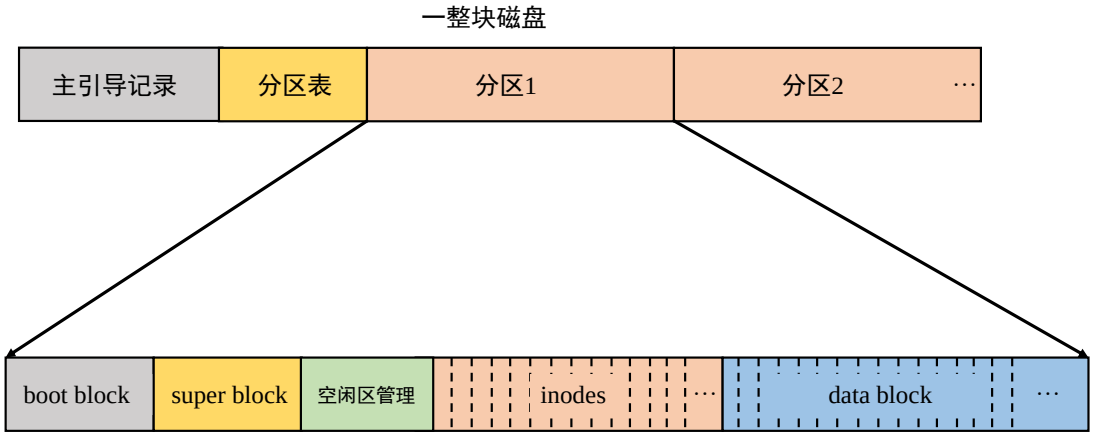
## 基本问题回答

**1. 了解 UNIX 文件系统的主要组成部分：超级块（superblock），i 节点（inode），数据块（data block），目录块（directory block），间接块（indirection block）。分别解释它们的作用。**

Unix 文件系统起源于 Berkeley Fast File System。通常来说，磁盘数据以扇区 Sector 为单位分布，在 Unix 文件系统中，按照 Block 为单位对磁盘数据进行读写操作，一个 Block 的大小为 512B（等同于一个扇区的大小）或 4KB（等同于 8 个扇区的大小）。

Unix 文件系统的布局如下图所示：




从上图可以看出，一块磁盘可以包含多个分区，而在一个分区中包含了一个文件系统，主要的组成部分是：

- ✧ boot block，启动块，装载 bootloader 程序，用于加载系统。
- ✧ super block，超级块，存储文件系统中的元信息。包括文件系统的类型、大小、状态和块的数量等。super block 对整个文件系统来说至关重要，在 Unix 文件系统中通常会保存多个 superblock 的副本。
- ✧ inode，又称索引节点，是 Unix 文件系统中对文件信息的抽象数据结构。存储了关于文件的元信息，可以简单理解为“文件名+inode=FCB”。每个文件应该要对应到 inode 才能进行操作。在 inode 中保存了除文件名之外的所有信息，包括：文件的大小（字节数）、访问存取权限、文件所有者（User ID）、文件所在组（Group ID）、时间戳、文件数据块所在位置等重要信息。需要注意的是，inode 本身也是一种资源，需要占据一定的磁盘空间。对于一个文件系统中的所有文件，文件系统会维护一个 inode 列表，也就是 inodes 区，这个区域可能占据一个或多个磁盘块。
- ✧ datablock，数据块。存储文件、目录等信息。一些文件系统可能会存在存放目录的 Directory Block 和 Indirection Block，但是在 Unix 文件系统中，目录视为是普通文件的一种，因此存放在 data block 中。而 indirection block 间接块指的是存放二级、三级索引的块。它们之间的唯一区别在于对于记录它们信息的 inode 中保存的属性不同。

**2. 阅读文件 ide.c。这是一个简单的 ide 硬盘驱动程序，对其内容作大致了解。**

IDE 全称是 Integrated Drive Electronics 集成驱动器电子，另外一个常见的名字是 ATA（Advanced Technology Attachment），这两个名词都有厂商在用，指的是同一个东西。这里的 ATA 其实指的是 PATA，向对比的是目前流行的 SATA。P 指的是并行的意思，S 指的是串行。

在 ide.c 文件中定义了一些关于 IDE 磁盘和端口规定的宏：

```
#define SECTOR_SIZE   512
#define IDE_BSY       0x80  // IDE 正忙
#define IDE_DRDY      0x40  // 磁盘块被修改
#define IDE_DF        0x20  // IDE
#define IDE_ERR       0x01  // IDE 错误

#define IDE_CMD_READ  0x20  // IDE 读
#define IDE_CMD_WRITE 0x30  // IDE 写
#define IDE_CMD_RDMUL 0xc4  // IDE 多个读，减少中断请求数量
#define IDE_CMD_WRMUL 0xc5  // IDE 多个写
```

在 ide.c 中定义了控制磁盘同步访问的数据结构，包括一个磁盘缓存队列 idequeue 和一个自旋锁 idelock。

```
static struct spinlock idelock;
static struct buf *idequeue;
```

之后是 havedisk1 这个静态变量，这个变量用来标识当前的系统中是否存在有多个 ide 磁盘，在启动时使用了主从式，首先启动一块磁盘，然后启动第二块磁盘。

磁盘请求的数据结构为 buf，定义如下：

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
```

其中，对 IDE 磁盘而言，需要关心的域是 flags（DIRTY，VALID），dev（设备），blockno 磁盘编号和 next（指向队列的下一个成员指针）。

磁盘读写实现的思路是：xv6 通过维护磁盘操作请求队列，当进程请求磁盘读写时，请求会被加入到队列，进程会进入睡眠状态（iderw()）。任何时候，队列的开头表示当前正在进行的磁盘读写请求。当一个磁盘读写操作完成时，会触发一个中断，中断处理程序(ideintr())会移除队列开头的请求，唤醒队列开头请求所对应的进程。如果还有后续的请求，就会将其移到队列开头，开始处理下一个磁盘请求。

下面是对 ide 磁盘操作的函数，他们的名称和功能如下：

| 函数名 | 功能描述 | 访问类型 |
|---|---|---|
| ideinit() | 初始化 IDE 磁盘 I/O | 公开访问 |
| idewait() | 等待磁盘进入空闲状态 | static，只允许文件内访问 |
| idestart() | 开始第一个磁盘读写请求 | static，只允许文件内访问 |
| iderw() | 上层文件系统调用的磁盘 I/O 接口 | 公开访问 |
| ideintr() | 磁盘请求完成后中断处理程序调用该函数 | 公开访问 |

操作系统启动时，main()函数会调用 ideinit()对 ide 磁盘进行初始化，初始化函数中会初始化 ide 锁，设定磁盘中断控制，并检查是否存在第二个磁盘。

iderw()函数提供了面向顶层文件系统模块的接口。iderw()既可用于读，也可用于写，只需通过判断 buf->flag 里的 DIRTY 位和 VALID 位就能判断出请求是读还是写。如果请求队列为空，证明当前磁盘不是工作状态，那么就需要调用 idestart()函数初始化磁盘请求队列，并设置中断。如果请求是个写请求，那么 idestart()中会向磁盘发出写出数据的指令。之后，iderw()会将调用者陷入睡眠状态。

当磁盘读取或者写操作完毕时，会触发中断进入 trap.c 中的 trap()函数，trap()函数会调用 ideintr()函数处理磁盘相关的中断。在 ideintr()函数中，如果当前请求是读请求，就读取目前已经在磁盘缓冲区中准备好的数据。最后，ideintr()会唤醒正在睡眠等待当前请求的进程，如果队列里还有请求，就调用 idestart()来处理新的请求。

## 3. 阅读文件 buf.h，bio.c。了解 XV6 文件系统中 buffer cache 层的内容和实现。描述 buffer 双链表数据结构及其初始化过程。了解 buffer 的状态。了解对 buffer 的各种操作。

首先是 buf.h 文件，在 buf.h 中定义了一个结构体 struct buf 和两个宏。结构体 buf 是磁盘中的块数据到内存中的映射，也就是说，把磁盘中某一块的内容拷贝到了内存中。所以 buf 里面有 uint 的类型的 dev 设备号和 blockno 块号。除此之外，buf 中还有用来标记当前 buffer cache 状态的 flags；一个 sleeplock 用来做同步互斥，需要注意的是，这里使用的是 sleeplock 睡眠锁而不是自旋锁，实际上 sleeplock 只被使用在文件系统中，因为对磁盘的访问和写入用自旋锁等待的话会严重影响整个系统的性能；refcnt 是 bget 函数打开 buffer cache 的一个计数，用来记录当前缓冲块被引用使用的次数；prev 和 next 是两个 buf 类型的指针，用来把 buf 结构体数组组织成双向链表的形式；qnext 也是一个 buf 类型的指针，但 qnext 的作用是在 ide.c 文件中，把磁盘内容读入缓冲时作为读入的排队顺序使用的，和 buffer cache 的双向链表形式并没有关系；date 是 512 字节的 uchar 数组，正好用来装载一个磁盘块内容的大小。此外，宏定义 B_VALID 利用第二位标志 buf 时候有磁盘载入的内容；B_DIRTY 利用二进制的第三位标志 buf 中的内容要被写入磁盘。

bio.c 文件中，具体的定义了 buffer cache 这个双向链表结构和在 buffer cache 上进行的一些操作。首先定义了一个 bcache 结构体，结构体中有一个 head 作为双向链表的头，head.next 被设置为最经常使用的节点；这里需要注意，内存中 buffer cache 的最大个数是固定为 30 个 buf，在 bcache 中还有一个 spinlock 用来做互斥。binit 函数初始化双向链表，通过 prev 和 next 指针把 30 个 buf 结构体连接起来；bget 方法是一个 static 方法，被 bread 调用，bget 函数传入的参数是设备号和块号，首先通过 next 节点向后遍历整个链表（MRU list 遍历），找到这个磁盘块是否被缓冲过，如果找到则 refcnt++，bcache 自旋锁放开，对应的 sleeplock 上锁，返回对应的 buf 指针；如果没有找到对应的 buf，则要分配 buf，此时就要利用 prev 遍历（LRU list 遍历），找到一个空的 buf 之后（次数的空 buf 必须是 refcnt==0 && B_DIRTY 位==0），分配给磁盘块，设置 flags，refcnt=1，解开 spinlock，上锁 sleeplock，返回 buf 指针。

bread 函数主要是调用了 bget 函数，bread 函数不是 static 的，它是提供给上层的一个频繁被使用的接口。bread 首先调用 bget 读取某个磁盘块，如果返回的 bget 指针使得 b->flags & B_VALID==0，说明这个磁盘块是第一次被读入缓冲，bget 中只是简单的占有了这个 buf，并没有执行实际的读入，所以在 B_VALID 位为 0 情况下，bread 调用 iderw 完成真正从磁盘读入内存的工作。简单的描述一下 iderw 这个函数：当 B_DIRTY 被设置，启动 ide 写操

作；当 B_VALID 没有被设置，启动 ide 读操作。iderw 首先 if 判断一些特殊情况（例如没有持有 buf 的 sleep 锁，B_DIRTY 未设置而 B_VALID 已经设置，当前不是第一个设备），然后把参数 b 放到 idequeue 的末尾（这个时候就需要使用到 buf 中的 qnext 指针了），当 idequeue==b 则启动读写（idestart），然后需要利用 while 循环一直等待 idestart 完成。

bwrite 函数也是提供向上的一个经常使用的接口，修改 B_DIRTY，并且启动 iderw，完成内存数据写到对应磁盘块的操作。

brelse 释放一个 buf 块，设置该 buf 缓冲块的 refcnt 被引用计数减一，只有当 refcnt==0 时才释放这个 buf 块，先从双向链表中删除这个 buf，然后把这个 buf 移动到 head.next 的位置，也就是如注释所说：移动到 MRU list 的头部，最后再解开 bcache 的自旋锁完成整个 brelse 操作。

## 4. 阅读文件 log.c。了解 XV6 文件系统中的 logging 和 transaction 机制。

- **日志系统的层次定位**

若将文件系统实现划分为七层：

1. 块读写驱动：底层设备相关的驱动程序
2. 数据块：原始磁盘数据块分配
3. 日志：提供多步更新中的异常恢复以及操作合并
4. 文件：i 节点分配、读取、写入、元数据维护
5. 目录：特殊 i 节点，包含其他 i 节点的列表
6. 文件名：像/usr/fs.c 一样，文件路径名使用起来更方便
7. 文件描述符：系统调用提供给用户使用的高层资源标识

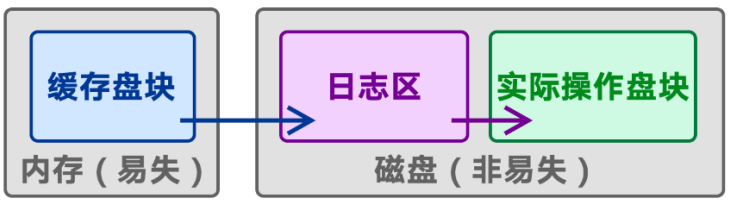
则 log.c 文件对应的日志系统处于第 3 层，负责提供对原始磁盘数据块的操作的抽象，提供多进程的文件操作同步、操作合并以及异常恢复支持，i 节点层会使用日志层所提供的服务。

日志层主要提供两个机制，logging 日志异常恢复，以及 transaction 操作事务化，下面分别讨论。

- **logging 日志异常恢复**

在执行数据块读写操作的过程中，如果系统发生异常，例如断电，导致数据写入不完全，就有可能对文件系统造成不可恢复的影响。

XV6 在文件系统中专门开辟了日志区，如此以来，数据可能存储在 3 个位置：内存中易失的数据块 buffer，磁盘中的日志区，磁盘中的相关的盘块（可能涉及数据区、i 节点区或空闲位图区），如下图所示：




XV6 中所有经过日志层做出的数据操作后，都不会立即写回相关的盘块，而是先写入磁盘中的日志区，而后再选取一个暂时没有磁盘操作的时机从日志区批量写入到修改的盘块里。如此方式大大降低了系统异常崩溃后文件系统受损的概率。

- **数据操作提交流程代码**

```
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

1. 从内存写入磁盘日志区：将块缓存中改动的盘块写入磁盘日志区。

2. 写入日志头：日志头定义如下图所示，由指示日志区当前盘块数量 n 以及每块日志区盘块对应的实际盘块地址构成的数组 block 组成。

```
struct logheader {
  int n;
  int block[LOGSIZE];
};
```

3. 实装到文件系统对应位置：将日志区中备份的盘块实际写入磁盘实际对应的盘块位置。

4. 清除日志头：将日志头的日志盘块数量设回 0，指示此次数据日志提交完成。

- **transaction 操作事务化**

上文提到在合适的时机下，系统会将记录的日志写回磁盘。这里“合适的时机”就体现了 XV6 中操作事务化的思想。

在 log.c 中维护了一个 log 结构体变量，其中的 outstanding 字段专用于记录当前已发起但尚未完成的数据操作数。只有在 outstanding 字段减少为 0 时，即在当前时刻所有发起的数据操作都结束而没有进行中的数据操作时，才会触发从日志区提交数据操作的流程。如图，在 end_op 函数中：

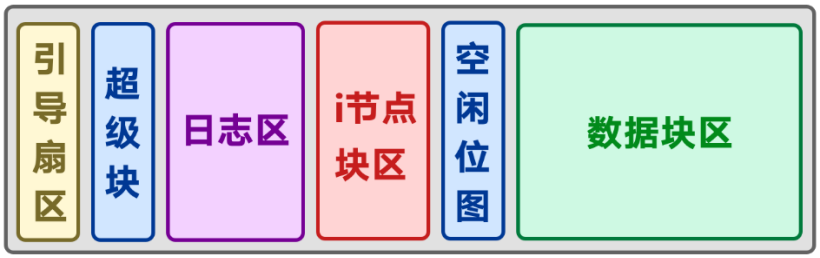
```
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
```

outstanding 字段成为提交更改的唯一前置条件。在这种安排下，只要所有数据操作在开始前都显式调用 begin_op 函数发起，在结束之后也显式调用 end_op 函数指示完成，这两次调用中的所有单个 log_write 写操作被包装成了一次事务，可以被认为要么全部提交更改，要么全部没有生效。

## 5. 阅读文件 fs.h，fs.c。了解 XV6 文件系统的硬盘布局。

- **XV6 文件系统布局**

fs.h 中定义了 XV6 文件系统模块所规定的磁盘数据物理布局，如下图所示：




可以看出，该文件系统主要由 6 个部分构成：

1. 引导扇区：存储操作系统启动时需要执行的引导代码，占用 0 号扇区。
2. 超级块：存储文件系统（一个 XV6 分区）的元配置信息，主要记录盘块区域划分要用到的区域起始块号或块总数，如下图结构体所示：

```
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
mkfs: make filesystem
mkfs算出超级块的参数，构建初始文件系统
struct superblock {
  uint size;         文件系统磁盘块总数量  // Size of file system image
(blocks)
  uint nblocks;      数据块数量     // Number of data blocks
  uint ninodes;      i节点数量  // Number of inodes.
  uint nlog;         日志块数量  // Number of log blocks
  uint logstart;     起始日志块编号  // Block number of first log block
  uint inodestart; 起始i节点块编号  // Block number of first inode block
  uint bmapstart;  起始空闲位图区块编号  // Block number of first free map
block
};
```

3. 日志区：即上文问题 5 所讨论的数据操作日志中转区。由日志头和日志数据盘块组成。
4. i 节点块区：集中存储 i 节点的磁盘区域。i 节点中主要保存了磁盘文件索引用于查找存储文件内容的物理盘块号，除此以外，还有一部分的文件元信息存于其中。i 节点在磁盘盘块中的存储结构如下图所示：

```
// On-disk inode structure
struct dinode {
  short type;  文件类型   (目录文件、普通文件、设备文件3种，见stat.h)        //
File type
  short major; 主设备号（仅针对设备文件类型）          // Major device number
(T_DEV only)
  short minor; 次设备号（仅针对设备文件类型）            // Minor device
number (T_DEV only)
  short nlink; 文件系统中存在的链接数          // Number of links to inode
in file system
  uint size;   文件大小，单位字节          // Size of file (bytes)
  uint addrs[NDIRECT+1]; 数据块的地址索引  // Data block addresses
};
```

根据该定义即可得出每个 i 节点结构恰好占用 8+4+13*4 = 64 字节空间大小、磁盘里每

个磁盘块可以存储 512/64 = 8 个 i 节点。结构体的最后一个字段，addrs 数组即文件内容块索引，索引方式如下图所示：

```
#define NDIRECT 12 直接索引可索引12块
#define NINDIRECT (BSIZE / sizeof(uint)) 间接索引可索引128块
#define MAXFILE (NDIRECT + NINDIRECT) 单个文件最大28块
```

可以看到，XV6 文件系统采用的数据索引方式与 UNIX 非常类似，都属于综合索引方式：直接索引和间接索引相结合。相较而言实现的比较简化，只支持一级间接索引。

5. 空闲位图：XV6 通过维护空闲位图表来实现磁盘空闲空间管理，空闲位图的相关定义如下图所示：

```
// Bitmap bits per block
#define BPB 每个磁盘块内包含的位数 512*8 = 4096          (BSIZE*8)

// Block of free map containing bit for block b
/* 定位 b号 数据块对应的空闲位表所处的磁盘块号 */
#define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
/* 以起始空闲位表编号开始算起，向后偏移 b/4096 下取整 */
```

6. 数据块区：存储具体的文件数据内容，VX6 中文件在 stat.h 中主要定义了三种类型，如下图所示：

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device
```

即目录文件、普通文件和设备文件。目录文件的结构定义如下图所示：

```
// Directory is a file containing a sequence of dirent structures.
/* 目录文件包含一系列目录项 */
#define DIRSIZ 14 文件名最大长度
dirent = Directory Entry 目录项
struct dirent {
  ushort inum; i节点号
  char name[DIRSIZ]; 文件名数组，最长14和字符
};
```

根据该定义可以得出一项目录项大小空间占用为 4+14 = 16 字节。

- **fs.c 中包含的实现综述**

  若依照问题 5 中对文件系统的 7 层划分，fs.c 文件中实现的逻辑贯穿了 4、5、6 三层，涉及到了超级块读取、空闲块管理、i 节点分配、i 节点内容操作、目录管理、路径查找等方面，下面将其分成 5 个部分依次梳理主要函数并一一简要说明。

- **超级块读取与空闲块管理**

| 函数/结构名 | 说明 | 外部可否调用？ |
|---|---|---|
| sb | 读入内存的超级块信息 | |
| readsb | 读超级块 | ✓ |
| bzero | 将磁盘块置零 | |

| | | |
|---|---|---|
| balloc | 分配全零磁盘块 | |
| bfree | 释放磁盘块 | |

- **i 节点分配**

| 函数 | 说明 | 外部可否调用？ |
|---|---|---|
| icache | i 节点内存缓存结构体 | |
| iinit | i 节点系统模块初始化 | ✓ |
| ialloc | 分配 i 节点 | ✓ |
| iupdate | 将内存中更改过后的 i 节点写入硬盘 | ✓ |
| iget | 查找 n 号 i 节点，读入内存 | |
| idup | 将指定 i 节点的引用计数加一 | ✓ |
| ilock | 将 i 节点上锁+从磁盘读入 | ✓ |
| iunlock | 将 i 节点解锁 | ✓ |
| iput | 释放 i 节点缓存 | ✓ |
| iunlockput | iunlock+iput | ✓ |

- **i 节点内容操作**

| 函数 | 说明 | 外部可否调用？ |
|---|---|---|
| bmap | 获取或创建 i 节点的第 n 个数据块号 | |
| itrunc | 删除 i 节点以及其对应的数据块内容 | |
| stati | 读 i 节点的文件状态信息 | ✓ |
| readi | 读 i 节点数据内容 | ✓ |
| writei | 向 i 节点写内容 | ✓ |

- **目录管理**

| 函数 | 说明 | 外部可否调用？ |
|---|---|---|
| namecmp | 对比两个文件名是否相等 | ✓ |
| dirlookup | 在目录文件中搜索目录项 | ✓ |
| dirlink | 向目录文件里写目录项 | ✓ |

- **路径查找**

| 函数 | 说明 | 外部可否调用？ |
|---|---|---|
| skipelem | 从路径中取出一个路径元素 | |
| namex | 返回指定路径对应的 i 节点/父 i 节点 | |
| namei | 返回指定路径对应的 i 节点 | ✓ |
| nameiparent | 返回指定路径对应的父 i 节点和文件名 | ✓ |

**6. 阅读文件 file.h, file.cc。了解 XV6 的“文件”有哪些，以及文件，i 节点，设备相关的数据结构。了解 XV6 对文件的基本操作有哪些。XV6 最多支持多少个文件？每个进程最多能打开多少个文件？**

- **文件类型**

  XV6 中“文件”可以分为设备文件、管道文件、目录文件和普通文件。

在 file.h 文件中的文件描述符结构 file 存在枚举属性 type，取值包括：FD_NONE(未使用)、FD_PIPE(管道文件)、FD_INODE(使用 i 节点的文件)。另外，file 中存在 i 节点结构 inode 的指针，被使用的 inode 的 type 属性取值有三种（若 inode 空闲则 type 取值为 0），在 stat.h 文件中定义，分别为：T_DIR(目录文件)、T_FILE(普通文件)、T_DEV(设备文件)。

- **文件相关数据结构**

file.h 文件中的 file 类是 XV6 文件描述符层中表示文件的数据结构，它是每个进程 PCB 中文件打开表 ofile(proc.h 中)和全局文件打开表 file(file.c 中)的组成条目。如图 1 所示，file 结构的成员属性包括：表示 file 类型的 type、表示引用计数的 ref，表示可读性的 readable、表示可写性的 writable、表示 pipe 结构指针的*pipe，表示 inode 结构的指针*ip 和表示文件读写偏移的 off。

```
1  struct file {
2    enum { FD_NONE, FD_PIPE, FD_INODE } type;
3    int ref; // reference count
4    char readable;
5    char writable;
6    struct pipe *pipe;
7    struct inode *ip;
8    uint off;
9  };
```

*图1　file 结构*

- **i 节点相关数据结构**

file.h 文件中的 inode 类表示被读入内存中的 i 节点，它包含了一个磁盘上 i 节点的拷贝，以及一些内核需要的附加信息。如图 2 所示，inode 结构的成员属性包括：表示设备号的 dev、表示 i 节点号的 inum、表示引用计数的 ref、表示访问睡眠锁的 lock，表示 i 节点是否被读入内存的 valid 和磁盘 i 节点结构 dinode 的各种属性和数据。

```
12 // in-memory copy of an inode
13 struct inode {
14   uint dev;           // Device number
15   uint inum;          // Inode number
16   int ref;            // Reference count
17   struct sleeplock lock; // protects everything below here
18   int valid;          // inode has been read from disk?
19
20   short type;         // copy of disk inode
21   short major;
22   short minor;
23   short nlink;
24   uint size;
25   uint addrs[NDIRECT+1];  // 索引数组，12个直接索引和1个间接索引
26 };
```

*图2　inode 结构*

- **设备相关数据结构**

file.h 文件中与设备相关的数据结构为 devsw 类。如图 3 所示，devsw 类中定义了设备通过 inode 读取或写入设备文件数据内容的两个方法，在初始化设备时重写这两个方法。文件中还定义了全局的设备读写方法数组 devsw[]，每个条目为 devsw 类。

```
28  // table mapping major device number to
29  // device functions
30  struct devsw {
31    int (*read)(struct inode*, char*, int);
32    int (*write)(struct inode*, char*, int);
33  };
34
35  extern struct devsw devsw[];
```

*图3　devsw 结构*

- **管道文件相关数据结构**

管道文件相关结构在 pipe.c 文件中，系统在内核空间中申请一页空间作为管道文件（其中包含了一块的数据空间和一些控制信息），并申请读文件描述符和写文件描述符供进程在一页的空间中交换信息（读写的方式为取模循环读写）。如图 4 所示，pipe 结构的属性成员包括：表示访问自旋锁的 lock、表示管道文件数据空间的 data 数组、表示读光标的 nread、表示写光标的 nwrite、表示持有读描述符打开引用计数的 readopen 和表示写描述符打开引用计数的 writeopen。

```
11  #define PIPESIZE 512
12
13  struct pipe {
14    struct spinlock lock;
15    char data[PIPESIZE];
16    uint nread;     // number of bytes read
17    uint nwrite;    // number of bytes written
18    int readopen;   // read fd is still open
19    int writeopen;  // write fd is still open
20  };
```

*图4　pipe 结构*

- **对文件的基本操作**

file.c 文件中定义了与文件基本操作相关的 file 结构管理方法，文件基本操作和 file 结构相关方法的对应关系如下表所示。

| 基本操作 | 相关方法 | 方法描述 |
|---|---|---|
| 打开文件 | filealloc() | 在全局打开文件表中初始化新表项，ref 设为 1 |
| 描述符引用数加 1 | filedup() | ref 加 1 |
| 关闭文件 | fileclose() | ref 减 1。若 ref==0，type 设为 FD_NONE，释放内存中管道文件或 i 节点 |
| 读取文件元数据 | filestat() | 将对应 i 节点的元数据读取到 stat 结构中 |
| 读文件 | fileread() | 根据文件类型调用不同的读取方法 |
| 写文件 | filewrite() | 根据文件类型使用不同的写入方法 |

- 支持文件数

如图 5 所示，param.h 文件中定义了 XV6 系统所支持的文件数。在一个文件系统中，XV6 最多同时支持 100（NFILE）个文件，每个进程最多能同时打开 16（NOFILE）个文件。

```
3  #define NCPU          8  // maximum number of CPUs
4  #define NOFILE       16  // open files per process
5  #define NFILE       100  // open files per system
6  #define NINODE       50  // maximum number of active i-nodes
7  #define NDEV         10  // maximum major device number
8  #define ROOTDEV       1  // device number of file system root disk
```

*图 5　param.h*

## 7. 阅读 sysfile.c。了解与文件系统相关的系统调用，简述各个系统调用的作用。

sysfile.c 文件中定义了各种文件系统调用相关的调用方法，下表将各种文件系统相关的系统调用与 sysfile.c 文件中方法的调用关系。表中包括了 13 个系统调用，描述了各个系统调用的系统调用号、相应函数和作用。

| 系统调用 | 调用号 | 相应函数 | 作用 |
|---|---|---|---|
| SYS_dup | 10 | sys_dup() | 调用 filedup()对文件描述符的引用计数加 1 |
| SYS_read | 5 | sys_read() | 调用 fileread()读取文件相应位置相应大小的数据 |
| SYS_write | 16 | sys_write() | 调用 filewrite()写相应大小数据到文件相应位置 |
| SYS_close | 21 | sys_close() | 释放进程的文件描述符，调用 fileclose()关闭文件 |
| SYS_fstat | 8 | sys_fstat() | 调用 filestat()获得文件的元数据 |
| SYS_link | 19 | sys_link() | 为文件创建一个新的硬链接（不可用于目录文件） |
| SYS_unlink | 18 | sys_unlink() | 移除文件的某个硬连接，若硬链接全部被移除，释放 i 节点（可用于删除文件夹） |
| SYS_open | 15 | sys_open() | 打开相应文件，返回文件描述符 |
| SYS_mkdir | 20 | sys_mkdir() | 创建一个目录文件（文件夹） |
| SYS_mknod | 17 | sys_mknod() | 创建一个设备文件 |
| SYS_chdir | 9 | sys_chdir() | 切换进程当前工作目录 |
| SYS_exec | 7 | sys_exec() | 将 elf 可执行文件读入内存并执行 |
| SYS_pipe | 4 | sys_pipe() | 创建一个管道文件，返回读、写文件描述符 |

## 源代码阅读

### 1. file.h & file.c

file.h 文件主要声明了内存中文件描述符的结构 file 和 i 节点的结构 inode。inode 是 file 结构的一个组成成分，而 file 则是打开文件表的表项。

XV6 为每个进程分配一个独立的打开文件表，每个打开文件表项都由 file 结构表示。进程每次使用 SYS_open 系统调用都会创建一个新的打开文件，即新的 file 结构体实例，它们是全局打开文件表表项的一份拷贝，但拥有不同的文件读写偏移。

XV6 系统在 file.c 文件中维护着一个全局的打开文件表 ftable，依靠一下五个方法对 ftable 进行管理：

- filealloc()：在 ftable 中遍历寻找一个引用计数 ref 为 0 的表项，将其 ref 设为 0 并返回其引用。
- filedup()：为 ftable 中的某个表项增加引用计数 ref。
- fileclose()：减少 ftable 表项的引用计数，当一个文件的引用计数为 0 时，根据文件类型释放当前的 i 节点或管道。
- filestat()：读取 ftable 表项所表示文件的 i 节点元数据信息。
- fileread()：从文件当前偏移位置读取确定字节的数据到内存，根据文件的类型（i 节点文件或管道）调用不同的读取方法。
- filewrite()：将内存中确定字节的数据写到文件当前偏移位置，根据文件的类型（i 节点文件或管道）调用不同的读取方法。需要注意的是，如果是写磁盘上的文件，由于日志系统的存在，需要分多次写数据，一次写入不超过最大块数地数据到磁盘的日志区。

### 2. sysfile.c

sysfile.c 文件中实现了 XV6 系统中与文件相关的系统调用函数，大多数的系统调用的实现都是简单调用 file.c 文件中的方法，如 sysdup()和 sysread()等。其他需要介绍的系统调用函数如下：

- sys_link()：为 old 文件创建硬链接 new。首先，获取 old 文件的 i 节点 ip，检查文件类型（无法为目录创建硬链接），ip->nlink++。其次，得到 new 路径所在目录的 i 节点 dp，检查 ip 和 dp 是否为同一个文件系统（不能跨文件系统创建硬链接），为 dp 的添加（新文件名，ip 的 i 节点号）的索引条目，创建链接。最后，写回 ip 和 dp，完成硬链接创建过程。
- sys_unlink()：删除某个硬链接或目录。首先，找到路径名文件所在目录的 i 节点 dp 和文件的 i 节点 ip。其次，将 dp 中文件对应索引项置为全 0。第三，若该文件为目录文件，dp 的 nlink 减 1；ip 的 nlink 减 1。最后，写回 ip 和 dp，完成删除过程。
- create()：根据路径创建某种类型的文件。首先，找到路径名文件所在目录的 i 节点 dp，查找文件的 i 节点 ip，若存在，便直接返回。其次，若 ip 不存在，创建新的 i 节点，将其设为 ip，初始化 ip 的元数据并更新。第三，若需要创建的文件类型为目录文件，为该文件添加“.”和“..”两个目录项。随后，为父目录添加 ip 的目录项，返回 ip。

- sys_open(): 打开一个文件，返回文件句柄。首先，若文件打开方式含有 O_CREATE，便调用 create()获取 i 节点 ip，否则调用 namei()获取 i 节点。其次，为该文件创建全局打开文件描述符表项和进程的打开文件描述符表项。最后，初始化文件描述符表的信息项，返回进程打开文件描述符。
- sys_mkdir()：创建一个目录文件。调用 create()方法，传入 T_DIR 作为类型参数，便可创建目录文件。
- sys_mknod()：创建一个设备文件。调用 create()方法，传入 T_DEV 作为类型参数，便可创建设备文件。
- sys_chidir()：切换当前进程的当前工作目录。首先，找到路径名表示的文件的 i 节点 ip；其次，将当前进程的工作目录指向的 i 节点写回，将工作目录重新设为 ip。
- sys_pipe()：创建一个管道，返回读、写描述符。首先，初始化 pipe，声明全局的读、写文件描述符。其次，声明进程的读、写描述符，并将它们返回，供进程使用。

## 参考文献


[1] XV6 源码阅读-文件系统 [EB/OL]. 荒野之萍. 2019.06.08. https://icoty.github.io/2019/06/08/xv6-filesystem/

[2] XV6 操作系统代码阅读心得（五）：文件系统 [EB/OL]. Hao_He. 2019.05.31. https://www.cnblogs.com/hehao98/p/10953777.html

[3] linux 系统调用之 sys_link（基于 linux0.11）[EB/OL]. theanarkh. 2019.05.04. https://blog.csdn.net/theanarkh/article/details/89814934

[4] xv6-Chinese [EB/OL]. Ranxian. 2020.03.31. https://github.com/ranxian/xv6-chinese

[5] 北京大学操作系统高级课程文件系统课件. 陈向群. 2020.