前操作系统中的应用情况分析

[前操作系统中的应用情况分析](#header-n0)  
 [一．前言](#header-n3)  
 [二．进程调度](#header-n16)  
 [（一）进程调度所用到的数据结构](#header-n26)  
 [（二）进程调度相关算法](#header-n68)  
 [（三）新版Linux出现的调度机制](#header-n71)  
 [三. 死锁问题](#header-n79)  
 [（一）Linux检测死锁方法](#header-n83)  
 [（二）Linux解决死锁方案](#header-n87)  
 [四. 虚拟存储器](#header-n92)  
 [（一）虚拟存储器与虚拟页面](#header-n98)  
 [（二）逻辑地址与物理地址翻译](#header-n114)  
 [五. 文件管理](#header-n139)  
 [（一）Linux文件系统模型](#header-n143)  
 [（二）如今Linux主要存储方式](#header-n146)  
 [1. inode table](#header-n150)  
 [2. data block](#header-n152)  
 [3. superblock](#header-n154)  
 [4. 文件（档案）系统描述](#header-n156)  
 [5. 块对照表](#header-n158)  
 [6. inode对照表](#header-n160)  
 [六. 系统安全性](#header-n164)  
 [七. 结语](#header-n182)  
 [引用](#header-n185)

### 一．前言

Linux操作系统，已经成为世界第二大主流操作系统，其中，最吸引人的特性便是开源，这使得无数开发者可以随心所欲的进行二次开发，做出最适合自己的操作系统。并且Linux开源的特性使得我们可以更加便捷的了解一个操作系统的原貌与整体设计。

Linux主要有下面及部分组成：

* Linux内核
* Linux Shell
* Linux 文件系统
* Linux 应用程序

这几部分构成了整个Linux系统，撑起了整个Linux大系，下面我不会根据组成来分析，而是根据《计算机操作系统》一书相关目录来进行，但是无论如果，都不会脱开上面四部分的联系。

### 二．进程调度

首先便是进程调度问题，众所周知，这个世界是异步的，操作系统也是如此，在Linux系统中，总共有三类可执行调度：

* SCHED\_OTHER:是以非实时的进程为主，结合进度调度分析，进行动态性的先进性（FIFO）进行调度分析，进行系统化的进程分析，对于非实时调度进程来说，当CPU运行时，可利用优先级的进程抢占，只有当CPU实时进程开展过程中，应以SCHED-OTHER为主，相应进程都不可运行。
* SCHED\_FIFO：作为一种实时进程，应遵守POSIX1.b的标准化调度原则，并直到其能够主动释放相关CPU，从而针对性相应更高优先级 被迫化释放。在Linux的实现的过程，应以适当的进程结合时间片用完时才会被迫释放对应的CPU。
* SCHED*RR：也作为一种实时进度，应遵循相关规范和标准，实时控制调度规则，使得其适当的适应SCHED*FIFO。类似进行分析，当SCHED\_RR的时间进程片段信息进行用完后，可充分的结合优先级信息进行适应性调整。

下面则是针对这些进程调度与Linux源码所作的分析：

#### （一）进程调度所用到的数据结构

在kernel/sched/core.c文件中，内核为每一个cpu创建了一个进程就绪队列

DEFINE\_PER\_CPU\_SHARED\_ALIGNED(struct rq, runqueues);

在这里， 有一个结构体rq， 通过查找， 我们发现其在kernel/sched/sched.h中有着定义。该结构体是本地cpu所有进程组成的就绪队列，在linux内核中，进程被分为普通进程和实时进程，这两种进程的调度策略是不同的。

同时，进程结构中还有一些调度参数的结构体，比如sched\_parma:

#include <sched.h>  
  
struct sched\_param   
{   
 int32\_t sched\_priority;   
 int32\_t sched\_curpriority;   
 union   
 {   
 int32\_t reserved[8];   
 struct   
 {   
 int32\_t \_\_ss\_low\_priority;   
 int32\_t \_\_ss\_max\_repl;   
 struct timespec \_\_ss\_repl\_period;   
 struct timespec \_\_ss\_init\_budget;   
 } \_\_ss;   
 } \_\_ss\_un;   
}  
  
#define sched\_ss\_low\_priority \_\_ss\_un.\_\_ss.\_\_ss\_low\_priority  
#define sched\_ss\_max\_repl \_\_ss\_un.\_\_ss.\_\_ss\_max\_repl  
#define sched\_ss\_repl\_period \_\_ss\_un.\_\_ss.\_\_ss\_repl\_period  
#define sched\_ss\_init\_budget \_\_ss\_un.\_\_ss.\_\_ss\_init\_budget

其中的成员包括：

* sched\_priority
  + 当获得调度参数时，该成员反映出分配给线程或进程的优先级。它不反映任何由于优先级继承而造成的临时调整。
  + 当您设置调度参数时，请将此成员设置为您想要使用的优先级。优先级必须介于sched*get*priority*min()和sched*get*priority*max()为调度策略返回的最小值和最大值之间。
* sched\_curpriority
  + 当您获得调度参数时，该成员被设置为线程或进程当前运行的优先级。这是内核在做出调度决策时使用的值。
  + 当您设置调度参数时，该成员将被忽略。
* sched*ss*low\_priority
  + 正在执行的后台或低优先级线程
* sched*ss*max\_repl
  + 补给计划的最大次数，通常是由于阻塞操作。在一个线程多次阻塞之后，它会自动将其执行的剩余部分降至低优先级，直到其执行预算得到补充。
* sched*ss*repl\_period
  + The time that should be used for scheduling the replenishment of an execution budget after being blocked or having overrun the maximum number of replenishments. This time is used as an offset against the time that a thread is made READY.
* sched*ss*init\_budget
  + 应该用于线程执行预算的时间。当线程运行在其高优先级级别时，它的执行时间就从这个预算中划分出来了。一旦预算完全耗尽，线程就会降到低优先级，如果可能的话，由于优先级安排，线程可以继续运行，直到执行预算得到补充。

#### （二）进程调度相关算法

//调度策略  
#define SCHED\_OTHER 0  
#define SCHED\_FIFO 1  
#define SCHED\_RR 2  
#ifdef \_\_USE\_GNU  
# define SCHED\_BATCH 3  
#endif  
  
struct sched\_param {  
/\* ... \*/  
int sched\_priority; //优先级  
/\* ... \*/  
};  
  
//设置进程的调度策略和优先级  
//第一个参数为进程pid;  
//第二个参数为调度策略；  
//第三个参数为优先级；  
int sched\_setscheduler (pid\_t pid,  
 int policy,  
 const struct sched\_param \*sp);

可以看到， 根据不同的宏定义值， 可以选择不同的进程调度机制。

#### （三）新版Linux出现的调度机制

**限制实时线程的CPU使用时间**

SCHED*FIFO, SCHED*RR的线程如果内部是一个非阻塞的死循环，那么它将一直占用CPU，使得其它线程没有机会运行；

在2.6.25以后出现了限制实时线程运行时间的新方式，可以使用RLIMIT\_RTTIME来限制实时线程的CPU占用时间；Linux也提供了两个proc文件，用于控制为非实时线程运行预留CPU时间；

/proc/sys/kernel/sched\_rt\_period\_us

这个文件中的数值指定了总CPU(100%)时间的宽度值，默认值是1,000,000；

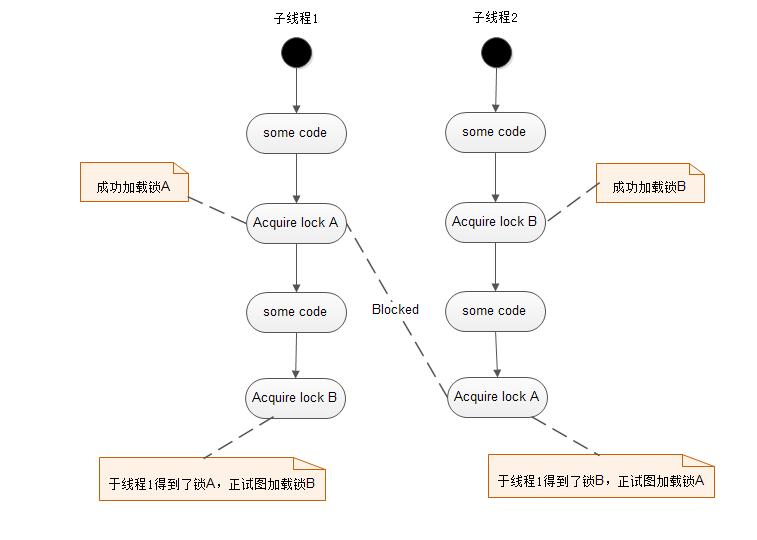
/proc/sys/kernel/sched\_rt\_runtime\_us

这个文件中的数值指定了实时线程可以运行的CPU时间宽度，如果设置为-1，则认为不给非实时线程预留任何运行时间，默认值是950,000，因为第一个文件的总量是1,000,000，也就是说默认配置为非实时线程预留了5%的CPU时间；

### 三. 死锁问题

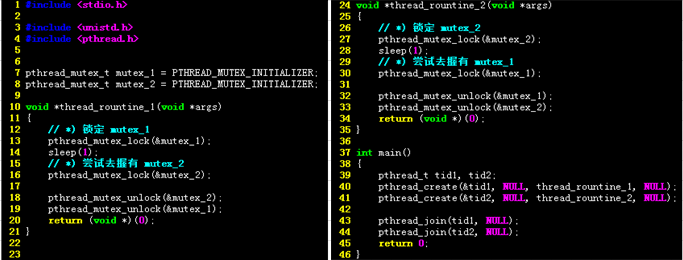
死锁（DeadLock)是指两个或者两个以上的进程（线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程(线程）。由于资源占用是互斥的，当某个进程提出申请后，使得有关进程（线程）在无外力协助下，永远分配不到必需的资源而无法继续进行，这就产生了一种特殊现象——死锁。

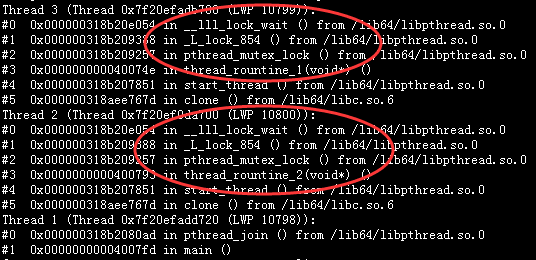
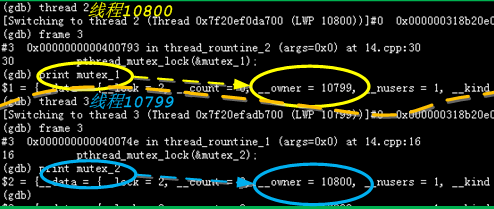
一种交叉持锁死锁的情形，此时执行程序中两个或多个线程发生永久堵塞（等待），每个线程都在等待被其他线程占用并堵塞了的资源。例如，如果线程1锁住了记录A并等待记录B，而线程2锁住了记录B并等待记录A，这样两个线程就发生了死锁现象。在计算机系统中，如果系统的资源分配策略不当，更常见的可能是程序员写的程序有错误等，则会导致进程因竞争资源不当而产生死锁的现象。



#### （一）Linux检测死锁方法

我们来模拟一个出现死锁的程序, 然后通过常规方式来确定是否出现了死锁, 以及在那些线程上出现的. 如下是经典的死锁程序:



使用**pstack来快速扫描堆栈**:  发现有两个线程均在lock中等待, 存在死锁的嫌疑, 需要gdb后具体确认.  解读: **线程10800申请mutex\_1**(此时被线程10799所有), 而**线程10799申请mutex\_2**(被线程10800所有), 于是线程10800在等待线程10799的释放, 而线程10799在等待线程10800的释放, 于是我们可以确定发生死锁了。 但这种方式, 需要开发人员自己去**验证和排除**, 复杂的案例就并不轻松了。 在gdb中, 我们可以只能看到mutex对应的线程, 却无法**反向获取到线程拥有的mutex列表**, 如果有这个信息, 就像**jstack**工具那样, 获取对死锁的判定, 只要扫下堆栈信息, 就能基本的判定了。

#### （二）Linux解决死锁方案

通过上面的分析和猜测，我们可能想到如果我们能像java反射一样, 拦截lock/unlock操作, 并添加汇报线程与锁关系的功能, 那自然能构建有向图. 进而实现自动检测死锁情况。 但是C/C++没有反射, 不过可以在所有的lock/unlock代码处添加桩代码, 并得以解决.。但这对使用方的代码具有侵入性, 能否改善呢?

在Linux源码里，我们发现了解决方案——宏扩展

#include <sys/syscall.h>  
   
#define gettid() syscall(\_\_NR\_gettid) // 用于获取id  
   
// 拦截lock, 添加before, after操作, 记录锁与线程的关系  
#define pthread\_mutex\_lock(x) \  
 do { \  
 printf("before=>thread\_id:%d apply mutex:%p\n", gettid(), x); \  
 pthread\_mutex\_lock(x); \  
 printf("after=>thread\_id:%d acquire mutex:%p\n", gettid(), x); \  
 } while (false);  
   
// 拦截unlock, 添加after操作, 解除锁和线程的关系  
#define pthread\_mutex\_unlock(x) \  
 do { \  
 pthread\_mutex\_unlock(x); \  
 printf("unlock=>thread\_id: %d release mutex:%p\n", gettid(), x); \  
 } while(false);

这样，Linux就实现了检测与解决死锁的方案了。

### 四. 虚拟存储器

系统中的进程共享CPU和主存资源，但存储器空间是有限的，而且还容易被破坏。现代系统提供了一种对主存的抽象，称为虚拟存储器，以更有效地管理存储器。虚拟存储器将主存看作磁盘上的地址空间的高速缓存，为每个进程提供了一致的地址空间，并保护进程的地址空间不被其他进程破坏。

1、虚拟存储器机制，是现代计算机采用的内存机制，其优点是让进城拥有独立的地址空间（虚拟地址空间），给进程独占内存的假象，其好处有很多。

2、虚拟存储器的概念，广义来说是指统一以逻辑地址（虚拟地址）来标识地址，再进行地址转换到实际地址的整套机制。狭义来说，是一个虚拟的存储器区域，包括物理内存，和磁盘上的swap区间，如下图所示，虚拟内存 = 物理内存 + 交换空间。

在Linux中，主要则是分页式管理。

#### （一）虚拟存储器与虚拟页面

虚拟存储器（VM）为一个由存放在磁盘上的N个连续的字节大小的单元组成的数组，每个字节有唯一的虚拟地址。磁盘上数组的内容以块为单位被缓存在主存中，虚拟存储器分割的块称为虚拟页（VP），物理存储器分割的块称为物理页（PP，页帧），它们的大小都为P = 2^p字节。

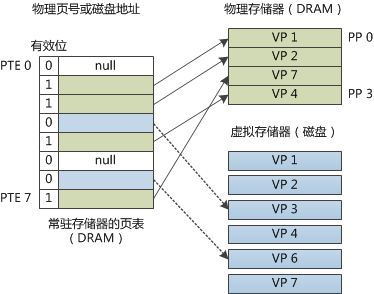
虚拟页面的集合可以分为：

* 未分配的：VM系统还未分配的页，它们没有关联的数据，不占用磁盘空间。
* 未缓存的：没有缓存在物理存储器中的已分配页。
* 缓存的：缓存在物理存储器中的已分配页。

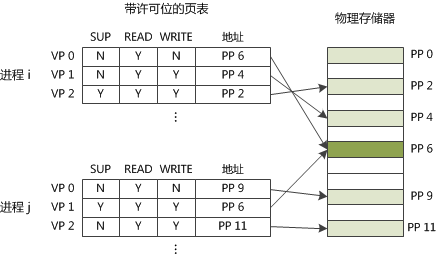
将虚拟存储器系统的缓存称为DRAM缓存。磁盘比DRAM慢得多，它的不命中处罚和访问第一字节的开销都很大。因此虚拟页较大（一般为4 ~ 8KB），而且DRAM缓存是全相联的（虚拟页可以放在任何物理页），替换策略也很重要，并总是使用写回。

页表将虚拟页映射到物理页，它是存放在物理存储器中的一个数据结构，地址翻译硬件转换地址时都会读取页表。操作系统负责维护页表的内容和在磁盘与DRAM之间传送页。

页表是一个PTE（页表条目）的数组，为了简化，设PTE由一个有效位和一个n位地址字段组成。设置了有效位表明虚拟页为已缓存的，地址指向DRAM中相应物理页的起始位置；未设置有效位且地址为空表明虚拟页未分配，否则地址指向磁盘上虚拟页的起始位置。



操作系统为每个进程都提供了一个独立的页表，即一个独立的虚拟地址空间。多个虚拟页面可以映射到同一个共享物理页面上。

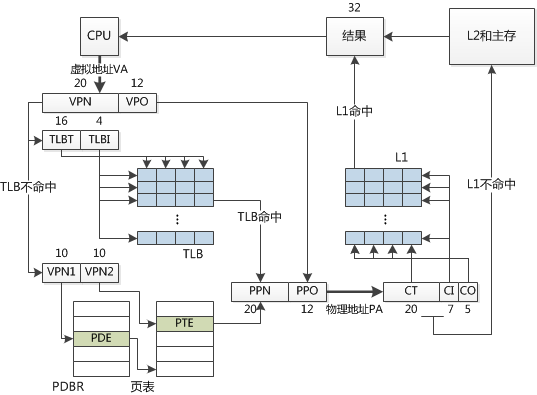


#### （二）逻辑地址与物理地址翻译

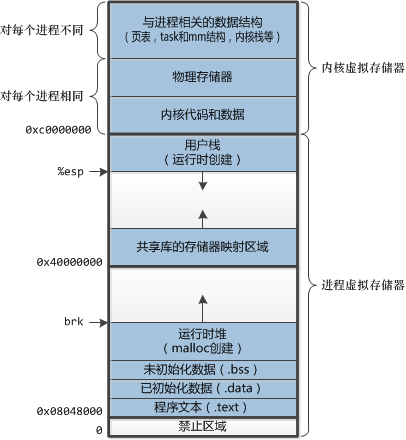
在Pentium/Linux存储器系统，有32位地址空间、4KB的页大小、TLB、L1、L2为4路组相联：

* 指令TLB：32个条目、8组。
* 数据TLB：64个条目、16组。
* L1 i-cache和d-cache：16KB、128组、32B块大小。
* L2高速缓存：128KB ~ 2MB、32B块大小。

Pentium系统使用两级页表，第一级页表称为页面目录，包含1024个32位的PDE（页面目录条目），PDE指向二级页表，每个二级页表包含1024个32位的PTE，PTE指向物理存储器或磁盘上的页面。每个进程有唯一的页面目录和页表集合，页表可以换进换出，页表目录和已分配页面的相关页表常驻存储器。PDBR（页面目录基址寄存器）指向页表目录的起始位置。



同时，Linux中虚拟存储器布局有着严格的规定



虚拟存储器被组织成区域（段）的集合，区域是已分配的连续组块，以某种方式相关联。不属于某区域的虚拟页是不存在的，不能被引用。区域允许了虚拟地址空间有间隙。

内核为每个进程维护一个任务结构 task\_struct ，结构中包含运行进程所需的信息。其中一个条目指向 mm\_struct ，它描述了虚拟存储器的当前状态。其中两个字段， pgd 指向页面目录表的基址， mmap 指向 vm\_area\_struct 链表。每个 vm\_area\_struct 描述当前虚拟地址空间的一个区域。

在源码中有如下定义：

/\*\* <linux/mm\_types.h> \*/  
struct vm\_area\_struct {  
 unsigned long vm\_start; /\* 区域开始处 \*/  
 unsigned long vm\_end; /\* 区域结束处 \*/  
 struct vm\_area\_struct \*vm\_next; /\* 链表中下一个区域结构 \*/  
 pgprot\_t vm\_page\_prot; /\* 区域内所有页面的读写权限 \*/  
 unsigned long vm\_flags; /\* 区域内的页面是共享的还是私有的, ... \*/  
 /\* ... \*/  
};

当查询时，需要对虚拟存储器与磁盘对象做映射查询，而将虚拟存储器区域和磁盘上的对象关联起来，来初始化这个区域的内容，称为存储器映射。

fork 函数创建新进程时，内核创建当前进程的 mm\_struct 、 vm\_area\_struct 和页表的原样副本，标记两个进程的每个页面为只读的，标记两个进程的每个区域结构为私有的写时复制的。

execve 函数加载和执行程序时，替代当前程序。具体地，删除已存在的用户区域，映射私有区域（文本和数据区映射为程序文件的文本和数据区，bss区映射到匿名文件，栈和堆初始长度为0），映射共享区域，设置程序计数器（PC）。

mmap 函数提供了用户级的存储器映射：

#include <sys/mman.h>  
  
/\*\* 要求内核创建新的虚拟存储器区域，最好从addr开始，将fd指定的对象的一个连续的组块映射到该新区域  
 \* @return 返回指向映射区域的指针，出错返回-1  
 \* @addr 建议起始地址，一般用NULL  
 \* @length 组块字节大小  
 \* @prot 访问权限位  
 \* PROT\_EXEC：区域内页面由可执行指令组成  
 \* PROT\_READ：区域内页面可读  
 \* PROT\_WRITE：区域内页面可写  
 \* PROT\_NONE：区域内页面不能访问  
 \* @flags 描述映射对象类型  
 \* MAP\_ANON：且fd为NULL，匿名对象  
 \* MAP\_PRIVATE：私有的写时复制对象  
 \* MAP\_SHARED：共享对象  
 \* @fd 文件描述符  
 \* offset 距文件开始处的字节偏移 \*/  
void \*mmap(void \*addr, size\_t length, int prot, int flags,  
 int fd, off\_t offset);  
/\*\* 删除从虚拟地址addr开始的length字节的区域  
 \* @return 返回0，出错返回-1 \*/  
int munmap(void \*addr, size\_t length);

这样，逻辑地址就可以被映射为物理地址了。

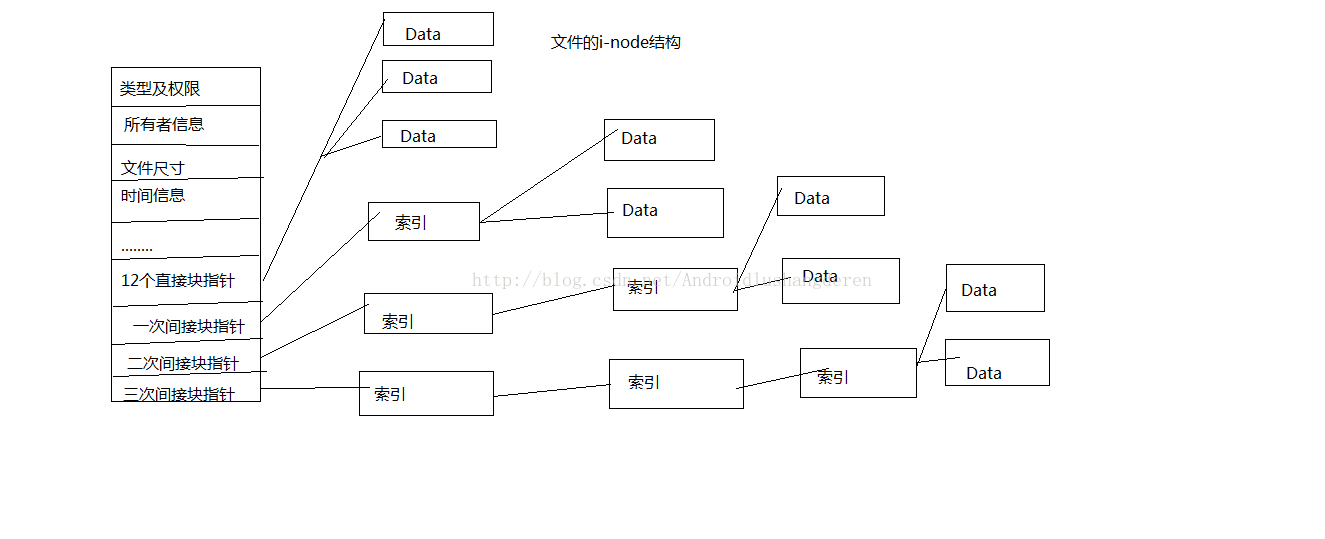
### 五. 文件管理

[计算机](https://zh.wikipedia.org/wiki/计算机)的**文件系统**是一种存储和组织计算机数据的方法，它使得对其访问和查找变得容易，文件系统使用**文件**和**树形目录**的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念，用户使用文件系统来保存数据不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的那个块地址没有被使用，硬盘上的存储空间管理（分配和释放）功能由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。

而Windows下的文件系统又与Linux的文件系统大相径庭，但在这里我们只讨论Linux的文件系统。

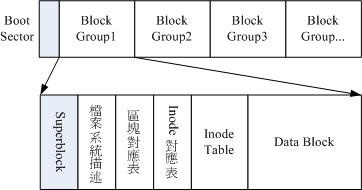
#### （一）Linux文件系统模型

在Linux中，主要采用索引链表法，这其实也就是数据库中的索引的方式。把索引的地址与真实物理数据分开存放，避免了多次的IO操作，直接将索引表放在内存中，一个索引表项中记录了下一个物理块的位置。在内存中逐个查找，总比在磁盘中提供反复的打开文件等这么频繁的IO操作来的效率高。



#### （二）如今Linux主要存储方式

如今Linux主要使用的时EXT4文件系统，示意图如下：



其中，每一个Block Group又有着很多分支，其中每个分支结构又如下：

##### 1. inode table

主要记录文件的属性以及该文件实际数据是放置在哪些block中，它记录的信息至少有这些： 大小、真正内容的block号码（一个或多个） 访问模式(read/write/excute) 拥有者与群组(owner/group) 各种时间：建立或状态改变的时间、最近一次的读取时间、最近修改的时间 没有文件名！文件名在目录的block中！ 一个文件占用一个 inode，每个inode有编号 Linux 系统存在 inode 号被用完但磁盘空间还有剩余的情况 注意，这里的文件不单单是普通文件，目录文件也就是文件夹其实也是一个文件，还有其他的也是 inode 的数量与大小在格式化时就已经固定了，每个inode 大小均固定为128 bytes (新的ext4 与xfs 可设定到256 bytes) 文件系统能够建立的文件数量与inode 的数量有关，存在空间还够但inode不够的情况 系统读取文件时需要先找到inode，并分析inode 所记录的权限与使用者是否符合，若符合才能够开始实际读取 block 的内容 inode 要记录的资料非常多，但偏偏又只有128bytes ， 而inode 记录一个block 号码要花掉4byte ，假设我一个文件有400MB 且每个block 为4K 时， 那么至少也要十万条block 号码的记录！inode 哪有这么多空间来存储？为此我们的系统很聪明的将inode 记录block 号码的区域定义为12个直接，一个间接, 一个双间接与一个三间接记录区（详细见附录）

##### 2. data block

放置文件内容数据的地方 在格式化时block的大小就固定了，且每个block都有编号，以方便inode的记录 原则上，block 的大小与数量在格式化完就不能够再改变了(除非重新格式化) 每个block 内最多只能够放置一个文件的资料，但一个文件可以放在多个block中（大的话） 若文件小于block ，则该block 的剩余容量就不能够再被使用了(磁盘空间会浪费) 所以如果你的档案都非常小，但是你的block 在格式化时却选用最大的4K 时，可能会产生容量的浪费

##### 3. superblock

记录整个文件系统相关信息的地方，一般大小为1024bytes，记录的信息主要有： block 与inode 的总量 未使用与已使用的inode / block 数量 一个valid bit 数值，若此文件系统已被挂载，则valid bit 为0 ，若未被挂载，则valid bit 为1 block 与inode 的大小 (block 为1, 2, 4K，inode 为128bytes 或256bytes)； 其他各种文件系统相关信息：filesystem 的挂载时间、最近一次写入资料的时间、最近一次检验磁碟(fsck) 的时间 Superblock是非常重要的， 没有Superblock ，就没有这个文件系统了，因此如果superblock死掉了，你的文件系统可能就需要花费很多时间去挽救 每个块都可能含有superblock，但是我们也说一个文件系统应该仅有一个superblock 而已，那是怎么回事？事实上除了第一个块内会含有superblock 之外，后续的块不一定含有superblock，而若含有superblock则该superblock主要是做为第一个块内superblock的备份，这样可以进行superblock的救援

##### 4. 文件（档案）系统描述

这个区段可以描述每个block group的开始与结束的block号码，以及说明每个区段(superblock, bitmap, inodemap, data block)分别介于哪一个block号码之间

##### 5. 块对照表

如果你想要新增文件时要使用哪个block 来记录呢？当然是选择「空的block」来记录。那你怎么知道哪个block 是空的？这就得要通过block bitmap了，它会记录哪些block是空的，因此我们的系统就能够很快速的找到可使用的空间来记录 同样在你删除某些文件时，那些文件原本占用的block号码就得要释放出来， 此时在block bitmap 中对应该block号码的标志位就得要修改成为「未使用中」

##### 6. inode对照表

与block bitmap 是类似的功能，只是block bitmap 记录的是使用与未使用的block 号码， 至于inode bitmap 则是记录使用与未使用的inode 号码

通过对于的Linux文件系统的分析，我知晓了为什么公司基本都选用Linux系统作为服务器系统，不仅是因为其资源占用少，还是因为其文件系统的稳健性，其可以快速通过日志文件恢复数据，尽可能小的避免损失，实在是公司极度需要的需求。

### 六. 系统安全性

“Linux 有可能是最安全的，但要求用户是资深用户。”——Moore

正如Moore所言，世界上的专家们的明确共识是便是—— Linux 是最安全的操作系统。

然而，这种安全却是建立在基础知识完善的前提下，毕竟，Linux是开源内核，所有bug与漏洞都显露无疑，但同时，这给了Linux无数免费劳动力修改这些代码，正如Simplex Solutions 的 CISO Igor Bidenko 所说，编码人员可以阅读和审查彼此工作的现实看起来像是一场安全噩梦，但这确实是让 Linux 如此安全的重要原因。 “Linux 是最安全的操作系统，因为它的源代码是开放的。任何人都可以查看它，并确保没有错误或后门。”

而Linux安全技术又有

* 最小化安装
* 关闭端口
* seLinux安全策略
* 防火墙
* shell脚本监控

等等等等，这些每个都是非常大的方面，可以说，开源给了Linux一定的风险，却又给了Linux无尽的安全性和扩展性，属实利大于弊。

### 七. 结语

无论是Linux还是Windows操作系统，都为计算机处理与工作提供了一个大环境。做好环境维护工作是用户开展工作的基础。操作系统依旧只有不断优化和提升，才能防止黑客通过最底层入侵计算机设备。所以，本篇文章或许时效性不长，但却是用心之作。

2019.12.20

### 引用

1. [Linux调度器免锁优化方法研究](https://kns.cnki.net/kcms/detail/detail.aspx?filename=XXWX201704006&dbcode=CJFQ&dbname=CJFD2017&v=)[J]. 张旭,顾乃杰,苏俊杰. 小型微型计算机系统. 2017(04)
2. [Linux操作系统探讨](https://kns.cnki.net/kcms/detail/detail.aspx?filename=WXXJ201912107&dbcode=CJFQ&dbname=CJFDTEMP&v=)[J]. 刘鲁昊. 计算机产品与流通. 2019(12)
3. 一种Linux内核自旋锁死锁检测机制的设计与实现\_张文盛 2012(0)
4. [嵌入式Linux任务调度算法实时性优化](https://kns.cnki.net/kcms/detail/detail.aspx?filename=ZXDB201026025&dbcode=CJFQ&dbname=CJFD2010&v=)[J]. 李玉奇,和晓军,王子复. 科技创新导报. 2010(26)
5. [Linux下的多线程编程](https://kns.cnki.net/kcms/detail/detail.aspx?filename=YYKJ200105011&dbcode=CJFQ&dbname=cjfd2001&v=)[J]. 杨传安,王国夫,张海勋. 应用科技. 2001(05)
6. [Linux同步机制研究](https://kns.cnki.net/kcms/detail/detail.aspx?filename=DNZS201004046&dbcode=CJFQ&dbname=CJFD2010&v=)[J]. 陈友贵,王兆平. 电脑知识与技术. 2010(04)
7. [linux虚拟存储器详解](https://blog.csdn.net/abc86319253/article/details/45506805)
8. [Linux文件系统存储方式](https://blog.csdn.net/Androidlushangderen/article/details/41843169)