Linux系统编程-文件系统

杨劲松 yjs@oldhand.org

2011.04.13

课程目标

- 了解文件系统的基础知识
- 了解Linux文件系统的发展历史
- 了解ext2文件系统
- 了解VFS

参考资料

- 《深入理解Linux内核》(《Understanding the Linux Kernel》)s
  - http://oreilly.com.cn/samplechap/understandinglinuxkernel2/und_linux_kern2-17.pdf
- 《深入Linux内核架构》
  - 第8章 虚拟文件系统
  - 第9章 EXT2文件系统族
- 亚嵌教育《Linux系统编程》
  - 第2章 文件系统

内容提纲

- 概述
- ext2文件系统
- VFS

1. 概述

1.1文件系统是什么？

- 什么是文件系统(file system)，以下是几种解释：
  - A directory structure contained within a disk drive or disk area. The total available disk space can be composed of one or more file systems. A file system must be mounted before it can be accessed. To mount a file system, you must specify a directory to act as the mount point. Once mounted, any access to the mount point directory or its subdirectories will access the separate file system.
  - A method of organising files on a disk, eg NTFS, FAT.
  - A data structure or a collection of files. In Unix, filesystem can refer to two very distinct things, the directory tree or the arrangement of files on disk partitions.
  - the structure of files on a disk medium which is visible via the operating system, ie the structure of files which a Unix user can see using "ls" and other tools
  - A software mechanism that defines the way that files are named, stored, organized, and accessed on logical volumes of partitioned memory.
  - In computing, a file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, or they may be virtual and exist only as an access method for virtual data or for data over a network (e.g. NFS).

1.2 Comparison of file systems

- General information
  - Creator
  - Year introduced
  - Original operating system
- Limits
  - Maximum filename length
  - Allowable characters in directory entries
  - Maximum pathname length
  - Maximum file size
  - Maximum volume size
- Metadata
  - Stores file owner
  - POSIX file permissions
  - Creation timestamps
  - Last access/read timestamps
  - Last modification of content
  - This copy created
  - Last metadata change timestamps
  - Last archive timestamps
  - Access control lists
  - Security/MAC labels
  - Extended attributes/Alternate data streams/forks
  - Checksum/ECC

1.2 Comparison of file systems

- Features
  - Hard links
  - Symbolic links
  - Block journaling
  - Metadata-only journaling
  - Case-sensitive
  - Case-preserving
  - File Change Log
  - Snapshot
  - XIP
  - Encryption
  - COW
  - integrated LVM
  - Data deduplication
- Allocation and layout policies
  - Block suballocation
  - Variable file block size
  - Extents
  - Allocate-on-flush
  - Sparse files
  - Transparent compression

1.2 Comparison of file systems

- OS support
  - Windows 9x
  - Windows NT
  - Linux
  - Mac OS
  - Mac OS X
  - FreeBSD
  - BeOS
  - Solaris
  - AIX
  - z/OS
  - OS/2
  - ……

1.3 Linux主要的文件系统

- Minix
  - Minix 是Linux支持的第一个文件系统，对用户有很多限制，性能低下，有些没有时间标记，文件名最长l4个字符。
  - Minix文件系统最大缺点是只能使用64MB的硬盘分区，所以目前已经没有人使用它了。
- ext
  - ext是第一个专门为开发的Linux的文件系统类型，叫做扩展文件系统。它是1992年4月完成的，对Linux早期的发展产生了重要作用。但是，由于其在稳定性、速度和兼容性上存在许多缺陷，现在已经很少使用了。
- ext2
  - ext2是为解决ext文件系统的缺陷而设计的可扩展的、高性能的文件系统，它又被称为二级扩展文件系统。ext2是1993年发布的，设计者是 Rey Card。它是Linux文件系统类型中使用最多的格式，并且在速度和CPU利用率上较为突出，是GNU/Linux系统中标准的文件系统。它存取文件的 性能极好，对于中、小型的文件更显示出优势，这主要得益于其簇快取层的优良设计。ext2可以支持256字节的长文件名，其单一文件大小和文件系统本身的 容量上限与文件系统本身的簇大小有关。在常见的Intel x86兼容处理器的系统中，簇最大为4KB，单一文件大小上限为2048GB, 而文件系统的容量上限为6384GB。尽管Linux可以支持种类繁多的文件系统，但是2000年以前几乎所有的Linux发行版都使用ext2作为默认 的文件系统。
  - ext2也有一些问题。由于它的设计者主要考虑的是文件系统性能方面的问题，而在写入文件内容的同时，并没有写入文件的meta-data（和文件有关的 信息，例如权限、所有者及创建和访问时间）。换句话说，Linux先写入文件的内容，然后等到有空的时候才写入文件的meta-data。如果出现写入文 件内容之后，但在写入文件的meta-data之前系统突然断电，就可能造成文件系统就会处于不一致的状态。在一个有大量文件操作的系统中，出现这种情况 会导致很严重的后果。
- ext3
  - ext3是由开放资源社区开发的日志文件系统，早期主要开发人员是Stephen Tweedie。ext3被设计成是ext2的升级版本，尽可能方便用户从ext2向ext3迁移。ext3在ext2的基础上加入了记录元数据的日志功 能，努力保持向前和向后的兼容性，也就是在保有目前ext2的格式之下再加上日志功能。和ext2相比，ext3提供了更佳的安全性，这就是数据日志和元 数据日志之间的不同。ext3是一种日志式文件系统，日志式文件系统的优越性在于由于文件系统都有快取层参与运作，如不使用时必须将文件系统卸下，以便将 快取层的资料写回磁盘中。因此，每当系统要关机时，必须将其所有的文件系统全部卸下后才能进行关机。如果在文件系统尚未卸下前就关机 (如停电)，那么重开机后就会造成文件系统的资料不一致，故这时必须做文件系统的重整工作，将不一致与错误的地方修复。然而，这个过程是相当耗时的，特别 是容量大的文件系统不能百分之百保证所有的资料都不会流失，特别在大型的服务器上可能会出现问题。除了与ext2兼容之外，ext3还通过共享ext2的 元数据格式继承了ext2的其它优点。比如，ext3用户可以使用一个稳固的fsck工具。由于ext3基于ext2的代码，所以它的磁盘格式和ext2 的相同，这意味着一个干净卸装的ext3文件系统可以作为ext2文件系统毫无问题地重新挂装。如果现在使用的是ext2文件系统，并且对数据安全性要求 很高，这里建议考虑升级使用ext3。
  - ext3最大的缺点是，它没有现代文件系统所具有的、能提高文件数据处理速度和解压的高性能。此外，使用ext3文件系统要注意硬盘限额问题，在这个问题解决之前,不推荐在重要的企业应用上采用ext3＋Disk Quota（磁盘配额）。

1.3 Linux主要的文件系统

- ReiserFS
  - ReiserFS的第一次公开亮相是在1997年7月23日，Hans Reiser把他的基于平衡树结构的ReiserFS文件系统在网上公布。ReiserFS 3.6.x（作为Linux 2.4一部分的版本）是由Hans Reiser和他的Namesys开发组共同开发设计的。SuSE Linux也对它的发展起了重大的帮助。Hans和他的组员们相信最好的文件系统是能够有助于创建独立的共享环境或命名空间的文件系统，应用程序可以在其 中更直接、有效和有力地相互作用。为了实现这一目标，文件系统就应该满足使用者对性能和功能方面的需要。那样使用者就能够继续直接地使用文件系统，而不必 建造运行在文件系统之上（如数据库之类）的特殊目的层。ReiserFS 使用了特殊的、优化的平衡树（每个文件系统一个）来组织所有的文件系统数据，这为其自身提供了非常不错的性能改进，也能够减轻文件系统设计上的人为约束。 另一个使用平衡树的好处就是，ReiserFS 能够像其它大多数的下一代文件系统一样，根据需要动态地分配索引节，而不必在文件系统创建时建立固定的索引节。这有助于文件系统更灵活地适应面临的各种存 储需要，同时提供附加的空间有效率。
  - ReiserFS被看作是一个更加激进和现代的文件系统。传统的Unix文件系统是按磁盘块来进行空间分配的，对于目录和文件等的查找使用了简单的线性查 找。这些设计在当时是合适的，但随着磁盘容量的增大和应用需求的增加，传统文件系统在存储效率、速度和功能上已显得落后。在ReiserFS的下一个版本 —Reiser 4，将提供了对事务的支持。ReiserFS突出的地方还在于其设计上着眼于实现一些未来的插件程序，这些插件程序可以提供访问控制列表、超级链接，以及 一些其它非常不错的功能。在 http://www.namesys.com/v4/v4.html中，有Reiser 4的介绍和性能测试。
  - ReiserFS一个最受批评的缺点是，每升级一个版本都将要将磁盘重新格式化一次，而且它的安全性能和稳定性与ext3相比有一定的差距。因为 ReiserFS文件系统还不能正确处理超长的文件目录，如果创建一个超过768字符的文件目录，并使用ls或其它echo命令，将有可能导致系统挂起。 在http://www.namesys.com/ 网站可以了解关于ReiserFS的更多信息。
- XFS
  - XFS是一种非常优秀的日志文件系统，它是由SGI于20世纪90年代初开发的。XFS推出后被业界称为先进的、最具可升级性的文件系统技术。它是一个全64位、快速、稳固的日志文件系统，多年用于SGI的IRIX操作系统。当SGI决定支持Linux社区时，它将关键的基本架构技术授权于 Linux，以开放资源形式发布了他们自己拥有的XFS的源代码，并开始进行移植。此项工作进展得很快，目前已进入beta版阶段。作为一个64位文件系统，XFS可以支持超大数量的文件（9000×1GB），可在大型2D和3D数据方面提供显著的性能。XFS有能力预测其它文件系统薄弱环节，同时提供了在不妨碍性能的情况下增强可靠性和快速的事故恢复。
  - XFS可为Linux和开放资源社区带来如下新特性：
    - 可升级性 XFS被设计成可升级，以面对大多数的存储容量和I/O存储需求；可处理大型文件和包含巨大数量文件的大型目录，以满足21世纪快速增长的磁盘需求。 XFS有能力动态地为文件分配索引空间，使系统形成高效支持大数量文件的能力。在它的支持下，用户可使用的文件远远大于现在最大的文件系统。
    - 优秀的I/O 性能典型的现代服务器使用大型的条带式磁盘阵列，以提供达数GB/秒的总带宽。XFS可以很好地满足I/O请求的大小和并发I/O请求的数量。XFS可作为 root文件系统，并被LILO支持，也可以在NFS服务器上使用，并支持软件磁盘阵列（RAID）和逻辑卷管理器（Logical Volume Manager,LVM）。SGI最新发布的XFS为1.0.1版，在http://oss.sgi.com/projects/XFS/ 可以下载它。
    - 由于XFS比较复杂，实施起来有一些难度（包括人员培训等），所以目前XFS主要应用于Linux企业应用的高端。

1.3 Linux主要的文件系统

- ext4 - Linux kernel 自 2.6.28 开始正式支持新的文件系统ext4。ext4是ext3的改进版，修改了ext3中部分重要的数据结构，而不仅仅像ext3对ext2 那样，只是增加了一个日志功能而已。ext4可以提供更佳的性能和可靠性，还有更为丰富的功能：
  - 与ext3兼容。执行若干条命令，就能从ext3在线迁移到ext4，而无须重新格式化磁盘或重新安装系统。原有ext3数据结构照样保留，ext4作用于新数据，当然，整个文件系统因此也就获得了ext4所支持的更大容量。
  - 更大的文件系统和更大的文件。较之ext3目前所支持的最大 16TB 文件系统和最大 2TB 文件，ext4分别支持 1EB（1,048,576TB，1EB=1024PB， 1PB=1024TB）的文件系统，以及 16TB 的文件。
  - 无限数量的子目录。ext3目前只支持 32,000 个子目录，而ext4支持无限数量的子目录。
  - extents。ext3采用间接块映射，当操作大文件时，效率极其低下。比如一个 100MB 大小的文件，在ext3中要建立 25,600 个数据块（每个数据块大小为 4KB）的映射表。而ext4引入了现代文件系统中流行的extents 概念，每个extent 为一组连续的数据块，上述文件则表示为“该文件数据保存在接下来的 25,600 个数据块中”，提高了不少效率。
  - 多块分配。当写入数据到ext3文件系统中时，ext3的数据块分配器每次只能分配一个 4KB 的块，写一个 100MB 文件就要调用 25,600 次数据块分配器，而ext4的多块分配器“multiblock allocator”（mballoc） 支持一次调用分配多个数据块。
  - 延迟分配。ext3的数据块分配策略是尽快分配，而ext4和其它现代文件操作系统的策略是尽可能地延迟分配，直到文件在 cache 中写完才开始分配数据块并写入磁盘，这样就能优化整个文件的数据块分配，与前两种特性搭配起来可以显著提升性能。
  - 快速 fsck。以前执行 fsck 第一步就会很慢，因为它要检查所有的 inode，现在ext4给每个组的 inode 表中都添加了一份未使用 inode 的列表，今后 fsckext4文件系统就可以跳过它们而只去检查那些在用的 inode 了。
  - 日志校验。日志是最常用的部分，也极易导致磁盘硬件故障，而从损坏的日志中恢复数据会导致更多的数据损坏。ext4的日志校验功能可以很方便地判断日志数据是否损坏，而且它将ext3的两阶段日志机制合并成一个阶段，在增加安全性的同时提高了性能。
  - “无日志”（No Journaling）模式。日志总归有一些开销，ext4允许关闭日志，以便某些有特殊需求的用户可以借此提升性能。
  - 在线碎片整理。尽管延迟分配、多块分配和extents 能有效减少文件系统碎片，但碎片还是不可避免会产生。ext4支持在线碎片整理，并将提供 e4defrag 工具进行个别文件或整个文件系统的碎片整理。
  - inode 相关特性。ext4支持更大的 inode，较之ext3默认的 inode 大小 128 字节，ext4为了在 inode 中容纳更多的扩展属性（如纳秒时间戳或 inode 版本），默认 inode 大小为 256 字节。ext4还支持快速扩展属性（fastextended attributes）和 inode 保留（inodes reservation）。
  - 持久预分配（Persistent preallocation）。P2P 软件为了保证下载文件有足够的空间存放，常常会预先创建一个与所下载文件大小相同的空文件，以免未来的数小时或数天之内磁盘空间不足导致下载失败。ext4在文件系统层面实现了持久预分配并提供相应的 API（libc 中的 posix_fallocate()），比应用软件自己实现更有效率。
  - 默认启用 barrier。磁盘上配有内部缓存，以便重新调整批量数据的写操作顺序，优化写入性能，因此文件系统必须在日志数据写入磁盘之后才能写 commit 记录，若 commit 记录写入在先，而日志有可能损坏，那么就会影响数据完整性。ext4默认启用 barrier，只有当 barrier 之前的数据全部写入磁盘，才能写 barrier 之后的数据。（可通过 "mount -o barrier=0" 命令禁用该特性。）

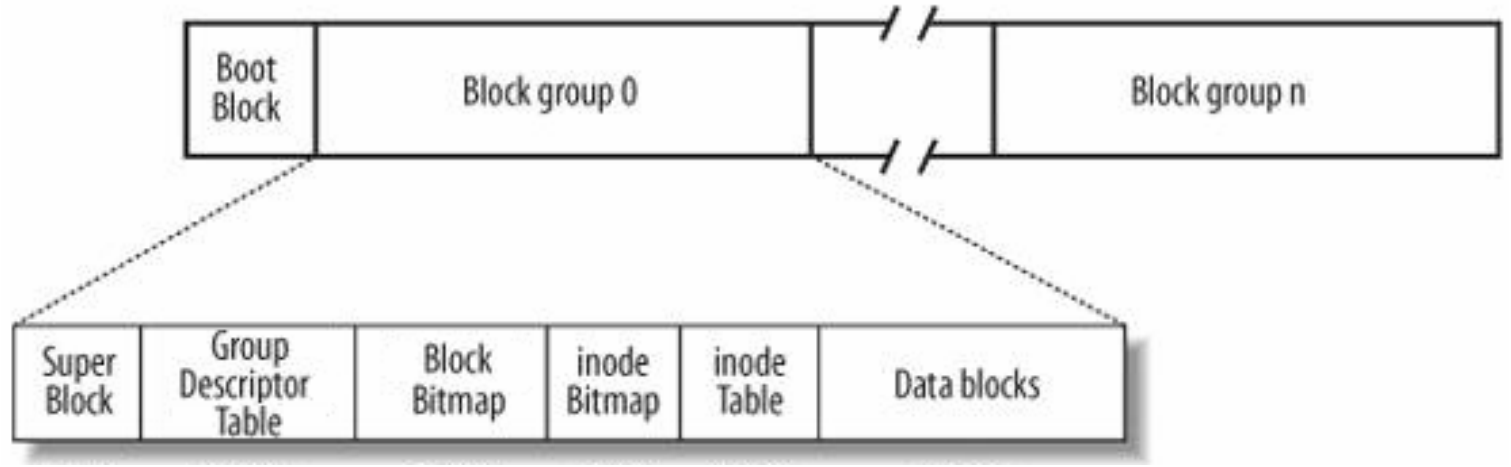
1.4 其他文件系统

- Xia 是Minix文件系统修正后的版本，在一定程度上解决了文件名和文件系统大小的局限。但是没有新的特色，目前很少有人使用。
- ISO9660 标准CDROM文件系统，通用的Rock Ridge增强系统，允许长文件名。
- NFS Sun公司推出的网络文件系统，允许多台计算机之间共享同一文件系统，易于从所有这些计算机上存取文件。
- SysV 是System V/Coherent在Linux平台上的文件系统。
- UMSDOS Linux下的扩展MSDOS文件系统驱动，支持长文件名、所有者、允许权限、连接和设备文件。允许一个普通的MSDOS文件系统用于Linux，而且无须为它建立单独的分区。
- MSDOS 是在DOS、Windows和某些OS/2操作系统上使用的一种文件系统，其名称采用“8+3”的形式，即8个字符的文件名加上3个字符的扩展名。
- VFAT 是Windows 9x和Windows NT/2000下使用的一种DOS文件系统，其在DOS文件系统的基础上增加了对长文件名的支持。
- HPFS 高性能文件系统（High Performance File System，HPFS）是微软的LAN Manager中的文件系统，同时也是IBM的LAN Server和OS／2的文件系统。HPFT能访问较大的硬盘驱动器，提供了更多的组织特性，并改善了文件系统的安全特性。
- SMB 是一种支持Windows for Workgroups、Windows NT和Lan Manager的基于SMB协议的网络操作系统。
- NCPFS 是一种Novell NetWare使用的NCP协议的网络操作系统。
- NTFS 是Windows NT/2000操作系统支持的、一个特别为网络和磁盘配额、文件加密等管理安全特性设计的磁盘格式。

2. ext2文件系统

2.1 总体存储布局

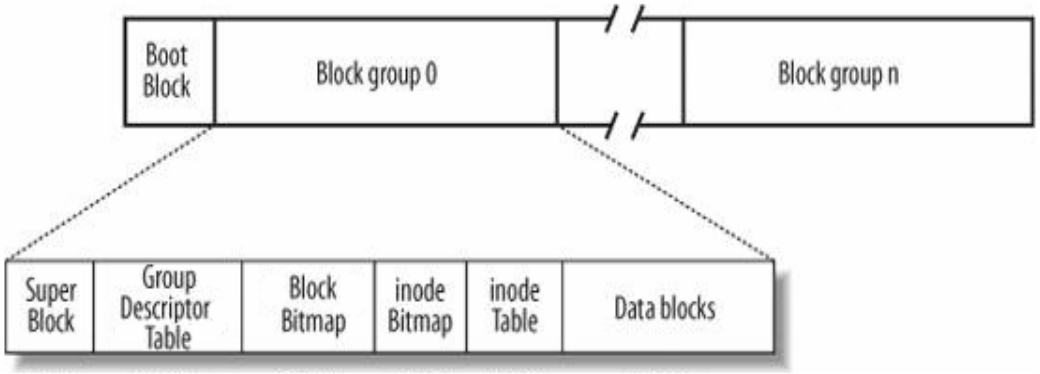
- 一个磁盘可以划分成多个分区，每个分区必须先用格式化工具（例如某种mkfs 命令）格式化成某种格式的文件系统，然后才能存储文件，格式化的过程会在磁盘上写一些管理存储布局的信息。
- 下图是一个磁盘分区格式化成ext2 文件系统后的存储布局



- 文件系统中存储的最小单位是块（Block），一个块究竟多大是在格式化时确定的，例如mke2fs 的-b 选项可以设定块大小为1024、2048 或4096 字节。而上图中启动块（Boot Block）的大小是确定的，就是1KB，启动块是由PC 标准规定的，用来存储磁盘分区信息和启动信息，任何文件系统都不能使用启动块。启动块之后才是ext2 文件系统的开始，ext2 文件系统将整个分区划成若干个同样大小的块组（Block Group），每个块组都由以下部分组成。

2.1.1 ext2文件系统的磁盘组织

- 除了引导扇区之外，EXT2磁盘分区被顺序划分为若干个磁盘块组（Block Group）。
- 每个块组由若干个磁盘块，按照相同的方式组织，具有相同的大小。
- ext2磁盘块组中的磁盘块按顺序被组织成：
  - 一个用作超级块的磁盘块。
    - 在这个磁盘块里，存放了文件系统超级块的一个拷贝；
  - N个记录组描述符的磁盘块；
  - 1个记录数据块位图的磁盘块；
  - 1个记录索引结点位图的磁盘块；
  - N个用作索引结点表的磁盘块；
  - N个用作数据块的磁盘块。



2.1.2 总体存储布局

- 超级块（Super Block）
  - 描述整个分区的文件系统信息，例如块大小、文件系统版本号、上次mount 的时间等等。超级块在每个块组的开头都有一份拷贝。
- 块组描述符表（GDT，Group Descriptor Table）
  - 由很多块组描述符组成，整个分区分成多少个块组就对应有多少个块组描述符。每个块组描述符（Group Descriptor）存储一个块组的描述信息，例如在这个块组中从哪里开始是inode 表，从哪里开始是数据块，空闲的inode 和数据块还有多少个等等。和超级块类似，块组描述符表在每个块组的开头也都有一份拷贝，这些信息是非常重要的，一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。通常内核只用到第0 个块组中的拷贝，当执行e2fsck 检查文件系统一致性时，第0 个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第0 个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。
- 块位图（Block Bitmap）
  - 块位图就是用来描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个bit 代表本块组中的一个块，这个bit 为1 表示该块已用，这个bit 为0 表示该块空闲可用。
- inode位图（inode Bitmap）
  - 和块位图类似，本身占一个块，其中每个bit 表示一个inode 是否空闲可用

2.1.2 总体存储布局(cont.)

- inode表（inode Table）
  - 我们知道，一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类型（常规、目录、符号链接等），权限，文件大小，创建/修改/访问时间等，也就是*ls -l*命令看到的那些信息，这些信息存在inode中而不是数据块中。每个文件都有一个inode，一个块组中的所有inode组成了inode表。inode表占多少个块在格式化时就要决定并写入块组描述符中，mke2fs格式化工具的默认策略是一个块组有多少个8KB 就分配多少个inode。由于数据块占了整个块组的绝大部分，也可以近似认为数据块有多少个8KB就分配多少个inode，换句话说，如果平均每个文件的大小是8KB，当分区存满的时候inode表会得到比较充分的利用，数据块也不浪费。如果这个分区存的都是很大的文件（比如电影），则数据块用完的时候inode会有一些浪费，如果这个分区存的都是很小的文件（比如源代码），则有可能数据块还没用完inode就已经用完了，数据块可能有很大的浪费。如果用户在格式化时能够对这个分区以后要存储的文件大小做一个预测，也可以用mke2fs 的-i 参数手动指定每多少个字节分配一个inode。
- 数据块（Data Blocks）
  - 根据不同的文件类型有以下几种情况
    - 对于常规文件，文件的数据存储在数据块中。
    - 对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外，ls -l 命令看到的其它信息都保存在该文件的inode 中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。
    - 对于符号链接，如果目标路径名较短则直接保存在 inode 中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。
    - 设备文件、FIFO 和socket 等特殊文件没有数据块，设备文件的主设备号和次设备号保存在inode 中。

2.2 实例剖析

- Step 1, 如果要格式化一个分区来研究文件系统格式则必须有一个空闲的磁盘分区，为了方便实验，我们把一个文件当作分区来格式化，然后分析这个文件中的数据来印证前 面所讲的要点。首先创建一个1MB的文件并清零：

```
[yjs@boss tmp]$ dd if=/dev/zero of=fs count=256 bs=4K
256+0 records in
256+0 records out
1048576 bytes (1.0 MB) copied, 0.00635302 seconds, 165 MB/s
[yjs@boss tmp]$ ls -al fs
-rw-r--r-- 1 yjs boss 1048576 Dec 26 02:28 fs
```

- Step 2,对文件fs进行格式化，也就是把这个文件的数据块合起来看成一个1MB的磁盘分区，在这个分区上再划分出块组。

```
[yjs@boss tmp]$ /sbin/mke2fs fs
mke2fs 1.39 (29-May-2006)
fs is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
128 inodes, 1024 blocks
51 blocks (4.98%) reserved for the super user
First data block=1
Maximum filesystem blocks=1048576
1 block group
8192 blocks per group, 8192 fragments per group
128 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 27 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

2.2 实例剖析

- Step 3, 用*dumpe2fs*工具可以查看这个分区的超级块和块组描述符表中的信息

```
[yjs@boss tmp]$ /sbin/dumpe2fs fs
dumpe2fs 1.39 (29-May-2006)
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          03be43ce-e301-4e02-a0cf-1ec45351de36
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      resize_inode dir_index filetype sparse_super
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              128
Block count:              1024
Reserved block count:     51
Free blocks:              986
Free inodes:              117
First block:              1
Block size:               1024
Fragment size:            1024
Reserved GDT blocks:      3
Blocks per group:         8192
Fragments per group:      8192
Inodes per group:         128
Inode blocks per group:   16
Filesystem created:       Sat Dec 26 02:32:29 2009
Last mount time:          n/a
Last write time:          Sat Dec 26 02:32:29 2009
Mount count:              0
Maximum mount count:      27
Last checked:             Sat Dec 26 02:32:29 2009
Check interval:           15552000 (6 months)
Next check after:         Thu Jun 24 02:32:29 2010
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Default directory hash:   tea
Directory Hash Seed:      9bdd19ac-3689-4906-9f58-8cdca7093394


Group 0: (Blocks 1-1023)
  Primary superblock at 1, Group descriptors at 2-2
  Reserved GDT blocks at 3-5
  Block bitmap at 6 (+5), Inode bitmap at 7 (+6)
  Inode table at 8-23 (+7)
  986 free blocks, 117 free inodes, 2 directories
  Free blocks: 38-1023
  Free inodes: 12-128
```

2.2 实例剖析

- Step 4, 用常规文件制作而成的文件系统也可以像磁盘分区一样mount到某个目录

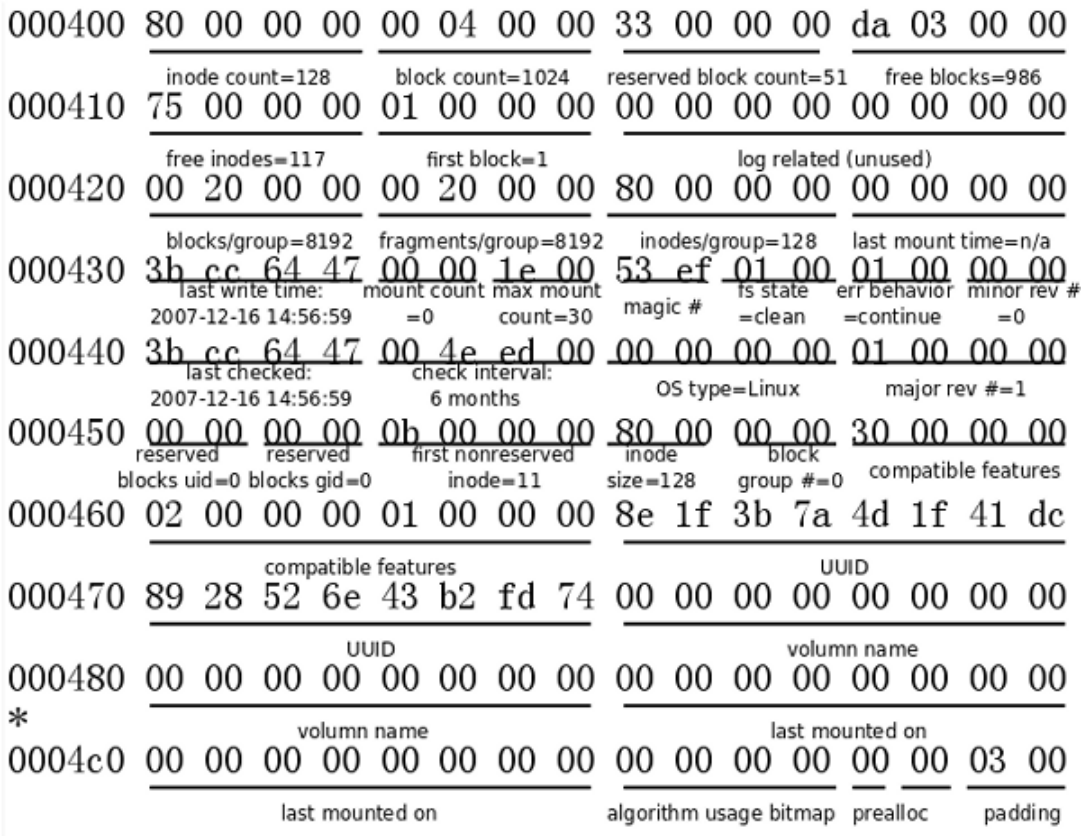
```
[yjs@boss tmp]$ mkdir test_mount_point
[yjs@boss tmp]$ mount -o loop ds test_mount_point/
mount: only root can do that
[yjs@boss tmp]$ su -
Password:
[root@boss ~]# cd /home/yjs/work/akae/20091226/tmp/
[root@boss tmp]# mount -o loop fs test_mount_point/
[root@boss tmp]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda2             3.9G  292M  3.4G   8% /
/dev/hda7              20G   14G  5.2G  72% /data
tmpfs                 506M     0  506M   0% /dev/shm
/dev/hda6              39G   31G  6.5G  83% /home
/dev/hda11             15G   13G  1.6G  89% /mnt/exchange
/dev/hda5              39G   30G  6.8G  82% /opt
/dev/hda9             494M   11M  458M   3% /tmp
/dev/hda8             7.8G  3.5G  3.9G  48% /usr
/dev/hda10            494M  346M  123M  74% /var
/home/yjs/work/akae/20091226/tmp/fs
                      1003K   17K  935K   2% /home/yjs/work/akae/20091226/tmp/test_mount_point
```

- Step 5, 查看目录

```
[yjs@boss tmp]$ cd test_mount_point/
[yjs@boss test_mount_point]$ ls -al
total 21
drwxr-xr-x 3 yjs  boss  1024 Dec 26 02:32 .
drwxr-xr-x 3 yjs  boss  4096 Dec 26 02:39 ..
drwx------ 2 root root 12288 Dec 26 02:32 lost+found
```

2.2 实例剖析-超级块

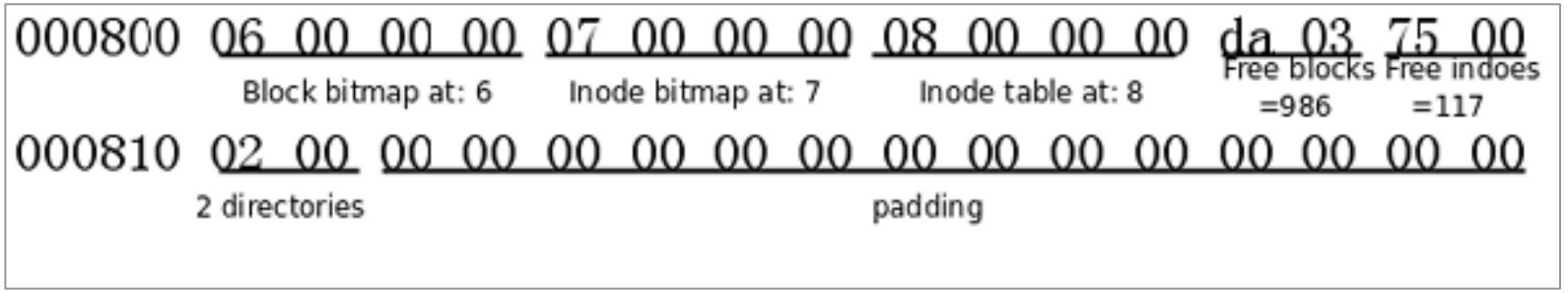
- Step 5, 用二进制查看工具查看这个文件系统的所有字节，并且同*dumpe2fs*工具的输出信息相比较，就可以很好地理解文件系统的存储布局了。
  - 从000000开始的1KB是启动块，由于这不是一个真正的磁盘分区，启动块的内容全部为零。
  - 从000400到0007ff的1KB是超级块，对照着*dumpe2fs*的输出信息，详细分析如下：



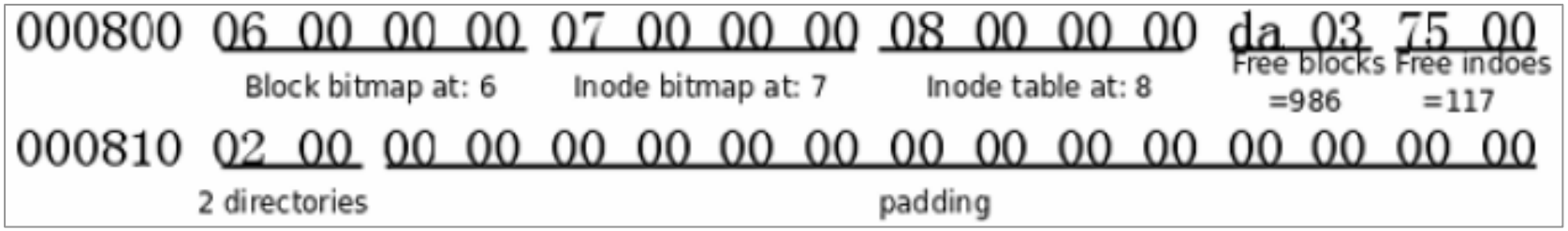
2.2 实例剖析-块组描述符

- 超级块中从0004d0到末尾的204个字节是填充字节，保留未用
- 从000800开始是块组描述符表，这个文件系统较小，只有一个块组描述符，对照着*dumpe2fs*的输出信息如下：

```
...
Group 0: (Blocks 1-1023)
  Primary superblock at 1, Group descriptors at 2-2
  Reserved GDT blocks at 3-5
  Block bitmap at 6 (+5), Inode bitmap at 7 (+6)
  Inode table at 8-23 (+7)
  986 free blocks, 117 free inodes, 2 directories
  Free blocks: 38-1023
  Free inodes: 12-128
...
```



2.2 实例剖析-块组描述符



- 整个文件系统是1MB，每个块是1KB，应该有1024个块，除去启动块还有1023个块，分别编号为1-1023，它们全都属于Group 0。
  - Block 1：超级块
  - Block 6：块位图
  - Block 2：块组描述符表
  - Block 3-5：保留未用
  - Block 7：inode位图
  - Block 8-23：inode表
    - inode表是从Block 8开始的，到Block 23结束（共16个块)
      - 由于超级块中指出每个块组有128个inode，每个inode的大小是128字节，因此共占16个块，inode表的范围是Block 8-23。
  - Block 24-1023：数据块
    - 空闲的数据块有986个
    - 空闲的inode有117个
    - 目录有2个

2.2 实例剖析-块位图(Block bitmap)

- Block 6是块位图
  - 从块位图中可以看出，前37位（前4个字节加最后一个字节的低5位）都是1，就表示Block 1-37已用
  - 在块位图中，Block 38-1023对应的位都是0（一直到001870那一行最后一个字节的低7位），接下来的位已经超出了文件系统的空间，不管是0还是1都没有意义
  - 块位图每个字节中的位应该按从低位到高位的顺序来看
  - 以后随着文件系统的使用和添加删除文件，块位图中的1就变得不连续了
- Block 24开始就是数据块
- 块组描述符中指出，空闲的数据块有986个，由于文件系统是新创建的，空闲块是连续的Block 38-1023，用掉了前面的Block 24-37。

```
001800 ff ff ff ff 1f 00 00 00 00 00 00 00 00 00 00 00
001810 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001870 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80
001880 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
*
```

2.2 实例剖析- inode bitmap

- Block 7是inode bitmap
  - 从inode位图可以看出，前11位都是1，表示前11个inode已用
  - 001c00这一行的128位就表示了所有inode，因此下面的行不管是0还是1都没有意义
  - 已用的11个inode中，前10个inode是被ext2文件系统保留的
    - 其中第2个inode是根目录，
    - 第11个inode是lost+found目录
  - 随着文件系统的使用和添加删除文件，inode位图中的1就变得不连续了
- 块组描述符指出，空闲的inode有117个，由于文件系统是新创建的，空闲的inode也是连续的，inode编号从1到128，空闲的inode编号从12到128。
- 块组描述符也指出该组有两个目录，就是根目录和lost+found

```
001c00 ff 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001c10 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
*
```

2.2 debugfs

- 探索文件系统还有一个很有用的工具debugfs，它提供一个命令行界面，可以对文件系统做各种操作，例如查看信息、恢复数据、修正文件系统中的错误。
- 使用debugfs打开fs文件，然后在提示符下输入help看看它都能做哪些事情：

```
$ debugfs fs
debugfs 1.40.2 (12-Jul-2007)
debugfs:  help
```

- 在debugfs的提示符下输入stat /命令，这时在新的一屏中显示根目录的inode信息：

```
Inode: 2   Type: directory    Mode:  0755   Flags: 0x0   Generation: 0
User:  1000   Group:  1000   Size: 1024
File ACL: 0    Directory ACL: 0
Links: 3   Blockcount: 2
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x4764cc3b -- Sun Dec 16 14:56:59 2007
atime: 0x4764cc3b -- Sun Dec 16 14:56:59 2007
mtime: 0x4764cc3b -- Sun Dec 16 14:56:59 2007
BLOCKS:
(0):24
TOTAL: 1
```

- 按q退出信息输出界面，然后用quit命令退出debugfs

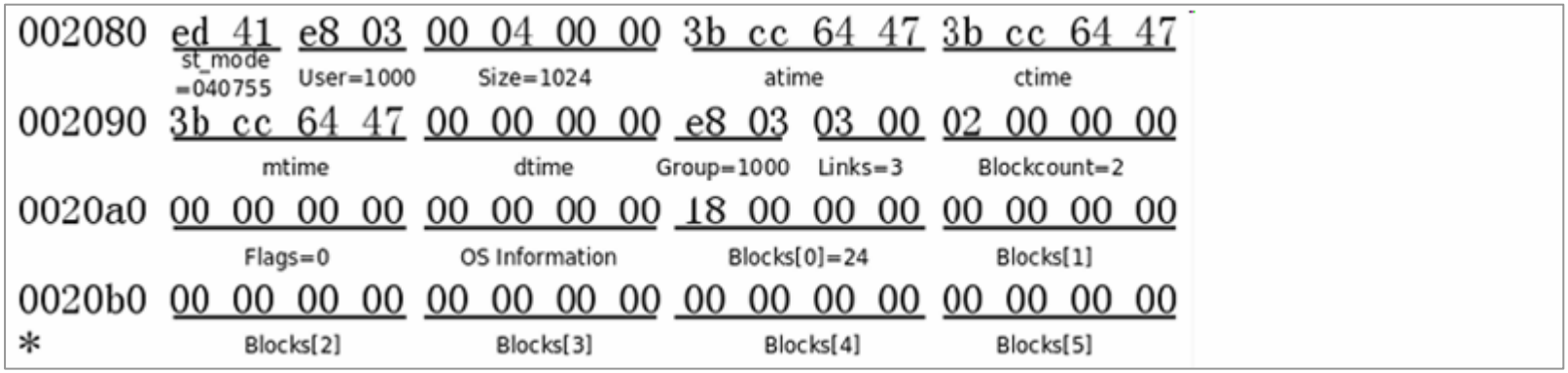
```
debugfs:  quit
```

- debugfs提供了cd、ls等命令，不需要mount就可以查看文件系统中的目录，例如用ls查看根目录：

```
2  (12) .    2  (12) ..    11  (1000) lost+found
```

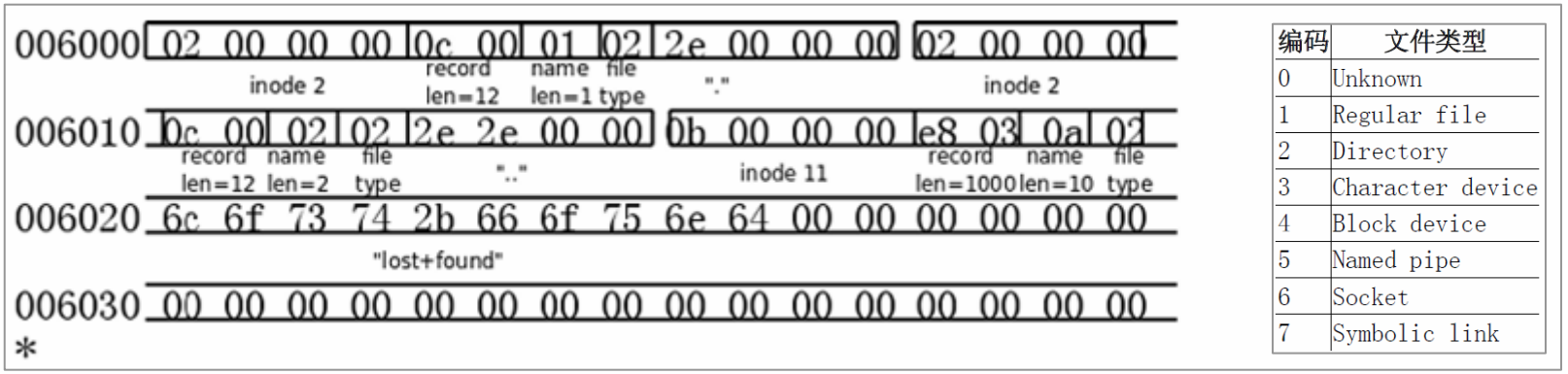
2.2 实例剖析-块组描述符根目录的inode

- st_mode以八进制表示，包含了文件类型和文件权限，最高位的4表示文件类型为目录（各种文件类型的编码详见stat(2)），低位的755表示权限。
- Size是1024，说明根目录现在只有一个数据块
- Links为3表示根目录有3个硬链接，分别是:
  - 根目录下的.
  - 根目录下的..
  - lost+found子目录下的..
  - 注意，虽然我们通常用/表示根目录，但是并没有名为/的硬链接，事实上，/是路径分隔符，不能在文件名中出现。
- Blockcount是以512字节为一个块来数的，并非格式化文件系统时所指定的块大小，磁盘的最小读写单位称为扇区（Sector），通常是512字节，所以Blockcount是磁盘的物理块数量，而非分区的逻辑块数量。



2.2 实例剖析-块组描述符-根目录的数据块

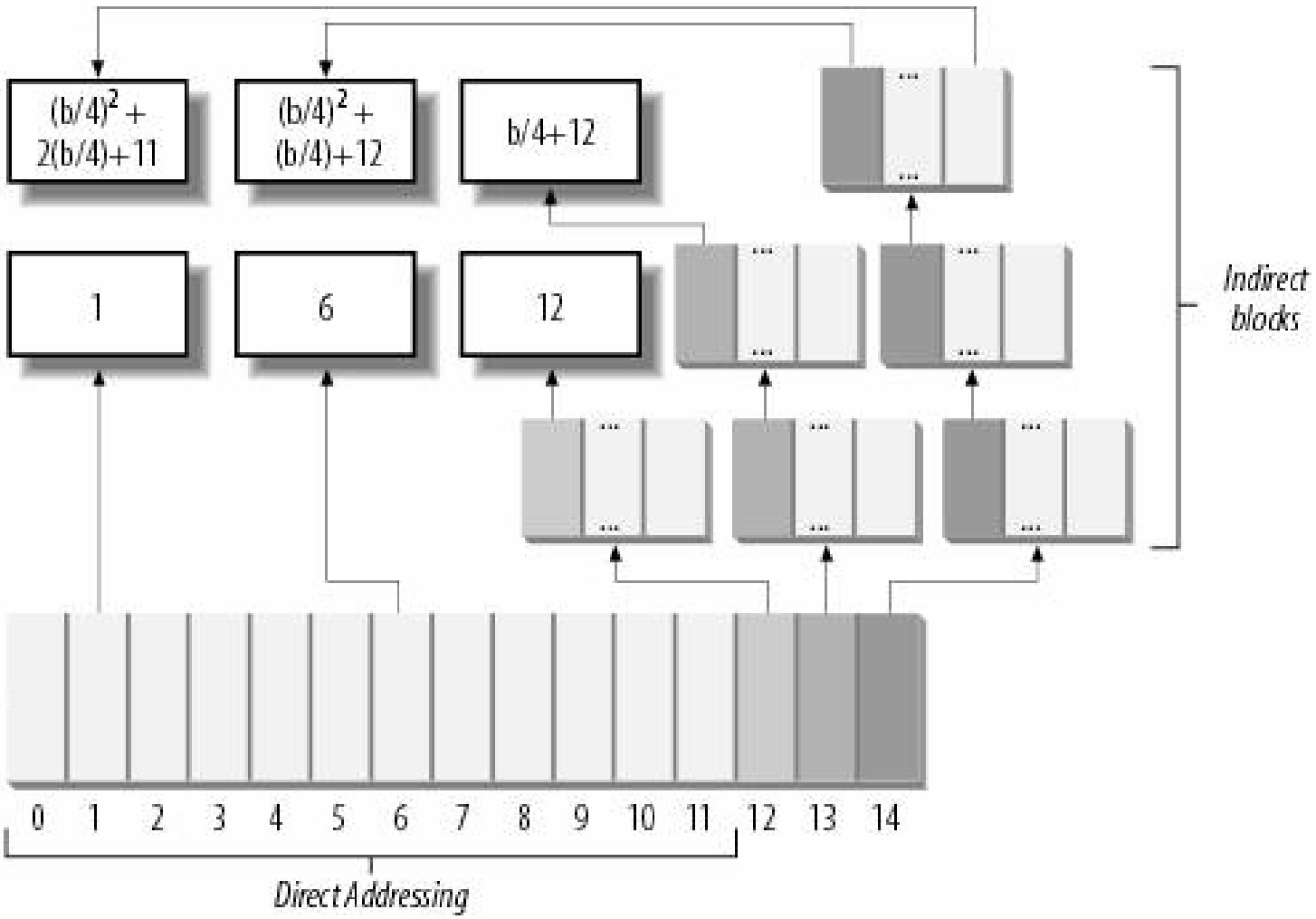
- 根目录数据块的位置由根目录inode信息中的Blocks[0]指出，也就是第24个块，它在文件系统中的位置是24*0x400=0x6000，从od命令的输出中找到006000地址，它的格式是这样：



编码	文件类型
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

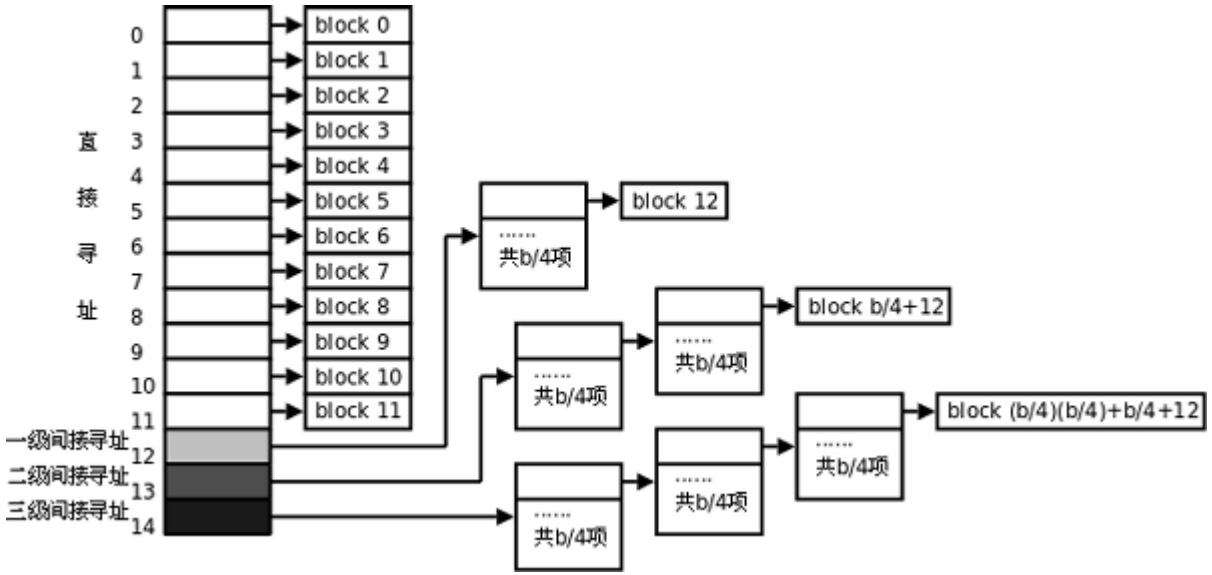
- 目录的数据块由许多不定长的记录组成，每条记录描述该目录下的一个文件
  - 第一条记录描述inode号为2的文件，也就是根目录本身，该记录的总长度为12字节，其中文件名的长度为1字节，文件类型为2，文件名是.
  - 第二条记录也是描述inode号为2的文件（根目录），该记录总长度为12字节，其中文件名的长度为2字节，文件类型为2，文件名字符串是..
  - 第三条记录一直延续到该数据块的末尾，描述inode号为11的文件（lost+found目录），该记录的总长度为1000字节（和前面两条记录加起来是1024字节），文件类型为2，文件名字符串是lost+found，后面全是0字节。
  - 如果要在根目录下创建新的文件，可以把第三条记录截短，在原来的0字节处创建新的记录。
  - 如果该目录下的文件名太多，一个数据块不够用，则会分配新的数据块，块编号会填充到inode的Blocks[1]字段。

2.3 数据块寻址


(b/4)² + 2(b/4)+11
(b/4)² + (b/4)+12
b/4+12
1
6
12
Indirect blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Direct Addressing

2.3.1 数据块寻址流程

- 如果一个文件有多个数据块，这些数据块很可能不是连续存放的，应该如何寻址到每个块呢？根据上面的分析，根目录的数据块是通过其inode中的索引项Blocks[0]找到的，事实上，这样的索引项一共有15个，从Blocks[0]到Blocks[14]，每个索引项占4字节。前12个索引项都表示块编号，例如上面的例子中Blocks[0]字段保存着24，就表示第24个块是该文件的数据块，如果块大小是1KB，这样可以表示从0字节到12KB的文件。如果剩下的三个索引项Blocks[12]到Blocks[14]也是这么用的，就只能表示最大15KB的文件了，这是远远不够的，事实上，剩下的三个索引项都是间接索引。
- 索引项Blocks[12]所指向的块并非数据块，而是称为间接寻址块（Indirect Block），其中存放的都是类似Blocks[0]这种索引项，再由索引项指向数据块。设块大小是b，那么一个间接寻址块中可以存放b/4个索引项，指向b/4个数据块。所以如果把Blocks[0]到Blocks[12]都用上，最多可以表示b/4+12个数据块，对于块大小是1K的情况，最大可表示268K的文件。如下图所示，注意文件的数据块编号是从0开始的，Blocks[0]指向第0个数据块，Blocks[11]指向第11个数据块，Blocks[12]所指向的间接寻址块的第一个索引项指向第12个数据块，依此类推。



- 从上图可以看出，索引项Blocks[13]指向两级的间接寻址块，最多可表示(b/4)2+b/4+12个数据块，对于1K的块大小最大可表示64.26MB的文件。索引项Blocks[14]指向三级的间接寻址块，最多可表示(b/4)3+(b/4)2+b/4+12个数据块，对于1K的块大小最大可表示16.06GB的文件。
- 可见，这种寻址方式对于访问不超过12个数据块的小文件是非常快的，访问文件中的任意数据只需要两次读盘操作，一次读inode（也就是读索引项）一次读数 据块。而访问大文件中的数据则需要最多五次读盘操作：inode、一级间接寻址块、二级间接寻址块、三级间接寻址块、数据块。实际上，磁盘中的inode 和数据块往往已经被内核缓存了，读大文件的效率也不会太低。

2.4.1 文件和目录操作的系统函数

- stat(2)/fstat(2)/lstat(2)
  - stat(2)函数读取文件的inode，然后把inode 中的各种文件属性填入一个struct stat 结构体传出给调用者。
  - stat(1)命令是基于stat 函数实现的
  - stat 需要根据传入的文件路径找到inode，假设一个路径是/opt/file，则查找的顺序是：
    1. 读出 inode 表中第2 项，也就是根目录的inode，从中找出根目录数据块的位置
    2. 从根目录的数据块中找出文件名为 opt 的记录，从记录中读出它的inode 号
    3. 读出 opt 目录的inode，从中找出它的数据块的位置
    4. 从 opt 目录的数据块中找出文件名为file 的记录，从记录中读出它的inode 号
    5. 读出 file 文件的inode
  - fstat(2)函数传入一个已打开的文件描述符，传出inode 信息，
  - lstat(2)函数也是传入路径传出inode 信息，但是和stat 函数有一点不同，当文件是一个符号链接时，stat(2)函数传出的是它所指向的目标文件的inode，而lstat 函数传出的就是符号链接文件本身的inode。
- access(2)
  - assess(2)函数检查执行当前进程的用户是否有权限访问某个文件，传入文件路径和要执行的访问操作（读/写/执行），access 函数取出文件inode 中的st_mode 字段，比较一下访问权限，然后返回0 表示允许访问，返回-1 表示错误或不允许访问。

2.4.2 文件和目录操作的系统函数

- chmod(2)/fchmod(2)
  - chmod(2)和fchmod(2)函数改变文件的访问权限，也就是修改inode 中的st_mode 字段。这两个函数的区别类似于stat/fstat。chmod(1)命令是基于chmod 函数实现的。
- chown(2)/fchown(2)/lchown(2)
  - chown(2)/fchown(2)/lchown(2)改变文件的所有者和组，也就是修改inode 中的User 和Group 字段，只有超级用户才能正确调用这几个函数，这几个函数之间的区别类似于stat/fstat/lstat。chown(1)命令是基于chown 函数实现的。
- utime(2)
  - utime(2)函数改变文件的访问时间和修改时间，也就是修改inode 中的atime和mtime 字段。touch(1)命令是基于utime 函数实现的。
- truncate(2)/ftruncate(2)
  - truncate(2)和ftruncate(2)函数把文件截断到某个长度，如果新的长度比原来的长度短，则后面的数据被截掉了，如果新的长度比原来的长度长，则后面多出来的部分用0 填充，这需要修改inode 中的Blocks 索引项以及块位图中相应的bit。这两个函数的区别类似于stat/fstat。
- link(2)/symlink(2)
  - link(2)函数创建硬链接，其原理是在目录的数据块中添加一条新记录，其中的inode 号字段和原文件相同。symlink(2)函数创建一个符号链接，这需要创建一个新的inode，其中st_mode 字段的文件类型是符号链接，原文件的路径保存在inode 中或者分配一个数据块来保存。ln(1)命令是基于link 和symlink函数实现的。
- unlink(2)
  - unlink(2)函数删除一个链接。如果是符号链接则释放这个符号链接的inode 和数据块，清除inode 位图和块位图中相应的位。如果是硬链接则从目录的数据块中清除一条文件名记录，如果当前文件的硬链接数已经是1 了还要删除它，就同时释放它的inode 和数据块，清除inode 位图和块位图中相应的位，这样就真的删除文件了。unlink(1)命令和rm(1)命令是基于unlink 函数实现的。
- rename(2)
  - rename(2)函数改变文件名，需要修改目录数据块中的文件名记录，如果原文件名和新文件名不在一个目录下则需要从原目录数据块中清除一条记录然后添加到新目录的数据块中。mv(1)命令是基于rename 函数实现的，因此在同一分区的不同目录中移动文件并不需要复制和删除文件的inode 和数据块，只需要一个改名操作，即使要移动整个目录，这个目录下有很多子目录和文件也要随着一起移动，移动操作也只是对顶级目录的改名操作，很快就能完成。但是，如果在不同的分区之间移动文件就必须复制和删除inode 和数据块，如果要移动整个目录，所有子目录和文件都要复制删除，这就很慢了。
- readlink(2)
  - readlink(2)函数读取一个符号链接所指向的目标路径，其原理是从符号链接的inode 或数据块中读出保存的数据，这就是目标路径。

2.4.3 文件和目录操作的系统函数

- mkdir(2)
  - mkdir(2)函数创建新的目录，要做的操作是在它的父目录数据块中添加一条记录，然后分配新的inode 和数据块，inode 的st_mode 字段的文件类型是目录，在数据块中填两个记录，分别是.和..，由于..表示父目录，因此父目录的硬链接数要加1。
  - mkdir(1)命令是基于mkdir()函数实现的。
- rmdir(2)
  - rmdir(2)函数删除一个目录，这个目录必须是空的（只包含.和..）才能删除，要做的操作是释放它的inode 和数据块，清除inode 位图和块位图中相应的位，清除父目录数据块中的记录，父目录的硬链接数要减1。
  - rmdir(1)命令是基于rmdir()函数实现的。
- opendir(3)/readdir(3)/closedir(3)
  - opendir(3)/readdir(3)/closedir(3)用于遍历目录数据块中的记录。opendir打开一个目录，返回一个DIR *指针代表这个目录，它是一个类似FILE *指针的句柄，closedir 用于关闭这个句柄，把DIR *指针传给readdir 读取目录数据块中的记录，每次返回一个指向struct dirent 的指针，反复读就可以遍历所有记录，所有记录遍历完之后readdir 返回NULL。

3. VFS

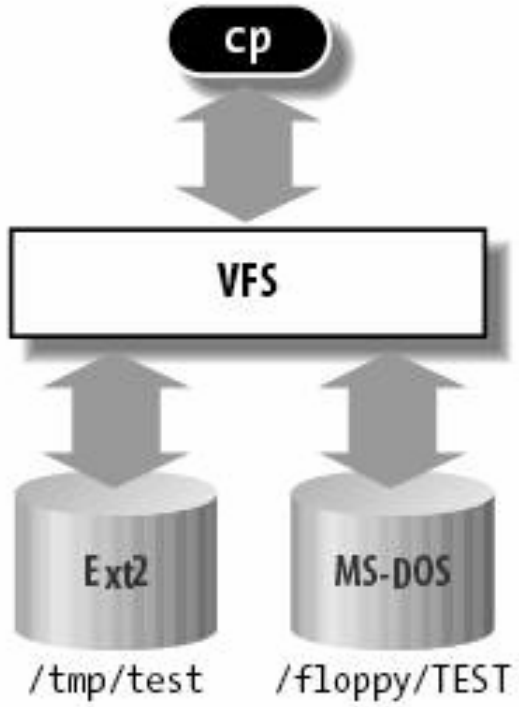
3.1 虚拟文件系统VFS的作用

- 虚拟文件系统
  - Virtual Filesystem
  - Virtual Filesystem Switch
  - VFS是一个软件层，用来处理与Unix标准文件系统相关的所有系统调用。
    - 是用户应用程序与文件系统实现之间的抽象层
  - 能为各种文件系统提供一个通用的、统一的接口
- Linux与其他类Unix系统一样，采用虚拟文件系统VFS来达到支持多种文件系统格式的目标

3.1 VFS在一个简单文件复制操作中的作用

- 假设用户输入以下shell命令
  - $ cp /floppy/TEST /tmp/test
- 其中，
  - /floppy是MS-DOS的磁盘的一个挂载点（安装点）
  - /tmp是ext2文件系统中的一个目录
- 对于cp命令而言，它不需要知道/floppy/TEST和/tmp/test分别是什么文件系统类型
  - 在cp命令中，它通过VFS提供的系统调用接口进行文件操作

Figure 12-1. VFS role in a simple file copy operation



(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

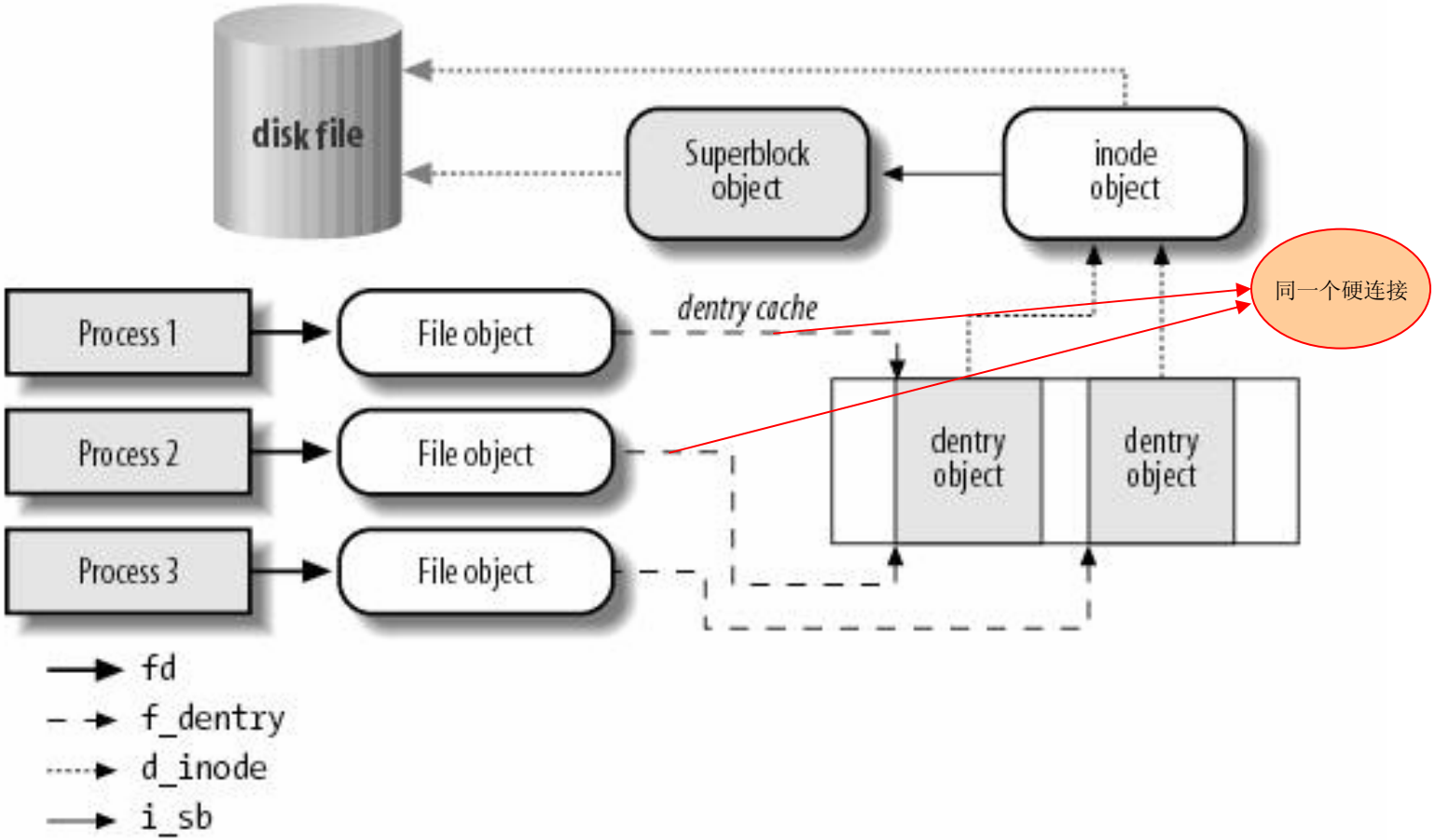
3.2 VFS支持的文件系统类型

- VFS支持的文件系统可以划分为三种主要类型
  - 基于磁盘的文件系统：它们管理在本地磁盘分区中可用的存储空间
    - Linux使用的文件系统：ext2、ext3、ReiserFS
    - Unix家族的文件系统：SYSV文件系统，UFS，MINIX文件系统以及VERITAS VxFS
    - 微软公司的文件系统：MS-DOS、VFAT以及NTFS
    - ISO9660CD-ROM文件系统和通用磁盘格式的DVD文件系统
    - 其他有专利权的文件系统，如HPFS、HFS、AFFS、ADFS
    - 起源于非Linux系统的其他日志文件系统，JFS，XFS
  - 网络文件系统：用于访问属于其他网络计算机的文件系统所包含的文件
    - NFS、Coda、AFS、SMB、NCP
  - 特殊文件系统
    - 不同于上述两大类
    - 不管理具体的磁盘空间
    - /proc
- 各种不同的文件系统通过mount（挂载、安装）到根文件系统中
  - 在Linux中，根文件系统即根目录所代表的文件系统
  - 通常是ext2文件系统

3.3 VFS中通用文件模型概念

- VFS的基本思想：引入一个通用文件模型，这个模型能够表示所有支持的文件系统
  - 对于一个具体实现的文件系统，在处理时，需要将其进行概念上的转换
    - 例如，在通用文件模型中，目录被看成是普通文件
  - 在实现上，read()->sys_read->file数据结构->f_op->MS_DOS文件操作指针（其中的read()操作）
  - 类似面向对象的概念
- 通用文件模型有下列对象类型组成
  - 超级块对象（superblock object）
    - 存放文件系统相关信息：例如文件系统控制块
  - 索引节点对象（inode object）
    - 存放具体文件的一般信息：文件控制块/inode
  - 文件对象（file object）
    - 存放已打开的文件和进程之间交互的信息
  - 目录项对象（dentry object）
    - 存放目录项与文件的链接信息

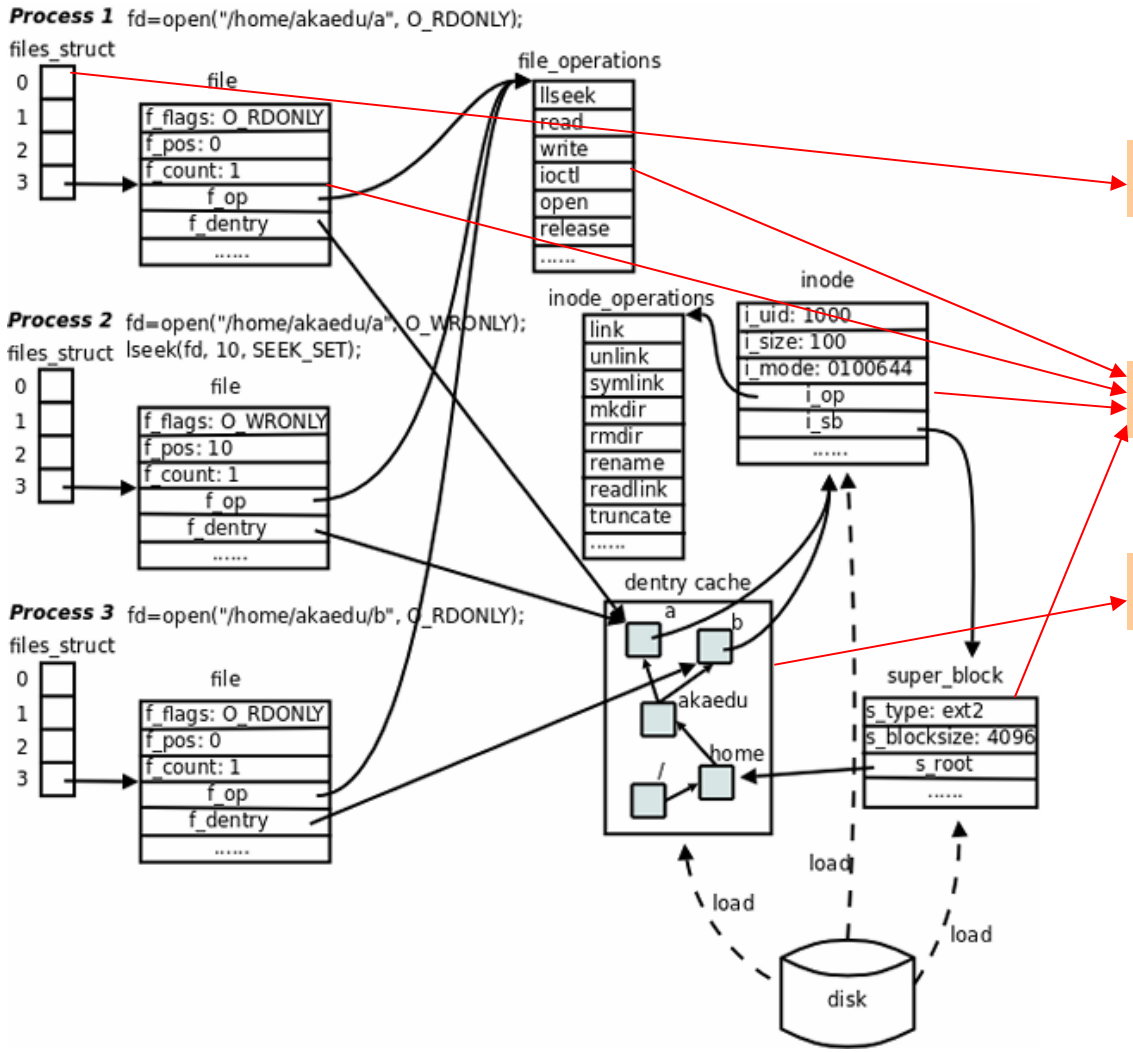
Figure 12-2. Interaction between processes and VFS objects



3.4 VFS所处理的系统调用

- VFS所处理的系统调用
  - mount、umount：挂载/卸载文件系统
  - sysfs ： 获取文件系统信息
  - statfs、fstatfs、ustat ： 获取文件系统统计信息
  - chroot ： 更改根目录
  - chdir、fchdir、getcwd ： 操纵当前工作目录
  - mkdir、rmdir ： 创建/删除目录
  - getdents、readdir 、link 、unlink 、rename ： 对目录项进行操作
  - readlink 、symlink ： 对符号链接进行操作
  - chown 、fchown 、lchown ： 更改文件所有者
  - chmod 、fchmod 、utime ： 更改文件属性
  - open、close、create …
- 上述大部分操作之需要与通用文件模型中的一些对象打交道，而不需要真正操作具体的文件系统和文件，因此可以把VFS看成是一个“通用”的文件系统，在必要时依赖某种具体的文件系统

3.5 内核数据结构



- include/linux/file.h
- include/linux/fs.h
- include/linux/dcache.h

3.5 内核数据结构

- 每个进程在PCB（Process Control Block）中都保存着一份文件描述符表，文件描述符就是这个表的索引，每个表项都有一个指向已打开文件的指针，现在我们明确一下：已打开的文件在内核中用file结构体表示，文件描述符表中的指针指向file 结构体。
- 在file 结构体中维护File Status Flag（file 结构体的成员f_flags）和当前读写位置（file 结构体的成员f_pos）。在上图中，进程1和进程2 都打开同一文件，但是对应不同的file 结构体，因此可以有不同的File StatusFlag 和读写位置。file 结构体中比较重要的成员还有f_count，表示引用计数（Reference Count），dup()、fork() 等系统调用会导致多个文件描述符指向同一个file 结构体，例如有fd1 和fd2 都引用同一个file 结构体，那么它的引用计数就是2，当close(fd1)时并不会释放file 结构体，而只是把引用计数减到1，如果再close(fd2)，引用计数就会减到0 同时释放file结构体，这才真的关闭了文件。
- 每个file 结构体都指向一个file_operations 结构体，这个结构体的成员都是函数指针，指向实现各种文件操作的内核函数。比如在用户程序中read 一个文件描述符，read 通过系统调用进入内核，然后找到这个文件描述符所指向的file 结构体，找到file 结构体所指向的file_operations 结构体，调用它的read 成员所指向的内核函数以完成用户请求。在用户程序中调用lseek()、read()、write()、ioctl()、open()等函数，最终都由内核调用file_operations 的各成员所指向的内核函数完成用户请求。file_operations 结构体中的release成员用于完成用户程序的close 请求，之所以叫release 而不叫close 是因为它不一定真的关闭文件，而是减少引用计数，只有引用计数减到0 才关闭文件。对于同一个文件系统上打开的常规文件来说，read()、write() 等文件操作的步骤和方法应该是一样的，调用的函数应该是相同的，所以图中的三个打开文件的file 结构体指向同一个file_operations 结构体。如果打开一个字符设备文件，那么它的read()、write() 操作肯定和常规文件不一样，不是读写磁盘的数据块而是读写硬件设备，所以file 结构体应该指向不同的file_operations 结构体，其中的各种文件操作函数由该设备的驱动程序实现。
- 每个file 结构体都有一个指向dentry 结构体的指针，“dentry”是directory entry（目录项）的缩写。我们传给open()、stat()等函数的参数的是一个路径，例如/home/akaedu/a，需要根据路径找到文件的inode。为了减少读盘次数，内核缓存了目录的树状结构，称为dentry cache，其中每个节点是一个dentry 结构体，只要沿着路径各部分的dentry 搜索即可，从根目录/找到home 目录，然后找到akaedu 目录，然后找到文件a。dentry cache 只保存最近访问过的目录项，如果要找的目录项在cache 中没有，就要从磁盘读到内存中。
- 每个dentry 结构体都有一个指针指向inode 结构体。inode 结构体保存着从磁盘inode 读上来的信息。在上图的例子中，有两个dentry，分别表示/home/akaedu/a 和/home/akaedu/b，它们都指向同一个inode，说明这两个文件互为硬链接。inode 结构体中保存着从磁盘分区的inode 读上来信息，例如所有者、文件大小、文件类型和权限位等。每个inode 结构体都有一个指向inode_operations 结构体的指针，后者也是一组函数指针指向一些完成文件目录操作的内核函数。和file_operations 不同，inode_operations 所指向的不是针对某一个文件进行操作的函数，而是影响文件和目录布局的函数，例如添加删除文件和目录、跟踪符号链接等等，属于同一文件系统的各inode 结构体可以指向同一个inode_operations 结构体。
- inode 结构体有一个指向super_block 结构体的指针。super_block 结构体保存着从磁盘分区的超级块读上来的信息，例如文件系统类型、块大小等。
- super_block 结构体的s_root 成员是一个指向dentry 的指针，表示这个文件系统的根目录被mount 到哪里，在上图的例子中这个分区被mount 到/home 目录下。

3.5.1 struct files_struct


```
struct fdtable {
        unsigned int max_fds;
        int max_fdset;
        struct file ** fd;      /* current fd array */
        fd_set *close_on_exec;
        fd_set *open_fds;
        struct rcu_head rcu;
        struct files_struct *free_files;
        struct fdtable *next;
};

/*
 * Open file table structure
 */
struct files_struct {
  /*
   * read mostly part
   */
        atomic_t count;
        struct fdtable *fdt;
        struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
        spinlock_t file_lock ____cacheline_aligned_in_smp;
        int next_fd;
        struct embedded_fd_set close_on_exec_init;
        struct embedded_fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];
};
```

3.5.2 struct file

```
struct file {
        /*
         * fu_list becomes invalid after file_free is called and queued via
         * fu_rcuhead for RCU freeing
         */
        union {
                struct list_head        fu_list;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        const struct file_operations    *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        struct file_ra_state    f_ra;

        unsigned long           f_version;
        void                    *f_security;

        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head        f_ep_links;
        spinlock_t              f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
};
```

3.5.3 struct file_operations

```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev, unlocked_ioctl and compat_ioctl
 * can be called without the big kernel lock held in all filesystems.
 */
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
};
```

3.5.4 struct inode_operations


```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int, struct nameidata *);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
};
```

3.5.5 struct inode

```
523 struct inode {
524         struct hlist_node       i_hash;
525         struct list_head        i_list;
526         struct list_head        i_sb_list;
527         struct list_head        i_dentry;
528         unsigned long           i_ino;
529         atomic_t                i_count;
530         umode_t                 i_mode;
531         unsigned int            i_nlink;
532         uid_t                   i_uid;
533         gid_t                   i_gid;
534         dev_t                   i_rdev;
535         loff_t                  i_size;
536         struct timespec         i_atime;
537         struct timespec         i_mtime;
538         struct timespec         i_ctime;
539         unsigned int            i_blkbits;
540         unsigned long           i_version;
541         blkcnt_t                i_blocks;
542         unsigned short          i_bytes;
543         spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
544         struct mutex            i_mutex;
545         struct rw_semaphore     i_alloc_sem;
546         struct inode_operations *i_op;
547         const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
548         struct super_block      *i_sb;
549         struct file_lock        *i_flock;
550         struct address_space    *i_mapping;
551         struct address_space    i_data;
552 #ifdef CONFIG_QUOTA
553         struct dquot            *i_dquot[MAXQUOTAS];
554 #endif
```

3.5.5 struct inode(cont.)

```
555         struct list_head        i_devices;
556         union {
557                 struct pipe_inode_info  *i_pipe;
558                 struct block_device     *i_bdev;
559                 struct cdev             *i_cdev;
560         };
561         int                     i_cindex;
562
563         __u32                   i_generation;
564
565 #ifdef CONFIG_DNOTIFY
566         unsigned long           i_dnotify_mask; /* Directory notify events */
567         struct dnotify_struct   *i_dnotify; /* for directory notifications */
568 #endif
569
570 #ifdef CONFIG_INOTIFY
571         struct list_head        inotify_watches; /* watches on this inode */
572         struct mutex            inotify_mutex;  /* protects the watches list */
573 #endif
574
575         unsigned long           i_state;
576         unsigned long           dirtied_when;   /* jiffies of first dirtying */
577
578         unsigned int            i_flags;
579
580         atomic_t                i_writecount;
581         void                    *i_security;
582         void                    *i_private; /* fs or device private pointer */
583 #ifdef __NEED_I_SIZE_ORDERED
584         seqcount_t              i_size_seqcount;
585 #endif
586 };
```

3.5.6 struct super_block

```
884 struct super_block {
885         struct list_head        s_list;         /* Keep this first */
886         dev_t                   s_dev;          /* search index; _not_ kdev_t */
887         unsigned long           s_blocksize;
888         unsigned char           s_blocksize_bits;
889         unsigned char           s_dirt;
890         unsigned long long      s_maxbytes;     /* Max file size */
891         struct file_system_type *s_type;
892         struct super_operations *s_op;
893         struct dquot_operations *dq_op;
894         struct quotactl_ops     *s_qcop;
895         struct export_operations *s_export_op;
896         unsigned long           s_flags;
897         unsigned long           s_magic;
898         struct dentry           *s_root;
899         struct rw_semaphore     s_umount;
900         struct mutex            s_lock;
901         int                     s_count;
902         int                     s_syncing;
903         int                     s_need_sync_fs;
904         atomic_t                s_active;
905         void                    *s_security;
906         struct xattr_handler    **s_xattr;
907
908         struct list_head        s_inodes;       /* all inodes */
909         struct list_head        s_dirty;        /* dirty inodes */
910         struct list_head        s_io;           /* parked for writeback */
911         struct hlist_head       s_anon;         /* anonymous dentries for (nfs) exporting */
912         struct list_head        s_files;
913
914         struct block_device     *s_bdev;
915         struct list_head        s_instances;
916         struct quota_info       s_dquot;        /* Diskquota specific options */
917
918         int                     s_frozen;
919         wait_queue_head_t       s_wait_unfrozen;
920
921         char s_id[32];                          /* Informational name */
922
923         void                    *s_fs_info;     /* Filesystem private info */
924
925         /*
926          * The next field is for VFS *only*. No filesystems have any business
927          * even looking at it. You had been warned.
928          */
929         struct mutex s_vfs_rename_mutex;        /* Kludge */
930
931         /* Granularity of c/m/atime in ns.
932            Cannot be worse than a second */
933         u32               s_time_gran;
934 };
```

3.5.7 struct dentry

```
 82 struct dentry {
 83         atomic_t d_count;
 84         unsigned int d_flags;           /* protected by d_lock */
 85         spinlock_t d_lock;              /* per dentry lock */
 86         struct inode *d_inode;          /* Where the name belongs to - NULL is
 87                                          * negative */
 88         /*
 89          * The next three fields are touched by __d_lookup.  Place them here
 90          * so they all fit in a cache line.
 91          */
 92         struct hlist_node d_hash;       /* lookup hash list */
 93         struct dentry *d_parent;        /* parent directory */
 94         struct qstr d_name;
 95
 96         struct list_head d_lru;         /* LRU list */
 97         /*
 98          * d_child and d_rcu can share memory
 99          */
100         union {
101                 struct list_head d_child;       /* child of parent list */
102                 struct rcu_head d_rcu;
103         } d_u;
104         struct list_head d_subdirs;     /* our children */
105         struct list_head d_alias;       /* inode alias list */
106         unsigned long d_time;           /* used by d_revalidate */
107         struct dentry_operations *d_op;
108         struct super_block *d_sb;       /* The root of the dentry tree */
109         void *d_fsdata;                 /* fs-specific data */
110         void *d_extra_attributes;       /* TUX-specific data */
111 #ifdef CONFIG_PROFILING
112         struct dcookie_struct *d_cookie; /* cookie, if any */
113 #endif
114         int d_mounted;
115         unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
116 };
```

3.6 dup()/dup2()

- dup()和dup2()都可用来复制一个现存的文件描述符，使两个文件描述符指向同一个file结构体。
  - 如果两个文件描述符指向同一个file结构体，File Status Flag和读写位置只保存一份在file结构体中，并且file结构体的引用计数是2。
  - 如果两次open()同一文件得到两个文件描述符，则每个描述符对应一个不同的file结构体，可以有不同的File Status Flag和读写位置。
- 它们经常用来重定向进程的stdin、stdout和stderr

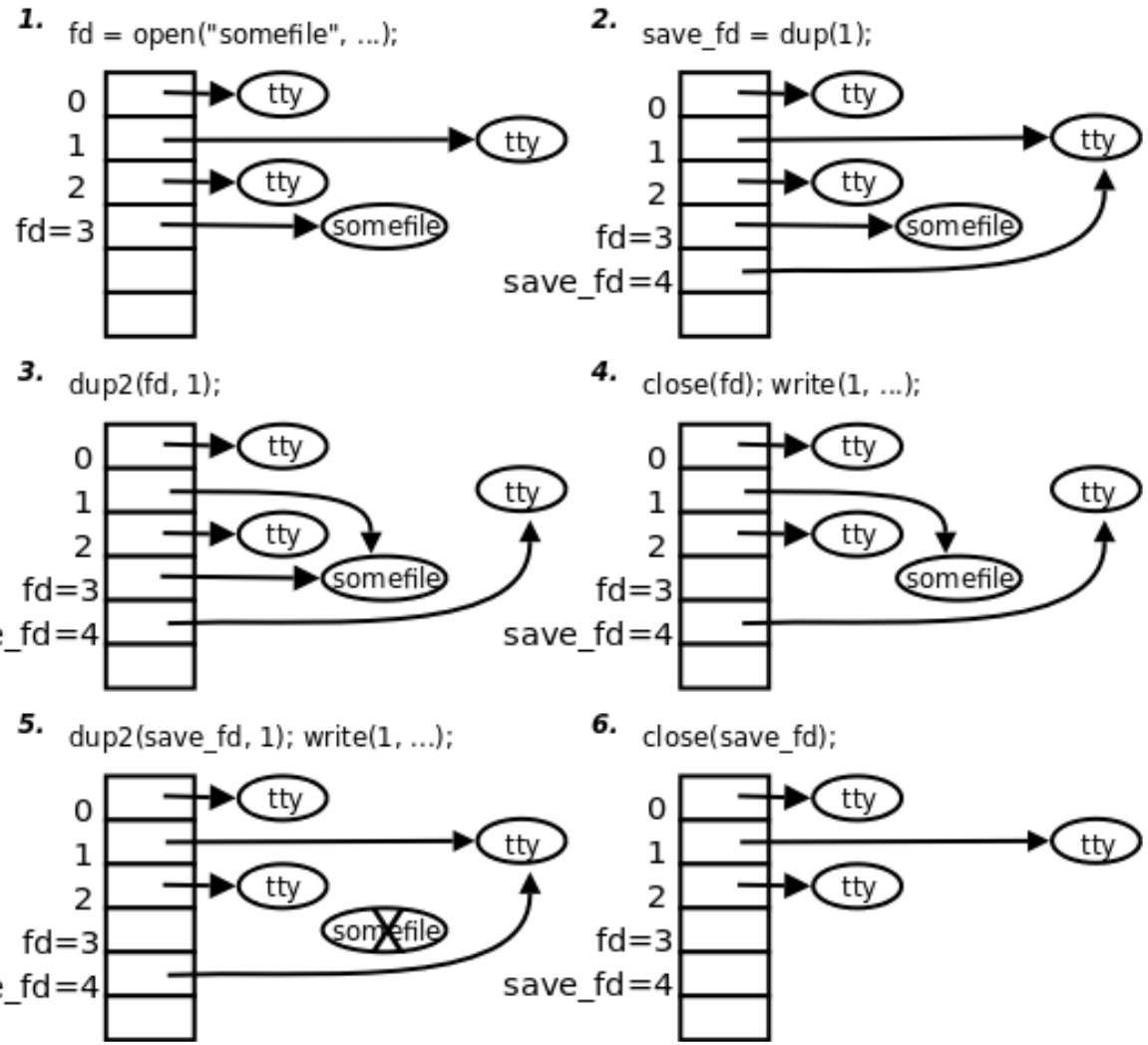
```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- 如果调用成功，这两个函数都返回新分配或指定的文件描述符，如果出错则返回-1。
- dup()返回的新文件描述符一定该进程未使用的最小文件描述符，这一点和open()类似。
- dup2()可以用newfd 参数指定新描述符的数值。如果newfd当前已经打开，则先将其关闭再做dup2()操作，如果oldfd等于newfd，则dup2()直接返回newfd而不用先关闭newfd再复制。

3.6.1 demo code

```c
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(int argc, char **argv)
{
  int fd, save_fd;
  char msg[] = "This is a test\n";

  fd = open("somefile", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);

  if (fd < 0)
  {
    fprintf(stderr, "open() failed: %s\n", strerror(errno));
    exit(1);
  }

  save_fd = dup(STDOUT_FILENO);
  dup2(fd, STDOUT_FILENO);
  close(fd);

  write(STDOUT_FILENO, msg, strlen(msg));
  dup2(save_fd, STDOUT_FILENO);
  write(STDOUT_FILENO, msg, strlen(msg));
  close(save_fd);

  return 0;
}
```

3.6.2 图解



第3幅图，要执行dup2(fd,1);，文件描述符1原本指向tty，现在要指向新的文件somefile，就把原来的关闭了，但是tty这个文件原本有两个引用计数，还有文件描述符save_fd也指向它，所以只是将引用计数减1，并不真的关闭文件。

第5幅图，要执行dup2(save_fd,1);，文件描述符1原本指向somefile，现在要指向新的文件tty，就把原来的关闭了，somefile原本只有一个引用计数，所以这次减到0，是真的关闭了。

Let's DO it!

Thanks for listening!