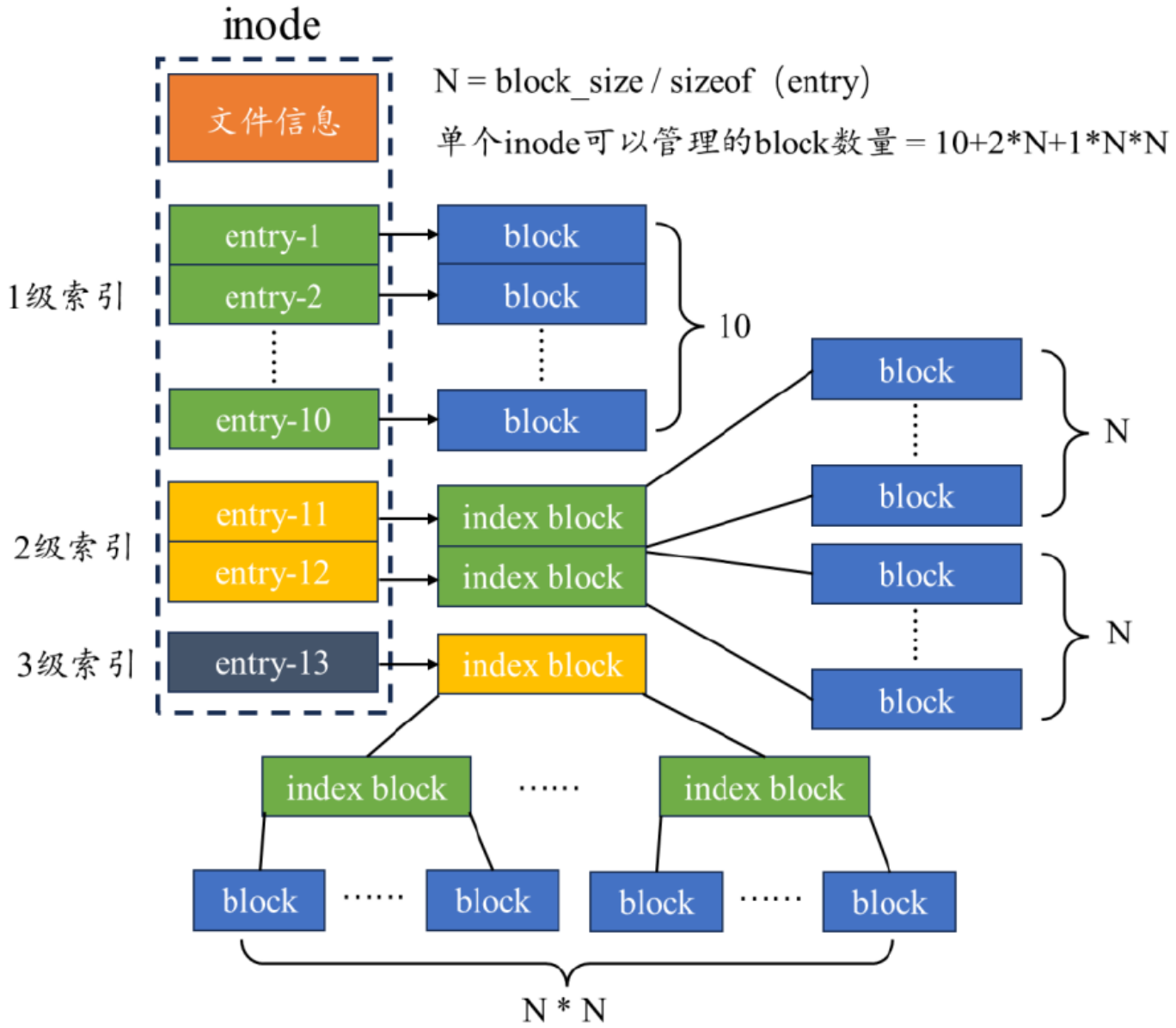
bitmap 是一种非常经典的文件系统管理方法, 它使用一块区域的每一个 bit 标记某种资源是否被占用.

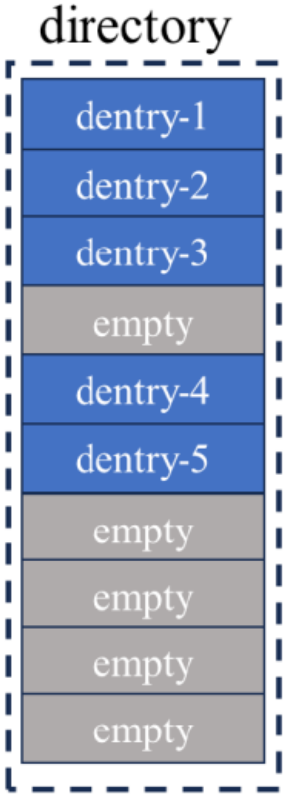
- 我们用到了两块 bitmap (出于简化考虑, 它们各占1个block)
- 当我们申请一个 data block 或 inode 时, 对应 bitmap 的一个 bit 被置为 1
- 当我们释放一个 data block 或 inode 时, 对应 bitmap 的对应 bit 被置为 0

实验七：文件系统-中层抽象

- 中层抽象主要依赖索引节点（index node，inode）、目录项（directory entry，dentry）、文件路径（path）三个重要元素构建。
- inode主要存储了两类信息：
  - 一类是文件信息（如文件名、时间戳、读写权限等），
  - 一类是索引信息（用于记录文件内容存储在哪些block中）。文件索引采用10+2*N+N*N的组织格式，可以很好适应小型文件（10个block足够存储）、中型文件（10+2*N个block足够存储）和大型文件（10+2*N+N*N个block足够存储）。
- 目录是一种特殊的结构化文件，一个目录包含多个目录项，目录项用于建立文件名到inode序号的映射。

实验七：文件系统-中层抽象





目录项基本操作：

- add_entry
- delete_entry
- search_entry
- get_entries

实验七：文件系统-中层抽象

- 多个目录和文件构成一颗目录树。路径解析的过程包括两件事：
  - 路径的切割（字符串处理和信息提取）和目录访问，目标是拿到路径对应的inode。
  - 读到inode后可以通过其中存储的索引信息查询文件放在哪些block，最终完成数据读取。

实验七：文件系统-中层抽象

1. 理解 mkfs.c 做了哪些事情, 和上次相比, 磁盘发生了什么变化。

2. 完成 inode.c 里面的函数, 理解 “磁盘里的inode与内存里的inode”、 "inode元数据与数据"

3. 完成 dir.c 里的第一部分, 理解 "目录" 这种结构化文件的构成和管理

4. 完成 dir.c 里的第二部分, 理解文件路径是如何确定 inode 的

实验七：文件系统-中层抽象

```
typedef struct inode {
    // 磁盘里的inode信息 (由slk保护)
    uint16 type;              // inode 管理的文件类型
    uint16 major;             // 设备文件使用: 主设备号
    uint16 minor;             // 设备文件使用: 次设备号
    uint16 nlink;             // 链接数量 (nlink个文件名链接到这个inode)
    uint32 size;              // 文件大小 (字节)
    uint32 addrs[N_ADDRS];    // 文件存储在哪些block里 (分为一级 二级 三级)


    // 内存里的inode信息
    uint16 inode_num;         // inode序号
    uint32 ref;               // 引用数 (由lk_icache保护)
    bool valid;               // 上述磁盘里inode字段的有效性 (由slk保护)
    sleeplock_t slk;          // 睡眠锁
} inode_t;
```

实验七：文件系统-中层抽象

- inode的管理

// 内存中的inode资源 + 保护它的锁

#define N_INODE 32

static inode_t icache[N_INODE];

static spinlock_t lk_icache;

```

声明一个全局的资源仓库,采用数组的形式组织,并向外部提供申请和释放资源的接口

inode_init()函数负责仓库的初始化
```

实验七：文件系统-中层抽象

- inode_init()：
- inode_rw ()：函数实现让磁盘和内存同步的工作。
- inode_alloc()：确定磁盘里有inode-x，那么直接申请一个空闲inode并把它的inode_num设为x即可
- inode_create()：创建一个磁盘里没有的inode-x, 首先申请一个空闲inode(磁盘和内存都要申请), 赋值成你想要的样子, 随后写入磁盘, 最后返回这个inode即可
- inode_free()：释放inode资源使用的，（1）只是在内存里释放这个inode；（2）在磁盘里也要删除这个inode，调用 inode_destroy()。
- inode_lock() 和 inode_unlock() 函数负责上锁和解锁。
- inode_unlock_free() 函数一口气解锁和释放inode。

实验七：文件系统-中层抽象

- inode 里的 size 和 addrs 字段服务于数据管理

addrs数组的结构:

- 最前面的 N_ADDRS_1 个元素直接存放 block_num (一级映射)
- 中间的 N_ADDRS_2 个元素存放的 block_num 不是数据block, 而是索引block，索引block由连续的entry构成, 每个entry存放一个block_num (二级映射)
- 最后的 N_ADDRS_3 个元素存放的是索引block的索引block (三级映射)

- inode_locate_block() 函数, 它负责把逻辑上的inode第bn块block转换成物理上的磁盘第bx个block, 如果bn不存在(比现有的最大bn大1), 则申请新的block并返回block_num。
- inode_read_data() 、 inode_write_data()和inode_data_free()。

目录管理

目录是一种特殊的结构化文件, 它由若干目录项 (directroy entry 简称 dirent) 组成

```
typedef struct dirent {
    uint16 inode_num;
    char name[DIR_NAME_LEN];
} dirent_t;
```

目录在磁盘和内存的定义是一样的, 由 inode_num 和 目录名 两部分组成

从目录名到inode_num的单向映射关系(一个目录名对应一个inode, 一个inode可以有多个目录名)
一个inode可以有多个目录名涉及链接的问题(nlink),
简化假设: 目录文件只能管理一个block(32个entry), 只有addrs[0]是有效的

目录项管理包括三个函数: dir_search_entry()、 dir_add_entry() 和dir_delete_entry()

实验七：文件系统-中层抽象

文件路径解析

文件路径, 指的是由一串文件和目录构成的结构化信息,

如："/user/work/hello.txt"

这里面包含了根目录 user目录、 work目录 和 hello.txt文件

本次实验中, 所有路径都是绝对路径, 也就是从根目录出发的路径

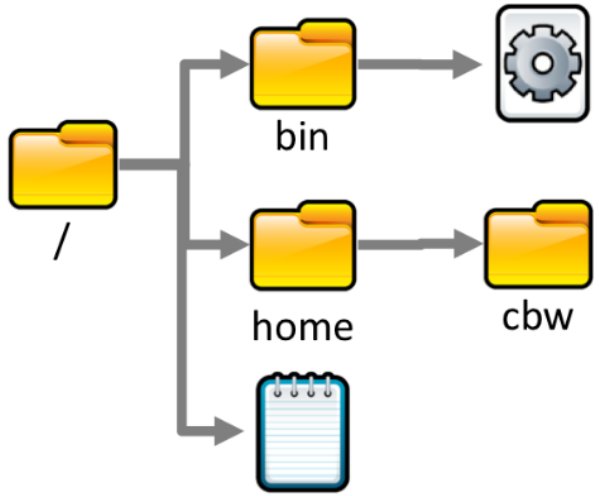
两个接口函数: path_to_inode() 和 path_to_pinode()

search_inode() ： 提供路径到inode的解析能力

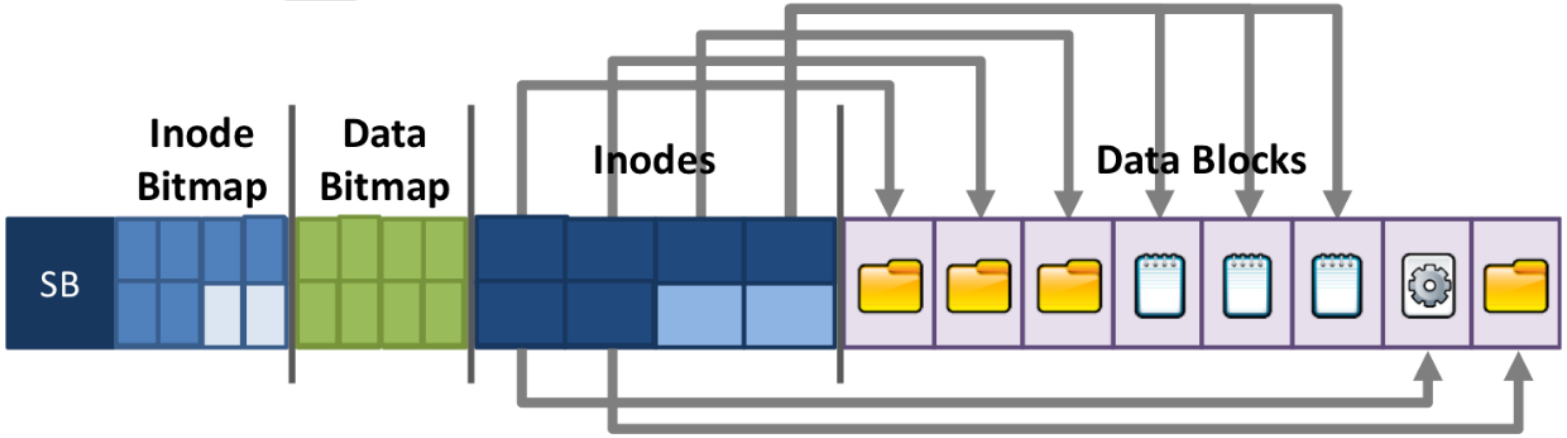
实验七：文件系统-中间抽象

- 文件结构体集合在一起就是文件表，整个系统只有一张文件表，配置了一个锁避免竞争条件。
- 文件描述符表每个进程有一张表，在进程控制块中。
- 文件操作：file_alloc,file_open, file_close,file_read, file_write等等
- 实现fs目录 file.c文件中的函数

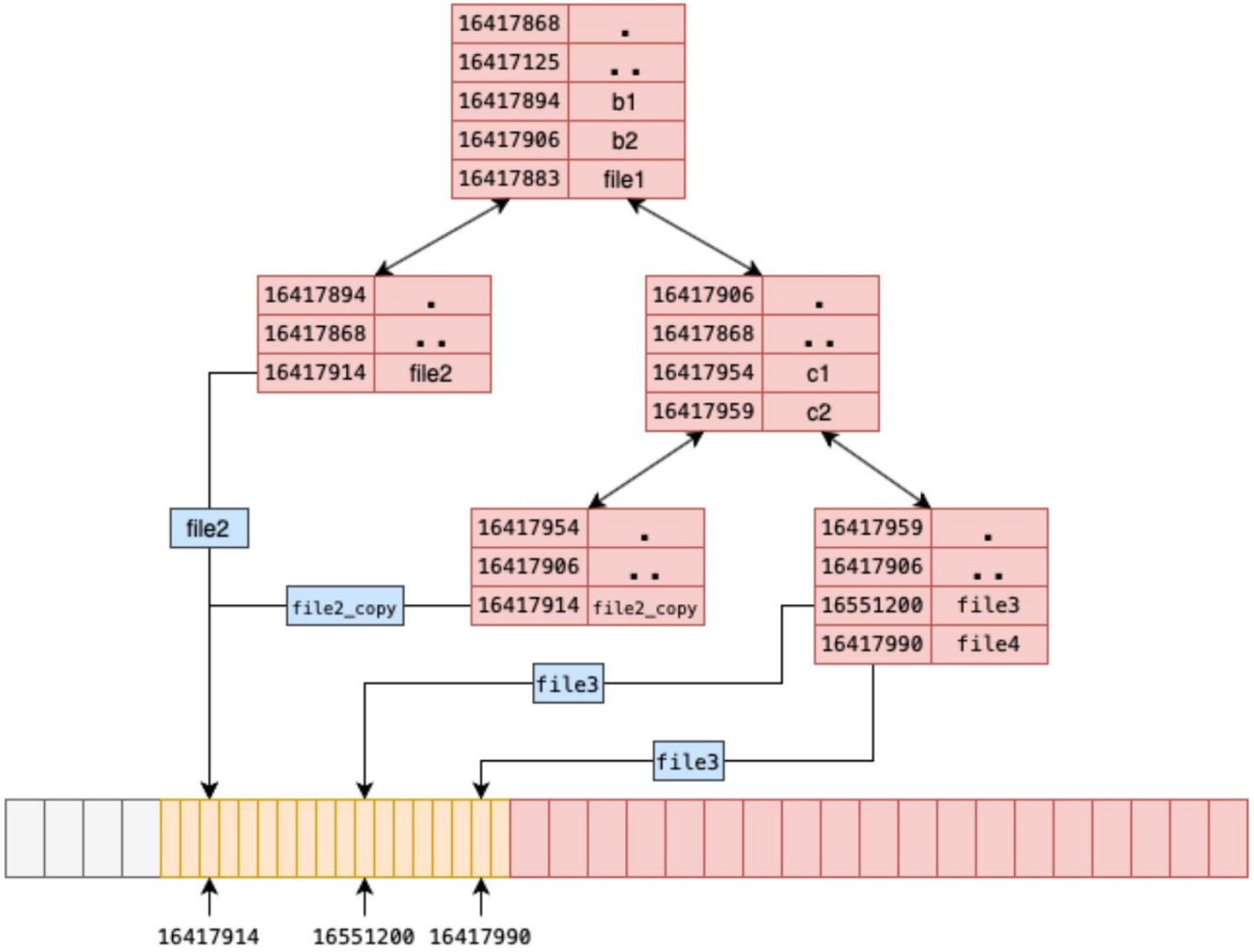
实验七：文件系统-中层抽象



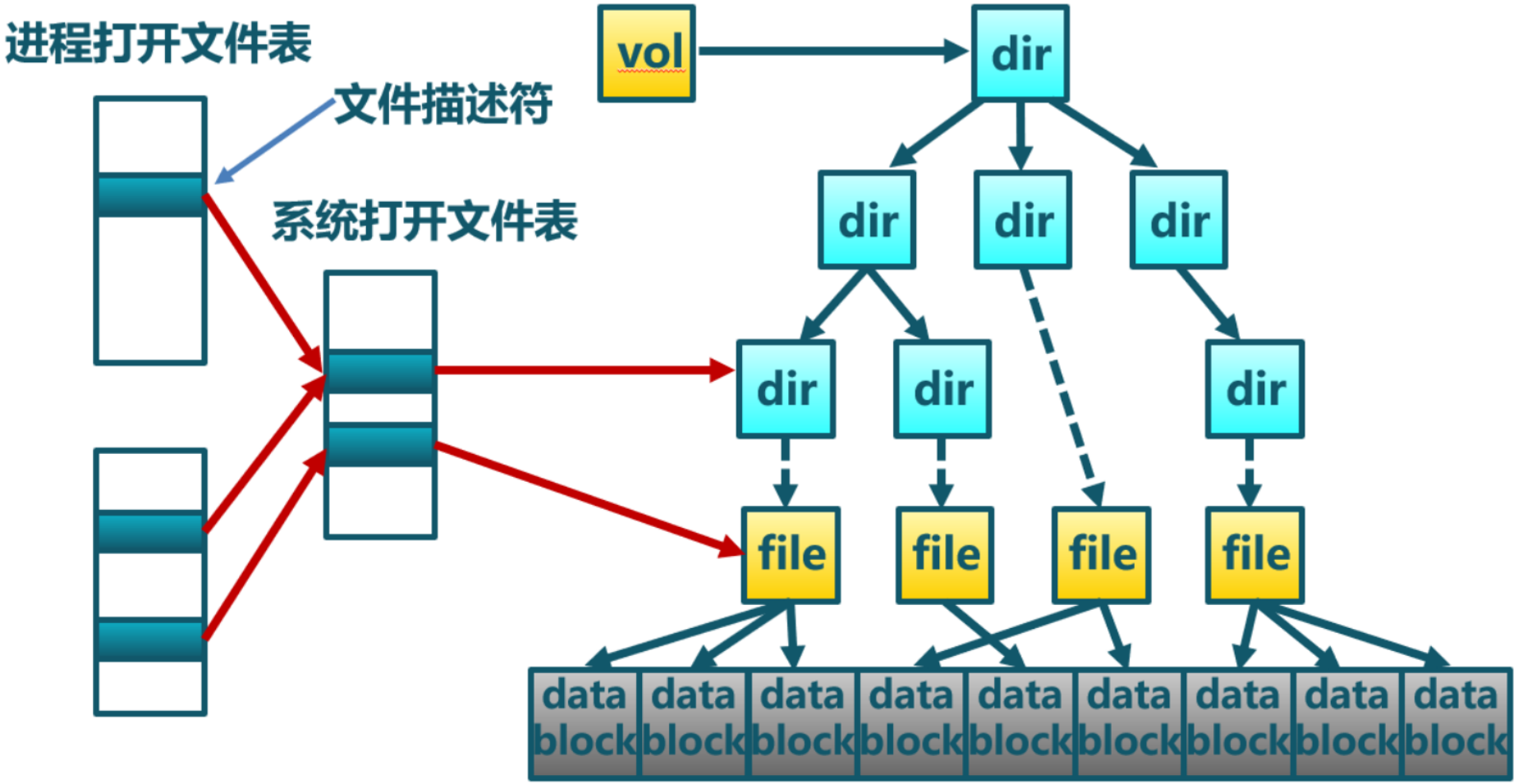
Name	inode
.	0
bin	1
home	2
initrd.img	3



实验七：文件系统-中层抽象

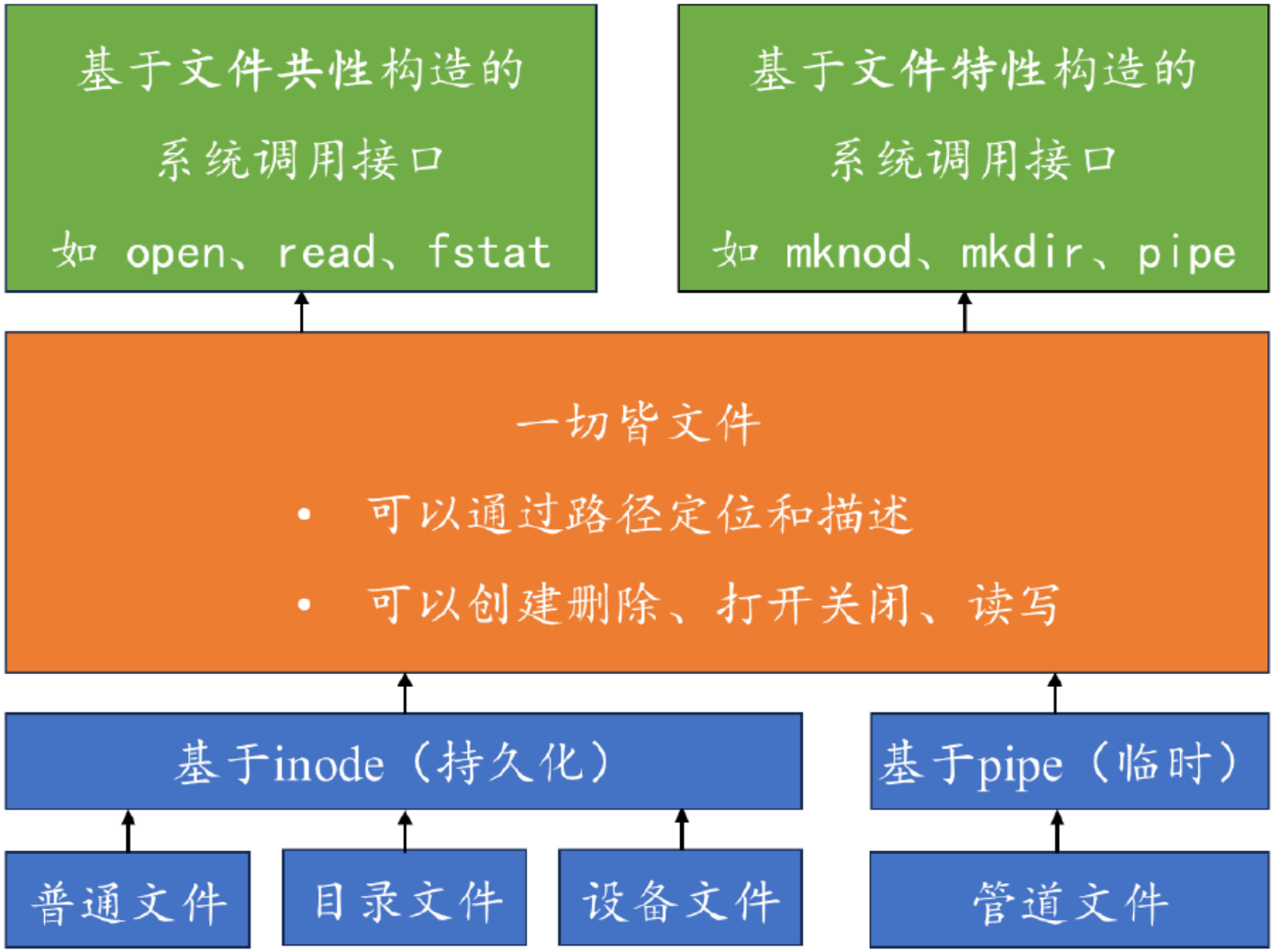


实验七：文件系统-中层抽象



实验七：文件系统-上层接口

- 上层将普通文件（包括可执行文件）、目录文件、设备文件、管道文件这四类常见文件以统一的框架组织起来。这样的好处是，上层的系统调用可以用统一的语言语义来操作各种各样的文件。



实验七：文件系统-上层接口

- 对于文件的共性操作（如open、read等），可以用同一个系统调用接口，在接口内部依据文件类型，执行类似switch-case的语句，分别调用文件注册的操作函数；对于文件的特性操作（如mknod、mkdir等），可以设立不同的系统调用接口，满足文件的私有需求。
- 实现syscall目录 sysfile.c文件中的函数

实验七：文件系统-上层接口

- 最后需要实现一个proc_exec函数, 这个函数可以用于执行一个ELF文件, 它涉及进程、内存、文件系统等多个模块。

实验七：文件系统-中间抽象