# Persistence (II)
# File System - 1
# (OSTEP Section 39,40)

# 讲授目的

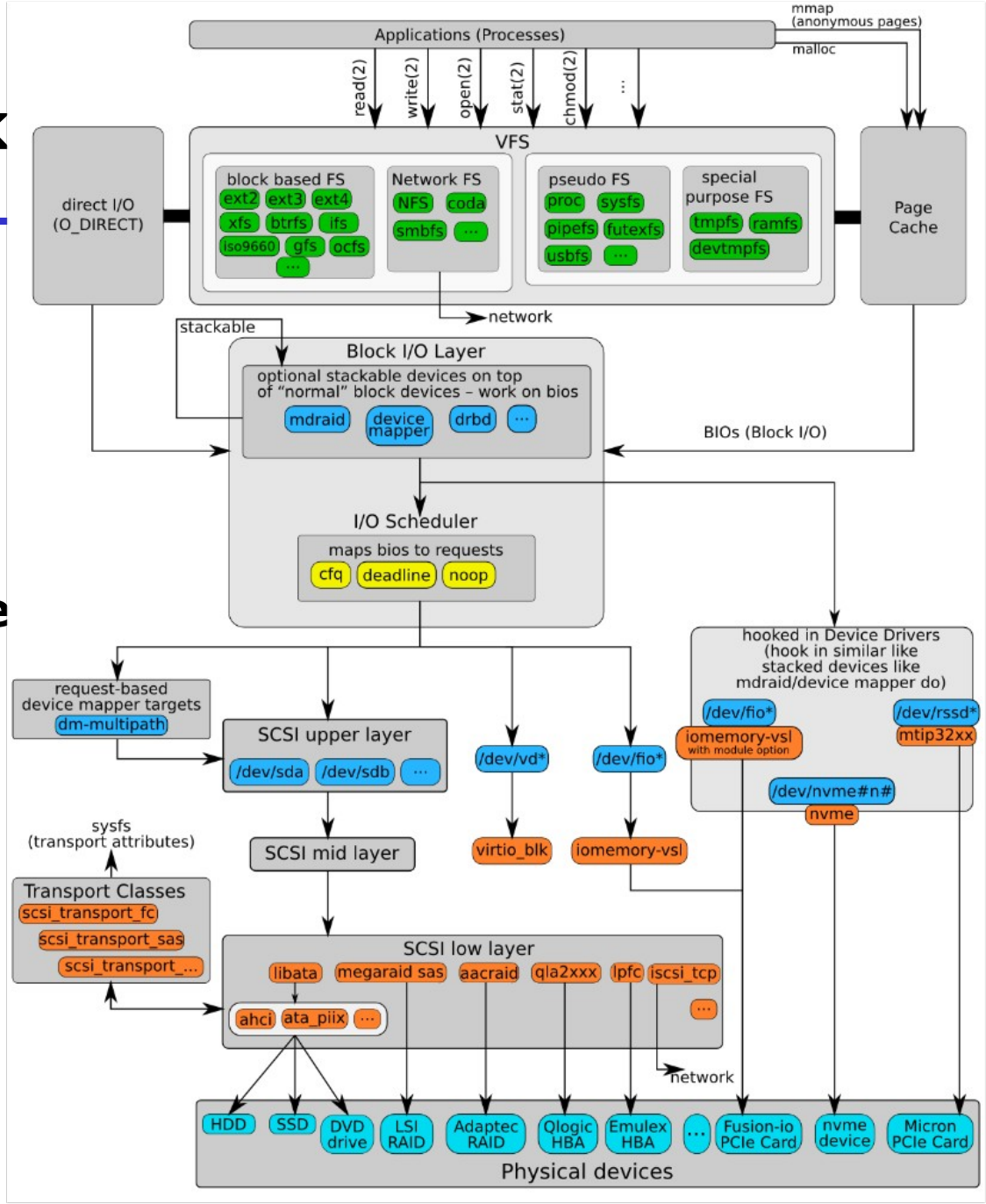
- 了解文件系统的实现原理
- 接触复杂系统的设计
  - 从简单逐步到复杂
  - 分层抽象
  - 避免灾难， instead of best
- 体会数据管理系统的异同
  - 内存管理、文件系统、数据库、 KV 存储、……

# Outline

- **Linux I/O Stack**
- **File System**
- **File System Implementation**
- **Fast File System**
- **FSCK and Journaling**
- **Log-structured File Systems （不讲）**
- **Data Integrity and Protection （不讲）**

# Linux I/O Stack

- VFS
- FS
  - (Page Cache)
- Block I/O Layer
  - I/O Scheduler
- Block Device Drive



# Linux I/O Stack

- **VFS (Virtual File System)**
  - 很薄的一层
  - 对上提供各种文件操作接口
  - 屏蔽不同文件系统的细节，提供统一接口（Unix IO）
    - Open
    - Read
    - Write
    - Close
    - Stat
    - Chmod
    - …

# Linux I/O Stack

- **File System**
  - 具体文件系统，有很多经典实现
  - **Ext** 系列、 **xfs** 、 **btrfs**
  - 网络文件系统（ **nfs, ceph** ）
  - **Special purpose fs (tmpfs, ramfs)**
  - **Pseudo fs (proc, pipefs)**

# Linux I/O Stack

- **Page Cache**
  - 充分利用内存空闲空间，为慢速 I/O 设备提供写缓冲（buffering）和读缓存（caching）
  - Dirty 数据定期写回磁盘
  - 清空 page cache 方法:
    - To free pagecache, use echo 1 > /proc/sys/vm/drop_caches;
    - To free dentries and inodes, use echo 2 > /proc/sys/vm/drop_caches;
    - to free pagecache, dentries and inodes, use echo 3 >/proc/sys/vm/drop_caches.
  - Ps. Storage Device 和 RAID 卡上可能还有 cache

扩展内容

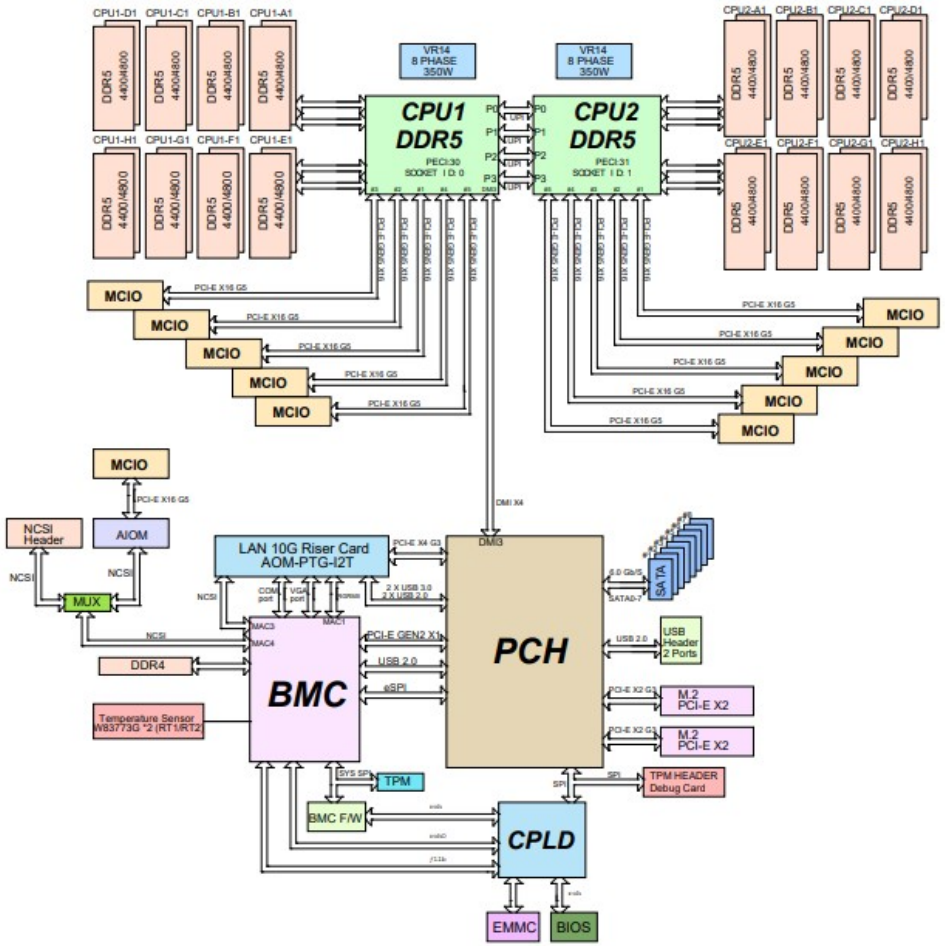
# How Computer Architecture Looks

扩展内容

# How Computer Architecture Looks Like

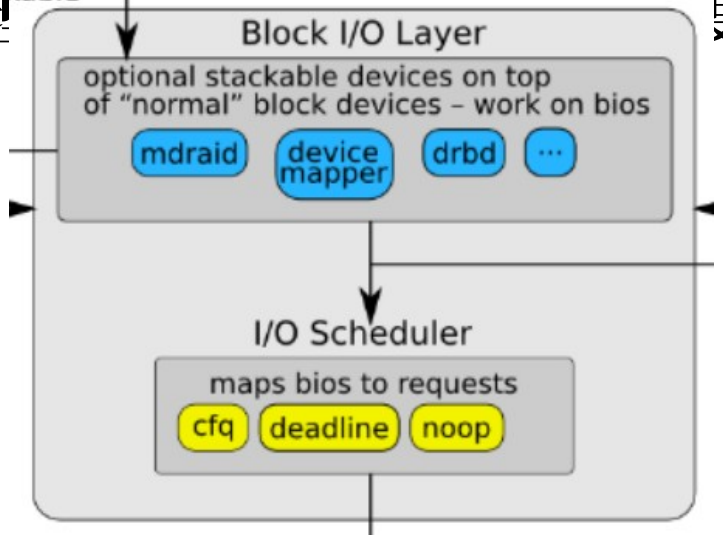
## System Block Diagram

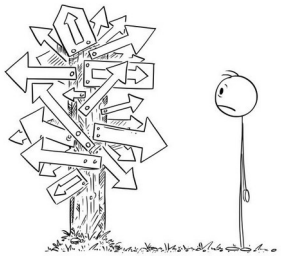
The block diagram below shows the connections and relationships between the subsystems and major components of the overall system.



# Linux I/O Stack

- **Block I/O Layer (General Block Layer)**
  - 为 FS 提供抽象的“ block device”
  - 实现“ software RAID”，即 muiltple-device (md)
  - 对发往物理设备的 I/O 请求进行调度



各种 block、 page 之类的，很混乱。。。

扩展内容

# page, block, logical, physical...

- These words are somehow abused in CS

- 不同上下文中的含义不同
  - SSD 中的 page、 physical block、 logical block，和 OS 中的 page 和 block，是不同的概念，他们之间存在一定的对应关系
  - SDD FTL 中的 logical block 在 OS 中通常叫 physical sector 或者 physical block

sector 不是磁盘内部的 physical unit 吗？怎么 OS 里也有 sector。。。

扩展内容

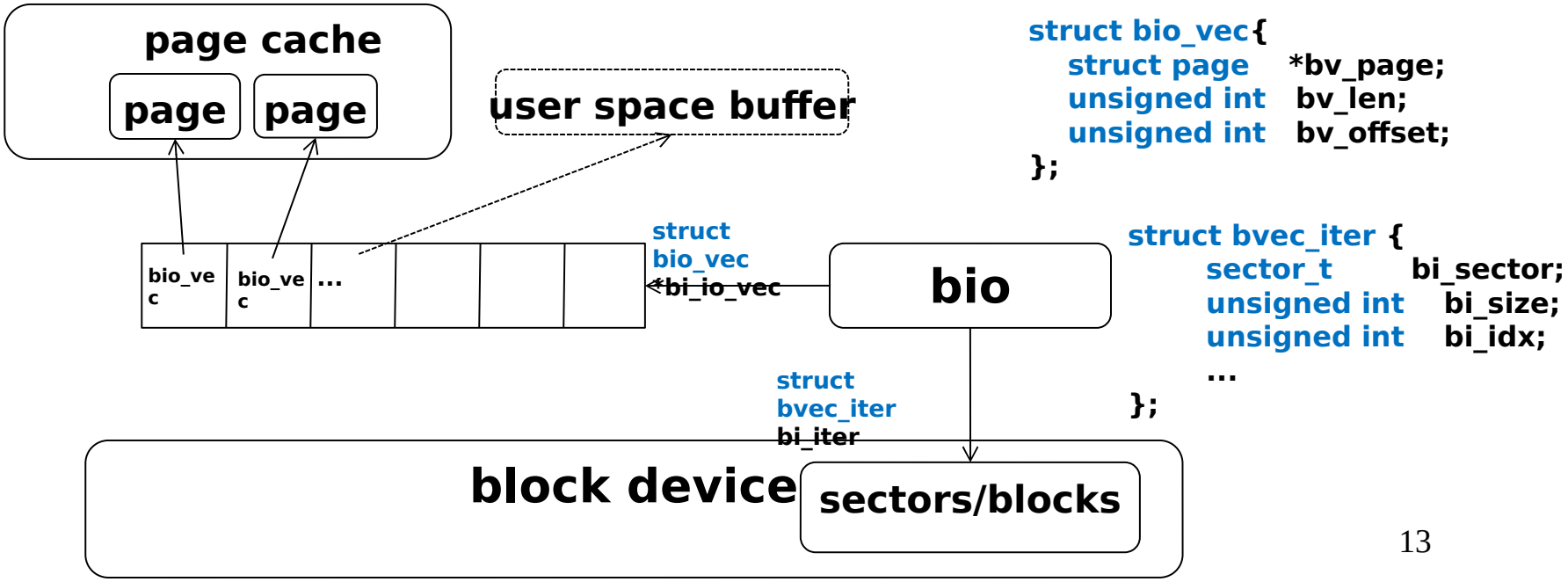
# 两种 addressing method 的纠葛： CHS vs. LBA

- **CHS: cylinder-head-sector**
  - 早期 disk 上没有 controller，只能把 CHS 的三维结构暴露给 OS， OS 中只能使用 CHS addressing 来访问 device
  - late 1980s: disk 越来越复杂，开始加入 controller 以隐藏内部的几何结构，只向 OS 暴露 logical 的 CHS 地址
- **LBA: logical block addressing**
  - mid 1990s: 更加简洁的 LBA 开始逐渐取代 CHS
  - 但直到今天，很多 OS 和磁盘管理工具中还有 CHS 的身影，一些 device 依然支持 CHS addressing，很多资料里还在用 sector 作为 OS 访问 device 的基本单位

Linux 中为了兼容老设备，依然有 512B 的 sector，但对较新的设备会使用 4KB block/sector（advanced format）

# Linux I/O Stack

- **Block I/O Layer**
  - **struct bio 代表 FS 对 block I/O layer 的 I/O 请求**
  - **bio 就像一个快递员，从内存中收取多个包裹，送往 block device 上的一个目的地，或者反之**



# Linux I/O Stack

- **Block I/O Layer**
  - **Linux Device Mapper 机制在 block I/O layer**
  - **可以将多个 physical device 封装为一个 block device**
    - **支持 software RAID （i.e., multiple device, 缩写 md）**
  - **或者自定义功能**
    - **例如实现 SSD 为 HDD 提供 cache 功能（Flashcache, bcache, dm-cache）**
    - **实现几个 kernel 接口，编译为 xxx.ko， insmod xxx.ko**
      - **ko = kernel object**
      - **insmod 或者 modprode 命令可以用来加载模块到 kernel**

# Linux I/O Stack

- **I/O Scheduler**
  - 对发往 device 的 request 进行排序和调度
  - 常见的调度算法
    - **noop: no operation**，没有排序，最简单（适合低层是 SSD）
    - **deadline:** 保证 request 在一定时间内被发往 device
      - 四个队列，其中两个分别为正常 read 和 write 队列，另外两个为超时 read 和 write 的队列
      - 正常队列：按 sector addr 排序，合并相邻的请求；如果新请求总在相邻地址，那么其他地址的 io 请求可能饿死
      - 超时 read 和 write 的队列：按请求创建时间排序，保证超时的请求会优先被处理，防止请求被饿死

# Linux I/O Stack

- **I/O Scheduler**
  - 对发往 device 的 request 进行排序和调度
  - 常见的调度算法
    - CFQ （completely fair queuing）
      - 试图给所有进程提供完全公平的 IO 调度
      - 它为每个进程创建一个同步 IO 队列， OS 以时间片 (time slice) 轮转的方式来执行每个进程队列中的请求，以此保证每个进程的 IO 资源占用是公平的

# Linux I/O Stack

- **I/O Scheduler**
  - 新型的 SSD，内部硬件特征复杂，与磁盘有很大区别
    - 例如多通道， FTL 算法
  - 对于一些新设备， OS 并不能给出很好的调度（厂家也不愿意公开某些细节的关乎性能的设计）
  - 所以，对于某些新存储硬件，可以 by-pass I/O scheduler，直接由应用程序配合设备上的控制器来完成 I/O 调度

# Outline

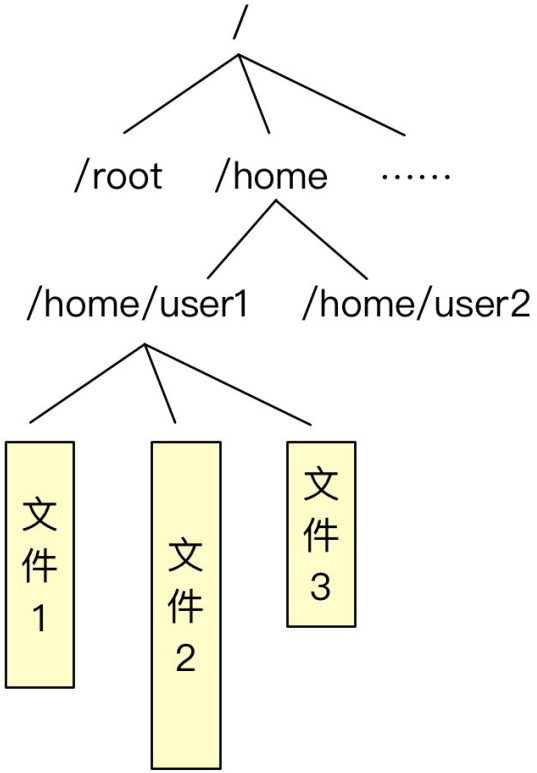
- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems
- Data Integrity and Protection

# File System

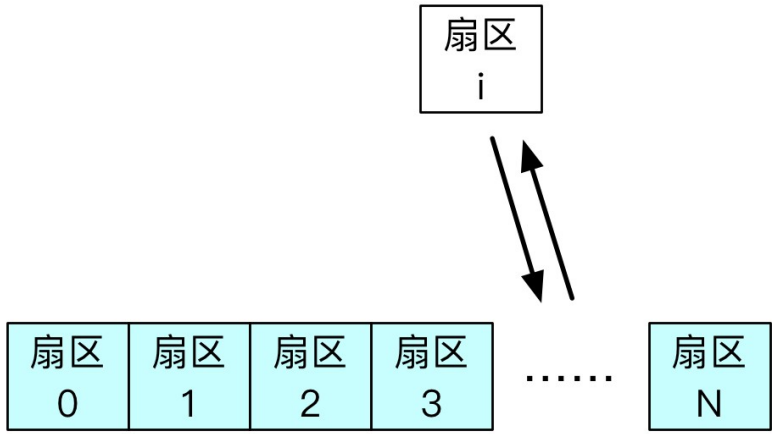
- 虚拟内存和 Block I/O：硬件抽象
- 文件：数据抽象
  - 文件是一个字节向量，不用关心其在设备上的存储细节
  - Naming，文件之间的命名和关系，重启后还能找到
    - 目录树
- 更高级的抽象
  - Key-Value store
  - object store
  - Relation (Table)
  - Graph
  - …

# File System

- 文件数据模型比 block 模型、磁盘模型更便于用户使用（更高级 / 复杂）



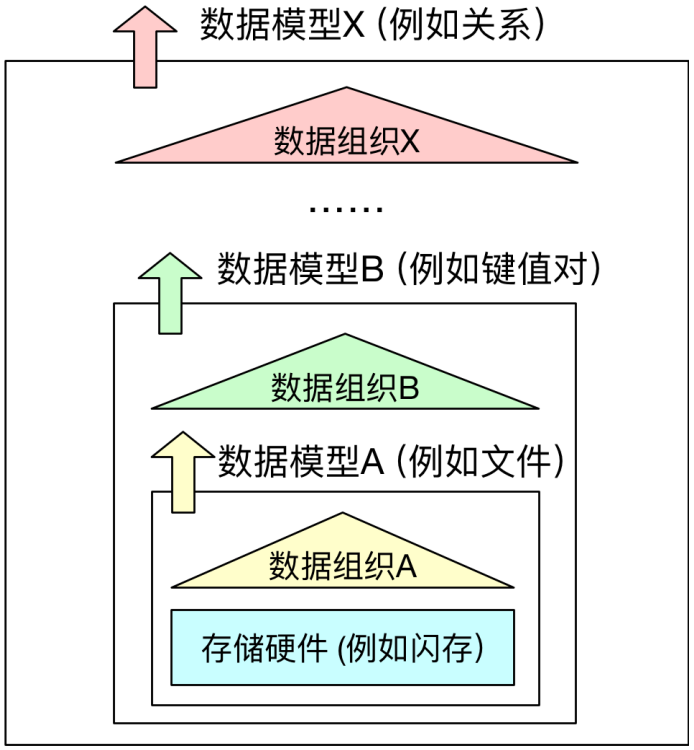

(A) 文件数据模型




(B) 磁盘的访问模型

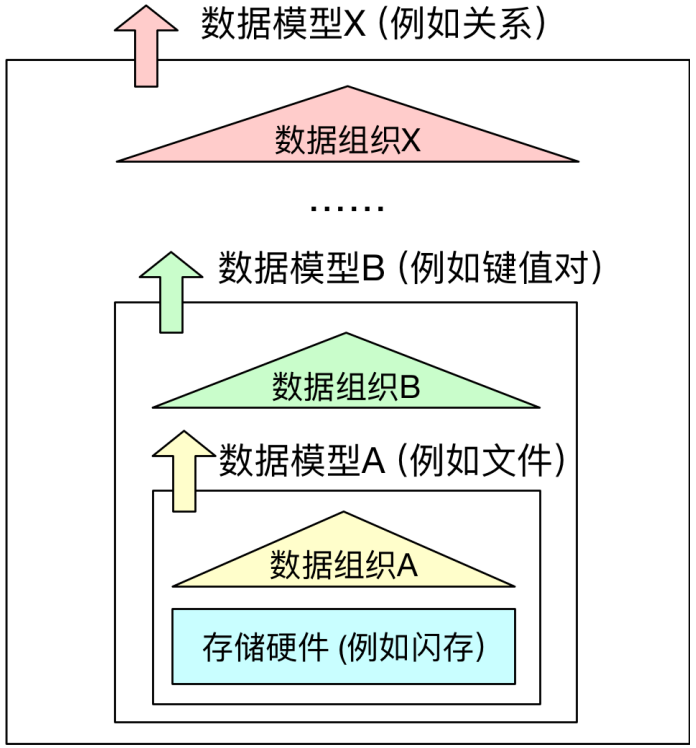
# File System

- 数据模型之间的嵌套关系
  - 分布式文件系统 HDFS 系统是在每个结点的本地文件系统的基础上，将分布式的资源进行封装，对上提供文件数据模型的接口
  - HBase 系统就是在 HDFS 的基础上，进一步封装，提供键值访问和列存数据模型的接口



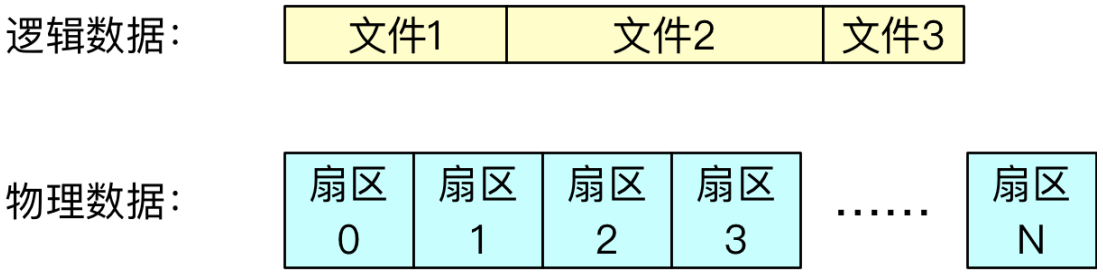
# File System

- 数据模型之间的嵌套关系
  - 例如 **RocksDB** 键值存储引擎就是在本地文件系统（例如 **ext4** ）的基础上通过日志合并（ **LSM** ）树结构实现了键值数据组织，提供键值数据模型的访问接口。
  - 分布式系统 **Ceph** 首先实现一套分布式的对象存储（键值数据模型），然后在对象存储的基础上，又进一步封装为块数据模型和文件数据模型的访问接口（通过额外实现文件系统的目录结构等功能）。



# File System

- 如何实现文件系统?
  - 最直接的想法是将所有文件首尾拼接，从目录树的复杂结构退化为一维线性结构，以便和磁盘的一维 sector/block 模型相匹配
  - 但是这样做存在一些比较严重问题：
    - 文件大小增加如何处理？需要在其他地方放置文件新增的数据，那么就破坏了文件首尾相接的结构；
    - 文件删除如何处理？那么文件首尾相接的结构中会出现一些空洞，想要回收这些空间要额外的设计；
    - 如何能够快速找到用户想要访问的某个文件？按照目前的方法，可能需要遍历整个磁盘空间才找到目标文件，性能非常差。
    - 在哪里记录文件的属性信息？包括文件的访问权限、修改时间等等。



# File System

- **File System Interface**
  - **Create Files**
  - **Reading and Writing Files**
  - **Reading and Writing Files Randomly**
  - **Writing Immediately**
  - **Renaming Files**
  - **Getting Information About Files**
  - **Removing Files**
  - **Making, Reading, and Deleting Directories**
  - **Hard Links**
  - **Symbolic Links**

# File System

- **Creating Files**
  - **int fd = open(“foo”, O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S+IRUSR);**
  - **O_TRUNC，文件截断为 0 （对应 O_APPEND ）**
  - **文件描述符，进程私有**

# File System

- **Reading and Write Files**
  - **跟踪文件访问的系统调用**
    - **Strace 工具（MacOS 上有 dtruss）**
      - **0: 标准输入； 1: 标准输出；2: 标准错误**
      - **Read/write 第二个参数是 buffer，已显示内容；返回长度**

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
prompt>
```

# File System

- **Reading And Writing, But Not Sequentially**
  - 文件的 current 指针
    - off_t lseek(int fildes, off_t offset, int whence);
  - lseek 本身不会直接导致磁盘的 seek 操作，后续的 read/write 才会
  - ssize_t pread (int fd, void* buf, size_t count, off_t offset);
  - ssize_t pwrite (int fd, void* buf, size_t count, off_t offset);
    - 把 lseek 和 read/write 合起来作为一个原子操作

# File System

- **Writing Immediately**
- **Write 的语义**
  - **只是告诉 OS 要将数据写入持久化设备，但可以过一段时间再写**
  - **写到 memory buffer 中就返回，性能好**
  - **Dirty 数据过多，或停留时间过长， OS 会将其刷回设备**
- **fsync(int fd)**
  - **强制 dirty 数据写回设备**
  - **数据写完后返回，性能比较差**

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

# File System

- **Writing Immediately**
  - **fdatasync()**
    - 只同步 **FS** 中的数据，不同步元数据
  - **O_DIRECT**
    - **open** 可增加这个选项，跳过 **page cache**，直接写到通用块层、**I/O** 调度层，自己处理块对齐
    - 只是承诺不拷贝到 **page cache**，不承诺等所有数据都写完才返回，所以会比单独用 **O_SYNC, fsync** 要快一些
  - **O_SYNC**
    - **open** 可增加这个选项，同步模式，每次 **write** 后都调用 **fsync**，所有数据写入设备成功才返回
    - 但也可能在磁盘的缓存中，数据不一定 **100%** 安全

```
fd = open("myfile",
 O_RDWR | O_CREAT | O_SYNC | O_DIRECT, S_IRUSR | S_IWUSR);
```

# File System

- **Renaming Files**
  - **prompt> mv foo bar**
  - **原子操作，防止系统崩溃**

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

# File System

- **Getting Information About Files**
  - **stat() or fstat()**

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

# File System

- **Removing Files**
  - 为什么删除的 system call 是 unlink?

```
prompt> strace rm foo
...
unlink("foo")                                   = 0
...
```

# File System

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

- **Making, Reading, and Deleting Di**

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                          = 0
...
prompt>
```

  - 自己写 ls，读目录

```
struct dirent {
    char           d_name[256]; /* filename */
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file */
};
```

# File System

- **Hard Links**
  - **Ln: create another way to refer to the same file**
  - **Same inode**

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

# File System

- **Hard Links**
  - 创建文件的两个步骤
    - 创建 **inode**，记录文件的相关信息（**size, block location, …**）
    - **Link a human-readable name to that file, and put the link in the directory**
  - 删除文件的操作是 **unlink**
    - 不影响 **inode** 和文件数据，链到同一个 **file** 的其他 **reference** 仍然可以访问
    - 当文件的 **reference count = 0** 时，就真的删除

# File System

- **Symbolic Links / Soft Links**
  - **Hard links 的限制**
    - 不能链接到 **directory** （可能在目录树上形成闭环）
      - **A** 目录下有 **B** 目录， **B link** 到 **A** （不是树形结构）
    - 不能链接到其他 **disk partition** 的文件（因为 **inode** 编号只是在每个 **Partition** 上是唯一的）
  - **Ln –s 创建软链接**
    - 软链接是 **file, directory** 之外的第三种类型
    - 删除原文件后，访问软链接也会出错

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

```
prompt> stat file
  ... regular file ...
prompt> stat file2
  ... symbolic link ...
```

# File System

- **Making and Mounting a File System**
  - **fdisk /dev/sda**
  - **mkfs.ext4 /dev/sda1**
  - **Mount /dev/sda1 /home**
  - **mount point**
    - 把一个文件系统（一个磁盘分区）挂载到目录树的一个点中
    - 从 / 开始的整个目录树可以包括多个（不同的）文件系统

# Outline

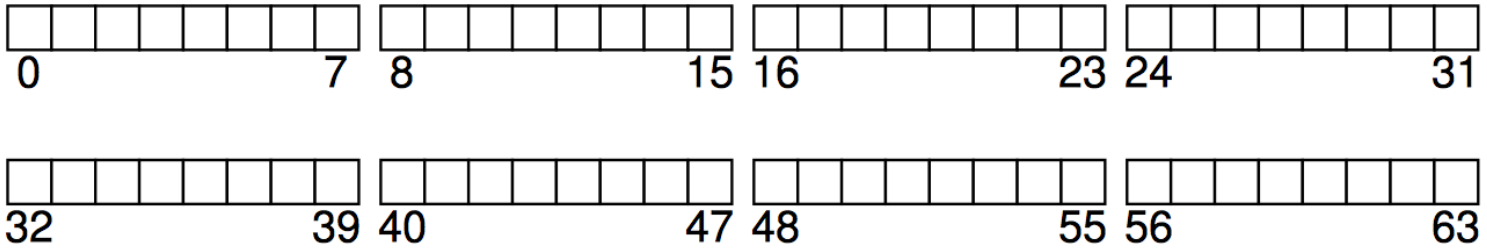
- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems
- Data Integrity and Protection

# File System Implementation

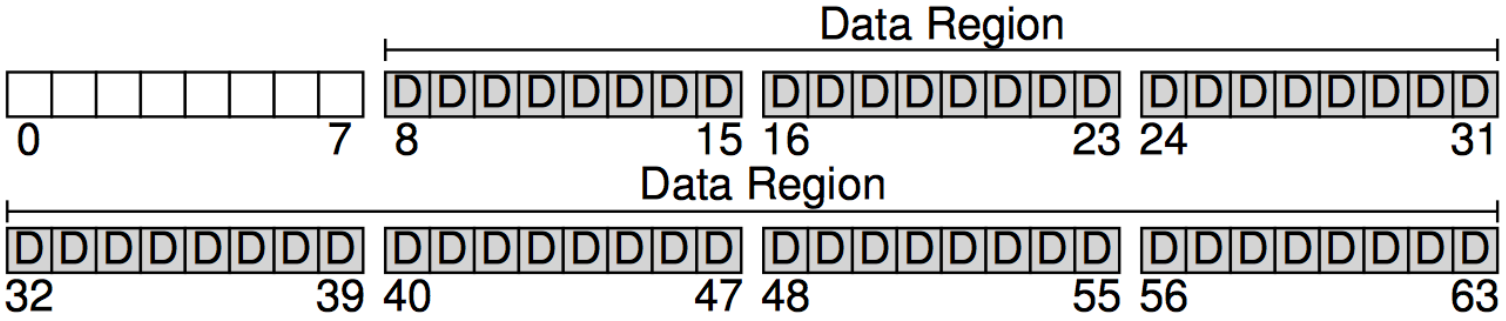
- FS 本质
  - 硬件提供的接口非常简单
    - CHS 或 LBA 地址
    - sector 或 block 访问粒度（例如 512B 或 4KB）
  - FS 是在简单硬件接口上提供文件数据模型
    - 目录树、文件元数据、空间管理
  - KV、数据库、 Graph 也是如此，只是结构比 FS 更复杂，可能基于 FS 来做

# File System Implementation

- **VSFS (Very Simple File System)**
  - 假设硬件只有 64 个 block，每个 block 4KB
  - 暂时不考虑针对硬件特性的优化

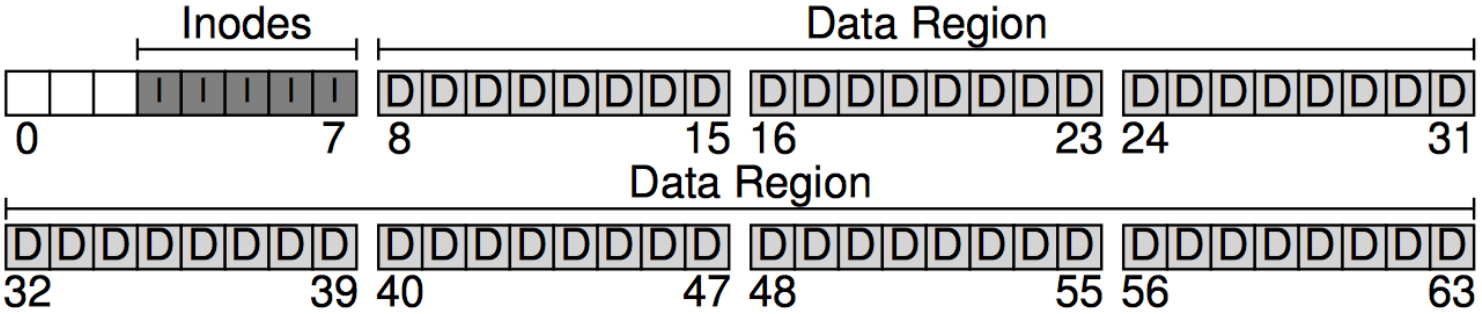



  - 后面 56 个块作为 data region



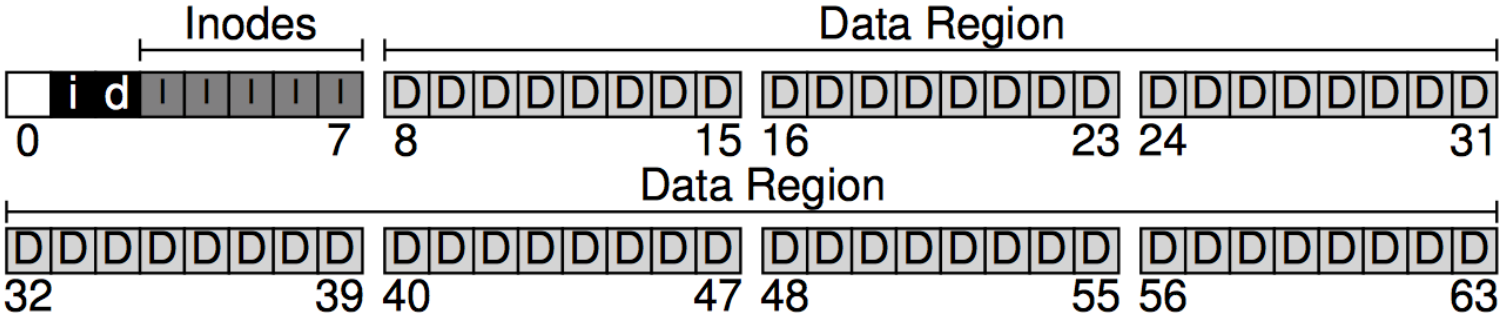
# File System Implementation

- **Inode**
  - 文件的元数据，每个文件对应一个 inode
- **Inode Table**
  - 比文件小很多，假设占用 5 个 block，每个 inode 256B, 共 80 个 inode，最多支持 80 个文件



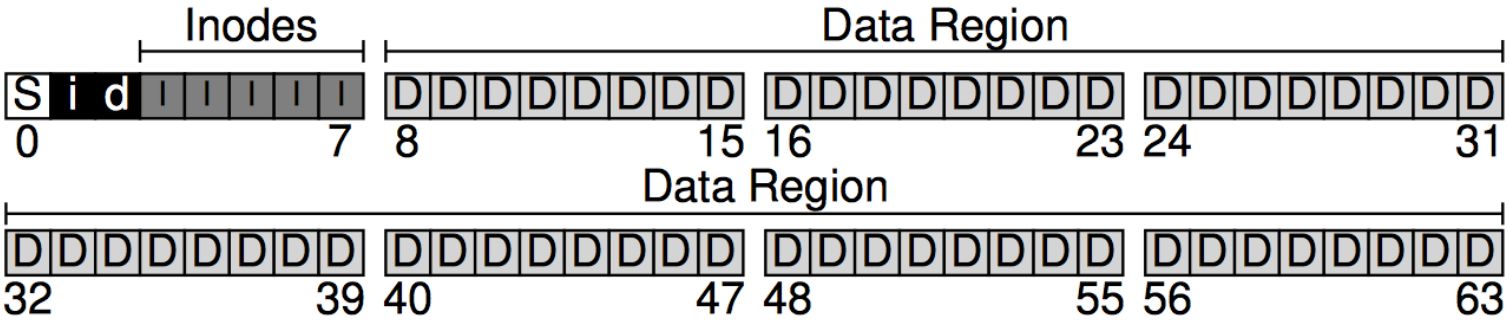
# File System Implementation

- **Bitmap**
  - 记录 **inode** 和 **data block** 是否被占用
    - **Inode bitmap (I)**
    - **Data bitmap (D)**



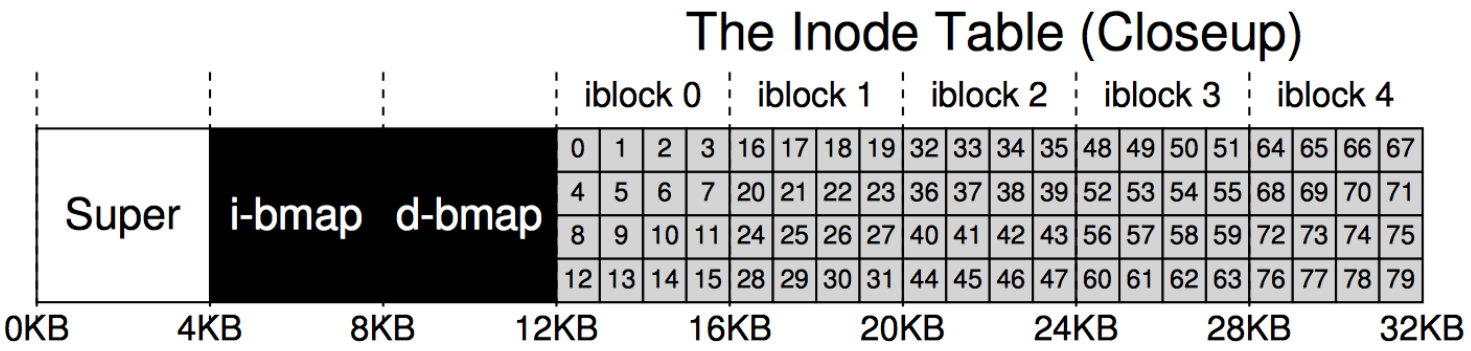
# File System Implementation

- **Superblock**
  - 文件系统的一些参数，例如有多少 **inode, data block, inode table** 的位置等



# File System Implementation

- **Inode**
  - **i-number: 每个 inode 都有一个隐式的编号**
  - **根据 i-number 能知道 inode 在磁盘上的准确位置**



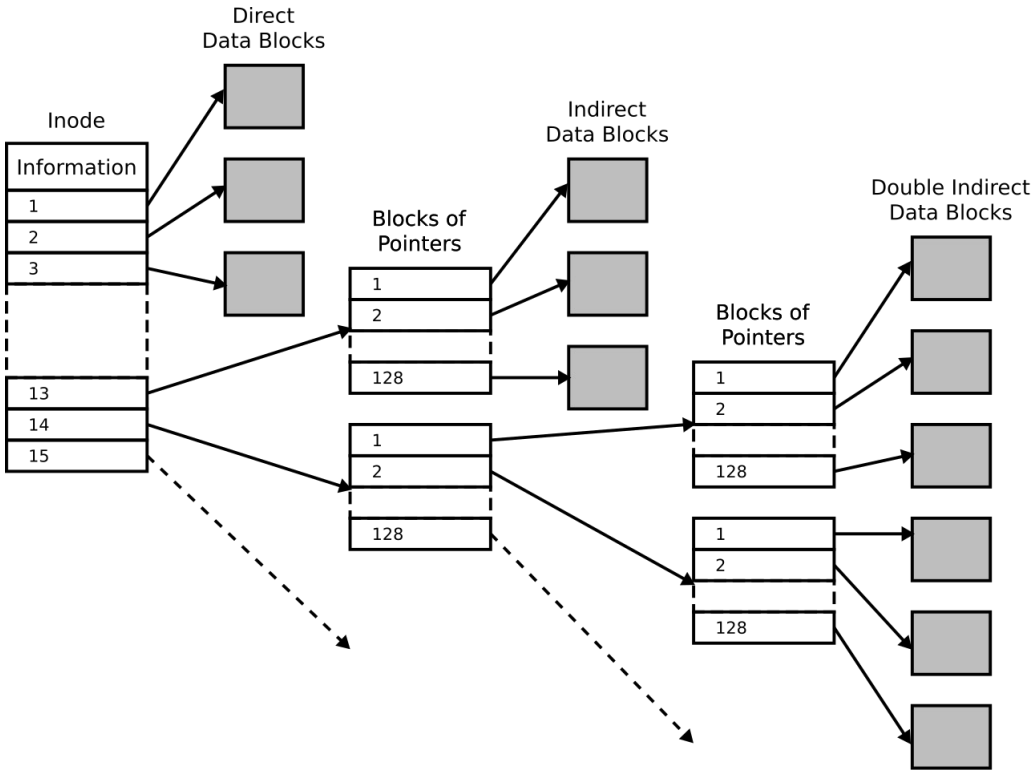
# File System Implementation

- **Inode 数据结构举例（ext2 简化）**
  - **15 个指针 (direct pointer)，指向 data blocks； 15 个不够用的话，要用到 multi-level i**

| Size | Name | What is this inode field for? |
|---|---|---|
| 2 | mode | can this file be read/written/executed? |
| 2 | uid | who owns this file? |
| 4 | size | how many bytes are in this file? |
| 4 | time | what time was this file last accessed? |
| 4 | ctime | what time was this file created? |
| 4 | mtime | what time was this file last modified? |
| 4 | dtime | what time was this inode deleted? |
| 2 | gid | which group does this file belong to? |
| 2 | links_count | how many hard links are there to this file? |
| 4 | blocks | how many blocks have been allocated to this file? |
| 4 | flags | how should ext2 use this inode? |
| 4 | osd1 | an OS-dependent field |
| 60 | block | a set of disk pointers (15 total) |
| 4 | generation | file version (used by NFS) |
| 4 | file_acl | a new permissions model beyond mode bits |
| 4 | dir_acl | called access control lists |

# File System Implementation

- **Multi-level index**
  - **ext2, ext3, Netapp's WAFL**



# File System Implementation

- **Multi-level index**
  - **12 direct pointer + 1 single indirect pointer + 1 double indirect pointer**
  - **一个文件最大的大小是多少?**
    - **4GB**
    - **一个 block 4KB，一个指针 4B，可放 1024 个指针**
    - **(12+1024+1024*1024)*4KB ≈ 4GB**
  - **为什么不用平衡树?**
    - **Most files are small**

# File System Implementation

- **Extent**
  - **Point + length (in blocks)**
  - **当一个文件包含很多连续的 data blocks 时，节省 Inode 空间**
    - **例如 ext4, XFS**
  - **此外，也能更好地保证 I/O 的连续性**
    - **所有存储设备的顺序访问性能都好于随机访问， RAM 也不例外**
    - **世界上没有真正的‘’ Random‘’ Access Storage**

# File System Implementation

- **Linked-based approach**
  - **Inode 只包含一个指针，指向第一个 data block，data block 最后增加一个指针，指向下一个 data block**
  - **做 random 访问时性能很差**
    - **把这些指针先放到内存中来加速**
  - **例如 FAT**

# File System Implementation

- 文件系统的一些特性

| | |
|---|---|
| **Most files are small** | Roughly 2K is the most common size |
| **Average file size is growing** | Almost 200K is the average |
| **Most bytes are stored in large files** | A few big files use most of the space |
| **File systems contains lots of files** | Almost 100K on average |
| **File systems are roughly half full** | Even as disks grow, file systems remain ~50% full |
| **Directories are typically small** | Many have few entries; most have 20 or fewer |

# File System Implementation

- **Directory Organization**
  - 目录被当做文件，有自己的 inode 和 data block
    - Inode 的类型是 directory，而不是 regular file
  - 目录的数据内容就是所包括的文件和子目录的信息，例如

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 12 | 2 | . |
| 2 | 12 | 3 | .. |
| 12 | 12 | 4 | foo |
| 13 | 12 | 4 | bar |
| 24 | 36 | 28 | foobar_is_a_pretty_longname |

# File System Implementation

- **Free Space Management**
  - **Bitmap/freelist/B-tree**
  - 创建文件
    - 查找 **inode bitmap**，找到一个 **free inode**
    - 分配该 **inode** 给这个文件，并标记已使用
    - 修改 **bitmap**
    - 分配 **data block** 也是类似
  - 优化技巧
    - 尽量提高数据连续性，如 **extent**

# File System Implementation

- **Reading a File from Disk**
  - **Open （“ /foo/bar”, O_RDONLY)**
  - **从 path 找到 bar 文件的 inode**
    - **1) 根目录 / 的 inode (well known, e.g., 2 )**
    - **2) 根目录 / 的 data，找到下级目录 /foo 的 i-number**
    - **3) /foo 的 inode**
    - **4) /foo 的 data**
    - **5) /foo/bar 的 inode**
  - **Read**
    - **1) 读 /foo/bar 的 inode**
    - **2) 读 /foo/bar 的 data**
    - **3) 写 /foo/bar 的 inode，更新 last accessed time**
    - **4) 更新 in-memory open file table， offset**

# File System Implementation

- **Writing to Disk**
  - 类似的过程
  - 分配 **inode, data blocks**，更新 **bitmap**
  - 写一个文件至少产生 **5** 次 **I/O**
    - **Read data bitmap**
    - **Write data bitmap**
    - **Read inode**
    - **Write inode**
    - **Write data**
  - 文件系统的额外开销，尤其是小文件

# File System Implementation

- **Caching and Buffering**
  - 没有 cache， I/O 太多，尤其是目录较深
  - 管理算法，例如 LRU
  - Write buffer
    - 批处理写入，合并一些 I/O （例如同一个 inode 的修改）
    - 请求合并、磁盘调度，性能更好
  - 从 Linux 2.4 开始， Caching 和 Buffering 统一由 Page Cache 提供

# 课堂练习

- 以下哪个不是文件的一部分?
  - 文件名
  - 文件大小
  - 文件数据
  - 文件数据块的存储位置

# 课堂练习

- 假设计算机中，每个 data block 的大小是 4KB，每个 pointer 4B，inode 结构中包括 12 个 direct pointers，2 个 indirect pointers，2 个 double indirect pointer。
  - 请问该文件系统支持的一个文件最大为多大?
  - 假设一个文件是 100KB，请画出这些指针的结构。

# Outline

- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems
- Data Integrity and Protection