

引导

冯氏结构的特点：可编程、计算和存储分离

操作系统的特征

操作系统的分类

进程

进程状态

进程和线程

调度

CPU调度

CPU资源调度(决策指标)

调度策略

数据

内存管理

内存管理需要满足的目标

页面替换

覆盖技术

输入输出设备

设备控制器

文件系统

文件

目录

文件和卷

文件基本操作

设计磁盘文件系统的挑战

磁盘文件系统

概述

获取数据

文件存储方式

FAT文件系统

优缺点

FFS文件系统

特点

目录机构

空闲地址映射和局部性分配

优缺点

NTFS：引入新概念Extents

文件系统布局

备份策略

目录存储格式

硬连接和符号连接的存储格式

虚拟文件系统

磁盘文件系统的镜像

mount系统调用

open系统调用

进程间通信

进程间为什么要通信？

进程间如何通信？

管道模型

消息队列模型

信号

共享内存模型

信号量

计算机网络系统

TCP

存储方式

引导

冯氏结构的特点：可编程、计算和存储分离

- 1.采用指令和数据存储在一起的结构
- 2.存储器是按地址访问、线性编址
- 3.指令由操作码和地址码组成
- 4.指令在CPU的运算单元处理
- 5.控制流由指令流产生

CPU：执行各种计算机指令的逻辑机器

操作系统的特征

1. **并发**：同时**存在**多个运行的程序（进程），需要OS管理和调度
2. **共享**：互斥共享，某种**资源只能同时被一个进程独享**；同时访问：**允**

许在一段时间内由多个进程交替访问

3. **虚拟**：利用多道程序设计技术，让每个进程和每个用户**觉得有一个处理器(CPU)专门为他服务**。

4. **异步**：进程执行不是一贯到底，而是**走走停停且执行速度不可预知**

命令接口：联机控制(交互式)；脱机控制(批处理)

程序接口：一组系统调用命令组成，用户需要使用这些系统调用来请求操作系统提供服务(如外设、磁盘、内存分配和回收等)

操作系统的分类

单道操作系统：多个进程按照先来先服务的方式顺序执行(排队系统)；内存中只有一个进程运行且它需要执行完毕(独占内存)；在运行时，高速CPU会等待低速I/O完成，限制系统吞吐量

多道操作系统：内存中**同时存放**多个相互独立的进程；**宏观上并行**(先后开始各自的运行,但都未运行完毕)；**微观上串行**(轮流占有CPU)；**资源利用率和系统吞吐量均增大**；CPU、内存、I/O设备**资源分配问题复杂**；大量进程和数据组织和存放的**安全性和一致性问题**

分时操作系统：**分时技术**:处理器的运行时间分成很短的时间片；按**时间片轮流**把处理器分配给各个用户或进程使用；存在系统延时(银行系统、购票系统等)

实时操作系统：计算机系统接收到程序请求后会**立即处理**，并在**严格的时限**；内处理完成该程序
可靠性和实时性保证较难；一般需要**专门设计**，**普适性较差**

网络操作系统(5G)

分布式操作系统(云，CDN)

中断：来自CPU执行指令之外发生的事件(I/O、时间中断)，它与当前程序运行无关

异常：来自CPU执行指令内部的事件(除0、算术越界等)；它依赖当前程序的运行而且不能被屏蔽
都用于用户态切换至内核态

大内核：将操作系统的主要功能模块作为整体运行在核心态，以此来为应用程序提供高性能服务

模块间信息共享->性能优势明显；程序服务需求增加、接口形式更复杂->功能耦合度高、模块化程度下降，内核设计庞大且复杂

微内核：解决操作系统的内核代码难以维护问题，即将内核最基本功能(e.g., 进程、线程管理)保留，剩余功能移到用户态执行

降低了内核设计的复杂度；具有更好的分层和模块化接口；**性能问题!!!!** 频繁的在用户态和内核态切换

进程

程序：包含代码和数据，是由某种程序语言来刻画

可执行文件：CPU指令+其可使用的数据

进程：操作系统为正在运行的程序提供的**抽象**；包括地址空间+执行上下文(这两个由程序指令控制)+环境(由系统调用控制)

“好的”程序抽象：轻量级且隐藏了具体实现；易于使用的接口；可以被实例化多次；易于实现

小细腰结构：易于实现和维护；安全性(攻击点少)；互联网也是这种结构(IP层)

进程控制块PCB：对于任意一个进程，操作系统会维护一个进程控制块来**描述进程的基本信息和运行状态**，进而控制和管理进程(PCB是进程存在的**唯一标志**)

进程镜像：代码段、数据段以及PCB组成

进程的创建和撤销：本质是对PCB的**创建和撤销**

PCB包含的内容：进程在内存中的位置(page table)；对应的可执行文件在磁盘的位置

哪个用户执行这个进程(uid)；进程ID号(pid)；进程运行状态(运行态、等待态...)；调度信息、保存的内核SP、中断栈信息...

进程状态

创建态(Init)：申请一个**空白的PCB**，并向其中**填写进程信息**，由操作系统为其**分配必需资源**

就绪态 (Runnable): 进程获得除CPU外的一切所需资源, 一旦得到CPU调度即可运行

运行态 (Running): 进程在CPU上运行, 一个核同时只能运行一个

阻塞态 (Waiting): 进程等待某一事件而暂停运行

结束态 (Finished): PCB置位, 资源回收

僵尸进程: 当子进程被销毁时, 会归还从父进程获得的资源, 但并非马上就消失掉, 遗留下的数据结构

僵尸进程的销毁: 子进程的结束和父进程的运行是一个异步过程, 即父进程永远无法预测子进程到底什么时候结束; 子进程结束后, 应通知父进程需要调用wait()系统调用取得子进程的终止状态, 销毁其对应的僵尸进程; 由于编程错误或者异常 (子进程被kill) ->僵尸进程; 若父进程比子进程更早的结束, 则由init进程(1号进程)接管

进程和线程

引入**进程**的目的: 更好地使多道程序并发执行, 提高资源利用率和系统吞吐量, 增加并发程度

引入**线程**的目的: 减少程序在并发执行所付出的时空开销, 提高OS的并发能力

引入线程后, 进程将只作为除CPU外的系统资源的分配单元, 线程则作为CPU调度单元 (同一地址空间内的上下文切换代价较小)

线程: OS为正在运行的程序提供的**并发性抽象**, 它可被理解为**轻量级进程**

一个线程代表一组顺序执行的指令序列 (a.k.a, 任务); OS可以像对进程一样, 创建、挂起、恢复、运行线程; 一个线程必须从属于某一个进程, 由线程ID、PC、寄存器集合和堆栈组成, 是基本的CPU执行单元 (它不拥有系统资源); 属于同一个进程的多个线程相互共享进程资源且是互信的(i.e., 线程解耦了并发性 and 保护性)

进程为什么需要线程?

性能: 充分利用**多核CPU**

自然地呈现程序结构: 描述了属于同一程序下的**并发任务**; e.g., 更新屏幕、从磁盘获取数据、获取用户输入信息

高响应性: **交互性线程 (处理用户输入输出)** 与**功能性线程**分离; 功能性线程可以在**后台**运行

隐藏慢速的I/O操作: 在某个线程处于I/O等待状态时, **其他功能线程可以被调度执行**

- | | |
|--------------------------------|------------------------------|
| • 具有数据段, 代码段, 堆栈段 | ➤ 具有自己的栈 |
| • 至少拥有一个线程 | ➤ 必须从属于某一个进程 |
| • 进程终止→ 资源回收、线程终止 | ➤ 线程终止→ 线程拥有的栈回收 |
| • 进程间通信需要通过OS和数据传递 (e.g., 信号) | ➤ 线程间通信只需通过共享内存 (e.g., 全局变量) |
| • 拥有隔离的内存地址空间 (i.e., 其他进程不能访问) | ➤ 同一进程下的其他线程可以访问自己的栈 |
| • 创建进程和进程间切换会带来较高的系统开销 | ➤ 创建线程和线程间切换不会带来较高的系统开销 |

11

一个进程(PCB): 隔离的内存地址空间; 独立的代码段、数据段和堆栈段; 共享的I/O资源; 权限保护;
一个或多个线程(TCB), 拥有独立的栈、改变CPU寄存器(PC, SP)的取值

PCB中的cr3成员变量描述了进程的页表基地址, lcr(next->cr3)标识加载下一个调度的进程next的页表基地址

用户态程序的本质是使用资源来完成某种功能

系统态程序的本质就是管理资源进而更好地为用户服务

进程: **用户态程序所需资源**的抽象集合

线程: **用户态程序提供功能**的抽象集合

调度

可抢占式资源: OS可以很容易将进程A的某些资源收回, 用于其它进程, 并且过段时间可以将相等的资源再次分配给进程A (e.g., CPU)

非抢占式资源: OS不能很容易将进程A的某些资源收回, 只能等待进程A主动放弃这些资源 (e.g., 读写磁盘内容)

资源调度: 进程 (a.k.a., 任务) 拥有资源的时间 (e.g., CPU)

决定进程服务的顺序; 僧多粥少 (时分); 抢占式资源; 核心是**CPU调度**

资源分配: 进程获得**资源**的数量 (e.g., 内存)

决定进程可以获得多少资源; 狼少肉多 (空分); 非抢占式资源; 无线网络是时分和频分

CPU调度：从就绪态任务队列选择下一个使用CPU的任务

磁盘调度：选择下一个对文件或块设备的读写操作

网络调度：选择下一个发送或处理的数据包

内存页替换调度：选择cache中哪个内存页被替换

计算机通过操作系统主要完成**计算、通信和存储**三个方面功能

CPU调度

调度对象：**进程/线程** (i.e., 任务)，例如点击鼠标、网页请求

调度方式：从**就绪态任务队列**选择**下一个使用CPU的任务**

进程调度是非阻塞的过程

从资源角度来看，本质是**交替改变CPU和I/O操作**，分为CPU密集型和I/O密集型任务

从与用户参与度来看，可以分为**批处理**和**交互式**任务(批处理：用户不需要参与，一堆处理，后台自行执行；交互式：需要用户输入等，用户参与，前台处理)

从安全权限角度来看，可以分为**系统级**和**用户级**任务

CPU资源调度(决策指标)

响应时间：用户态程序完成某些功能所需的总时间

初始等待时间：任务第一次被调度的事件 - 到达时间

总体等待时间：线程在就绪态队列中等待的总时间

吞吐量：单位时间CPU完成的任务数

系统开销：完成任务所需的额外资源量(事件、空间)

公平性：每个任务使用CPU的时间或资源量

饥饿度：任务两次被调度之间的等待时间

调度策略

FIFO先来先服务策略：第一个到达的请求CPU的任务获取CPU资源；非抢占式调度

优点：实现简单，调度公平

缺点：任务等待时间取决于任务到来时间，即无法发生抢占操作；头阻塞问题；无法很好地支持交互式任务

SJF最短任务优先策略：试图最小化平均响应时间；使用预测来判断任务长短；可以优化给定任务集合的平均响应时间问题

非抢占式：一旦任务获得CPU后，只有当其完成后才行调度下一个任务

抢占式：若一个任务获得的CPU执行时间少于其所需CPU的执行时间，则发生抢占 (执行最短剩余时间优先)

通常使用**红黑树实现SJF**

优点：为每个任务分配相同的CPU资源；较低的平均等待时间；当任务所需执行时间差别较大时，较低的平均响应时间；当任务数不是特别大时，可以很好地支持交互式任务

缺点：设置时间过大越趋近FIFO；过小则导致过多切换

优先级调度：每个任务分配一个优先级

多级队列调度：根据任务类型存在多个就绪态任务队列；不同的任务队列可以采用不同的任务调度策略

多级反馈队列调度：类似多级队列，但是是动态的，分层调度，每层使用不同的策略，但是需要考虑的参数较多

实时任务调度：具有延时约束要求；终止时间要求；周期性时间；研究热点

多核任务调度：哪个任务在哪个CPU上运行

数据

bss：未初始化或者初始化为0的全局变量

data：初始化非0的全局变量

rodata：只读数据/局部变量

全局变量：线程共享进程的内存，便于线程间的数据通信

多线程访问数据：共享内存全局变量会出现问题。CPU切换，会导致写的顺序出现差错。线程私有数据不会出现问题。

进程和线程的本质就是**调用主线程**

上下文切换的主要任务：1.切换**进程内存空间(堆栈)**。2.切换**CPU寄存器和指令指针寄存器**。

抢占式调度：包含**时间驱动**和**任务被唤醒**两种形式

内存管理

内存管理需要满足的目标

隔离性：不希望不同的进程状态在物理内存中相互冲突

共享性：希望有选择性的共享某些物理内存，实现高效进程间通信

虚拟化：希望为每个进程提供一种“幻觉”，它可以独占所有物理内存->引入虚拟内存

利用率：希望能尽可能的高效利用有限的物理资源

内存管理的任务：虚拟内存的管理；物理内存的管理；虚拟->物理内存的映射

不同进程的相同用户态虚拟地址会映射到不同的物理地址 (隔离)

不同进程的相同内核态虚拟地址会映射到相同的物理地址 (共享)

内核本质也是**ELF文件**

页面替换

页换入操作: 1. 寻找一个**空闲的物理页**

2. 发起**磁盘块加载请求**

3. 阻塞**当前进程P1**

4. 上下文切换到**新的进程**

5. 当磁盘块**加载完毕后更新页表项** (frame号+标识)

6. 将**进程P1**放回到Runnable队列中

页换出操作: 1. 寻找页表中所有与**其相关的表项** (共享页, COW页)

2. 将这些页表项**均置为失效**

3. **清楚TLB表中对应的表项**

4. 如果需要的话, 将**物理内存页的内容**写回**磁盘块**(**代码段和只读数据**不需要写回)

页面替换算法:

Random: 随机换出一个物理页面 (worst case)

FIFO: 根据页面换入顺序, 换出**最早换入**的页面

OPT: Belady's algorithm (**offline**), 换出其未来被访问的时间距离当前最远的页面

LRU: 换出**未使用时间最长**的页面(利用**时间戳**进行页面的标识)

LFU: 换出**单位时间内最少使用**的页面

覆盖技术

原理: 一个程序的**代码段或数据段**, 按照时间先后来占用公共的内存空间, **操作系统可以将程序必要部分(常用功能)的代码和数据常驻内存, 可选部分(不常用功能)平时存放在外存(覆盖文件)中, 在需要时才装入内存**。不存在调用关系的模块不必同时装入到内存, 从而可以相互覆盖。

缺点: 编程时必须划分程序模块和确定程序模块之间的覆盖关系, **增加编程复杂度**; 从外存装入覆盖文件, **以时间延长换取空间节省**。

输入输出设备

设备控制器

CPU通过**写设备控制器的寄存器**, 实现对**控制器下发指令**

CPU通过**读设备控制器的寄存器**, 实现对**I/O设备状态查看**

CPU**读写寄存器**相比直接操作硬件要**更标准、简单**

I/O设备的分类: 块设备和字符设备

块设备: 将**信息存储在固定大小的块中**, 每个块**都有自己的地址**, 例如硬盘, 块大小为4KB且地址是扇区号

字符设备：发送或接受的是字节流，不用考虑任何块结构，没有方式寻址，例如鼠标

若I/O设备传输的**数据量较大**(磁盘或打印机), 其设备控制器会设置**数据缓冲区** (类似内存)。只有缓冲区的数据**达到阈值**才会真正执行读写I/O设备

非阻塞I/O模式->轮询等待

设备控制器的寄存器会有**状态标志位**, 通过该值来确定**某个指令操作是否完成**

while (true) {直至状态标志位显示完成}

会**参与进程/线程调度**, 影响**整体系统性能**

异步I/O模式->中断

OS提供一个**硬件中断控制器**

当硬件完成某个任务触发**中断到硬件中断控制器**, 该控制器会**通知CPU**, CPU会**停下当前执行任务去处理中断** (优先级)

软中断 (系统调用, INT xxH) vs. **硬件中断** (中断控制器触发)

DMA技术: CPU对DMA控制器下指令, 指定从I/O设备读取多少数据到内存的某个地方, DMA控制器会发指令给设备控制器, 后者执行上述读数据任务, 待传输完毕, 设备控制器会通知

设备控制器**不属于OS**的一部分

每种设备控制器的**寄存器、缓冲区的使用模式以及支持的指令都不同**->OS需要将它们的**差异进行屏蔽**

设备驱动程序属于OS的一部分, 它既**包含面向I/O设备控制器的代码**, 又**包括对OS统一的接口**(能够屏蔽设备控制器的差异)

设备驱动程序的本质是内核模块

构建内核模块: 1.**头文件部分**: 内核模块必须包括以下2个头文件#include<linux/module.h> 和 #include<linux/init.h>

2.定义函数用户**处理内核模块的逻辑**

3.定义一个**file_operations结构**, 对于文件系统的操作均需要这一结构进行描述

4.定义**整个模块的初始化函数和退出函数**, 例如在打印机驱动中定义了lp_init和lp_cleanup

5.调用**module_init**和**module_exit**, 分别指向上面的初始化和退出函数module_init(lp_init), module_exit(lp_cleanup)

6.最后调用**MODULE_LICENSE**, 声明一下license

I/O设备工作要素:

需要一个设备驱动程序 (.ko文件): 初始化函数、中断处理函数、设备操作函数; 加载设备驱动时, 初始化函数被调用, 并在cdev_map结构中注册该设备 (利用主/次设备号进行查找), 根据主设备号可以获得该设备的驱动程序

需要在/dev目录下创建设备文件：该文件属于特殊的文件系统devtmpfs, 也需要相应的dentry和inode (这里的inode与磁盘文件不同); 磁盘文件的inode指向文件数据, 而设备文件的inode指向设备的驱动程序 (通过主/次设备号在cdev_map中获得)

打开和读写设备文件, 类似磁盘文件操作：文件描述符fd, file结构, path结构 (inode), file_operations结构

中断处理过程：

中断是从**外部设备**发起, 形成外部中断, 外部中断会**到达中断控制器**, 控制器会发送**中断向量**给CPU

每个CPU都有一个**中断向量表(idt_table)**, 里面存放了**不同的中断向量处理函数**

硬件中断处理函数的**统一接口是do_IRQ()**, 它会把**中断向量通过vector_irq映射为irq_desc中断描述结构**

irq_desc中的成员irqaction描述了**用户注册的中断处理函数结构**

文件系统

背景：内存只能暂存数据，如果需要进程结束后数据仍然保存，就需要存储在外部存储中（例如磁盘）

知识（数据）->书籍（文件）->图书馆（磁盘）

文件系统：图书馆书籍管理系统

块：4KB 扇区：512B

文件

元数据：由操作系统添加的信息，包括文件大小、文件创建者、文件修改日期、访问权限等（**inode、index node**）

应用数据：用户产生的具体数据

磁盘数据的存储形式

文件创建时必须命名，进程根据文件名访问文件内容

文件名=ascii码 + 和扩展名

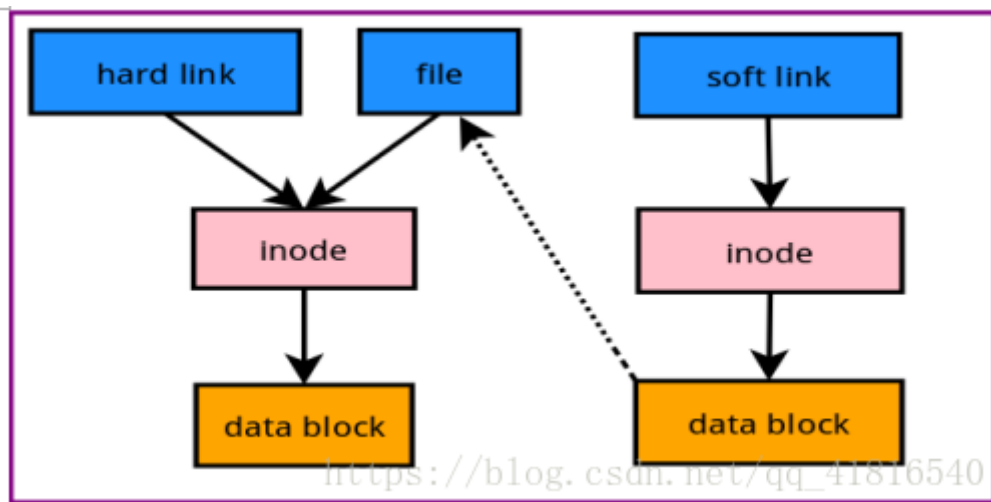
目录

绝对路径：从根目录开始到文件

相对路径：从当前工作目录开始到文件

硬连接：将一个文件名映射到某个文件或目录（多个文件绑定，同时修改；相当于文件的另一个路口）

符号连接：将一个文件名映射到另一个文件名（类似快捷方式，符号链接文件存储的是源文件的路径）



文件和目录都有**标号**

文件和卷

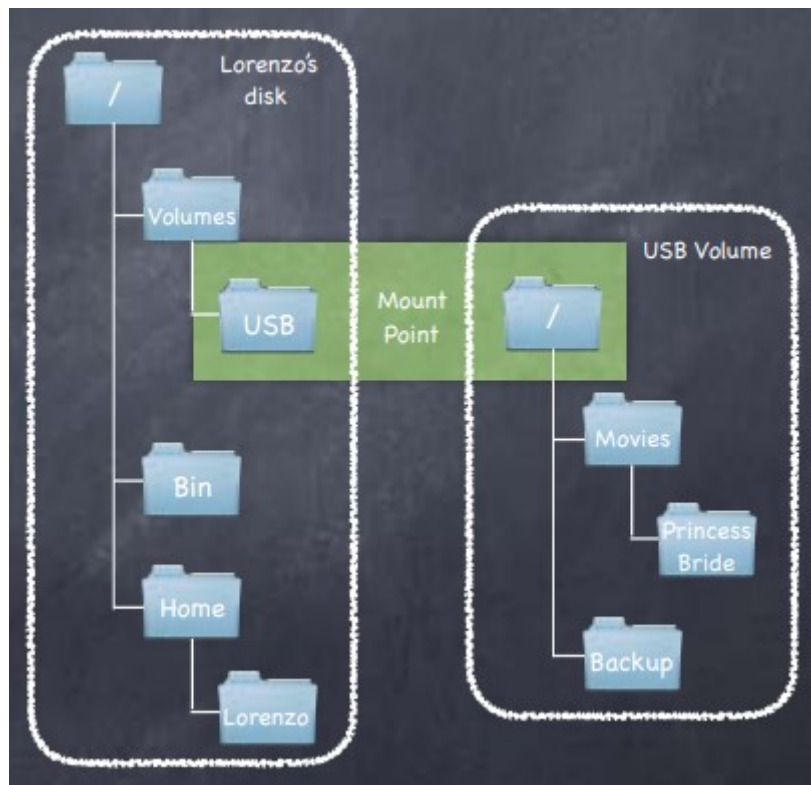
卷：一系列物理资源块形成的逻辑存储设备

挂载：将某个卷和某个文件系统建立连接（usb）

本质：将文件系统中的某个路径与被挂载的卷的根目录建立映射关系

磁盘文件系统

虚拟文件系统



文件基本操作

创建、写、读、删除、在文件末尾添加新内容

本质：将文件名与磁盘块进行映射

fd：文件描述符

作用域是当前进程，所有操作都需要fd

设计磁盘文件系统的挑战

高性能（磁盘的空间仍然有限）

普适性（文件系统需要支持各种类型的文件；大多数文件属于小文件，所以体积不是很大，因此设计的磁盘块不宜过大；对大文件的访问需要实现高效性，即如何快速寻找相关的磁盘块）

持久性（在磁盘上高效地管理和更新用户数据）

可靠性（发生os异常或读写异常时可以保持存储数据不被破坏，或可以很快回复）

（注：目录：会根据文件名找到文件号（Inode=index node）；文件：根据文件号确定所属地磁盘块；好的文件系统一定要让同一文件的磁盘块位于相近的位置？？）

磁盘文件系统

概述

磁盘文件系统<-格式化磁盘

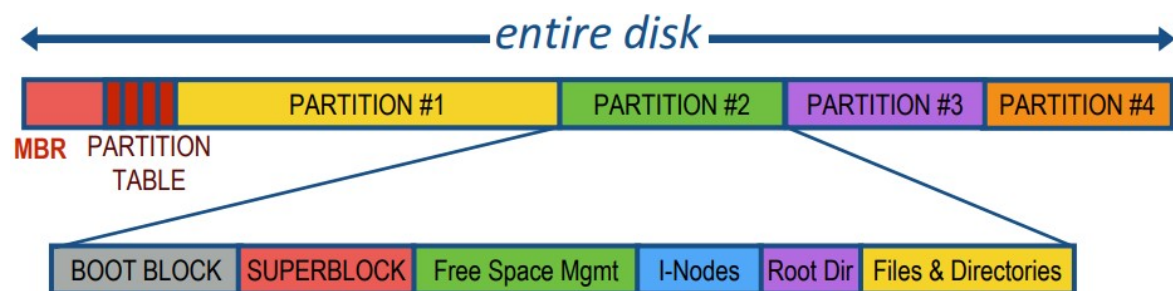
存储在磁盘上

分区方式划分磁盘

0盘0面1扇区为引导扇区MBR（Master Boot Record）

紧跟着MBR的是磁盘分区表

每个分区第一个块是启动块，由MBR填入，系统启动时会执行MBR，会执行每个分区的启动块



获取数据

建立索引：用于定位每个文件的每个磁盘块（一般会按照树形结构存储）

空闲地址映射：提出快速的分配机制（一般会采用bitmap实现）

局部性原则：尽量将相同文件的靠近内容放置在物理相近的磁盘块（便于seek）

文件存储方式

连续分配原则：按创建顺序及大小依次占用磁盘块

优：实现简单，每个文件只需要记录器初始和终止磁盘块；高效，整个文件读写仅需要依次seek操作

劣：容易碎片化，特别是当大量文件被移动或删除时；不适合文件大小频繁变化的情景（编写代码、文档）

链表方式：

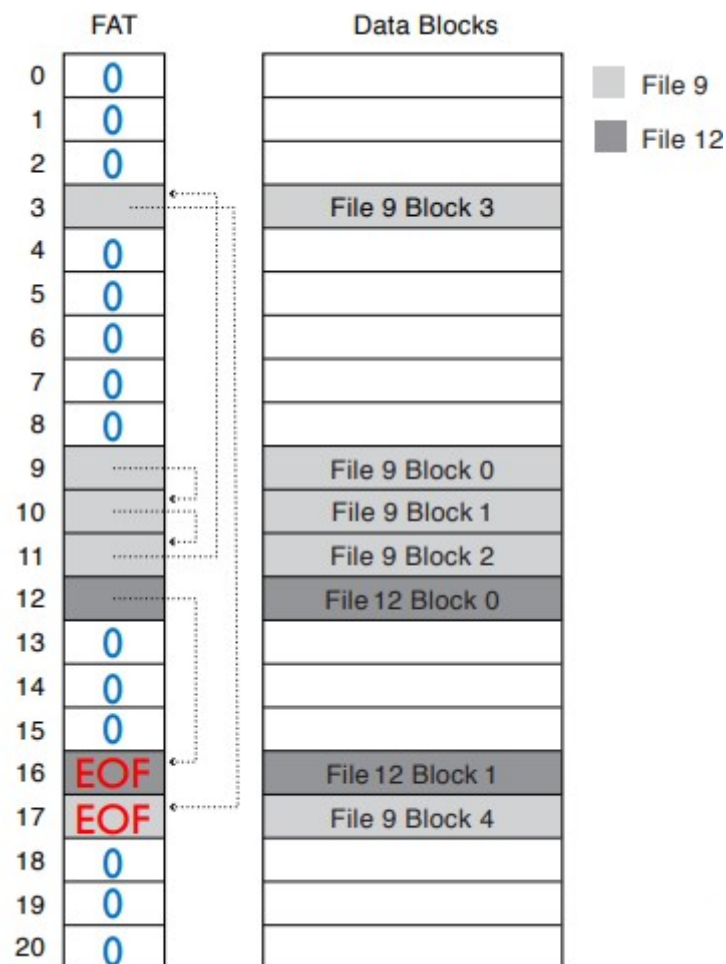
优：实现简单，文件的每个磁盘块只需维护指向下一个磁盘块的指针

劣：性能不稳定，特别是随机访问文件中的内容（seek频繁）；空间开销问题，每个磁盘块均需要空间来保存指针

FAT文件系统

70年代，用于DOS和Windows

文件分配表：采用链表形式；将指针部分分离，形成以<key, value, next>结构为表项的链表；需要为每个文件(key)指明其首个磁盘块(value)。



表项为0，没有使用；EOF：某文件的最后一块；每个磁盘块对应一个FAT表项；目录项描述的是一个文件名和其首个磁盘块的关系

目录项->FAT表项->磁盘块

优缺点

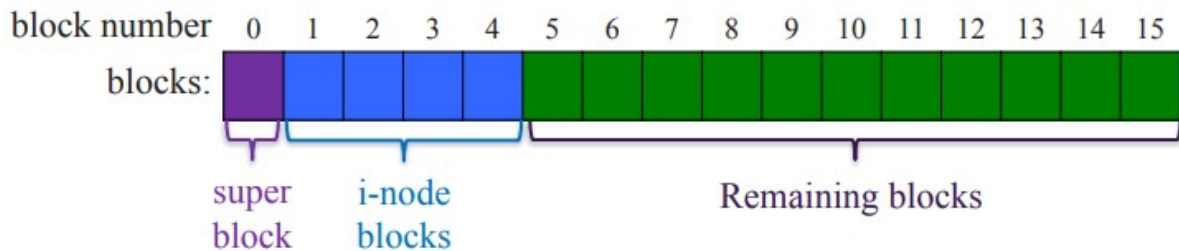
优点：实现简单；较少的磁盘碎片；目录和FAT表均在启动时加载到内存中，磁盘块只保存用户数据

缺点：磁盘Seek可能耗时较高；很难提供可靠性；存储目录和FAT表内存开销可能很大（e.g., 1TB (40 位) 磁盘, 块大小为4KB (12位), 则需要表项为 $2^{(40-12)} = 2^{28}$, 若每个表项为4 bytes, 则需要 $2^{(28+2)} = 2^{30}$ B)

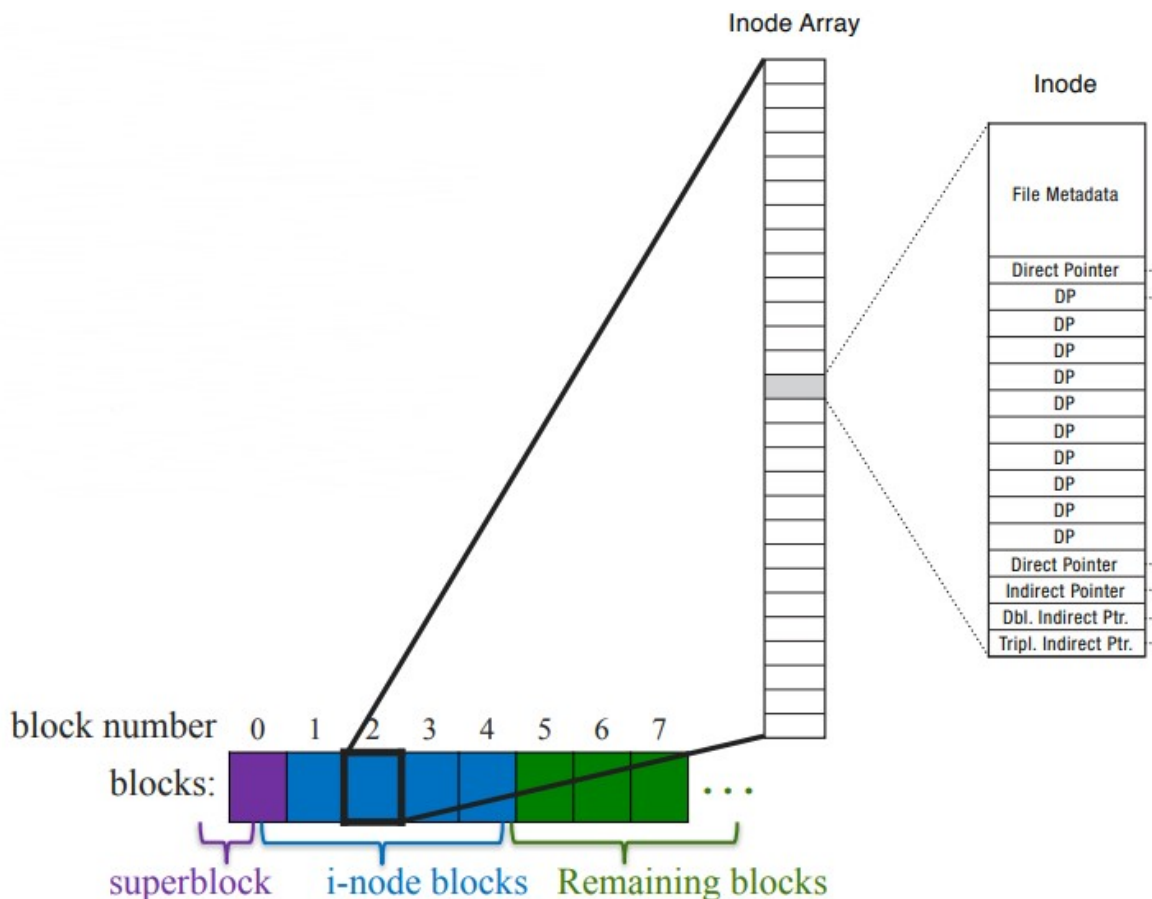
FFS文件系统

80年代，UNIX以及Linux ext2, ext3

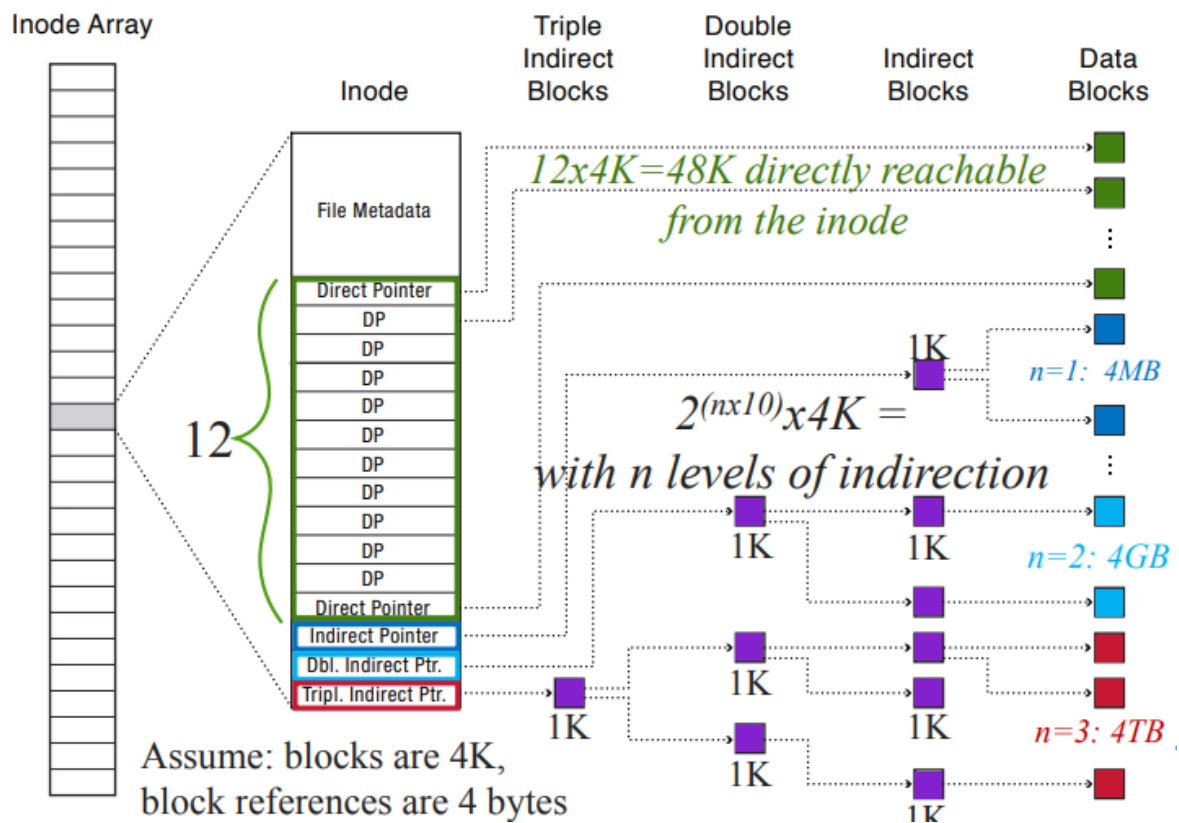
采用**树形-多级索引方式**：每个文件是棵非对称的树，根节点Inode，所有节点大小均固定(磁盘块大小 4KB)；多级索引可以很好地支持大体积文件和小体积文件；新颖的空间地址映射和局部性分配



超级块：指明文件系统的关键参数(文件系统类型；块大小，inode array的起始位置和长度；空闲块的起始位置)



每一个Inode块是一个Inode Array；Inode array中每个节点是Inode；Inode包含文件的元数据, 12个数据块指针DP以及3个间接指针



File Metadata存储着(类型: 一般文件, 目录, 符号连接等; 文件总大小; 文件所有者 (user id, group id); 权限; 时间: 创建、访问、修改等)

间接指针是多级索引, 适用于大文件的存储; 直接指针用于小文件的存储

为什么简介指针指向1K个数据块(4KB / 4B = 1K)

特点

树形结构->很快地找到所需地磁盘块数据

局部性分配(多级指针)->最小化Seek时间, 可以较好地支持连续读写

固定的结构->易于实现

非对称性: 数据磁盘块不在同一层->支持大体积文件->小体积文件不会产生过多的开销

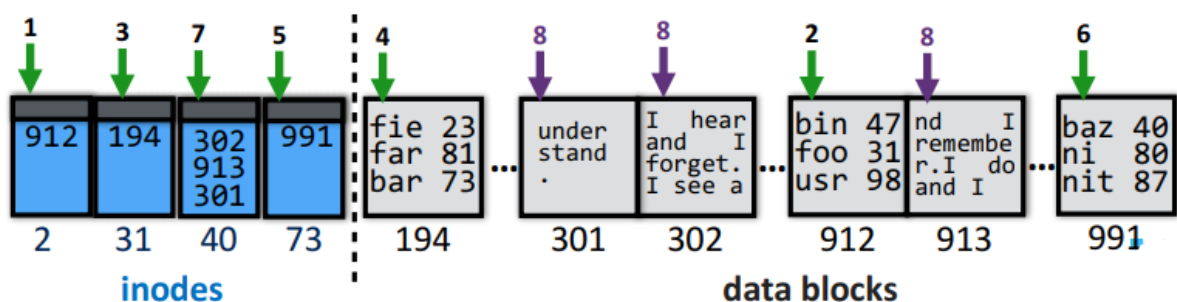
目录机构

采用**线性表结构**

数组项包括: inode号4位; 文件名长度; 文件名

第一个数组项为. 指向自己

第二个数组项为.. 指向上一个inode结点



打开并读取文件/foo/bar/baz

根据**根目录**的inode=2, 找到其所在的**磁盘块912**

根据根目录地址912, 找到**目录foo**的inode=31

根据foo目录的inode=31, 找到其所在**磁盘块194**

以此类推直至找到991块中的**bar文件**inode=40

从inode=40找到该**文件的内容**

空闲地址映射和局部性分配

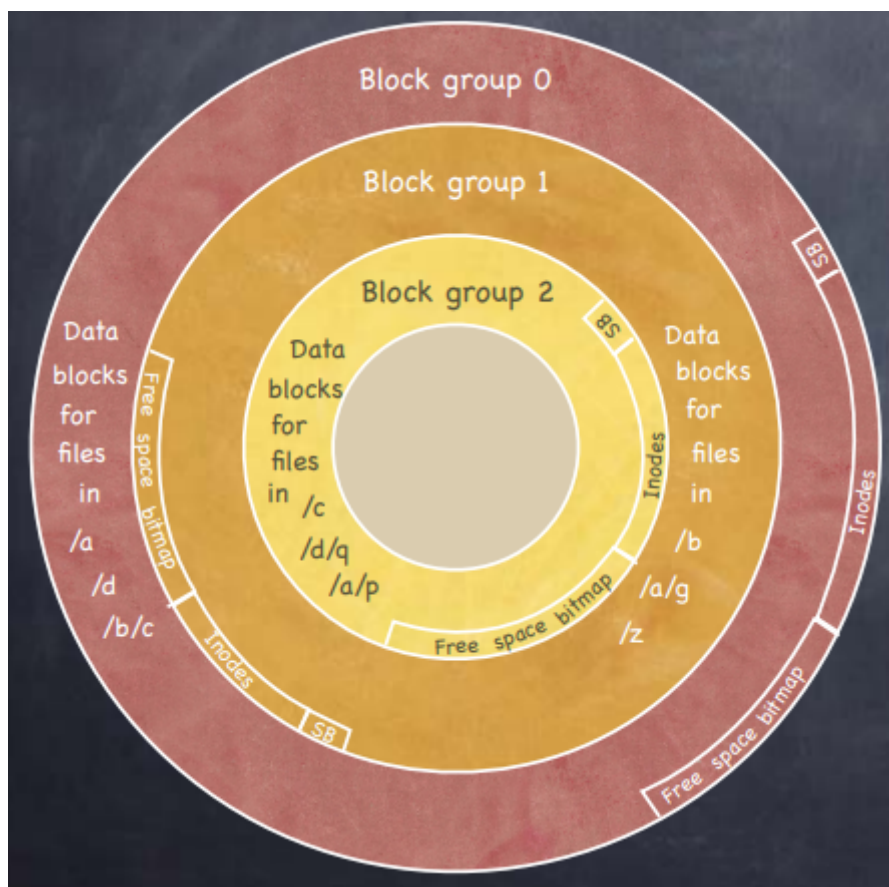
引入bitmap, 每一位表示一个磁盘块

将磁盘分成块组, 每个块组代表一系列相邻的磁道

均匀地将空闲bitmap和inode array分发给这些块组, 并且每个块组均具有一份超级块的拷贝

在当前目录下, 若创建一个新目录。FFS会将当前目录视为新目录地父目录并为其分配一个磁盘块

若创建文件, 则在当前磁盘块内找一个空闲inode, 根据bitmap将数据徐徐放置在空闲数据块内



优缺点

优点: 可以很好地支持大体积和小体积文件; 实现了局部性放置; 固定地字段格式使得实现较为简单

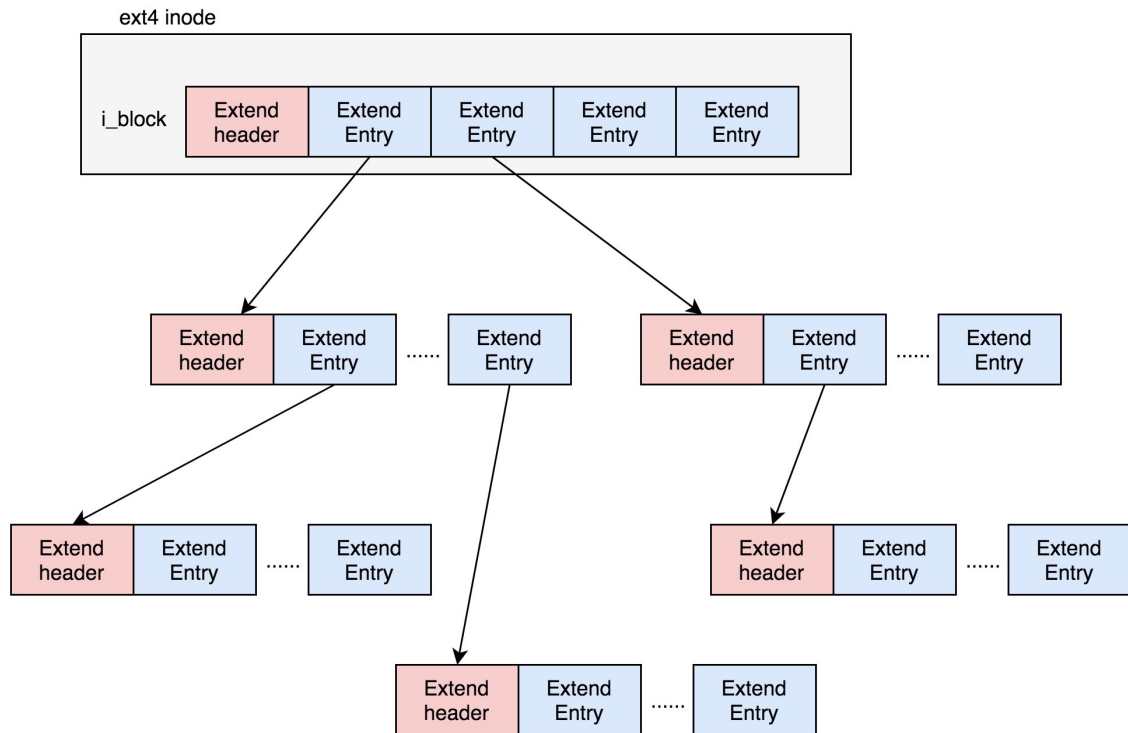
缺点: 不太适合体积极小地文件(tiny file), inode 可能比有效数据量还大, 空间利用率低; 需要预留 10%-20%磁盘空间给块组, 实现局部性放置

NTFS: 引入新概念Extents

extents动机: 加入一个文件128M, 磁盘块4KB, 则需要32K个块; 若使用FFS, 需要2级间接指针; 数据块的位置可能过于分散, 导致seek时间比较长; extents可以用来存放连续的块, 树形结构:

header+entry

其中entry类型为：ext4_extent(数据)和ext_extent_idx(索引)



头部表明：**entry最大128M？！**

数据中为什么是低32位高16位？

根节点只有4个entres，每个entry只想一个数据或索引

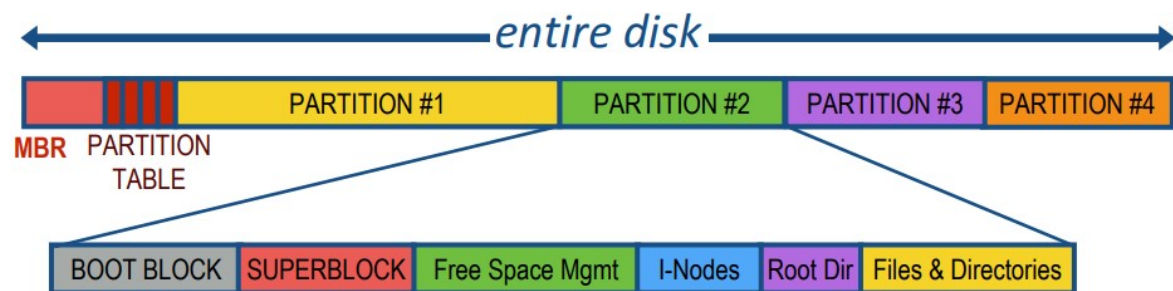
数据extent，叶子结点，每个entry指向了一篇连续的数据块，**最大128M**

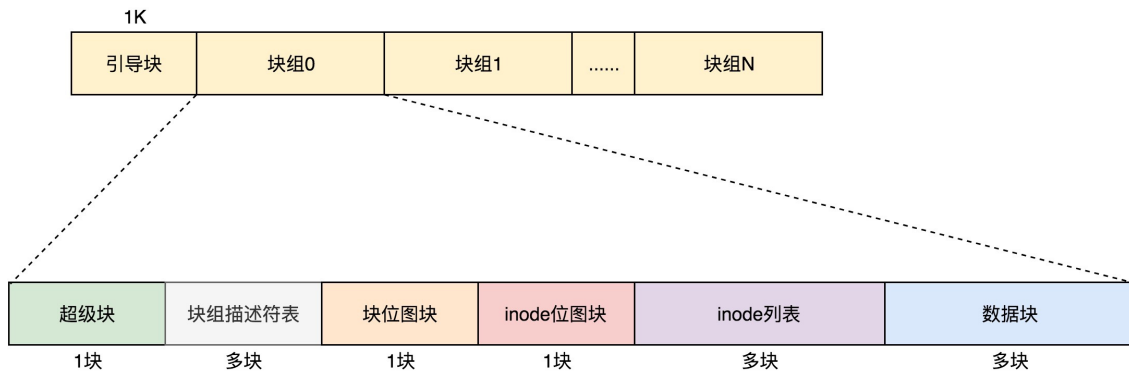
索引|extent，指向另一个数据或者索引|extent

head: 12B entry: 12B

索引|extent 的整体大小为4K

文件系统布局





为什么是128MNB：位图bitmap占1块描述4KB，4KB等于32K个数据块(bitmap 1位表示1个数据块)，所以32K个数据块存储128MB的大小

描述块组的数据结构(局部信息)：块组的块位图，inode位图，inode列表，数据块列表都有相应的成员变量

块组描述符表(全局信息)

超级块(整个文件系统的信息，全局信息)：inode总数, 块总数, 每个块组的inode数目, 每个块组的数据块数目

全局信息必须备份，这会产生冗余，采用何种备份方式？

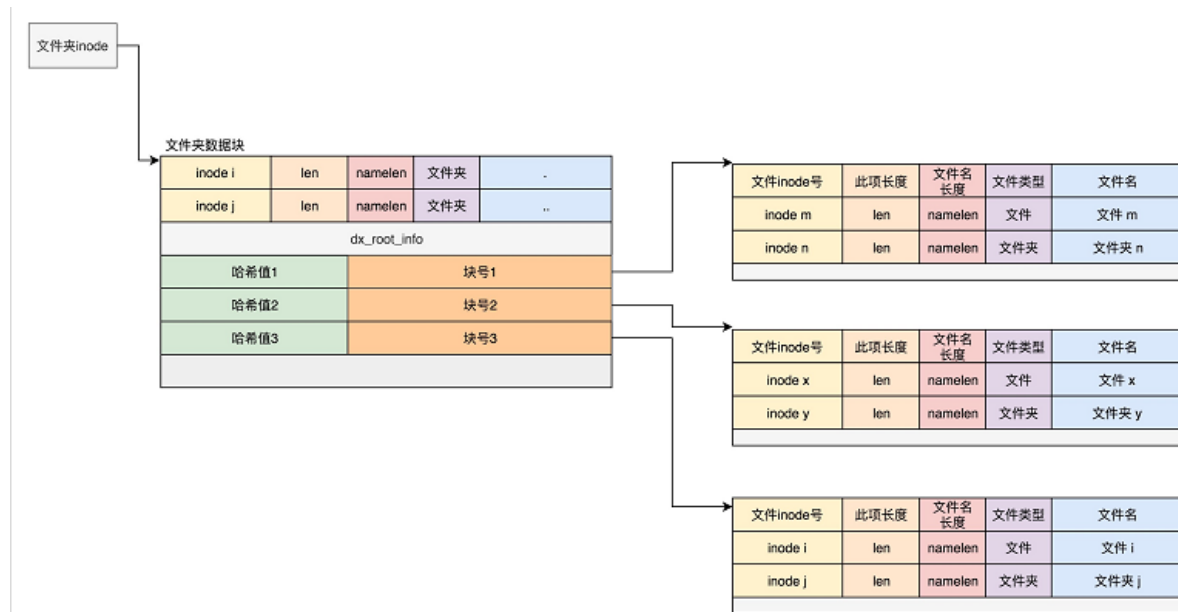
备份策略

超级块(通常所占内存不多)：每个块组都备份；sparse_super模式：副本只存在块组索引为0、3、5、7的整数幂

块组描述符表(所占内存较多)：每个块都备份(狼粪，限制文件系统大小)；**sparse_super模式(限制文件系统大小)：**一个块组最大128MB，若给定块组数目为N，则文件系统大小为128N；**meta block group模式：**将块组进行再次分组，64个块组为一个元块组，1个元块组的描述符表最多64项；每个块组备份自己的元块组描述符表，一般采用3个备份，分别为第一块、第二块以及最后一块

目录存储格式

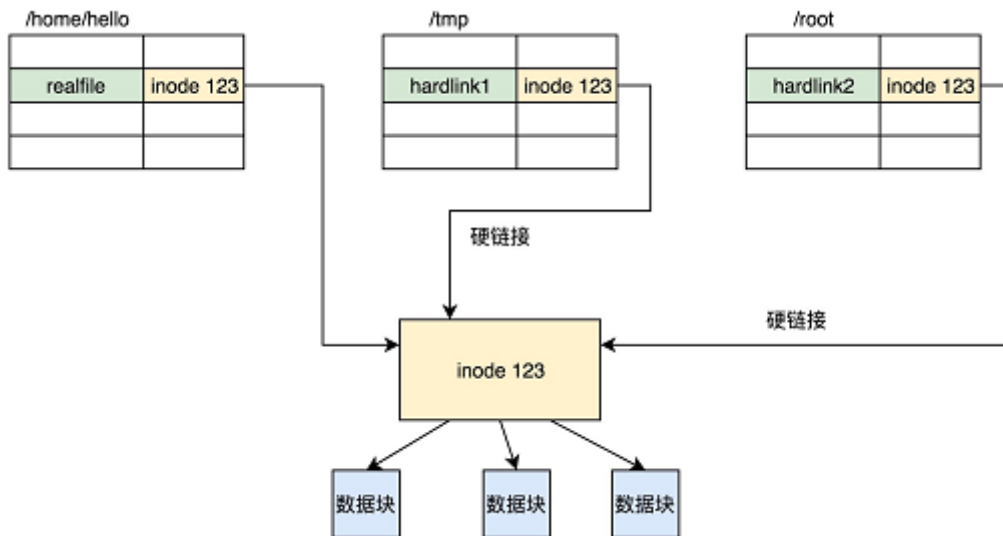
如果inode中iflags设置为EXT4_INDEX_FL：目录的块组织形式将从链表形式变为Hashed Tree结构



硬连接和符号连接的存储格式

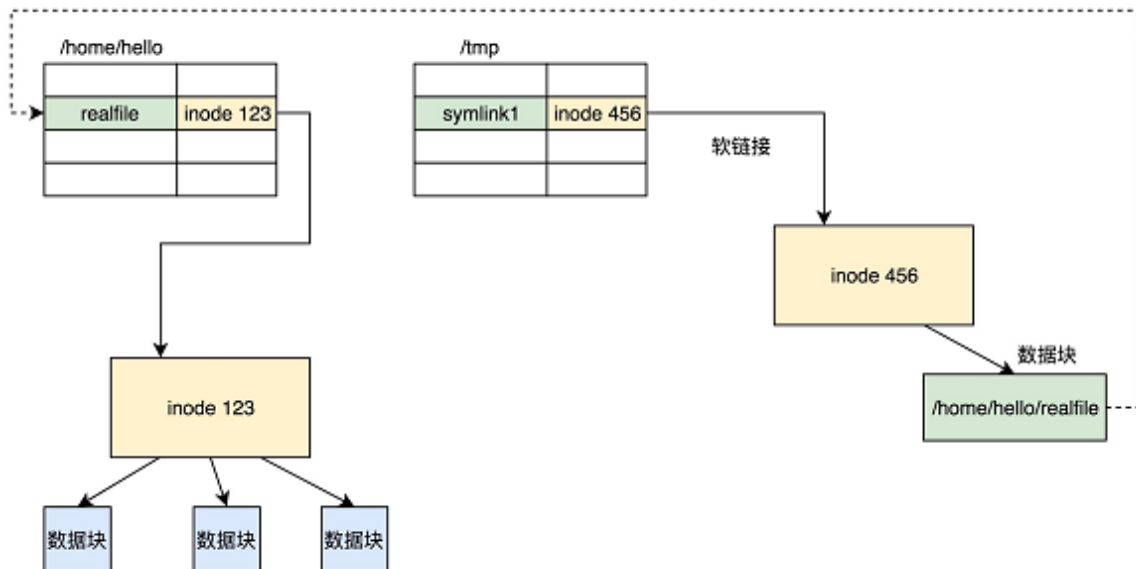
硬连接：与原始文件共用一个inode

inode不能跨文件系统->硬链接不能跨文件系统



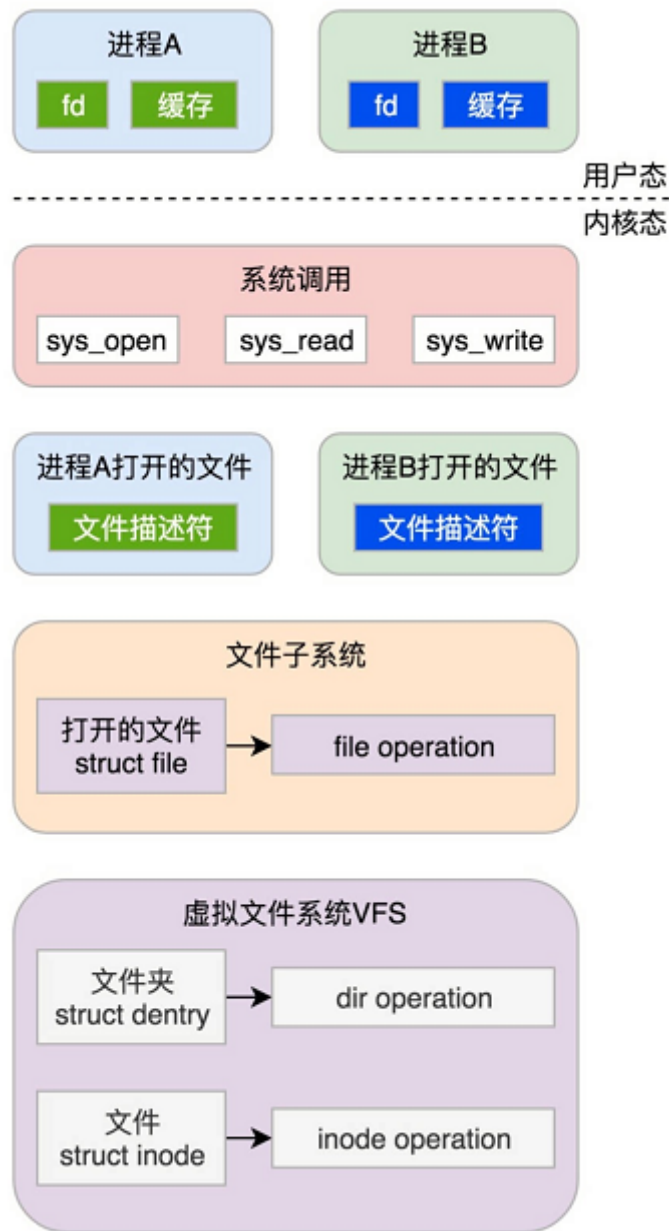
符号连接：具有独立的inode

访问符号链接文件，其实是访问指向的另一个文件，可跨文件系统



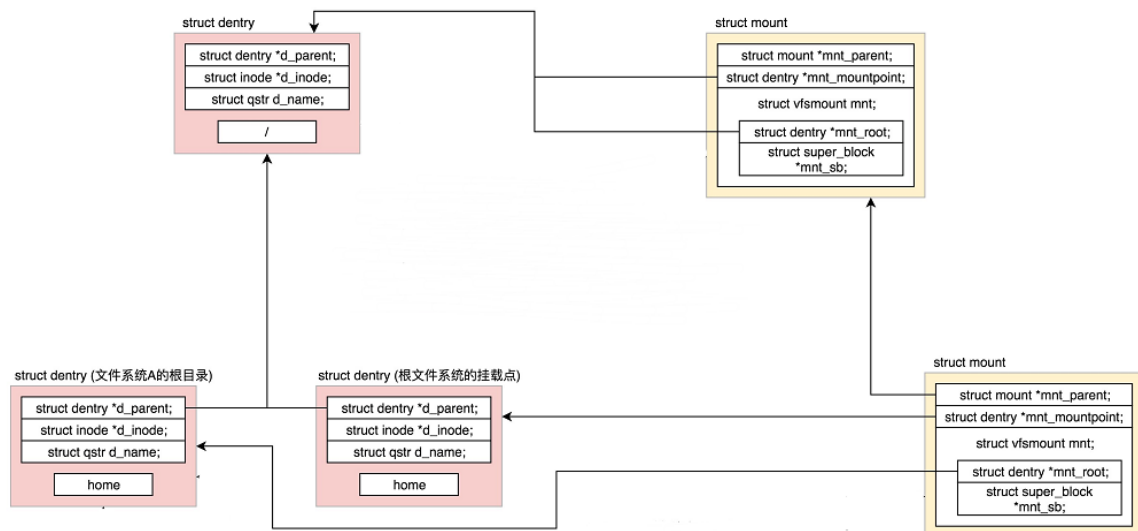
虚拟文件系统

磁盘文件系统的镜像



mount系统调用

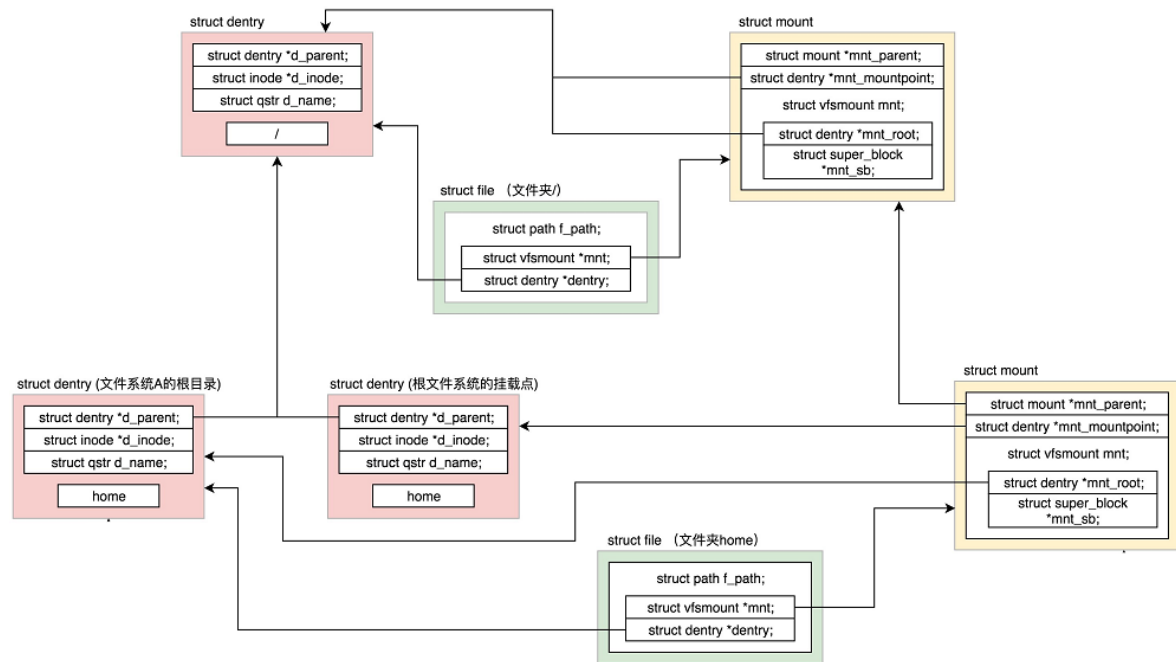
磁盘文件系统：需要首先要挂载文件系统，且必须在内核中注册



将磁盘文件系统A的根目录挂载到虚拟文件系统/home:

这里注意文件系统A的根目录和根文件系统的挂载点, 它们内容相同, 但是属于不同的内存空间 (原因是可以有其它的文件系统挂载到/home上, 此时用到的是挂载点那个内存, 当然前提是该挂载点与A的根目录unmount)

open系统调用



进程的task_struct有个指针成员, 类型是files_struct结构体; file_struct中存在**文件描述符表**, 每一项是 `<struct file*, in/out>` 实质就是fd

sys_open->do_sys_open->do_filp_open 创建struct file结构

->fd_install (fd, f) 创建文件描述符表项 (关联file和fd)

do_filp_open 会初始化struct nameidata, 即解析待打开文件的路径, nameidata中存在关键成员 struct path, 与文件系统建立连接

task_struct->files_struct (fd) ->file (fd与映射)->path (磁盘文件系统):

vfsmount (虚拟与磁盘文件系统映射) + dentry (inode)->文件读写操作

进程间通信

进程间为什么要通信?

数据传输 (如web服务、进程查询数据库): 一个进程需要将它的数据发送给另一个进程

事件通知 (如进程终止时要通知父进程): 一个进程需要向另一个或一组进程发送消息, 通知它们发生了某种事件

资源共享 (互斥和共享): 多个进程之间共享同类资源

进程控制 (如debug程序代码): 有些进程希望完全控制另一个进程的执行, 即能够拦截另一个进程的陷入和异常, 并能够及时知道另一个进程的状态改变

进程间如何通信?

单机:

管道模型 进程->内存文件系统(pipe对象)->进程

消息队列模型 进程->内存文件系统(消息队列对象)->进程

信号 进程->OS内核->进程

共享内存模型(同步和互斥) 进程->映射到相同的物理内存->进程

管道模型**消耗额外内存最多！！**

多机：

socket机制

RPC机制

管道模型

类比于项目开发

匿名管道(用完即销毁)和**命名管道**(以文件形式出现)

问题：**通信频率高、通信量大**(缓冲区有限)

消息队列模型

类比于邮件：需要对消息格式进行定义；进程间发送消息采用固定内存大小的数据单元 (i.e., 消息体)

消息队列的操作十分程式化(**目前使用度不高**)

信号

OS内核与进程之间的通信模式

信号触发的**动机**：内核发现一个系统事件，例如**除0错误或子进程退出**；一个进程调用**kill命令**，请求OS发送信号给目标进程，例如调试、挂起、恢复或超时等等

信号**接收的机制** (进程收到内核发送的信号)：忽略信号；终止本进程；用一个定义的用户态信号处理函数去捕捉信号

共享内存模型

管道模型、消息队列模型以及信号都是**端到端且数据传输量较小**的IPC, 有些情形需要**多进程协作, 且数据交换量较大**->**共享内存模型**

每个进程有**独立的虚拟内存空间**，映射到不同的物理内存中，即**不同进程访问同一虚拟内存**，本质访问的**物理内存不同 (隔离性)**

如果每个进程拿出一块虚拟内存, 映射到相同的物理地址, 这样一个进程写入的东西, 另一个进程就可以读出, 不需要发送多个消息

类似于进程内的多线程，需要**同步和互斥机制(信号量)**

信号量

P操作->申请资源操作(资源量减)

V操作->归还资源操作(资源量加)

联合体中val = 共享资源总量

计算机网络系统

TCP

TCP是**面向连接**的

TCP提供**可靠交付**, 无差错、不丢失、不重复

TCP面向**字节流**

TCP可以提供**流量控制和拥塞控制**

建立连接: 为了客户端和服务端建立一定的数据结构维护双方交互的状态

双方的状态符合TCP协议的规则, 就认为连接存在, 否则连接断开

流量控制和拥塞控制就是根据收到的对端数据包, **调整两端的数据结构状态**

存储方式

大端存储: 高位字节存放在内存的低地址端, 低位字节存放在内存的高地址端(便于强制转换)

小端存储: 低位字节存放在内存的低地址端, 高位字节存放在内存的高地址端(便于判断符号位)

TCP/IP协议规定网络字节顺序采用大端模式进行编址

listen函数

inet_csk函数的作用是将**sock指针**强制转换成**inet_connection_sock指针**

其本质是: tcp_sock结构第一个成员为inet_connection_sock对象,
inet_connection_sock结构第一个成员为inet_sock对象,
inet_sock结构第一个成员为sock对象, 而初始化套接字时
sock *sk其实指向的是tcp_sock对象 (**TCP套接字**)

Remote Method Invoke/Procedure Call模型

传统的socket编程

反射机制: 计算机程序在运行时可以访问, **检测和修改**它本身状态或行为的一种能力;

在面向对象的编程语言中, 反射允许在编译期间不知道接口(类)的名称, 字段、方法的情况下在运行时检查类、接口、字段和方法。

反射机制

静态编译：在编译时确定类型, 绑定对象

动态编译：运行时确定类型, 绑定对象，可以降低类之间的藕合性

动态编译在Web服务中的**优点：**当更新web服务时，若使用静态编译则需要把整个程序重新编译一次才可以实现功能的更新，而采用反射机制可以不用卸载，只需要在运行时才动态的创建和编译，就可以实现该功能。

缺点：相对于静态编译其响应时间会慢, 影响性能