Linux 系统结构详解

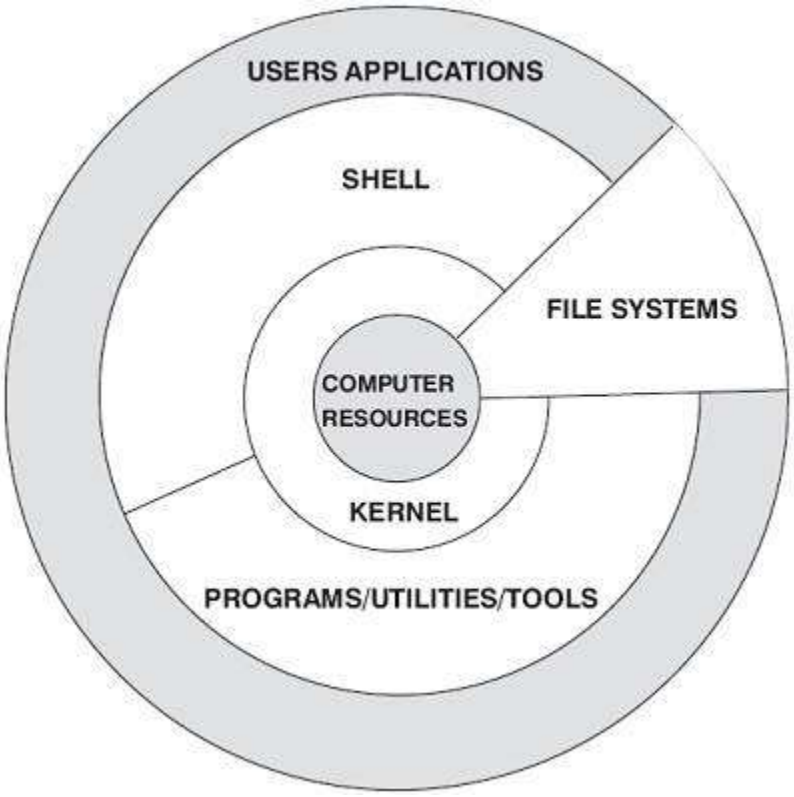
目录

Linux 操作系统

链接： http://blog.csdn.net/hguisu/article/details/6122513

Linux 系统一般有 4 个主要部分：

内核、shell、文件系统和应用程序。内核、shell 和文件系统一起形成了基本的操作系统结构，它们使得用户可以运行程序、管理文件并使用系统。部分层次结构如图 1-1 所示。



1. Linux 内核

内核是操作系统的核心，具有很多最基本功能，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

Linux 内核由如下几部分组成：内存管理、进程管理、设备驱动程序、文件系统和网络管理等。如图：

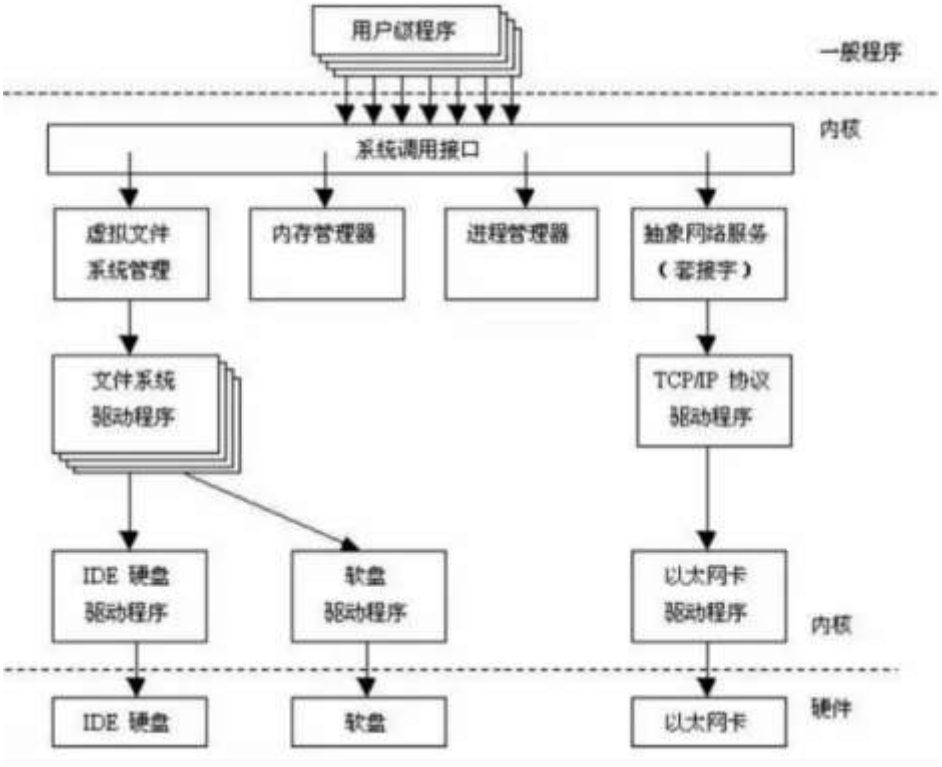


图 1

系统调用接口：SCI 层提供了某些机制执行从用户空间到内核的函数调用。这个接口依赖于体系结构，甚至在相同的处理器家族内也是如此。SCI 实际上是一个非常有用的函数调用多路复用和多路分解服务。在 ./linux/kernel 中您可以找到 SCI 的实现，并在 ./linux/arch 中找到依赖于体系结构的部分。

2. 内存管理

对任何一台计算机而言，其内存以及其它资源都是有限的。为了让有限的物理内存满足应用程序对内存的大需求量，Linux 采用了称为“虚拟内存”的内存管理方式。Linux 将内存划分为容易处理的“内存页”（对于大部分体系结构来说都是 4KB）。Linux 包括了管理可用内存的方式，以及物理和虚拟映射所使用的硬件机制。

不过内存管理要管理的可不止 4KB 缓冲区。Linux 提供了对 4KB 缓冲区的抽象，例如 slab 分配器。这种内存管理模式使用 4KB 缓冲区为基数，然后从中分配结构，并跟踪内存页使用情况，比如哪些内存页是满的，哪些页面没有完全使用，哪些页面为空。这样就允许该模式根据系统需要来动态调整内存使用。

为了支持多个用户使用内存，有时会出现可用内存被消耗光的情况。由于这个原因，页面可以移出内存并放入磁盘中。这个过程称为交换，因为页面会被从内存交换到硬盘上。内存管理的源代码可以在 ./linux/mm 中找到。

3. 进程管理

进程实际是某特定应用程序的一个运行实体。在 Linux 系统中，能够同时运行多个进程，Linux 通过在短的时间间隔内轮流运行这些进程而实现“多任务”。这一短的时间间隔称为“时间片”，让进程轮流运行的方法称为“进程调度”，完成调度的程序称为调度程序。

进程调度控制进程对 CPU 的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际上是仅等待 CPU 资源的进程，如果某个进程在等待其它资源，则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

通过多任务机制，每个进程可认为只有自己独占计算机，从而简化程序的编写。每个进程有自己单独的地址空间，并且只能由这一进程访问，这样，操作系统避免了进程之间的互相干扰以及“坏”程序对系统可能造成的危害。 为了完成某特定任务，有时需要综合两个程序的功能，例如一个程序输出文本，而另一个程序对文本进行排序。为此，操作系统还提供进程间的通讯机制来帮助完成这样的任务。Linux 中常见的进程间通讯机制有信号、管道、共享内存、信号量和套接字等。

内核通过 SCI 提供了一个应用程序编程接口（API）来创建一个新进程（fork、exec 或 Portable Operating System Interface [POSIX] 函数），停止进程（kill、exit），并在它们之间进行通信和同步（signal 或者 POSIX 机制）。

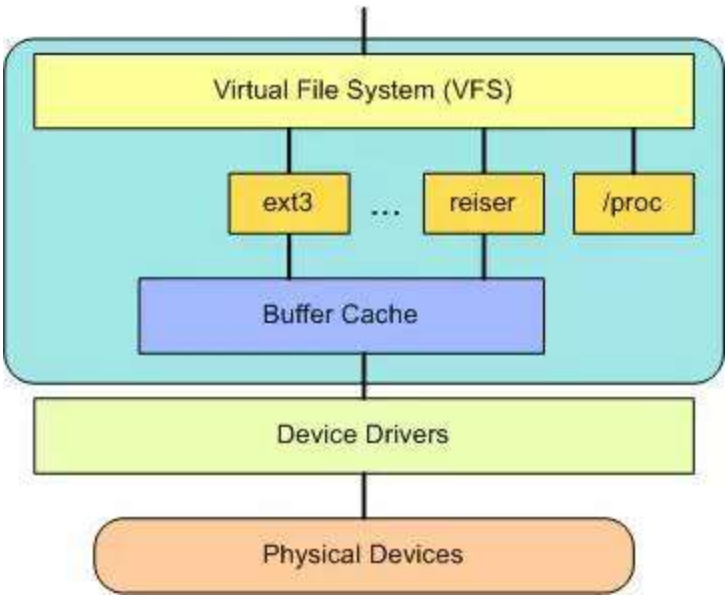
4. 文件系统

和 DOS 等操作系统不同，Linux 操作系统中单独的文件系统并不是由驱动器号或驱动器名称（如 A: 或 C: 等）来标识的。相反，和 UNIX 操作系统一样，Linux 操作系统将独立的文件系统组合成了一个层次化的树形结构，并且由一个单独的实体代表这一文件系统。Linux 将新的文件系统通过一个称为“挂装”或“挂上”的操作将其挂装到某个目录上，从而让不同的文件系统结合成为一个整体。Linux 操作系统的一个重要特点是它支持许多不同类型的文件系统。Linux 中最普遍使用的文件系统是 Ext2，它也是 Linux 土生土长的文件系统。但 Linux 也能够支持 FAT、VFAT、FAT32、MINIX 等不同类型的文件系统，从而可以方便地和其它操作系统交换数据。由于 Linux 支持许多不同的文件系统，并且将它们组织成了一个统一的虚拟文件系统。

虚拟文件系统（VirtualFileSystem,VFS）:隐藏了各种硬件的具体细节，把文件系统操作和不同文件系统的具体实现细节分离了开来，为所有的设备提供了统一的接口，VFS 提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext2,fat 等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

虚拟文件系统（VFS）是 Linux 内核中非常有用的一个方面，因为它为文件系统提供了一个通用的接口抽象。VFS 在 SCI 和内核所支持的文件系统之间提供了一个交换层。即 VFS 在用户和文件系统之间提供了一个交换层。

VFS 在用户和文件系统之间提供了一个交换层：



在 VFS 上面，是对诸如 open、close、read 和 write 之类的函数的一个通用 API 抽象。在 VFS 下面是文件系统抽象，它定义了上层函数的实现方式。它们是给定文件系统（超过 50 个）的插件。文件系统的源代码可以在 ./linux/fs 中找到。

文件系统层之下是缓冲区缓存，它为文件系统层提供了一个通用函数集（与具体文件系统无关）。这个缓存层通过将数据保留一段时间（或者随即预先读取数据以便在需要是就可用）优化了对物理设备的访问。缓冲区缓存之下是设备驱动程序，它实现了特定物理设备的接口。

因此，用户和进程不需要知道文件所在的文件系统类型，而只需要象使用 Ext2 文件系统中的文件一样使用它们。

5. 设备驱动程序

设备驱动程序是 Linux 内核的主要部分。和操作系统的其它部分类似，设备驱动程序运行在高特权级的处理器环境中，从而可以直接对硬件进行操作，但正因为如此，任何一

个设备驱动程序的错误都可能导致操作系统的崩溃。设备驱动程序实际控制操作系统和硬件设备之间的交互。

设备驱动程序提供一组操作系统可理解的抽象接口完成和操作系统之间的交互，而与硬件相关的具体操作细节由设备驱动程序完成。一般而言，设备驱动程序和设备的控制芯片有关，例如，如果计算机硬盘是 SCSI 硬盘，则需要使用 SCSI 驱动程序，而不是 IDE 驱动程序。

6. 网络接口（NET）

提供了对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序。网络协议部分负责实现每一种可能的网络传输协议。众所周知，TCP/IP 协议是 Internet 的标准协议，同时也是事实上的工业标准。

Linux 的网络实现支持 BSD 套接字，支持全部的 TCP/IP 协议。Linux 内核的网络部分由 BSD 套接字、网络协议层和网络设备驱动程序组成。网络设备驱动程序负责与硬件设备通讯，每一种可能的硬件设备都有相应的设备驱动程序。

更多网络通信内容可参考《网络通信原理》部分。

7. linux shell

shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行，是一个命令解释器。另外，shell 编程语言具有普通编程语言的很多特点，用这种编程语言编写的 shell 程序与其他应用程序具有同样的效果。

目前主要有下列版本的 shell。

1. Bourne Shell：是贝尔实验室开发的。

2. BASH：是 GNU 的 Bourne Again Shell，是 GNU 操作系统上默认的 shell,大部分 linux 的发行套件使用的都是这种 shell。

3. Korn Shell：是对 Bourne SHell 的发展，在大部分内容上与 Bourne Shell 兼容。

4. C Shell：是 SUN 公司 Shell 的 BSD 版本。

8. linux 文件系统详解

文件系统是文件存放在磁盘等存储设备上的组织方法。Linux 系统能支持多种目前流行的文件系统，如 EXT2、 EXT3、 FAT、 FAT32、 VFAT 和 ISO9660。

8.1 文件类型

Linux 下面的文件类型主要有：

1) 普通文件：C 语言元代码、SHELL 脚本、二进制的可执行文件等。分为纯文本和二进制。

2) 目录文件：目录，存储文件的唯一地方。

3) 链接文件：指向同一个文件或目录的的文件。

4) 设备文件：与系统外设相关的，通常在/dev 下面。分为块设备和字符设备。

5）管道(FIFO)文件 ： 提供进程建通信的一种方式

6）套接字(socket) 文件： 该文件类型与网络通信有关

可以通过 ls -l, file, stat 几个命令来查看文件的类型等相关信息。

8.2 Linux 目录

文件结构是文件存放在磁盘等存贮设备上的组织方法。主要体现在对文件和目录的组织上。

目录提供了管理文件的一个方便而有效的途径。

Linux 使用标准的目录结构，在安装的时候，安装程序就已经为用户创建了文件系统和完整而固定的目录组成形式，并指定了每个目录的作用和其中的文件类型。

完整的目录树可划分为小的部分，这些小部分又可以单独存放在自己的磁盘或分区上。这样，相对稳定的部分和经常变化的部分可单独存放在不同的分区中，从而方便备份或系统管理。目录树的主要部分有 root、/usr、/var、/home 等（图 2） 。这样的布局可方便在 Linux 计算机之间共享文件系统的某些部分。

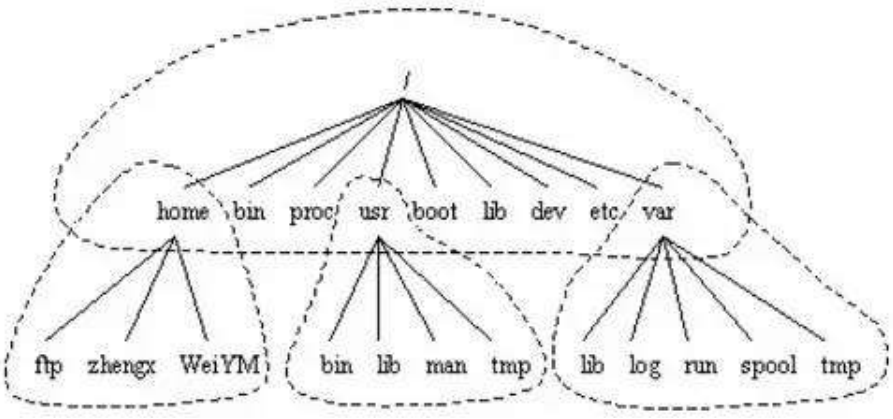


图 2

Linux 采用的是树型结构。最上层是根目录，其他的所有目录都是从根目录出发而生成的。

微软的 DOS 和 windows 也是采用树型结构，但是在 DOS 和 windows 中这样的树型结构的根是磁盘分区的盘符，有几个分区就有几个树型结构，他们之间的关系是并列的。最顶部的是不同的磁盘（分区），如：C，D，E，F 等。

但是在 linux 中，无论操作系统管理几个磁盘分区，这样的目录树只有一个。从结构上讲，各个磁盘分区上的树型目录不一定是并列的。

8.3 Linux 磁盘分区

一、主分区，扩展分区和逻辑分区：

linux 分区不同于 windows，硬盘和硬盘分区在 Linux 都表示为设备。

硬盘分区一共有三种：主分区，扩展分区和逻辑分区。

硬盘的分区主要分为主分区(Primary Partion)和扩展分区(Extension Partion)两种，主分区和扩展分区的数目之和不能大于四个。

n 主分区(Primary Partion)：可以马上被使用但不能再分区。

n 扩展分区(Extension Partion)：必须再进行分区后才能使用，也就是说它必须还要进行二次分区。

n 逻辑分区（(Logical Partion)）：由扩展分区建立起来的分区。逻辑分区没有数量上限制。

扩展分区只不过是逻辑分区的“容器”，实际上只有主分区和逻辑分区进行数据存储。

二、Linux 下硬盘分区的标识

硬盘分区的标识一般使用/dev/hd[a-z]X 或者/dev/sd[a-z]X 来标识，其中[a-z]代表硬盘号，X 代表硬盘内的分区号。

整块硬盘分区的块号标识:Linux 下用 hda、hdb、sda、sdb 等来标识不同的硬盘;

其中：

- IDE 接口硬盘：表示为/dev/hda1、/dev/hdb …；
- SCSI 接口的硬盘、SATA 接口的硬盘表示为/dev/sda、/dev/sdb … … ；
- 硬盘内的分区：如果 X 的值是 1 到 4,表示硬盘的主分区（包含扩展分区）；逻辑分区从是从 5 开始的，比如/dev/hda5 肯定是逻辑分区了；

例如：

用 hda1、hda2、 hda5、hda6 来标识不同的分区。其中，字母 a 代表第一块硬盘，b 代表第二块硬盘，依次类推。而数字 1 代表一块硬盘的第一个分区、2 代表第二个分区，依次类推。1 到 4 对应的是主分区(Primary Partition)或扩展分区(Extension Partition)。从 5 开始，对应的都是硬盘的逻辑分区(Logical Partition)。一块硬盘即使只有一个主分区，逻辑分区也是从 5 开始编号的，这点应特别注意。

总结：一个硬盘分区首先要大确认在哪个硬盘，然后再确认它所在硬盘内的哪个分区。

对于/dev/hda 类似的表示方法，也并不寞生吧；我们在 Linux 通过 fdisk -l 就可以查到硬盘是/dev/hda 还是/dev/hdb；

```
[root@localhost  ~]# fdisk -l

Disk  /dev/hda: 80.0GB,80026361856bytes

255heads,63sectors/track,9729cylinders

Units  = cylinders of16065  * 512  = 8225280bytes

Device Boot Start EndBlocks IdSystem

/dev/hda1  * 19707791493+ 7HPFS/NTFS

/dev/hda2971972970356667+ 5Extended

/dev/hda5971291515623181bW95 FAT32

/dev/hda6291641319767488+ 83linux

/dev/hda741325590117193868 3linux

/dev/hda8559168069767488+ 83linux

/dev/hda9680796572290062683linux

/dev/hda1096589729578308+ 82linux swap  / Solaris
```

请注意第一行， Disk /dev/hda: 80.0 GB, 80026361856 bytes ，这个就是表示机器中只有一个硬盘设备/dev/hda ，体积大小为 80.0G；下面的就是硬盘的分区，每个分区都有详细的信息，在这里不详细说了；

Linux 下磁盘分区和目录的关系如下：

- 任何一个分区都必须挂载到某个目录上。
- 目录是逻辑上的区分。分区是物理上的区分。
- 磁盘 Linux 分区都必须挂载到目录树中的某个具体的目录上才能进行读写操作。
- 根目录是所有 Linux 的文件和目录所在的地方，需要挂载上一个磁盘分区。

8.4 linux 主要目录的功用

/bin 二进制可执行命令

/dev 设备特殊文件

/etc 系统管理和配置文件

/etc/rc.d 启动的配置文件和脚本

/home 用户主目录的基点，比如用户 user 的主目录就是/home/user，可以用~user 表示

/lib 标准程序设计库，又叫动态链接共享库，作用类似 windows 里的.dll 文件

/sbin 系统管理命令，这里存放的是系统管理员使用的管理程序

/tmp 公用的临时文件存储点

/root 系统管理员的主目录（呵呵，特权阶级）

/mnt 系统提供这个目录是让用户临时挂载其他的文件系统。

/lost+found 这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows 下叫什么.chk）就在这里

/proc 虚拟的目录，是系统内存的映射。可直接访问这个目录来获取系统信息。

/var 某些大文件的溢出区，比方说各种服务的日志文件

/usr 最庞大的目录，要用到的应用程序和文件几乎都在这个目录。其中包含：

/usr/X11R6 存放 X window 的目录

/usr/bin 众多的应用程序

/usr/sbin 超级用户的一些管理程序

/usr/doc linux 文档

/usr/include linux 下开发和编译应用程序所需要的头文件

/usr/lib 常用的动态链接库和软件包的配置文件

/usr/man 帮助文档

/usr/src 源代码，linux 内核的源代码就放在/usr/src/linux 里

/usr/local/bin 本地增加的命令

/usr/local/lib 本地增加的库

8.5 linux 文件系统

文件系统指文件存在的物理空间，linux 系统中每个分区都是一个文件系统，都有自己的目录层次结构。linux 会将这些分属不同分区的、单独的文件系统按一定的方式形成一个系统的总的目录层次结构。一个操作系统的运行离不开对文件的操作，因此必然要拥有并维护自己的文件系统。

1.文件系统类型：

ext2 ： 早期 linux 中常用的文件系统

ext3 ： ext2 的升级版，带日志功能

RAMFS ： 内存文件系统，速度很快

NFS ： 网络文件系统，由 SUN 发明，主要用于远程文件共享

MS-DOS ： MS-DOS 文件系统

VFAT ： Windows 95/98 操作系统采用的文件系统

FAT ： Windows XP 操作系统采用的文件系统

NTFS： Windows NT/XP 操作系统采用的文件系统

HPFS ： OS/2 操作系统采用的文件系统

PROC : 虚拟的进程文件系统

ISO9660 ： 大部分光盘所采用的文件系统

ufsSun : OS 所采用的文件系统

NCPFS ： Novell 服务器所采用的文件系统

SMBFS ： Samba 的共享文件系统

XFS ： 由 SGI 开发的先进的日志文件系统，支持超大容量文件

JFS ：IBM 的 AIX 使用的日志文件系统

ReiserFS : 基于平衡树结构的文件系统

udf: 可擦写的数据光盘文件系统

2.文件系统特性

磁盘分区完毕后还需要进行格式化(format)，之后操作系统才能够使用这个分区。 格式化的目的是能使操作系统可以使用的文件系统格式（即我们上面提到文件系统类型）.

每种操作系统能够使用的文件系统并不相同. 如 windows 98 以前的微软操作系统主要利用的文件系统是 FAT (或 FAT16)，windows 2000 以后的版本有所谓的 NTFS 文件系统，

至于 Linux 的正统文件系统则为 Ext2 (Linux second extended file system, ext2fs)这一个。此外，在默认的情况下，windows 操作系统是不会认识 Linux 的 Ext2 的。

传统的磁盘与文件系统之应用中，一个分区就是只能够被格式化成为一个文件系统，所以我们可以说一个 filesystem 就是一个 partition。但是由于新技术的利用，例如我们常听到的 LVM 与软件磁盘阵列(software raid)， 这些技术可以将一个分区格式化为多个文件系统(例如 LVM)，也能够将多个分区合成一个文件系统(LVM, RAID)！ 所以说，目前我们在格式化时已经不再说成针对 partition 来格式化了， 通常我们可以称呼一个可被挂载的数据为一个文件系统而不是一个分区喔！

那么文件系统是如何运行的呢？这与操作系统的文件数据有关。较新的操作系统的文件数据除了文件实际内容外， 通常含有非常多的属性，例如 Linux 操作系统的文件权限(rwx)与文件属性(拥有者、群组、时间参数等)。 文件系统通常会将这两部份的数据分别存放在不同的区块，权限与属性放置到 inode 中，至于实际数据则放置到 data block 区块中。 另外，还有一个超级区块 (superblock) 会记录整个文件系统的整体信息，包括 inode 与 block 的总量、使用量、剩余量等。

对于一个磁盘分区来说，在被指定为相应的文件系统后，整个分区被分为 1024，2048 和 4096 字节大小的块。根据块使用的不同，可分为：

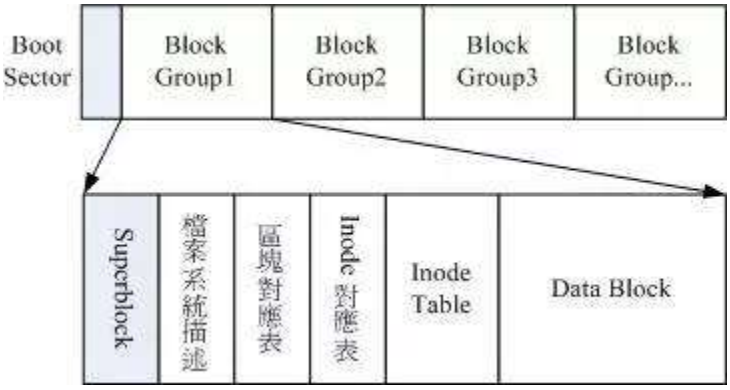
1. 超级块(Superblock): 这是整个文件系统的第一块空间。包括整个文件系统的基本信息，如块大小，inode/block 的总量、使用量、剩余量，指向空间 inode 和数据块的指针等相关信息。

2. inode 块(文件索引节点) : 文件系统索引,记录文件的属性。它是文件系统的最基本单元，是文件系统连接任何子目录、任何文件的桥梁。每个子目录和文件只有唯一的一个 inode 块。它包含了文件系统中文件的基本属性(文件的长度、创建及修改时间、权限、所属关系)、存放数据的位置等相关信息. 在 Linux 下可以通过 “ls -li” 命令查看文件的 inode 信息。硬连接和源文件具有相同的 inode 。

3. 数据块(Block) :实际记录文件的内容，若文件太大时，会占用多个 block。为了提高目录访问效率，Linux 还提供了表达路径与 inode 对应关系的 dentry 结构。它描述了路径信息并连接到节点 inode，它包括各种目录信息，还指向了 inode 和超级块。

就像一本书有封面、目录和正文一样。在文件系统中，超级块就相当于封面，从封面可以得知这本书的基本信息； inode 块相当于目录，从目录可以得知各章节内容的位置；而数据块则相当于书的正文，记录着具体内容。

Linux 正统的文件系统(如 ext2、3 等)将硬盘分区时会划分出超级块、inode Table 区块和 data block 数据区域。一个文件由一个超级块、inode 和数据区域块组成。Inode 包含文件的属性(如读写属性、owner 等，以及指向数据块的指针)，数据区域块则是文件内容。当查看某个文件时，会先从 inode table 中查出文件属性及数据存放点，再从数据块中读取数据。



ext2 文件系统示意图

我们将 inode 与 block 区块用图解来说明一下，如下图所示，文件系统先格式化出 inode 与 block 的区块，假设某一个文件的属性与权限数据是放置到 inode 4 号(下图较小方格内)，而这个 inode 记录了文件数据的实际放置点为 2, 7, 13, 15 这四个 block 号码，此时我们的操作系统就能够据此来排列磁盘的阅读顺序，可以一口气将四个 block 内容读出来！ 那么数据的读取就如同下图中的箭头所指定的模样了。

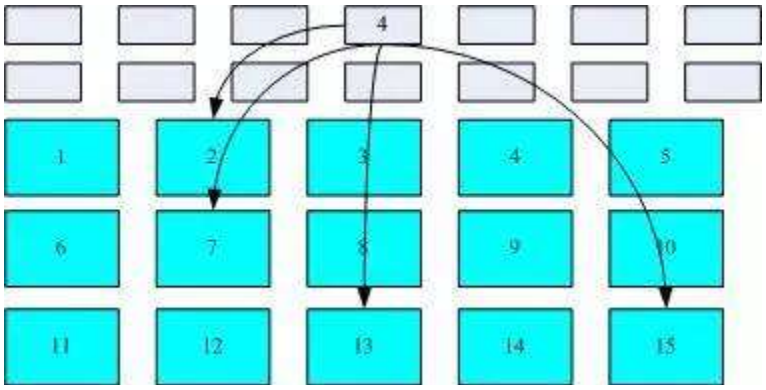
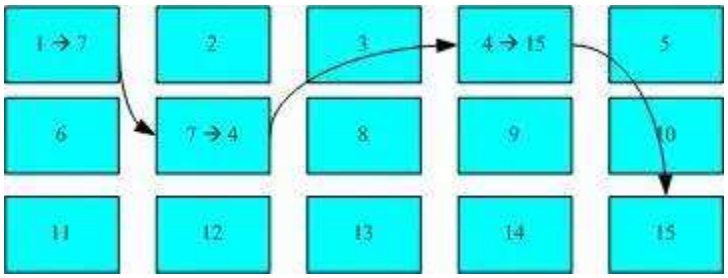


图 inode/block 数据存取示意图

这种数据存取的方法我们称为索引式文件系统(indexed allocation)。那有没有其他的惯用文件系统可以比较一下啊？ 有的，那就是我们惯用的闪盘(闪存)，闪盘使用的文件系统一般为 FAT 格式。FAT 这种格式的文件系统并没有 inode 存在，所以 FAT 没有办法将这个文件的所有 block 在一开始就读取出来。每个 block 号码都记录在前一个 block 当中， 其读取方式有点像下图所示:



图、FAT 文件系统数据存取示意图

上图中我们假设文件的数据依序写入 1->7->4->15 号这四个 block 号码中， 但这个文件系统没有办法一口气就知道四个 block 的号码，他得要一个一个的将 block 读出后，才会知道下一个 block 在何处。 如果同一个文件数据写入的 block 分散的太过厉害时，则我们的磁盘读取头将无法在磁盘转一圈就读到所有的数据， 因此磁盘就会多转好几圈才能完整的读取到这个文件的内容!

常常会听到所谓的“碎片整理”吧？ 需要碎片整理的原因就是文件写入的 block 太过于离散了，此时文件读取的效能将会变的很差所致。 这个时候可以透过碎片整理将同一个文件所属的 blocks 汇整在一起，这样数据的读取会比较容易啊！ 想当然尔，FAT 的文件系统需要经常的碎片整理一下，那么 Ext2 是否需要磁盘重整呢？

由于 Ext2 是索引式文件系统，基本上不太需要常常进行碎片整理的。但是如果文件系统使用太久， 常常删除/编辑/新增文件时，那么还是可能会造成文件数据太过于离散的问题，此时或许会需要进行重整一下的。 不过，老实说，鸟哥倒是没有在 Linux 操作系统上面进行过 Ext2/Ext3 文件系统的碎片整理说！似乎不太需要啦！^_^

可以用 ln 命令对一个已经存在的文件再建立一个新的连接，而不复制文件的内容。连接有软连接和硬连接之分，软连接又叫符号连接。它们各自的特点是：

硬连接：原文件名和连接文件名都指向相同的物理地址。目录不能有硬连接；硬连接不能跨越文件系统（不能跨越不同的分区）文件在磁盘中只有一个拷贝，节省硬盘空间；

由于删除文件要在同一个索引节点属于唯一的连接时才能成功，因此可以防止不必要的误删除。

符号连接：用 ln -s 命令建立文件的符号连接符号连接是 linux 特殊文件的一种，作为一个文件，它的数据是它所连接的文件的路径名。类似 windows 下的快捷方式。

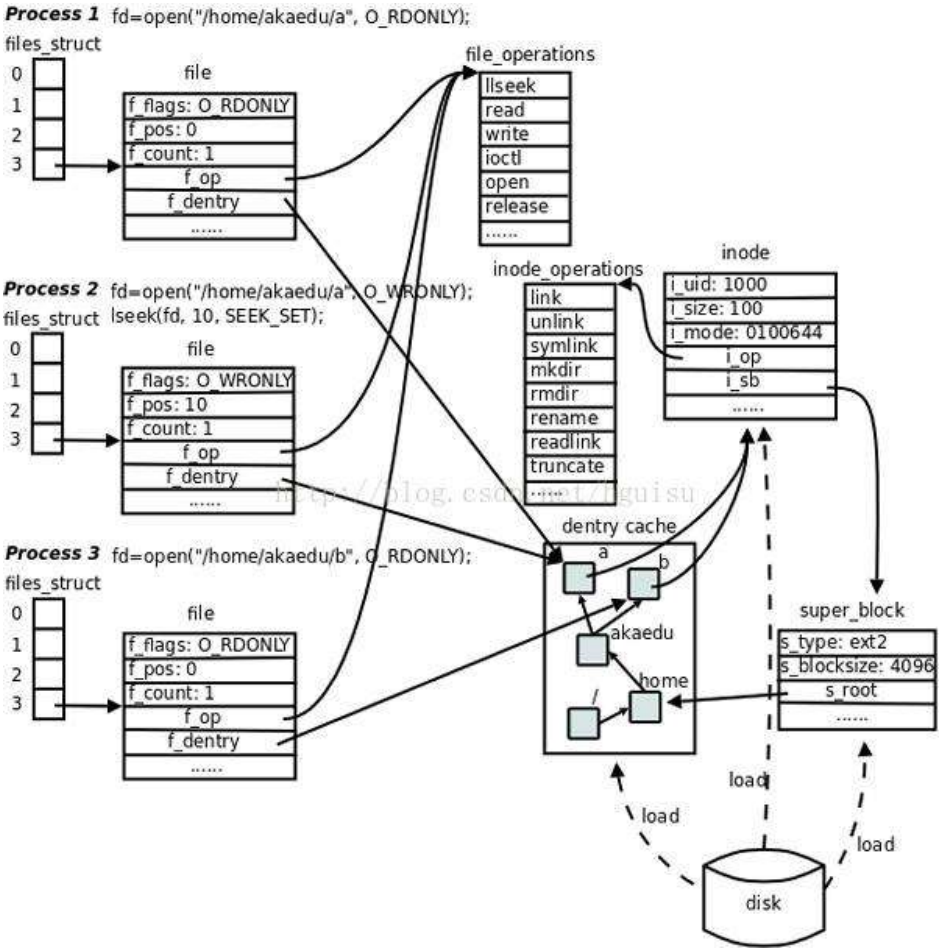
可以删除原有的文件而保存连接文件，没有防止误删除功能。

这一段的的内容过于抽象，又是节点又是数组的，我已经尽量通俗再通俗了，又不好加例子作演示。大家如果还是云里雾里的话，我也没有什么办法了，只有先记住，日后在实际应用中慢慢体会、理解了。这也是我学习的一个方法吧。

8.6 文件系统在内核中的表示

内核数据结构

Linux 内核的 VFS 子系统可以图示如下：



文件与 IO：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 Status Flag 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），后面我们会讲到，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。

file、dentry、inode、super_block 这几个结构体组成了 VFS 的核心概念。对于 ext2 文件系统来说，在磁盘存储布局上也有 inode 和超级块的概念，所以很容易和 VFS 中的概念建立对应关系。而另外一些文件系统格式来自非 UNIX 系统（例如 Windows 的 FAT32、NTFS），可能没有 inode 或超级块这样的概念，但为了能 mount 到 Linux 系统，也只好在驱动程序中硬凑一下，在 Linux 下看 FAT32 和 NTFS 分区会发现权限位是错的，所有文件都是 rwxrwxrwx，因为它们本来就没有 inode 和权限位的概念，这是硬凑出来的。

8.7 挂载文件系统

linux 系统中每个分区都是一个文件系统，都有自己的目录层次结构。linux 会将这些分属不同分区的、单独的文件系统按一定的方式形成一个系统的总的目录层次结构。这里所说的“按一定方式”就是指的挂载。

将一个文件系统的顶层目录挂到另一个文件系统的子目录上，使它们成为一个整体，称为挂载。把该子目录称为挂载点.

例如要读取硬盘中的一个格式化好的分区、光盘或软件等设备时，必须先把这些设备对应到某个目录上，而这个目录就称为“挂载点（mount point）”，这样才可以读取这些设备。 挂载后将物理分区细节屏蔽掉，用户只有统一的逻辑概念。所有的东西都是文件。

注意：1、挂载点必须是一个目录。

2、一个分区挂载在一个已存在的目录上，这个目录可以不为空，但挂载后这个目录下以前的内容将不可用。

对于其他操作系统建立的文件系统的挂载也是这样。但是需要理解的是：光盘、软盘、其他操作系统使用的文件系统的格式与 linux 使用的文件系统格式是不一样的。光盘是 ISO9660；软盘是 fat16 或 ext2；windows NT 是 fat16、NTFS；windows98 是 fat16、fat32；windows2000 和 windowsXP 是 fat16、fat32、 NTFS。挂载前要了解 linux 是否支持所要挂载的文件系统格式。

挂载时使用 mount 命令，其格式：mount [-参数] [设备名称] [挂载点]

其中常用的参数有

-t 指定设备的文件系统类型（什么提到的文件类型）

-o 指定挂载文件系统时的选项。有些也可用在/etc/fstab 中。常用的有

codepage=XXX 代码页

iocharset=XXX 字符集

ro 以只读方式挂载

rw 以读写方式挂载

nouser 使一般用户无法挂载

user 可以让一般用户挂载设备

例如：

1.挂载 windows 的文件系统：

1）首先我们使用 sudo fdisk -l 查看挂载的设备，例如最下面有：/dev/hda5

2）mkdir 创建一个目录，这里的目录是作为挂在目录，就是你要把 E 盘挂到这个目录下：mk /mnt/winc

3）windows 和 linux 使用的不是一个文件系统，一般情况下 linux 不挂载 windows 文件系统，所以要你手动 mount：

mount -t vfat /dev/hda5 /mnt/winc （ -t vfat 指出这里的文件系统 fat32）

现在就可以进入/mnt/winc 等目录读写这些文件了。

2.挂载光盘：# mk /mnt/cdrom

mount -t iso9660 /dev/cdrom /mnt/cdrom (关盘的名字一般都是 cdrom，这条命令一般都通用)

3.虚拟机共享文件夹：例如在 VirtualBox 下，主机是 Windows，Ubuntu 是 Guest。共分三步：

1). 首先要安装虚拟电脑工具包：在 VirtualBox 的菜单里选择”设备”->”安装虚拟电脑工具包”，你会发现在 Ubuntu 桌面上多出一个光盘图标，这张光盘默认被自动加载到了文件夹/media/cdom0，而且/cdrom 自动指向这个文件夹。默认设置下文件管理器会自动打开这张光盘，可以看到里面有个”VBoxLinuxAdditions.run”文件。打开一个命令行终端，依次输入”cd /cdrom”和”sudo sh ./VBoxLinuxAdditions.run”，不含双引号，开始安装工具包。安装完毕，会用英文提示要重启 Ubuntu，建议立刻重启。重启后，比较明显的变化是鼠标是共享模式，并且剪贴板也和 Windows 共享了。如果有这些变化，说明虚拟电脑工具包已经装成功。

2). 下一步设置共享文件夹。

在共享文件夹设置窗口中，单击右侧的”添加一个共享文件夹”，路径选择你想要共享的 Windows 文件夹，共享名任取一个自己喜欢的，比如”myshare”，选项 read-only 是指是否只允许 ubuntu 读这个文件夹，请根据需要选择这个选项。

3). 在 ubuntu 下挂载这个共享文件夹:sudo mount -t vboxsf myshare /media/share

其中”myshare”是之前取的共享文件夹的名字，”/media/share”是要挂载到的目标文件.

8.8 自动挂载 windows 分区

每次开机访问 windows 分区都要运行 mount 命令显然太烦琐，为什么访问其他的 linux 分区不用使用 mount 命令呢？

其实，每次开机时，linux 自动将需要挂载的 linux 分区挂载上了。那么我们是不是可以设定让 linux 在启动的时候也挂载我们希望挂载的分区，如 windows 分区，以实现文件系统的自动挂载呢？

这是完全可以的。在/etc 目录下有个 fstab 文件，它里面列出了 linux 开机时自动挂载的文件系统的列表。我的/etc/fstab 文件如下：

/dev/hda2 / ext3 defaults 1 1

/dev/hda1 /boot ext3 defaults 1 2

none /dev/pts devpts gid=5,mode=620 0 0

none /proc proc defaults 0 0

none /dev/shm tmpfs defaults 0 0

/dev/hda3 swap swap defaults 0 0

/dev/cdrom /mnt/cdrom iso9660 noauto,codepage=936,iocharset=gb2312 0 0

/dev/fd0 /mnt/floppy auto noauto,owner,kudzu 0 0

/dev/hdb1 /mnt/winc vfat defaults,codepage=936,iocharset=cp936 0 0

/dev/hda5 /mnt/wind vfat defaults,codepage=936,iocharset=cp936 0 0

在/etc/fstab 文件里，第一列是挂载的文件系统的设备名，第二列是挂载点，第三列是挂载的文件系统类型，第四列是挂载的选项，选项间用逗号分隔。第五六列不知道是什么意思，还望高手指点。

在最后两行是我手工添加的 windows 下的 C；D 盘，加了 codepage=936 和 iocharset=cp936 参数以支持中文文件名。参数 defaults 实际上包含了一组默认参数：

rw 以可读写模式挂载

suid 开启用户 ID 和群组 ID 设置位

dev 可解读文件系统上的字符或区块设备

exec 可执行二进制文件

auto 自动挂载

nouser 使一般用户无法挂载

async 以非同步方式执行文件系统的输入输出操作

大家可以看到在这个列表里，光驱和软驱是不自动挂载的，参数设置为 noauto。（如果你非要设成自动挂载，你要确保每次开机时你的光驱和软驱里都要有盘，呵呵。）

8.9 软连接、硬链接

可以用 ln 命令对一个已经存在的文件再建立一个新的连接，而不复制文件的内容。连接有软连接和硬连接之分，软连接又叫符号连接。它们各自的特点是：

硬连接：是给文件一个副本，原文件名和连接文件名都指向相同的物理地址。目录不能有硬连接；硬连接不能跨越文件系统（不能跨越不同的分区）文件在磁盘中只有一个拷贝，节省硬盘空间；

修改其中一个，与其连接的文件同时被修改。如果删除其中任意一个其余的文件将不受影响。

由于删除文件要在同一个索引节点属于唯一的连接时才能成功，因此可以防止不必要的误删除。

符号连接（软连接）：用 ln -s 命令建立文件的符号连接符号连接是 linux 特殊文件的一种，作为一个文件，它的数据是它所连接的文件的路径名。类似 windows 下的快捷方式。

当然删除这个连接，也不会影响到源文件，但对连接文件的使用、引用都是直接调用源文件的。

具体关系可以看下图：

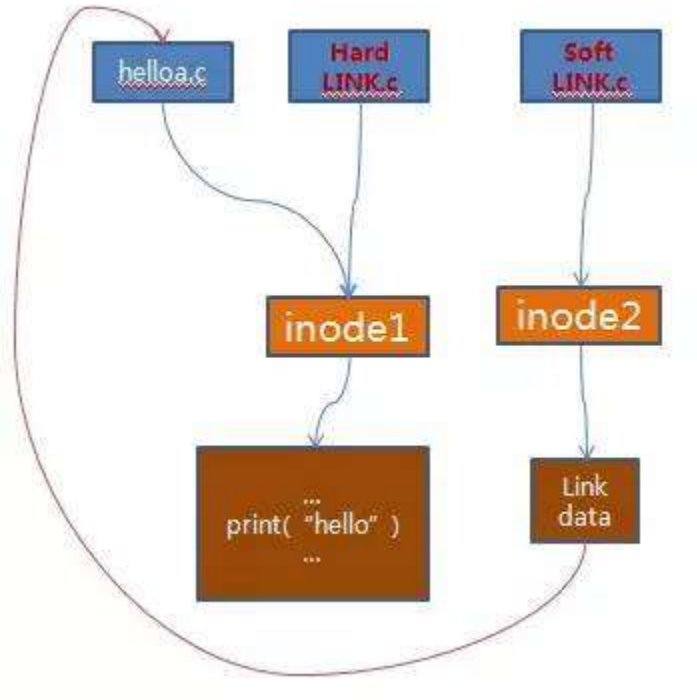


图 5：软链接和硬链接

从图上可以看出硬链接和软链接的区别：

1：硬链接原文件和新文件的 inode 编号一致。而软链接不一样。

2：对原文件删除，会导致软链接不可用，而硬链接不受影响。

3：对原文件的修改，软、硬链接文件内容也一样的修改，因为都是指向同一个文件内容的。

8.9 文件目录管理命令

磁盘和文件空间 ：fdisk df du

文件目录与管理： cd pwd mkdir rmdir ls cp rm mv

查看文件内容 cat、tac、more、less、head 、tail

文件目录与权限 ：chmod chown chgrp umask

文件查找：which、whereis、locate、find、find

9. Linux 应用

标准的 Linux 系统一般都有一套都有称为应用程序的程序集，它包括文本编辑器、编程语言、X Window、办公套件、Internet 工具和数据库等。

10. Linux 内核参数优化

内核参数是用户和系统内核之间交互的一个接口，通过这个接口，用户可以在系统运行的同时动态更新内核配置，而这些内核参数是通过 Linux Proc 文件系统存在的。因此，可以通过调整 Proc 文件系统达到优化 Linux 性能的目的。

11. Linux 启动流程

学习这部分之前，可先阅读 这里 通过了解 CPU 的工作流程洞悉 Linux 操作系统的启动流程。

但其实撇开现流行的微服务处理架构而言，服务器或个人主机的启动流程基本类似。

系统的启动需要借助 Linux 内核的一些模块。上面我们已经了解了一些内核的知识，这里再对内核相关内容做一个总结。

以 Linux 系统为例，其启动流程如下：

系统启动首先需要进行硬件可用性检查（避免因短路等硬件故障造成系统不可用），检查完毕后，系统需要开始工作，系统工作需要硬件，而硬件的识别需要驱动程序，获取驱动程序则需要先加载核心操作系统的核心信息，即基础内核信息。核心信息加载完成后，系统会尝试驱动所有硬件设备。

硬件启动完毕，系统还需要启动各个应用程序以进行工作。应用程序的启动具有优先级，内核会先启动第一个程序 init，并初始化其他应用程序所需的基础运行环境。之后 init 会根据系统定义好的程序启动顺序逐渐启动各个程序。

11.1 Linux 开机引导原理

以下步骤将概要介绍 Linux 引导进程的工作原理：

1、 计算机在启动时首先执行的是加载 BIOS 硬件信息，初始化测试 POST（开机自检，Power On Self Test），它将测试多个设备，包括处理器、内存、显卡和键盘。此外，还将测试引导介质（硬盘、软盘和 CD-ROM）。在 POST 之后，ROM 中的加载程序将载入引导扇区，该扇区随后将从活动分区中载入操作系统。您可以通过编辑服务器 BIOS 来更改引导介质顺序。

2、 引导扇区总是在同一个位置 — 引导设备的磁道 0、柱面 0、磁头 0。此扇区包含名为 loader 的程序（对于 Linux，它通常是 LILO 或 GRUB）；实际上是此程序引导操作系统。加载程序不是安装在 MBR 中，就是安装在活动主分区的第一个扇区中。

3、 如果服务器中安装了多个操作系统，您需要从引导加载程序菜单中选择需要引导的操作系统。如果安装了多个内核，您还可以在此菜单中选择要载入的内核。

4、 然后，引导加载程序将解压缩并载入内核。内核将首先载入内核模块，然后检测硬件（软驱、硬盘、网络适配器等），检验硬件配置，然后扫描和载入设备驱动程序。

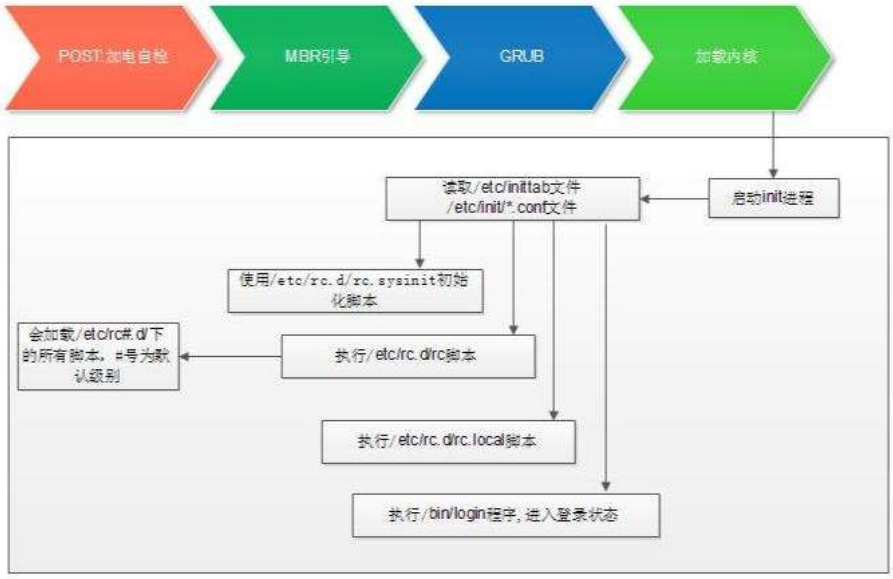
5、 在此阶段，内核将装入根文件系统和系统文件。在重新编译期间，可以（或通过其他程序）配置系统文件的位置。如果载入失败，kernel panic 将出现，并且系统将冻结（freeze）。这是先前提到的载入失败类型。

6、 接下来，内核将启动系统初始化进程 init，这将成为第一个进程。随后它将启动系统的其余部分。init 进程是 Linux 的第一个进程，它是所有其他进程的父进程。此进程是在任何 Linux/UNIX® 系统中第一个运行的进程；它的 PID 始终为 1。

7、 然后，init 将检验 /etc/inittab 文件以确定必须启动哪些进程。此文件将提供关于 runlevel 及在各个 runlevel 上应当启动的进程的 init 信息。然后，init 将查找第一个带有 sysinit（系统初始化）操作的行，然后将执行指定的命令文件，如 Red Hat Linux 中的 /etc/rc.d/rc.sysinit。在执行 /etc/rc.d/rc.sysinit 中的脚本之后，init 将开始启动与初始 runlevel 关联的进程。在执行完 runlevel 初始脚本时，Linux 将允许您登录。

11.2 CentOS6 启动流程

以 CentOS6 为例，其启动流程如下：



CentOS6 启动流程及其需要加载的文件如下：

1. **POST 加电自检**：加载 BIOS 的硬件信息，获取第一个启动设备（按次序查找引导设备，第一个有引导程序的设备为本次启动设备）

2. **Bootloader 引导**：读取第一个启动设备 MBR 的引导加载程序(grub/LILO)的启动信息

3.加载核心操作系统的核心信息，核心开始解压缩，并尝试驱动所有的硬件设备

n 加载核心操作系统的核心信息，核心开始解压缩

n 探测可识别到的所有硬件设备

n 加载硬件驱动程序（借助于 ramdisk 加载驱动）

4. 运行用户空间的第一个应用程序：/sbin/init，获取默认的运行信息

5. 读取 /etc/inittab 文件，/etc/init/*.conf 文件

6. init 程序执行/etc/rc.d/rc.sysinit 文件

7. 执行 /etc/rc.d/rc 脚本，启动核心的外挂模块

8. 加载 /etc/rc#.d/ 下的所有脚本，#号为默认级别：init 执行运行的各个批处理文件(scripts)

9. 执行 /etc/rc.d/rc.local 脚本

10. 执行 /bin/login，进入登录状态

11.登录之后开始以 Shell 控制主机

系统初始化步骤：

POST --> BootSequence (BIOS) -->Bootloader(MBR) --> kernel(ramdisk) --> rootfs(只读) --> init（systemd）

POST

Power-On-Self-Test，加电自检，是 BIOS 功能的一个主要部分。负责完成对 CPU、主板、内存、硬盘子系统、显示子系统、串并行接口、键盘、CD-ROM 光驱等硬件情况的检测。

ROM：BIOS，Basic Input and Output System，保存着有关计算机系统最重要的基本输入输出程序，系统信息设置、开机加电自检程序和系统启动自举程序等。

RAM：CMOS 互补金属氧化物半导体，保存各项参数的设定按次序查找引导设备，第一个有引导程序的设备为本次启动设备

Bootloader

bootloader: 引导加载器，引导程序

Windows: ntloader，仅是启动 OS

Linux：功能丰富，提供菜单，允许用户选择要启动系统或不同的内核版本；把用户选定的内核装载到内存中的特定空间中，解压、展开，并把系统控制权移交给内核

LILO：LInux LOader

GRUB: GRand Unified Bootloader

GRUB 0.X: GRUB Legacy, GRUB2

MBR:

446: bootloader, 64: 分区表, 2: 55AA

GRUB:

primary boot loader : 1st stage, 1.5 stage

secondary boot loader ：2nd stage，分区文件

kernel 自身初始化

kernel 自身初始化

探测可识别到的所有硬件设备

加载硬件驱动程序（借助于 ramdisk 加载驱动）

以只读方式挂载根文件系统

运行用户空间的第一个应用程序：/sbin/init

ramdisk：内核中的特性之一：使用缓冲和缓存来加速对磁盘上的文件访问，并加载相应的硬件驱动

ramdisk --> ramfs 提高速度

CentOS 5: initrd

工具程序：mkinitrd

CentOS 6, 7: initramfs

工具程序：mkinitrd, dracut

为当前正在使用的内核重新制作 ramdisk 文件：

mkinitrd /boot/initramfs-$(uname -r).img $(uname -r)

dracut /boot/initramfs-$(uname -r).img $(uname -r)

init 启动进程

init 初始化流程

n 读取其初始化文件/etc/inittab, /etc/init/*.conf 文件：初始运行级别(RUN LEVEL)

n 使用 /etc/rc.d/rc.sysinit 系统初始化脚本

n 执行 /etc/rc.d/rc 脚本

n 加载 /etc/rc#.d/ 下的所有脚本，# 号为默认级别：对应运行级别的脚本目录

n 捕获某个关键字顺序

n 定义 UPS 电源终端/恢复脚本

n 在虚拟控制台生成 getty

n 在运行级别 5 初始化 X

/etc/rc.d/rc.sysinit: 系统初始化脚本

(1) 设置主机名

(2) 设置欢迎信息

(3) 激活 udev 和 selinux

(4) 挂载/etc/fstab 文件中定义的文件系统

(5) 检测根文件系统，并以读写方式重新挂载根文件系统

(6) 设置系统时钟

(7) 激活 swap 设备

(8) 根据/etc/sysctl.conf 文件设置内核参数

(9) 激活 lvm 及 software raid 设备

(10) 加载额外设备的驱动程序

(11) 清理操作

/etc/rc.d/rcN.d/

该目录指的是系统进对应 N 的运行级别时候系统必须做的工作，目录下 S 打头的服务指进此运行级别时候启动的服务，而 K 打头的指离开此运行级别时关闭的服务。

总结：/sbin/init --> (/etc/inittab) --> 设置默认运行级别 --> 运行系统初始脚本、完成系统初始化 --> (关闭对应下需要关闭的服务)启动需要启动服务 --> 设置登录终端

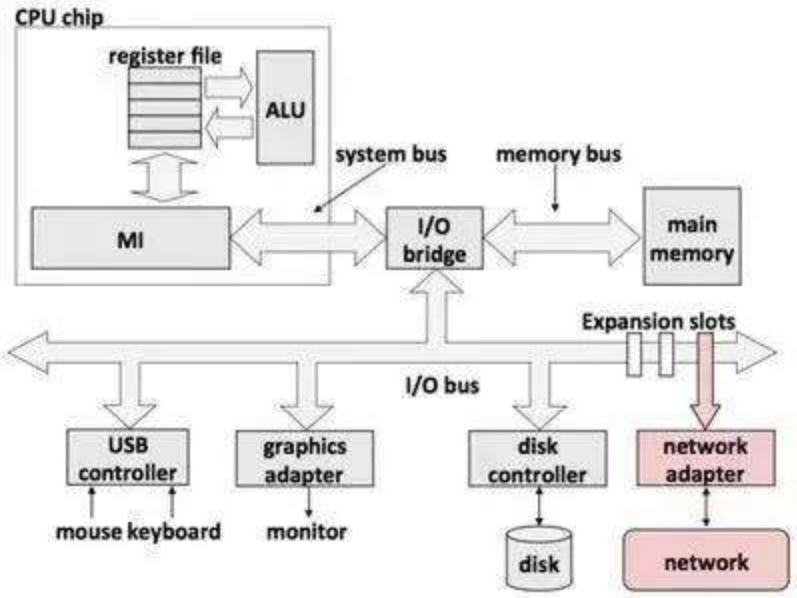
CPU

可以通过下面的文章对 CPU 的功能进行初步了解。

CPU 阿甘

原文链接：https://mp.weixin.qq.com/s/qWjoHuPazxKZHbs7j2OGYw

原创： 老刘 码农翻身



前言

上帝为你关闭了一扇门，就一定会为你打开一扇窗

这句话来形容我最合适不过了。

我是 CPU, 他们都叫我阿甘， 因为我和《阿甘正传》里的阿甘一样， 有点傻里傻气的。

上帝把我制造出来， 给我了一个很小的脑容量， 为数不多的寄存器能临时的记一点东西， 但是上帝给我打开了一扇特别的窗户， 那就是像阿甘一样，跑的飞快。

到底有多快呢？ 我这么比喻一下吧， 我的工作都是以纳秒为单位的， 你们人间的一秒， 我可能已经做了 1000,000,000 （10 亿）次动作了。

相比而言， 内存比我慢 100 倍， 硬盘比我慢 1000 多万倍， 你说我快不快？

启动

我住在一个机箱里，每天早上一阵电流把我叫醒， 还夹杂着嗡嗡的声音， 我知道我忠实的护卫电风扇又开始工作了， 我特别怕热， 又运行的飞快， 如果没有电风扇给我降温， 我很快就会生病， 生病的后果很严重， 那就是我的伙伴们像内存了，硬盘了。。全部都要罢工了， 没有我这个系统就会陷入的一片死寂。

我听说有些 CPU 的福利很好，竟然待在恒温恒湿，一尘不染的托管机房里，让我好生羡慕。

我的脑容量很小， 所以醒来后只想起了我的创造者告诉我的几件事情 ：

1. **你的工作就是运行指令**

2. **你不能保存指令， 你的指令全在内存里**

3. **你的第一条指令在内存的最顶端处 0xFFFFFFF0**

那还有什么可说的， 赶紧打电话给内存要指令，电话通过系统总线， 还得通过 I/O 桥电话局需要转接一下， 再通过存储总线接通 内存。

"哥们， 把这个地址处的指令给我说一下吧"

"你是谁？" 内存竟然把我忘了， 当然，他断了电和我一样，失忆了。

″我是阿甘啊， 我们经常聊天来着， 你忘了？″

内存磨磨唧唧半天才把数据发了过来（比我慢 100 倍啊）， 这是一条跳转指令， 我立刻回忆起来了， 这是我的老朋友 BIOS 等着我去运行他那一堆指令呢。

我给 BIOS 打电话: “老弟，今天干点啥？”

“阿甘，早上好 ″ BIOS 从不失忆，把所有人都记得清清楚楚 “ 还不是老一套啊，无非做一下系统的自检， 看看内存，硬盘，显卡等这些老伙计们有没有问题， 有问题的话用小喇叭提示一下主人 ”

这些过程我已经轻车熟路了， 很快搞定， 像往常一样，没有问题， 我还把一个叫做中断向量表的东西给弄好了， 我知道一会而要用

这些东西都搞完了，BIOS 果然告诉: ″阿甘， int 0x19″

我赶紧去刚弄好的中断向量表中去查第 19 号， 顺藤摸瓜又找到对应 0x19 的一大堆指令。

执行吧， 这堆指令把将磁盘的第一扇区（磁盘最开始的 512 字节）运到内存的 0X0000:0X7C00 处，然后我就从此处接着执行。

我想起来了， 接下来有一大堆精巧的指令把迷迷糊糊的操作系统从硬盘中唤醒， 运输到内存中来。

(此处实在是复杂， 略去 10 万字。。。。)

你看这就是为啥他们叫我阿甘， 我做事飞快，但非得别人告诉去哪里执行才行， 要不然我就只会坐在那里无所适从。

运行

操作系统一旦进入内存，立刻就是老大， 所有人都得听他指挥。

我也发现我的周围出现了一个屋子：进程屋

屋里堆着一大堆东西， 什么进程描述信息包裹了， 进程控制信息包裹了， 我都不太关心， 我只关心最最重要的两件东西：

1. **我工作必备的寄存器， 就放在我面前的工作台上。**
2. **程序计数器， 我用它记住我要执行的下一条指令地址。**

"阿甘， 别来无恙啊"，操作系统对我还是挺不错的， 先给我打招呼。

"Linux 老大， 今天有什么活啊"， 我每次都表现的积极主动。

"来，把这个 hello world 程序给运行了"

Hello world 程序还在硬盘上睡着呢， 得先把他也装载到内存里， 要不然我怎么执行啊。

于是我就拿起电话打给硬盘， 电话通过系统总线来到 IO 桥电话局， 再转接到 IO 总线，这才来到硬盘这里。

我在电话里请他把数据给我运过来， 然后我就无所事事的坐在那里等。

Linux 老大立刻就怒了 ： 阿甘， 告诉你多少次了， 你小子怎么还在等硬盘给你发数据！

是的， 我忘了一件事，硬盘比我慢太多了， 我执行一条指令大概是 1ns ，在用来读磁盘的 16ms 里， 我能潜在的执行 1600 多万条指令啊。

我感到深深的愧疚， 赶紧拿起电话打给硬盘 ： 哥们， 按我们之前商量好的，用直接内存访问（DMA）啊， 你直接把数据装载到内存吧， 不用经过我了， 装载完成以后给我发个信号。

"这还差不多" Linux 老大心情好了些

“阿甘，数据还没来， 别闲着， 这有一个菲波那切数列数列， 来算一下吧”

"肥波纳妾数列? 这名字好古怪，老大， 其实你也知道， 我脑子小，懒得去理解那是啥意思， 你把进程屋切换下，把程序计数器设置好，指向下一条指令， 我一条条指令执行就得了“ 我挺没追求的。

"真是个阿甘啊！ ”老大感慨到。

我所处的进程屋立刻发生了变化（当然，这也是我辅助 Linux 老大干的）， 各种包裹的信息都变了， 尤其是寄存器和程序计数器。

于是我就开始计算这个什么纳妾数列 ，但是这个数列似乎无穷无尽， 哪个无脑子的程序员写了个无限循环吧。

正在这时， 我便收到了一个电话， 说是 Helloworld 的数据已经装载到内存了， 让我去处理。

我放下手中的活， 保存好现场， 就去处理那个 Helloworld， 果然数据已经都好了，那就切换过去运行吧。

其实老大并不知道， 任何人，只要你运行了相当多的数量的指令以后， 你都能悟到这些程序的秘密。

我 CPU 阿甘虽然傻傻的， 但也架不住执行这数以万万亿的指令给我的熏陶啊。

这个秘密就是：程序都是由顺序，分支，循环来组成的。 其实分支和循环在我看来都是跳转而已。

所以我的工作就是打电话问内存要一条指令， 执行这个指令， 如果是个跳转指令的话，我就问内存要跳转的目标地址的那一条指令， 继续执行， 生活就是这么简单。

奥对了， 当然也有复杂的， 就是函数调用， 我得和内存紧密配合才能完成。 这个咱下回再说。

新装备：缓存

提到内存， 这真是我的好哥们， 没有他，我几乎什么事儿都干不成， 更重要的是他比硬盘快的多， 读取一次数据， 只需要 100 纳秒左右。 这样我们俩说起话来就轻松多了。

每次他都说：″阿甘， 幸亏有你给我聊天， 要不然我肯定被活活的闷死不可， 那个硬盘说话是在太慢了″

″它为啥那么慢？″ 我每次都问

″硬盘是个机械是的玩意， 一个磁头在一碟高速旋转的磁片上挪来挪去，光定位就慢死了″

″那主人为什么要用硬盘？″

″人家虽然慢， 但是不怕停电， 哪像你和我，一停电全部都失去记忆了。″

确实是， 人不能把好事都占全了啊。

我的指令中有些完全用我的寄存器就能完成， 但是有很多都需要读写内存的数据， 再加上所有的指令都在内存中存着， 虽然它只比我慢个 100 倍， 但指令多了我还是有点受不了。

我给内存说：″哥们， 你能不能再快点！″

内存说： 拜托， 这已经是我的极限了， 阿甘， 你自己再想想办法吧 ！ 我给你说啊，我留意了你最近访问的指令和数据， 我发现了个规律“

″啥规律？″

″比如说吧， 你访问了我一个内存位置以后过不多久还会多次访问， 还有，一个内存位置被访问了， 附近的位置很快也会访问到″

（码农翻身注： 这其实叫程序的局部性原理）

我还以为是啥规律， 其实我早就注意到了。

″这有啥用啊？”

″既然你经常访问同一块区域的东西， 你想想如果把这些东西缓存在你那里会怎么样....″

我一想有道理啊！ 加个缓存试试！

从此以后，我每次读写指令和数据， 都问缓存要， 缓存没有才给内存打电话。

果然， 由于局部性原理的存在， 我发现的确是快了不少啊。

当然也有缺点， 那就是 Linux 老大在做程序切换的时候， 缓存就会失效，因为两个程序之间没什么联系，局部性原理不起作用， 所以需要重建缓存。

自我提升：流水线

缓存让我的工作更有效率， 得到了 Linux 老大的表扬：″阿甘， 我看你很聪明嘛， 都会用缓存了″

″我哪有那么聪明，都是内存的点子。老大，不过我学会了一个重要的东西 ：当你改变不了别人的话，抱怨也没用， 还是先改变一下自己吧″

″挺有哲理的吗， 希望你明天重启后还能想起来″ Linux 老大笑话我。

″我最近又发现了一个问题， 正苦恼着呢， 你看我有四只手， 第一只手负责打电话问内存要指令， 第二只手翻译指令， 第三只手真正执行， 第四只手有时候还得把结果写回内存。 问题是， 我发现经常只有一只手在忙活， 其他都在闲着， 你看第一只手取指令， 其他手只能等着。 第二只手翻译指令的时候，其他三只也得等“

″看来以后我们不能叫你阿甘了， 你已经开始思考了″ Linux 老大笑了

“这问题好解决， 给你举个例子，你听说过洗车没有？ 和你差不多， 也是先喷水，再打洗洁剂， 再擦洗， 最后烘干， 但人家的工作方式和你不一样，人家是流水线作业， 你想想， 一辆车在烘干的时候， 后边是不是还有三辆车，分别在喷水， 打清洁剂和擦洗 ， 每个步骤都不会空闲。 ”

″这么简单的道理我怎么都没有想到呢？ 我也可以搞个流水线啊， 这样每只手都利用起来了″

别人都说我们高科技， 但其实原理都蕴含在生活之中啊。

有了缓存和流水线的帮助， 让我的工作大大的加快了，大家都对我刮目相看。 他们想给我起个新名字：超人 ， 不过我还是更喜欢 他们叫我“阿甘”， 多亲切。

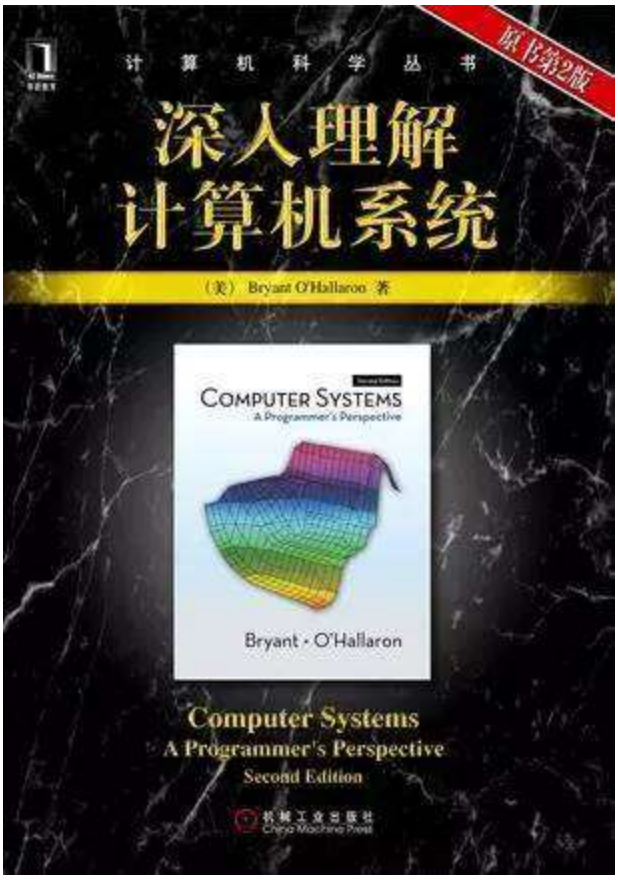
尾声

我一丝不苟，兢兢业业的运行指令， 时不时和伙伴们聊天， 很快一天就过去了， 又到了晚上，我知道关机的时刻到了， 赶紧挨个给他们道别。

很快那些让我兴奋的电流消失了，风扇的嗡嗡声也没有了， 我再也无法打出电话，整个世界沉寂了。

明天将会是新的一天。

备注： 本文内容主要来自于《深入理解计算机系统》， 这本书非常棒，值得每个程序员仔细研读。



现在 CPU 的程度指令运行机制并没有上述这么简单，操作系统也是在初代的基础上进行了进化又进化，直至达到一个相对理想的状态。我们可以通过下面的文章了解 CPU、操作系统及内存在处理进程上的演讲史。这也会方便我们对上述操作系统的概念有一个更加系统的认知。

CPU“阿甘”之烦恼

原文链接：https://mp.weixin.qq.com/s/iRqFqWEtOnEPCtpXdXEIxA

文章转载自公众号码农翻身，　作者 刘欣

1、批处理系统

CPU“阿甘”最近很烦。

原因很简单，内存和硬盘看他不顺眼。

阿甘心里很清楚， 是自己干活太快了，干完了活就歇着喝茶，这时候内存和硬盘还在辛辛苦苦的忙活，他们肯定觉得很不爽了。

中国有句古话叫什么来着？ “木秀于林，风必摧之”，“不患贫而患不均”，这就是阿甘的处境。

虽然阿甘自己也于心不忍， 可是有什么办法？谁让他们那么慢 ！一个比自己慢 100 倍，另外一个比自己慢 100 万倍！

这个世界的造物主为什么不把我们的速度弄的一样呢？

阿甘所在的是一个批处理的计算机系统， 操作系统老大收集了一批任务以后，就会把这一批任务的程序逐个装载的内存中，让 CPU 去运行，大部分时候这些程序都是单纯的科学计算，计算弹道轨迹什么的， 但有时候也会有 IO 相关的操作，这时候，内存和硬盘都在疯狂的加班 Load 数据， 可是阿甘只能等待数据到来，只能坐那儿喝茶了。

没多久， 内存向操作系统老大告了阿甘一状， 阿甘被老大叫去训话了： “阿甘，你就不能多干一点？ 老是歇着喝茶算是怎么回事？”

阿甘委屈的说： “老大， 这不能怪我啊， 你看你每次只把一个程序搬到内存那里 让我运行， 正常情况下，我可以跑的飞快， 可以是一旦遇到 IO 相关的指令，势必要去硬盘那里找数据，硬盘实在是太慢了， 我不得不等待啊”

操作系统说： “卧槽， 听你的口气还是我的问题啊， 一个程序遇到了 IO 指令， 你不能把它挂起，存到到硬盘里，然后再找另外一个运行吗？”

阿甘笑了：“老大我看你是气昏头了， 我要是把正在运行的程序存到硬盘里，暂时挂起，然后再从硬盘装载另外一个， 这可都是 IO 操作啊 ，岂不更慢？”

“这？！” 操作系统语塞了，沉默了半天说：“这样吧， 我以后在内存里多给你装载几个程序，一个程序被 IO 阻塞住了， 你就去运行另外一个如何？”

“这得问问内存，看他愿不愿意了， 我把内存叫来，我们一起商量商量” 阿甘觉得这个主意不错。

内存心思缜密，听了这个想法， 心想：自己也没什么损失啊，原来同一时间在内存里只有一个程序， 现在要装载多个，对我都一样。

可是往深处一想，如果有多个程序，内存的分配可不是个简单的事情， 比如说下面这个例子：

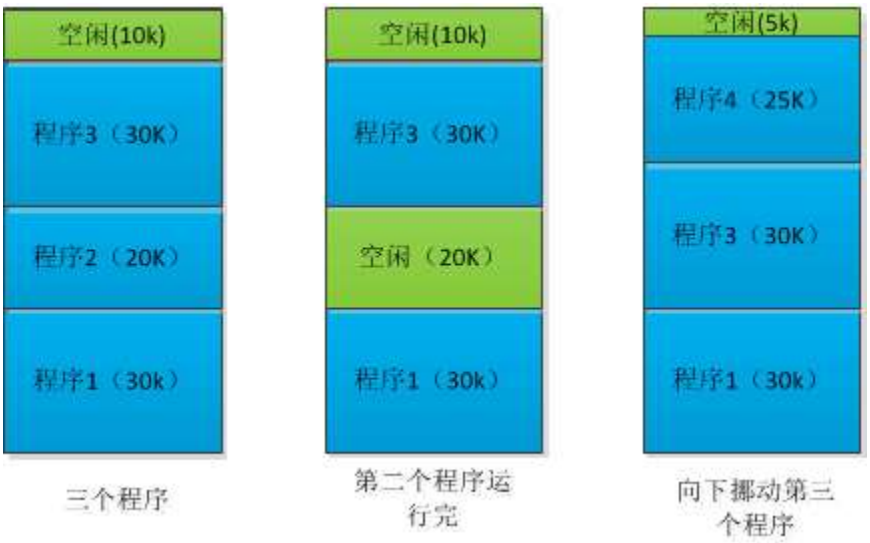


图 1 ：内存紧缩

(1) 内存一共 90k，一开始有三个程序运行，占据了 80k 的空间， 剩余 10k

(2) 然后第二个程序运行完了， 空闲出来 20k ，现在总空闲是 30K，但这两块空闲内存是不连续的。

(3) 第 4 个程序需要 25k，没办法只好把第三个程序往下移动， 腾出空间让第四个程序来使用了。

内存把自己的想法给操作系统老大说了说。

老大说： 阿甘，你要向内存学习啊， 看看他思考的多么深入，不过这个问题我有解决办法， 需要涉及到几个内存的分配算法， 你们不用管了。 咱们就这么确定下来，先跑两个程序试试。

2、地址重定位

第二天一大早，试验正式开始， 老大同时装载了两个程序到内存中：

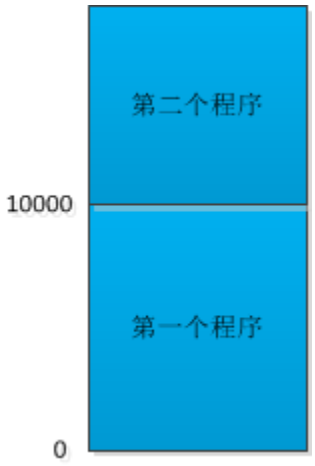


图 2：内存装入多个程序

第一个程序被装载到了内存的开始处，也就是地址 0 ，运行了一会，遇到了一个 IO 指令，在等待数据的时候， 老大立刻让 CPU 开始运行第二个程序，这个程序被装载到了地址 10000 处， 刚开始运行的好好的， 突然就来了这么一条指令：

```
MOV AX   [1000]
```

（作者刘欣注： AX 是一个寄存器， 你可以理解成在 CPU 内部的一个高速的存储单位，这个指令的含义是把 AX 寄存器的值写到内存地址 1000 处）

阿甘觉得似曾相识， 隐隐约约的记得第一个程序中也这么一条类似的指令： MOV BX   [1000]

“老大，坏了， 这两个程序操作了同一个地址！ 数据会被覆盖掉 ！” 阿甘赶紧向操作系统汇报。

操作系统一看就明白了，原来这个系统的程序引用的都是物理的内存地址， 在批处理系统中，所有的程序都是从地址 0 开始装载， 现在是多道程序在内存中， 第二个程序被装载到了 10000 这个地址，但是程序没有变化啊， 还是假定从 0 开始， 自然就出错了。

“看来老大在装载的时候得修改一下第二个程序的指令了，把每个地址都加上 10000（即第二个程序的开始处）， 原来的指令就会变成 MOV AX [11000] ” 内存确实反应很快。

（作者刘欣注： 直接修改程序的指令， 这叫静态重定位）

阿甘说： “ 如果用这种办法， 那做内存紧缩的时候可就麻烦了， 因为老大要到处移动程序啊， 对每个移动的程序岂不还都得做重定位？ 这多累啊！”

操作系统老大陷入了沉思， 阿甘说的没错， 这个静态重定位是很不方便， 看来想在内存中运行多道程序不是想象的那么容易。

但是能不能改变下思路，在运行时把地址重定位呢？

首先得记录下每个程序的起始地址， 可以让阿甘再增加一个**寄存器**，专门用来保存初始地址。

例如对第一个程序，这个地址是 0 ， 对第二个程序，这个地址是 10000，

运行第一个程序的时候，把寄存器的值置为 0 ， 当切换到第二个程序的时候，寄存器的值也应该切换成 10000。

只要遇到了地址有关的指令， 都需要把地址加上寄存器的值，这样才得到真正的内存地址，然后去访问。

（作者刘欣注： 这叫地址的**动态重定位**）

操作系统赶紧让阿甘去加一个新的寄存器， 重新装载两个程序，记录下他们的开始地址，然后切换程序，这次成功了， 不在有数据覆盖的问题了。

只是阿甘有些不高兴 ： “老大，这一下子我这里的活可多了不少啊， 你看每次访问内存， 我都得额外的做一次加法运算啊。”

老大说： “没办法，能者多劳嘛， 你看看我，我既需要考虑内存分配算法，还得做内存紧缩， 还得记住每个程序的开始地址， 切换程序的时候，才能刷新你的寄存器， 我比你麻烦多了！ ”

内存突然说到： ”老大， 我想到一个问题，假设有个不怀好意的恶意程序，它去访问别人的空间怎么办？ 比如说地址 2000 至 3000 属于一个程序， 但是这个程序来了一条这样的指令 MOV AX [1500]， 我们在运行时会翻译成 MOV AX [3500] ，这个 3500 有可能是别的程序的空间啊“

“唉，那就只好再加个寄存器了，阿甘， 用这个新寄存器来记录程序在内存中的长度吧， 这样每次访问的时候拿那个地址和这个长度比较一下，我们就知道是不是越界了”老大无可奈何了。

“好吧” 阿甘答应了，“ 我可以把这连个寄存器，以及计算内存地址的方法，封装成一个新的模块，就叫 **MMU （内存管理单元）**吧， 不过这个东西听起来好像应该内存来管啊”

内存笑着说： “那是不行的，阿甘， 能够高速访问的寄存器只有你这里才有啊 ， 我就是一个比你慢 100 倍的存储器而已！”。

3、分块装入程序

多道程序最近在内存中运行的挺好，阿甘没法闲下来喝茶了， 经常是一个还没运行完，很快就切换到另外一个。

那些程序也都是好事之徒，听说了这个新的系统，都拼了命，挤破头的往内存中钻。

内存很小，很快就会挤满， 操作系统老大忙于调度，也是忙的不可开交。

更有甚者，程序开始越长越大，有些图形处理的程序，还有些什么叫 Java 的程序，动不动就要几百 M 内存， 就这还嚷嚷着说不够。

操作系统头都大了，把 CPU 和内存叫来商量。

“世风日下，人心不古啊” 内存一边叹气一遍说 “原来批处理的时候那些程序规规矩矩的，现在是怎么了？”

“这也不能怪那些程序， 现在硬件的确比原来好多了，内存，你原来只有几十 K，现在都好几 G 了， CPU 在摩尔定律的关照下，发展的更快，每隔 18 个月，你的速度就翻一翻” 操作系统老大说。

“那也赶不上这些程序的发展速度，他们对我要求越来越高， 可是把我累坏了” CPU 垂头丧气的。

“我们还是考虑下怎么让有限的内存装下更多的程序吧”

“我有一个提议” 阿甘说 “对每个程序，不要全部装入内存，要分块装载，例如先把最重要的代码指令装载进来，在运行中按需装载别的东西。”

内存嘲笑说： “阿甘， 看来你又想偷懒喝茶了，哈哈， 如果每个程序都这样，IO 操作得多频繁， 我和硬盘累死， 你就整天歇着吧”

阿甘脸红了， 沉默了。

“慢着”老大说“阿甘，你之前不是发现过什么原理嘛， 就是从几千亿条指令中总结出的那个， 叫什么来着？”

“奥，那是局部性原理， 有两个：

(1) 时间局部性：如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行； 如果某数据被访问，则不久之后该数据可能再次被访问。

(2) 空间局部性：指一旦程序访问了某个存储单元，则不久之后。其附近的存储单元也将被访问。“

“这个局部性原理应该能拯救我们， 阿甘， 我们完全可以把一个程序分成一个个小块，然后按块来装载到内存中，由于局部性原理的存在， 程序会倾向于在这一块或几块上执行， 性能上应该不会有太大的损失。”

“这能行吗？ ” 内存和阿甘不约而同的问。

“试一试就知道了，这样我们把这一个个小块叫做**页框（page frame）**， 每个暂定 4k 大小， 装载程序的时候也按照页框大小来”

实验了几天， 果然不出老大所料， 那些程序在大部分时间真的只运行在几个页框中， 于是老大把这些页称为工作集(working set)

4、虚拟内存：分页

“既然一个程序可以用分块的技术逐步调入内存，而不太影响性能， 那就意味着，一个程序可以比实际的内存大的多啊”

阿甘躺在床上，突然间想到这一层， 心头突突直跳， 这绝对是一个超级想法。

“我们可以给每个程序都提供一个超级大的空间，例如 4G，只不过这个空间是虚拟的，程序中的指令使用的就是这些虚拟的地址，然后我的 MMU 把它们映射到真实的物理的内存地址上， 那些程序们浑然不觉，哈哈，实在是太棒了”

（作者刘欣注： 这就是我常说的增加一个中间层来解决问题）

内存听说了这个想法，惊讶的瞪大了双眼： “阿甘，你疯了吧”

“阿甘的想法是有道理的” 老大说 “只是我们还要坚持一点， 那就是分块装入程序，我们把虚拟的地址也得分块，就叫做页（page），大小和物理内存的页框一样， 这样好映射。”

“老大，看来你又要麻烦了， 你得维持一个页表， 用来映射虚拟页面和物理页面”

“不仅如此， 我还得记录一个程序那些页已经被装载到了物理内存， 那些没有被装载，如果程序访问了这些没被装载的页面，我还得从内存中找到一块空闲的地方， 如果内存已满， 只好把现有的页框置换一个到硬盘上了， 可是，怎么确定那个物理内存的页框可以置换呢？ 唉， 又涉及到很多复杂的算法，需要大费一番周折。 你看看，老大不是这么容易当的。”

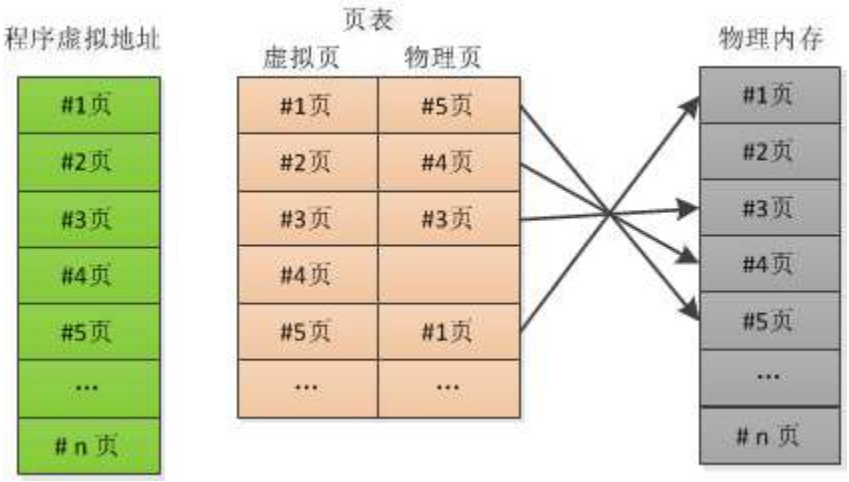


图 3： 分页

（作者刘欣： 这就是分页的工作原理，需要注意的是虚拟地址的#4 页， 在物理内存中不存在，如果程序访问第 4 页，就会产生缺页的中断，由操作系统去硬盘调取）

内存想起来一个问题： “如果程序运行时，每次都得查页表来获得物理的内存页， 而页表也是在内存里， 而我比你慢 100 倍， 你受得了吗， 阿甘？”

阿甘笑了： “这个问题其实我也考虑了，所以我打算增强我的内存管理单元， 把那些最常访问的页表项放到缓存里， 这样不就快了吗。 ”

内存想想也是， 还是局部性原理， 太牛了。

5、分段+分页

分页系统运行了一段时间以后， 又有程序表示不爽了，这些程序嚷嚷着说：

“你们能不能把程序“分家”啊， 例如代码段，数据段，堆栈段，这多么自然， 并且有利于保护，要是程序试图去写这个只读的代码段， 立刻就可以抛出保护异常！”

还有程序说：“页面太小了，实在不利于共享，我和哥们共享的那个图形库， 高达几十M ，得分成好多页来共享，太麻烦了，你们要是做一个共享段该多好！”

.....

这样的聒噪声多了， 大家都不胜其烦， 那就“分家”吧。

当然对每个程序都需要标准化， 一个程序被分成代码段，数据段和堆栈段等， 操作系统老大记录下每个段的开始和结束地址，每个段的保护位。

图 4：Linux 的虚拟内存示意图

但是在每个段的内部，仍然按分页的系统来处理，除了页表之外，操作系统老大又被迫维护了一个段表这样的东西 。

一个虚拟的内存地址来了以后，首先根据地址中的段号先找到相应的段描述表， 其中有页表的地址， 然后再从页表中找到物理内存， 过程类似这样：

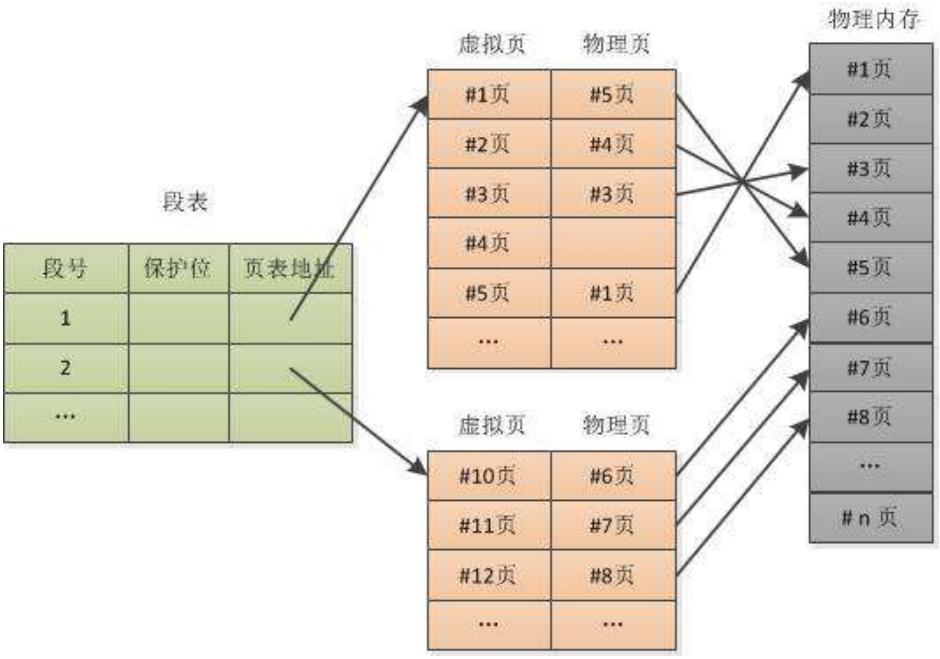


图 5：一个简化的段表和页表

所有事情都设置好了， 大家都喘了口气，觉得这样的结构大家应该没什么异议了。

老大心情大好，觉得一切尽在掌握，他笑着对 CPU 阿甘说：

“阿甘，从今天开始，如果有程序想非法的访问内存，例如一个不属于他的段， 我就立刻给他一个警告：Segmentation Fault !”

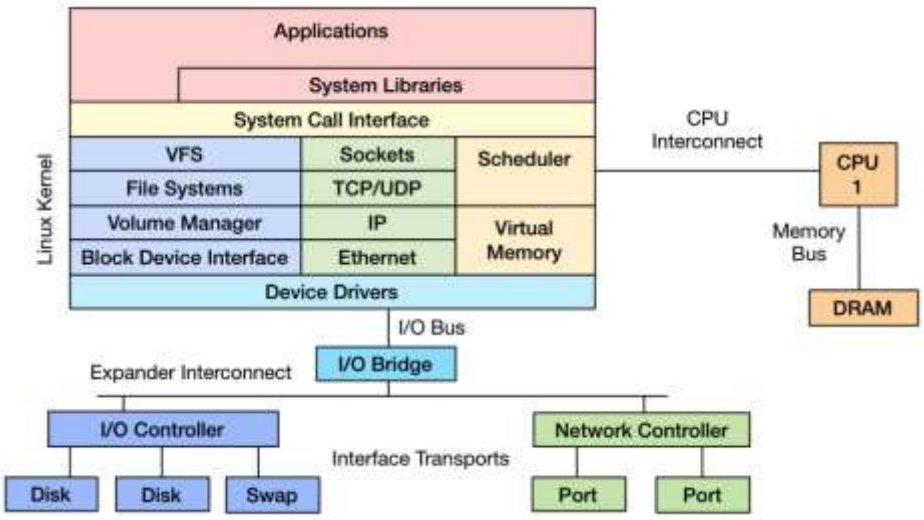
阿甘说：“那程序收到 Segmentation Fault 以后怎么处理？”

老大说： “通常情况下就被我杀死， 然后给他产生一个叫 core dump 的尸体，让那些码农们拿走分析去吧!”

（完）

Linux 性能分析

前面我们已经知道 Linux 操作系统大致是这么个样子。



有时候系统出现了问题，这些问题通常以某个应用程序的异常为表现。但当我们根据给现象进行问题排查事，会发现这些问题的本质往往并没有看起来的那么简单。

对于此类常见问题的分析思路是这样的：

1、**完善问题描述：**当故障发生时，我们会从发现故障者那里获取到一个对问题的描述。这些描述有些是完善的，有些却并不完善。从知情者那里获取并完善这些问题的描述，有助于后面的问题解决。哪里出了问题？为什么认为有问题？问题能否用数值表示（如延时、资源使用率）？

观测问题，确定有无其他可能引发问题出现的原因。

2、**排除低级错误，缩小调查范围：**

网络：检查防火墙、

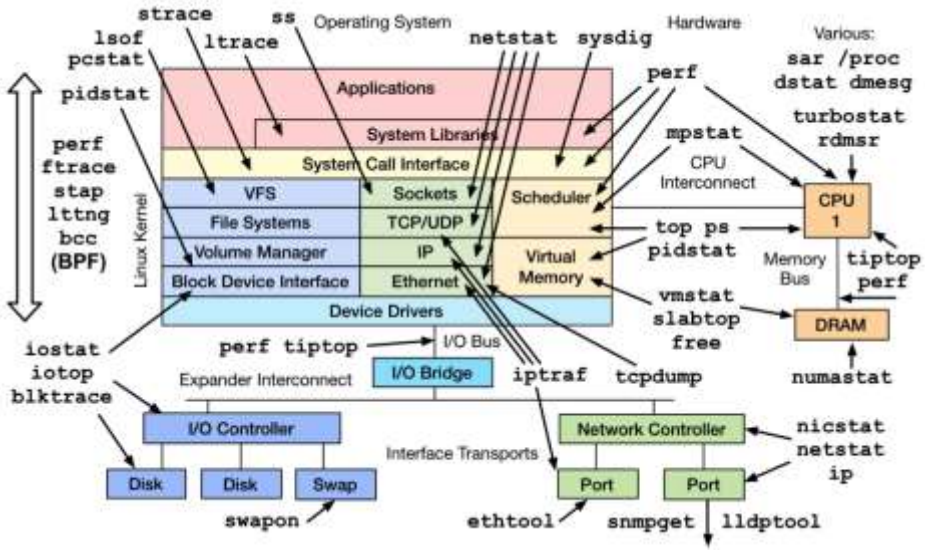
dmesg 查看系统日志

查看应用本身输出的日志

3、**根据问题描述确定排查思路：**列举可能导致该问题出现的若干原因。

根据程序的特性（如 web 服务器使用网络通信、数据库是高 CPU 程序、分布式程序 hadoop 要经常写数据）可大致确定程序运行的瓶颈。虽然通常程序运行需要多种资源，但我们仍可根据优先级逐渐对程序运行状况进行调试。

了解操作系统相关的资源使用情况（CPU、内存、磁盘、网络）可通过系统系统的观测命令。以 Linux 操作系统为例，我们可使用的观测工具如下。



这些资源的关注项大致如下：

系统负载层面：

1、关注系统负载；

2、查看系统开机之后是否出现什么问题 dmesg；

3、关注占用资源较高的资源（top/htop 按 CPU、内存、运行时间排序）

进程：

1、 关注运行中进程数，及阻塞中的进程数（vmstat 1--r、b）

2、 查看每个进程的情况：pidstat 1

CPU：

1、 **整体 CPU 使用率：mpstat -P ALL 1**（显示所有 CPU 的信息，间隔为 1s）

2、 系统**负载**：Load 是否大于 CPUs、load 是否正常

3、 每个 CPU 的使用情况：共多少**逻辑核** mpstat -P ALL 1、top（1）

4、 **主频多少**：是否因为主频偏低引起负载较高，cat /proc/cpuinfo |grep 'cpu MHz'、perf

5、 用户态和内核态的 CPU 占用情况：内核态过高，说明系统调用多，是有问题的

6、 每个进程/线程 CPU 用量，找出占用率最高的进程：perf、top、time（给一个命令计时，带 CPU 用量分析）

内存：

1、 内存使用情况：free -m、cat /proc/meminfo

2、 观察使用率，之后再决定是否排查有无换页情况：sar-B 1

磁盘 IO：

1、查看磁盘负载，包括下面这些具体指标：

1 多少存储空间正在进行 IO：（设备忙于处理 IO 请求的百分比：util）

1 IO 平均响应时间（包括驱动请求队伍里等待和设备的 IO 响应时间：await ）

1 理论上 IO 处理速度（推断磁盘设备的 IO 平均响应时间：svctm ）

1 实际处理请求数（在驱动请求队列里和在设备中活跃的平均请求数：avgqu-sz）

网络 IO：

1、查看网络吞吐：sar -n DEV 1

2、查看 TCP 连接情况：sar -n TCP,ETCP 1

3、查看网络相关的进程信息：netstat -p

当范围缩小到应用程序自身时，从进程维度（参考该资料下面《进程》部分）进行分析，直至定位问题源头。

系统负载

uptime

使用 uptime 可以查看当前机器负载。这里的负载（load）显示的是服务器在过去的 1 分钟、5 分钟、15 分钟的系统平均负载值。

这里的负载为 CPU+Disk 的负载，是个综合值，从这个值我们大体的可以估算出此系统是处于何种压力之下。

1 如果负载大于 CPU 核数（逻辑核），则代表此系统有压力，如果是 15 分钟持续很高，则更需注意。

1 如果机器负载很低也需要注意，大部分情况是进程挂掉了，或者进程在空跑

1 如果机器一直负载很高，但是小于 CPU 的核数（逻辑核）可以适当的注意，出现这种情况要么是 CPU 主频太低（本身就很差），要么就是数据量大。

1 如果机器负载很低，系统也可能是存在问题的。如进程挂掉，或者进程还活着，但是不工作了。

top/htop

可以查看系统总体情况及各资源使用情况。

1、查看 CPU（1）、平均负载、运行中进程数、内存使用情况、以及资源占用大的进程

dmesg

dmesg | tail 查看系统出现的问题

进程

根据进程的使用状态，

vmstat 1：查看进程是否阻塞（r）

pidstat 1：查看每个进程的情况

线程详解

线程状态

On-CPU

Off-CPU：等待上 CPU，或者等待 IO、锁、换页、工作等

线程的六种状态

On-CPU

执行：在 CPU 上

Off-CPU：等待上 CPU，或者等待 IO、锁、换页、工作等

可运行：等待轮上 CPU

匿名换页：可运行，但是 因等待匿名换页而受阻。

睡眠：等待包括网络、块设备和数据/文本换入在内的 IO

锁：等待获取同步锁（等待其他线程）

空闲：等待工作

根据线程状态做出如下研究：

1　执行：检查执行的是用户态时间还是内核态时间，用剖析来做 CPU 资源消耗分析。剖析可以确定那些代码路径消耗 CPU 和消耗了多久，其中包括花费在自旋锁上的时间。

1　可运行：等待轮上 CPU。在这个状态是哪个耗时意味着应用程序需要更多的 CPU 资源。检查整个系统的 CPU 负载，以及所有对该应用程序做出的 CPU 限制（如资源控制）。

1　匿名换页：可运行，但是 因等待匿名换页而受阻。应用程序缺少可用的主存会引起换页和延时。检查整个系统的内存使用情况，和所有对该应用程序做出的内存限制。

1　睡眠：等待包括网络、块设备和数据/文本换入在内的 IO。分析阻塞应用程序的资源。

1　锁：等待获取同步锁（等待其他线程）。识别锁和持有该锁的线程，确定线程持锁这么长时间的原因。原因可能是持锁线程阻塞在另一个锁上，这需要进一步梳理。这件事通常是熟悉应用和其锁机制的开发人员要做的事。

1　空闲：等待工作

睡眠和锁状态的时间实际上就是空闲的时间。一个应用程序可能会为了工作等待条件变量（锁状态），或者为了网络 IO（睡眠状态）。

当看到大量的睡眠和锁状态时间时，记住要深入检查应用程序是否真的空闲。

观测

执行：在 CPU 上

关注执行时间，即 CPU 占用时长（top 里的%CPU）

可运行：等待轮上 CPU

cat /proc/pid/schedstat

perf sched

匿名换页：可运行，但是 因等待匿名换页而受阻。

跟踪工具，starce 应 OK

睡眠：等待包括网络、块设备和数据/文本换入在内的 IO

pidstat -d 判断一个进程是在执行 IO，还是睡眠

iotop 如果启用了延时核算和 IP 核算的特征，可以用 iotop 观察到 IO 块阻塞的时间

statce 等追踪工具：调查其他造成阻塞的时间

监测点：根据应用程序的监测点精确跟踪 IO（磁盘、网络）的执行时间

快照：pstack 会对线程和用户栈做一个快照，以发现睡眠的线程和线程睡眠的原因。不过使用 pstack 可能会让目标有短暂停顿

锁：等待获取同步锁（等待其他线程）

statce 等追踪工具

空闲：等待工作

CPU 剖析

当应用程序在**执行**时，判断应用程序是如何消耗 CPU 资源的。

方法：

1、栈跟踪：一个有效的技术是对 CPU 上的用户栈跟踪做采样并将采样结果联系起来。栈跟踪告诉我们所选择的代码路径。但输出内容多，可以用火焰图做可视化。

2、对当前运行的函数做采样：perf、starce

系统调用分析

当应用程序 Off-CPU（等待上 CPU，或者等待 IO、锁、换页、工作等）时，应用程序实际上在等待系统调用。

研究系统调用在于找到系统调用的时间花在什么地方了、分析系统调用的类型以及使用该系统调用的原因。

方法：

1、断点跟踪：

strace -ttt -T -p 1884

-ttt：打印第一栏 UNIX 时间戳，以秒为单位，精确到毫秒级。

-T：打印最后一栏，这是系统调用的时间，以秒为单位，精确到毫秒级。

-p pid。

strace -c -p 1884

-c：用于系统调用活动的统计结果

```
$ strace -c -p 1884
Process 1884 attached - interrupt to quit
^CProcess 1884 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 83.29    0.007994           9       911       455 wait4
 14.41    0.001383           3       455           clone
  0.85    0.000082           0      2275           ioctl
  0.68    0.000065           0       910           close
  0.63    0.000060           0      4551           rt_sigprocmask
  0.15    0.000014           0       455           setpgid
  0.00    0.000000           0       455           rt_sigreturn
  0.00    0.000000           0       455           pipe
------ ----------- ----------- --------- --------- ----------------
100.00    0.009598                 10467       455 total
```

输出如下。

- **time**：显示系统 CPU 时间花在哪里的百分比。
- **seconds**：总的系统 CPU 时间，单位是秒。
- **usecs/call**：每次调用的平均系统 CPU 时间，单位是毫秒。
- **calls**：整个 strace(1)过程内的系统调用次数。
- **syscall**：系统调用的名字。

断点跟踪 starce 对该拼读系统调用和追踪函数调用的开销较高，所以不适合应用于度搜狐的生产环境场景。

2、缓冲跟踪

perf

IO 剖析

pidstat -d 判断一个进程是在执行 IO，还是睡眠

iotop 如果启用了延时核算和 IP 核算的特征，可以用 iotop 观察到 IO 块阻塞的时间

statce 等追踪工具：调查其他造成阻塞的时间

监测点：根据应用程序的监测点精确跟踪 IO（磁盘、网络）的执行时间

快照：pstack 会对线程和用户栈做一个快照，以发现睡眠的线程和线程睡眠的原因。不过使用 pstack 可能会让目标有短暂停顿

CPU

表 6.6 CPU 分析工具

Linux	Solaris	描述
uptime	uptime	平均负载
vmstat	vmstat	包括系统范围的 CPU 平均负载
mpstat	mpstat	单个 CPU 统计信息
sar	sar	历史统计信息
ps	ps	进程状态
top	prstat	监控每个进程/线程 CPU 用量
pidstat	prstat	每个进程/线程 CPU 用量分解
time	ptime	给一个命令计时，带 CPU 用量分解
DTrace, perf	DTrace	CPU 剖析和跟踪
perf	cpustat	CPU 性能计数器分析

top,htop

按 1 显示 CPU 核数，查看用户态和内核态（不能太高），以及 steal 值（虚拟机需要注意）。

```
top - 20:47:29 up 43 days,   5:49,   1 user,   load average: 0.79, 0.80, 0.81

Tasks: 131 total,     1 running, 130 sleeping,     0 stopped,     0 zombie

Cpu0   :   7.8%us,   7.8%sy,   0.0%ni,
83.3%id,   0.0%wa,   0.0%hi,   0.0%si,   1.0%st

Cpu1   :   6.1%us,   3.0%sy,   0.0%ni,
89.9%id,   0.0%wa,   0.0%hi,   0.0%si,   1.0%st

Cpu2   :  16.8%us,   3.0%sy,   0.0%ni,
77.2%id,   0.0%wa,   0.0%hi,   1.0%si,   2.0%st

Cpu3   :  15.0%us,   3.0%sy,   0.0%ni,
81.0%id,   0.0%wa,   0.0%hi,   0.0%si,   1.0%st

Mem:   16332272k total, 16116640k used,     215632k free,     125084k buffers
```

Swap: 5242872k total, 702196k used, 4540676k free, 1489164k cached

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

19816 rizhiyi   20   0 4625m 1.2g 8424 S 16.7  7.6   3065:02 /opt/rizhiyi/java/b

20002 rizhiyi   20   0 8210m 3.1g 8608 S 10.8 19.9 761:53.95 /opt/rizhiyi/java/b

30058 rizhiyi   20   0 2878m 301m 3260 S 10.8  1.9   1252:38 /opt/rizhiyi/parcel

 4047 rizhiyi   20   0 5295m 1.0g 8632 S  7.9  6.7 711:59.60 /opt/rizhiyi/java/b

 3914 rizhiyi   20   0 3435m 627m 9724 S  5.9  3.9  95:16.70 /opt/rizhiyi/java/b

 2543 rizhiyi   20   0 2316m  63m 2368 S  4.9  0.4   3046:21 /opt/rizhiyi/python

 2690 rizhiyi   20   0 2246m 181m 3980 S  2.9  1.1   1331:30 /opt/rizhiyi/mysql/

 4813 root      20   0 15028 1320  972 R  2.9  0.0   0:05.84 top
```

dstat

vmstat

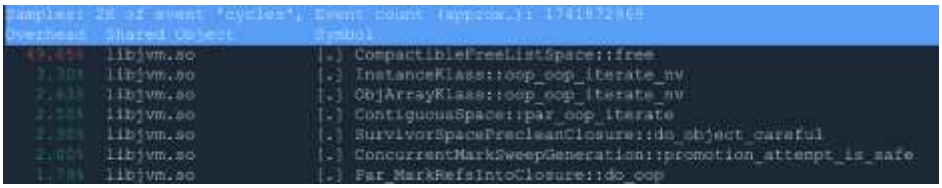
/proc/cpuinfo

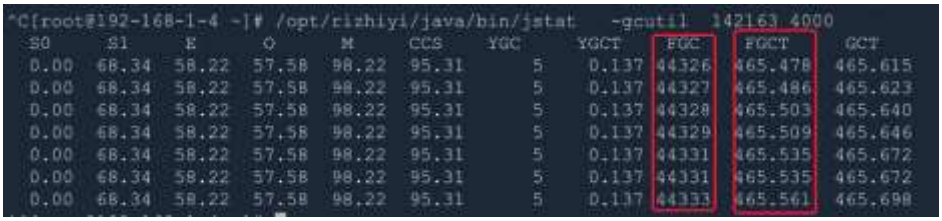
mpstat -P ALL 1：查看每个 CPU 的运行情况

pidstat：查看进程的 CPU 使用，可以使用 pidstat

sar -u 1

perf、top 可以看出系统中哪个函数在 CPU 中使用的很高，也可以更细致的分析某个进程，如 jstat





Memory

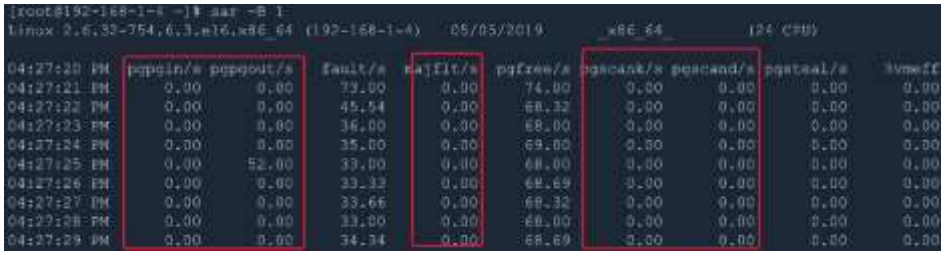
free -m：查看内存使用情况

vmstat

工具	使用方式	参数信息		
vmstat	vmstat 2 5 (vmstat -s、vmstat -Sm 1)	procs	r	可运行（正运行或等待运行）进程的个数，和核心数有关
			b	处于不可中断睡眠态的进程个数(被阻塞的队列的长度)
		memory	swpd	交换内存的使用总量
			free	空闲物理内存总量
			buff	用于 buffer 的内存总量
			cache	用于 cache 的内存总量
		swap	si	从磁盘交换进内存的数据速率(kb/s)
			so	从内存交换至磁盘的数据速率(kb/s)

		io	bi	从块设备读入数据到系统的速率(kb/s)
			bo	保存数据至块设备的速率
		system	in	interrupts 中断速率，包括时钟
			cs	context switch 进程切换速率
		cpu	us	运行非内核代码所花费的时间
			sy	运行内核代码所花费的时间
			id	闲着的时间。Linux 2.5.41 前，包括 IO 等待时间。
			wa	等待 IO 的时间。2.5.41 前，包括空转。
			st	从虚拟机中窃取的时间。2.6.11 前，未知。

sar -B 1



dstat：

sar -r

dmesg | grep killed

Disk

iostat -xzm 1：查看磁盘的负载情况

```
root@192.168.1.100:~$iostat -xmz 1
Linux 2.6.32-431.11.2.el6.x86_64 (liukang)      2019年09月24日   _x86_64_        (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          15.30    0.00    3.57    1.35    0.72   79.06

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
vda               0.18    65.64    0.93   20.50     0.03     0.32    33.44     0.57   26.70   18.11   27.09   3.06   6.55
vdb               0.04    31.67    0.46    8.53     0.02     0.14    36.46     0.10   10.84   12.08   10.77   1.72   1.55
vdc               0.01    15.38    0.09    7.34     0.01     0.07    21.74     0.06    8.44   13.66    8.37   1.97   1.46
dm-0              0.00     0.00    1.63  137.31     0.06     0.53     8.66     1.00    7.16   18.00    7.03   0.53   7.35
dm-1              0.00     0.00    0.00    0.27     0.00     0.00     8.00     0.06  163.48   12.36  210.00   0.80   0.03
```

如果 %util 接近 100%，说明产生的 I/O 请求太多，I/O 系统已经满负荷，该磁盘可能存在瓶颈。

如果 svctm （推断磁盘设备的 IO 平均响应时间）比较接近 await（磁盘设备的 IO 平均响应时间），说明 I/O 几乎没有等待时间；

如果 await 远大于 svctm，说明 I/O 队列太长，io 响应太慢，则需要进行必要优化。

如果 avgqu-sz（在驱动请求队列里和在设备中活跃的平均请求数）比较大，也表示有 io 在等待

sar -d 1

pidstat -d 1

smartctl

ss

iotop

Network

sar -n DEV 1：查看网路的吞吐情况

```
root@192.168.1.242:~$sar -n DEV 1
Linux 2.6.32-431.11.2.el6.x86_64 (192-168-1-242)        05/05/2019      _x86_64_        (4 CPU)

08:37:42 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
08:37:43 PM        lo    143.88    143.88     18.24     18.24      0.00      0.00      0.00
08:37:43 PM      eth0     44.90      2.04      5.26      0.29      0.00      0.00      0.00

08:37:43 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
08:37:44 PM        lo   1351.00   1351.00    826.95    826.95      0.00      0.00      0.00
08:37:44 PM      eth0     69.00      6.00      5.20      1.60      0.00      0.00      0.00

08:37:44 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
08:37:45 PM        lo    208.00    208.00     26.68     26.68      0.00      0.00      0.00
08:37:45 PM      eth0     26.00      1.00      1.80      0.05      0.00      0.00      0.00
```

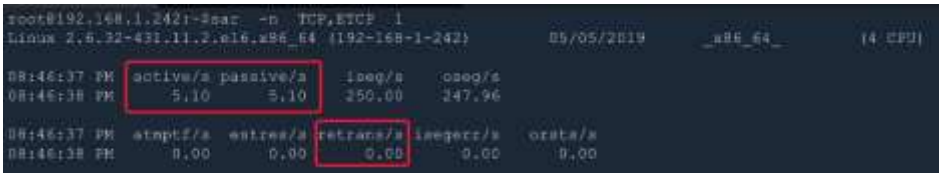
ip -s link

netstat -s

netstat -s 查看静态信息的详细输出

netstat -p 查看网络相关的进程信息

sar -n TCP,ETCP 1：查看 TCP 的情况

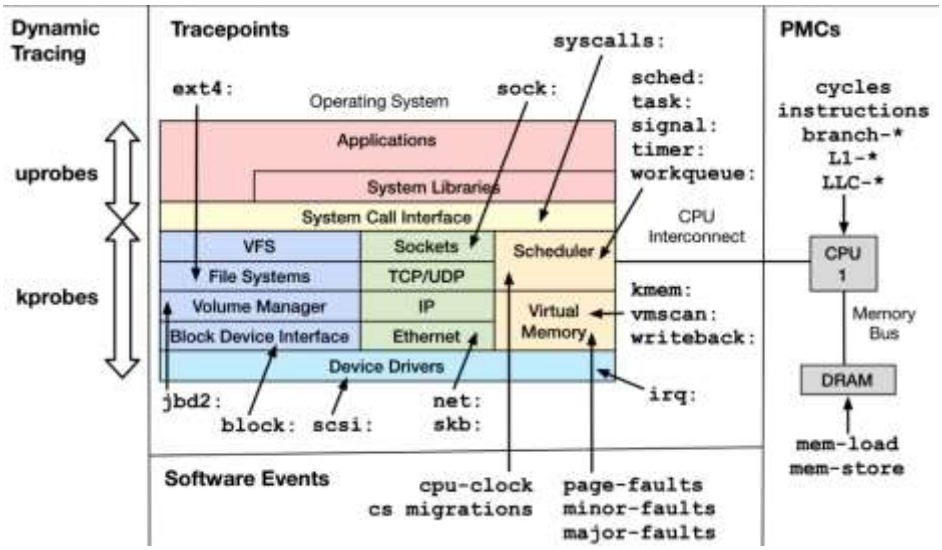


active/s ：从 CLOSED 状态变为 SYN-SENT 状态 --- 发送

passive/s：从 LISTEN 状态变为 SYN-RECV 状态 --- 接收

retrans/s：每秒的错误重传率

perf



工作负载

USE

向下挖掘分析

锁分析

静态性能调优

vmstat：

r 那列：查看运行中进程有多少个，看有没有阻塞

ps

pidstat

netstat/ss

参考《Shell 命令行的艺术》系统调试部分。

- curl 和 curl -I 可以被轻松地应用于 web 调试中，它们的好兄弟 wget 也是如此，或者也可以试试更潮的 httpie。
- 使用 netstat 和 ss 查看网络连接的细节。
- 使用 mtr 去跟踪路由，用于确定网络问题。
- ab 工具（Apache 中自带）可以简单粗暴地检查 web 服务器的性能。对于更复杂的负载测试，使用 siege。
- wireshark，tshark 和 ngrep 可用于复杂的网络调试。
- dstat 在你想要对系统的现状有一个粗略的认识时是非常有用的。然而若要对系统有一个深度的总体认识，使用 glances，它会在一个终端窗口中向你提供一些系统级的数据。
- 获取 CPU 和硬盘的使用状态，通常使用使用 top（htop 更佳），iostat 和 iotop。而 iostat -mxz 15 可以让你获悉 CPU 和每个硬盘分区的基本信息和性能表现。
    - 若要了解内存状态，运行并理解 free 和 vmstat 的输出。值得留意的是“cached”的值，它指的是 Linux 内核用来作为文件缓存的内存大小，而与空闲内存无关。
    - Java 系统调试则是一件截然不同的事，一个可以用于 Oracle 的 JVM 或其他 JVM 上的调试的技巧是你可以运行 kill -3 <pid> 同时一个完整的栈轨迹和堆概述（包括 GC 的细节）会被保存到标准错误或是日志文件。JDK 中的 jps，jstat，jstack，jmap 很有用。SJK tools 更高级。
    - 用 ncdu 来查看磁盘使用情况，它比寻常的命令，如 du -sh *，更节省时间。
    - 查找正在使用带宽的套接字连接或进程，使用 iftop 或 nethogs。
    - 了解 strace 和 ltrace。这俩工具在你的程序运行失败、挂起甚至崩溃，而你却不知道为什么或你想对性能有个总体的认识的时候是非常有用的。注意 profile 参数（-c）和附加到一个运行的进程参数 （-p）。
    - 了解使用 ldd 来检查共享库。但是永远不要在不信任的文件上运行。
    - 了解如何运用 gdb 连接到一个运行着的进程并获取它的堆栈轨迹。

- 学会使用 /proc。它在调试正在出现的问题的时候有时会效果惊人。比如：/proc/cpuinfo，/proc/meminfo，/proc/cmdline，/proc/xxx/cwd，/proc/xxx/exe，/proc/xxx/fd/，/proc/xxx/smaps（这里的 xxx 表示进程的 id 或 pid）。
- 当调试一些之前出现的问题的时候，sar 非常有用。它展示了 cpu、内存以及网络等的历史数据。
- 关于更深层次的系统分析以及性能分析，看看 stap（SystemTap），perf，以及 sysdig。
- 查看你当前使用的系统，使用 uname，uname -a（Unix / kernel 信息）或者 lsb_release -a（Linux 发行版信息）。
- 无论什么东西工作得很欢乐（可能是硬件或驱动问题）时可以试试 dmesg。
- 如果你删除了一个文件，但通过 du 发现没有释放预期的磁盘空间，请检查文件是否被进程占用： lsof | grep deleted | grep "filename-of-my-big-file"

案例

案例 1-负载高

现象：

- 负载异常高，load 达到了 100 多

- 机器非常卡顿，敲命令会被卡主

- 出现时间不规律

排查：

- CPU 共 16 核，虚拟机,使用率高但是正常；sys 很低，维持在 5%左右；iowait 很低，维持在 1 以下；

- steal 值很高，经常到 80

- 内存使用正常，没有进行换页

- 系统中 runable 进程的数量维持在正常水平（16 以内）

- 磁盘读写正常，因为此时机器已经非常卡顿，进程已经停止

- 网络正常

分析：

- 从 load 数值来看此机器是异常的，不是正常使用所能达到的一个数值，与 load 相关的基本上就是 CPU+DISKs。其中 CPU 有个异常值 steal 而且使用的机器是虚拟机。问题基本上可以确定—CPU 超卖

- 如果 steal 数值达到了 20 以上，我们所能做的就是迁移物理机上的虚拟机至其他位置，不要在此物理机上再运行虚拟机。

结果：

将此物理机上的与日志易无关的服务器迁移至其他机器，问题解决，客户没有在回复。

案例 2-磁盘

现象：

- 系统负载高，而且登录机器之后非常卡顿

排查：

- CPU 共 16 核，虚拟机,us 使用率正常；sys 很低，

维持在 5%左右；iowait 很高，经常 40 以上；steal 值正常，为 0

- 内存使用正常，没有进行换页

- 系统中 runable 进程的数量维持在正常水平（16 以内）

- 磁盘读写有问题，如右图：

分析：

- 从图中可以看出，是磁盘有问题，主要是写出现了

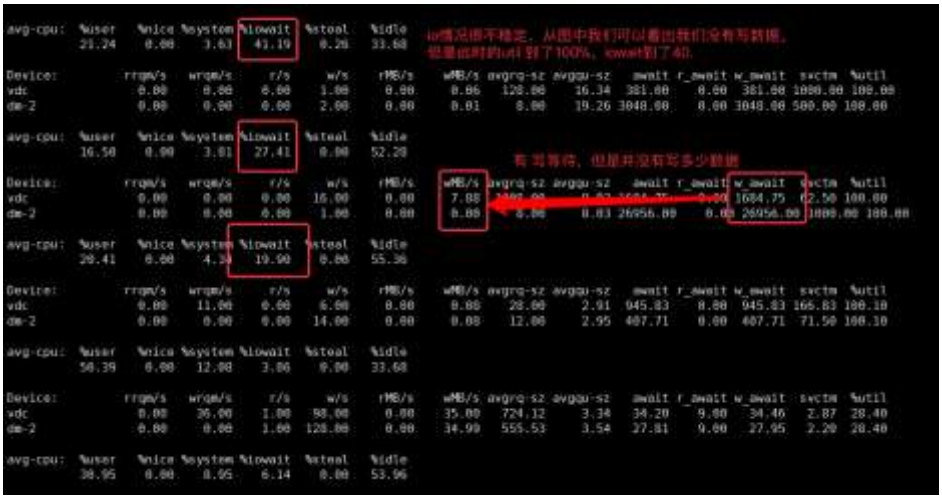
  问题；

- wMb/s 数值很小，但是 util 却很高，

- await 数值和 w_wait 数值相近，说明都是由写造成的

结果：

将此问题报告给客户，由客户进行调整



案例 3-网络

现象：

- logriver 回写 kafka 经常出错（sink error）

- 出现时间不固定

分析：

- kafka 是一个消息队列，它最依赖的就是磁盘+网络，logriver 回写 kafka 出错，需要查看是否是硬盘 io 不足导致

- 在 io 充足的情况下需要判断网络情况，如果判断网络情况需要计算当时的数据量，来粗略的估计一下每秒的数据量。

- 询问过研发，给的方案是增大 java_heap_size ，以及修改 ack 参数

排查：

- 使用 iostat 查看磁盘读写情况发现没问题（kafka 虽然依赖磁盘，但是他顺序写，在很烂的磁盘上都可以跑出高 eps）

- 根据每天的日志量计算一下平均每秒写入的量，发现基本上已经达到网络瓶颈，在后台查看网络使用，基本上已经达到千兆网络的的吞吐量。

案例 4-日志

在某些情况下，我们通过前面所说的命令查不到我们想要的结果，这时候我们就需要查看系统日志 dmesg /var/log/message

客户中海油

现象：

负载很高(60)，登录卡顿，但是登录之后执行命令很正常，不卡顿。kafka 已经不被消费说明此负载已经影响

到了系统。

分析：

CPU 使用率很低（us10% 左右，sys5%一下，iowait 没有，steal 为 0，zobine 为 0）内存使用正常，剩余近 35G。

磁盘 io 正常，队列没有挤压，因为没有消费 kafak 的数据，所以磁盘读写很低。查看系统中的运行 task 数，很

正常，没有超出 CPU（16 核）的数量。网络正常。

从上面可以看出所有可能出现问题的原因都被查看，

如何调优

CPU：

- 进程绑定
- 在 BIOS 中禁用省电模式

内存：

- vm.swappiness = 0
- kernel.numa_balancing = 0 （NUMA 架构，最好只访问自己本地寄存器的值）
- vm.dirty_ratio = 80 # 达到多少会进行刷盘，会阻塞所有的写操作
- vm.dirty_background_ratio = 5 # 异步刷盘，不会进行阻塞

- vm.dirty_expire_centisecs = 12000 # 内存中的脏数据过期时间，即超过多少秒刷盘（默认百万分之一）

磁盘/文件系统：

- 尽量使用 xfs 文件系统
- mount -o defaults,noatime,discard,nobarrier 挂载盘时可以考虑的挂载项
- /sys/block/*/queue/scheduler noop #如果是 ssd,可以使用电梯算法

网络：

- net.core.somaxconn = 1000
- net.core.netdev_max_backlog = 5000
- net.core.rmem_max = 16777216
- net.core.wmem_max = 16777216
- net.ipv4.tcp_wmem = 4096 12582912 16777216
- net.ipv4.tcp_rmem = 4096 12582912 16777216
- net.ipv4.tcp_max_syn_backlog = 8096
- net.ipv4.tcp_slow_start_after_idle = 0
- net.ipv4.tcp_tw_reuse = 1
- net.ipv4.ip_local_port_range = 10240 65535
- net.ipv4.tcp_abort_on_overflow = 1

上面的给的值是个建议值，需要根据不同的环境来进行调整。

JVM

调优的目的就是为减少 GC 次数，降低 GC 时间

1. Full GC >> Minor GC

- 尽量减少 Full GC 次数，可以将年轻代调大

2. 尽量减少大对象，尤其是短命的大对象

- 因为大对象非常占用里面的内存，所以尽量少的占用大对象内存，可以将大对象直接分配到年老代，

保持年轻代对象结构的完整性，这样可以提高 GC 的效率（-XX:PetenureSizeThreshold ）；也可以设置进入

老年代的年龄（XX:MaxTenuringThreshold 默认是 15 ），来让他在年轻代多待会

1. 是否使用稳定的堆与不稳定的堆看应用，对于我们系统来说，可以使用不稳定的堆

- 稳定堆 -Xms 和-Xmx 的大小一致，即最大堆和最小堆（初始堆）一样，减少 gc 次数，但是增加 gc 时间

- 非稳定堆 -XX:MinHeapFreeRatio 参数用来设置堆空间最小空闲比例，默认值是 40。当堆空间的空闲内存小于

这个数值时，JVM 便会扩展堆空间。-XX:MaxHeapFreeRatio 参数用来设置堆空间最大空闲比例，默认值是 70

1. 使用增大吞吐量方案

- 在 JVM 有两种评估 吞吐量(应用程序线程用时占程序总用时的比例)

- 暂停时间（一个时间段内应用程序线程让与 GC 线程执行而完全暂停）

2. 使用新版的 jdk 以及选用 G1 或者 ZGC 收集器

参考资料

参考链接：

Linux perf

http://www.brendangregg.com/overview.html

Linux USE Method

http://www.brendangregg.com/USEmethod/use-linux.html

JVM：

https://blog.csdn.net/yfqnihao/article/details/8289363

https://www.ibm.com/developerworks/cn/java/j-lo-jvm-optimize-experience/index.html

调优文档

https://www.lijiaocn.com/linux/chapter1/

https://lihz1990.gitbooks.io/transoflptg/content/

其他：自行查找相关资料