对不起，学会这些 Linux 知识后，我有点飘

[架构师修行之路](javascript:void(0);) 今天

以下文章来源于程序员cxuan ，作者cxuan

[程序员cxuan](https://mp.weixin.qq.com/s?__biz=MzIwNTc3OTAxOA==&mid=2247486641&idx=1&sn=a7bb78c17f1825efea4c70ac4bee4e82&chksm=972af594a05d7c823b65afc8e3d4f2c0d53ced05940a8308cfe388120f7ce34ffca2e4c96c91&scene=126&sessionid=1602752464&key=82003248020682faaa5cc09a198a10a9134ec79434462df07e8ea3f404e87639ec0dc39a200c099f8e897169b0f2cbe7a3b247650e6cc5bc865c260d0603d5881b66517850e8d8593cbc6326645408f22677d9b1af23a0c0e47b25030395b6982c0967973cf4cceff2181c22d2cdb6c3194099495414cf1b9ffe6a52ae06308f&ascene=1&uin=NDM2MDg2NzU1&devicetype=Windows+10+x64&version=6300002f&lang=zh_CN&exportkey=AXXwzMyQj06h4yYyAseUzX4%3D&pass_ticket=4bKw9fQ%2FYTkSBvgKjRPrY5nCsS%2FJ1M1xMETIXBFiTLY0kKtD5qwL3In5qJm6d5YM&wx_header=0)

[cxuan 写的文章还不错。会分享计算机底层、计算机网络、操作系统，Java基础、框架、源码等文章。](https://mp.weixin.qq.com/s?__biz=MzIwNTc3OTAxOA==&mid=2247486641&idx=1&sn=a7bb78c17f1825efea4c70ac4bee4e82&chksm=972af594a05d7c823b65afc8e3d4f2c0d53ced05940a8308cfe388120f7ce34ffca2e4c96c91&scene=126&sessionid=1602752464&key=82003248020682faaa5cc09a198a10a9134ec79434462df07e8ea3f404e87639ec0dc39a200c099f8e897169b0f2cbe7a3b247650e6cc5bc865c260d0603d5881b66517850e8d8593cbc6326645408f22677d9b1af23a0c0e47b25030395b6982c0967973cf4cceff2181c22d2cdb6c3194099495414cf1b9ffe6a52ae06308f&ascene=1&uin=NDM2MDg2NzU1&devicetype=Windows+10+x64&version=6300002f&lang=zh_CN&exportkey=AXXwzMyQj06h4yYyAseUzX4%3D&pass_ticket=4bKw9fQ%2FYTkSBvgKjRPrY5nCsS%2FJ1M1xMETIXBFiTLY0kKtD5qwL3In5qJm6d5YM&wx_header=0)

Linux 简介

UNIX 是一个交互式系统，用于同时处理多进程和多用户同时在线。为什么要说 UNIX，那是因为 Linux 是由 UNIX 发展而来的，UNIX 是由程序员设计，它的主要服务对象也是程序员。Linux 继承了 UNIX 的设计目标。从智能手机到汽车，超级计算机和家用电器，从家用台式机到企业服务器，Linux 操作系统无处不在。

大多数程序员都喜欢让系统尽量简单，优雅并具有一致性。举个例子，从最底层的角度来讲，一个文件应该只是一个字节集合。为了实现顺序存取、随机存取、按键存取、远程存取只能是妨碍你的工作。相同的，如果命令

ls A\*

意味着只列出以 A 为开头的所有文件，那么命令

rm A\*

应该会移除所有以 A 为开头的文件而不是只删除文件名是 A\* 的文件。这个特性也是最小吃惊原则(principle of least surprise)

❝

最小吃惊原则一半常用于用户界面和软件设计。它的原型是：该功能或者特征应该符合用户的预期，不应该使用户感到惊讶和震惊。

❞

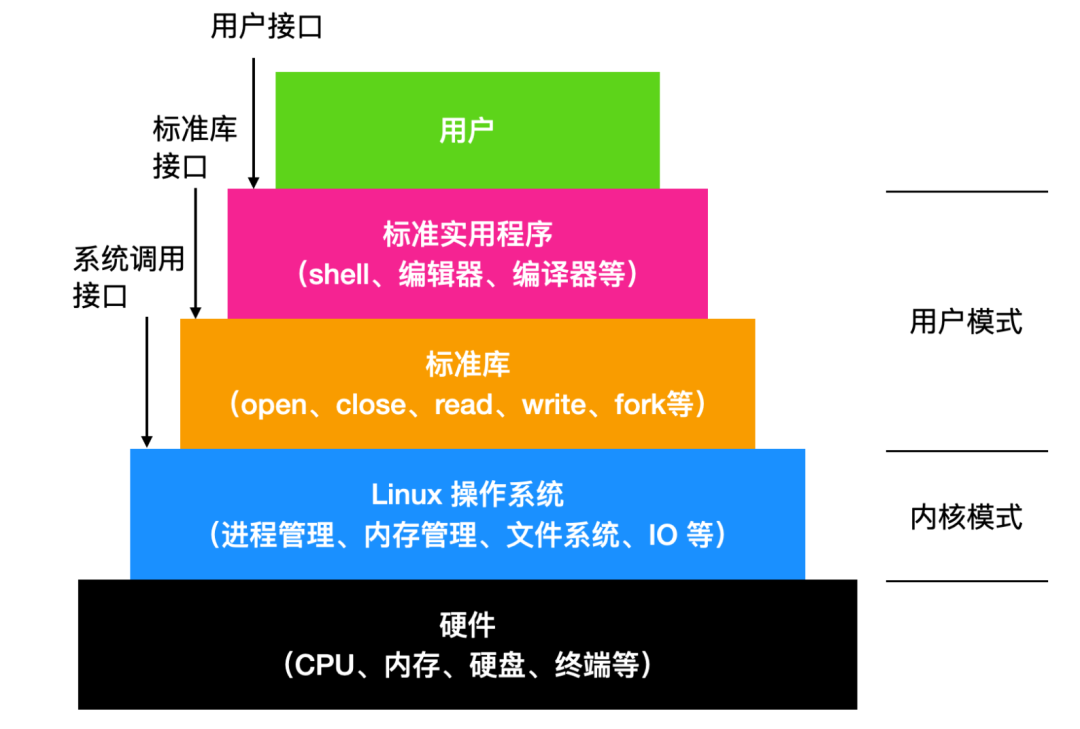
一些有经验的程序员通常希望系统具有较强的功能性和灵活性。设计 Linux 的一个基本目标是每个应用程序只做一件事情并把他做好。所以编译器只负责编译的工作，编译器不会产生列表，因为有其他应用比编译器做的更好。

很多人都不喜欢冗余，为什么在 cp 就能描述清楚你想干什么时候还使用 copy？这完全是在浪费宝贵的 hacking time。为了从文件中提取所有包含字符串 ard的行，Linux 程序员应该输入

grep ard f

Linux 接口

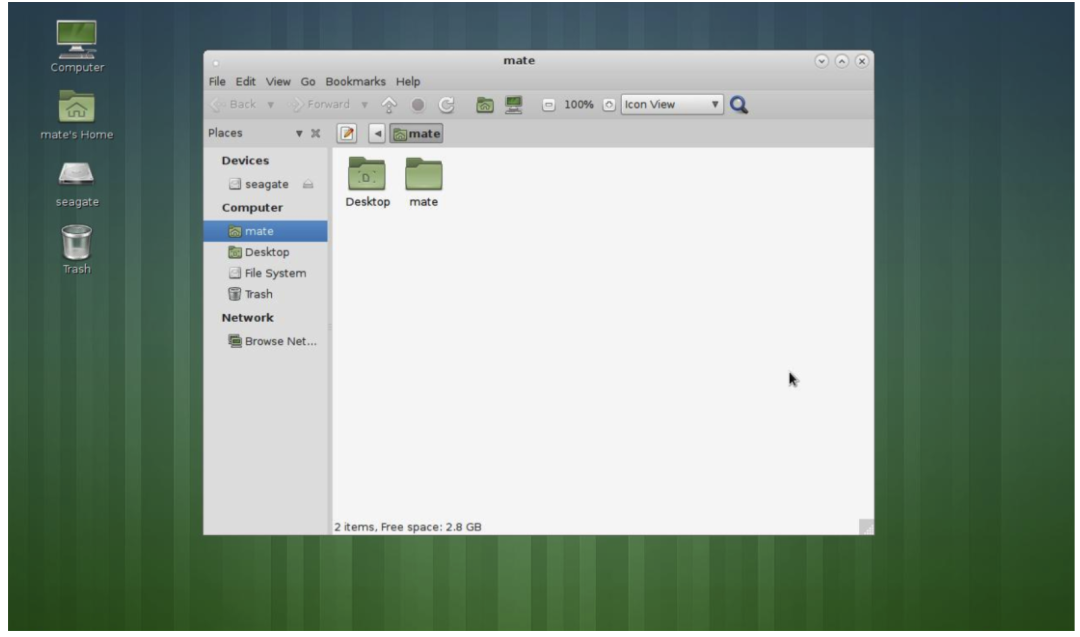
Linux 系统是一种金字塔模型的系统，如下所示



应用程序发起系统调用把参数放在寄存器中(有时候放在栈中)，并发出 trap 系统陷入指令切换用户态至内核态。因为不能直接在 C 中编写 trap 指令，因此 C 提供了一个库，库中的函数对应着系统调用。有些函数是使用汇编编写的，但是能够从 C 中调用。每个函数首先把参数放在合适的位置然后执行系统调用指令。因此如果你想要执行 read 系统调用的话，C 程序会调用 read 函数库来执行。这里顺便提一下，是由 POSIX 指定的库接口而不是系统调用接口。也就是说，POSIX 会告诉一个标准系统应该提供哪些库过程，它们的参数是什么，它们必须做什么以及它们必须返回什么结果。

除了操作系统和系统调用库外，Linux 操作系统还要提供一些标准程序，比如文本编辑器、编译器、文件操作工具等。直接和用户打交道的是上面这些应用程序。因此我们可以说 Linux 具有三种不同的接口：「系统调用接口、库函数接口和应用程序接口」

Linux 中的 GUI(Graphical User Interface) 和 UNIX 中的非常相似，这种 GUI 创建一个桌面环境，包括窗口、目标和文件夹、工具栏和文件拖拽功能。一个完整的 GUI 还包括窗口管理器以及各种应用程序。



Linux 上的 GUI 由 X 窗口支持，主要组成部分是 X 服务器、控制键盘、鼠标、显示器等。当在 Linux 上使用图形界面时，用户可以通过鼠标点击运行程序或者打开文件，通过拖拽将文件进行复制等。

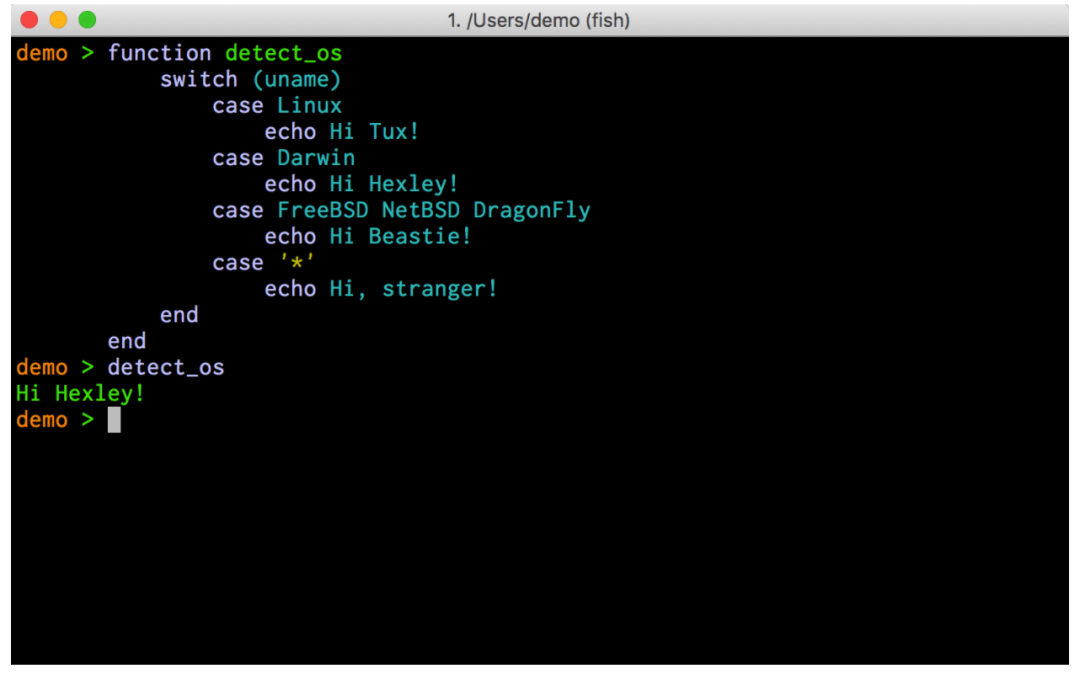
Linux 组成部分

事实上，Linux 操作系统可以由下面这几部分构成

* 引导程序(Bootloader)：引导程序是管理计算机启动过程的软件，对于大多数用户而言，只是弹出一个屏幕，但其实内部操作系统做了很多事情
* 内核(Kernel)：内核是操作系统的核心，负责管理 CPU、内存和外围设备等。
* 初始化系统(Init System)：这是一个引导用户空间并负责控制守护程序的子系统。一旦从引导加载程序移交了初始引导，它就是用于管理引导过程的初始化系统。
* 后台进程(Daemon)：后台进程顾名思义就是在后台运行的程序，比如打印、声音、调度等，它们可以在引导过程中启动，也可以在登录桌面后启动
* 图形服务器(Graphical server)：这是在监视器上显示图形的子系统。通常将其称为 X 服务器或 X。
* 桌面环境(Desktop environment)：这是用户与之实际交互的部分，有很多桌面环境可供选择，每个桌面环境都包含内置应用程序，比如文件管理器、Web 浏览器、游戏等
* 应用程序(Applications)：桌面环境不提供完整的应用程序，就像 Windows 和 macOS 一样，Linux 提供了成千上万个可以轻松找到并安装的高质量软件。

Shell

尽管 Linux 应用程序提供了 GUI ，但是大部分程序员仍偏好于使用命令行(command-line interface)，称为shell。用户通常在 GUI 中启动一个 shell 窗口然后就在 shell 窗口下进行工作。



shell 命令行使用速度快、功能更强大、而且易于扩展、并且不会带来肢体重复性劳损(RSI)。

下面会介绍一些最简单的 bash shell。当 shell 启动时，它首先进行初始化，在屏幕上输出一个 提示符(prompt)，通常是一个百分号或者美元符号，等待用户输入



等用户输入一个命令后，shell 提取其中的第一个词，这里的词指的是被空格或制表符分隔开的一连串字符。假定这个词是将要运行程序的程序名，那么就会搜索这个程序，如果找到了这个程序就会运行它。然后 shell 会将自己挂起直到程序运行完毕，之后再尝试读入下一条指令。shell 也是一个普通的用户程序。它的主要功能就是读取用户的输入和显示计算的输出。shell 命令中可以包含参数，它们作为字符串传递给所调用的程序。比如

cp src dest

会调用 cp 应用程序并包含两个参数 src 和 dest。这个程序会解释第一个参数是一个已经存在的文件名，然后创建一个该文件的副本，名称为 dest。

并不是所有的参数都是文件名，比如下面

head -20 file

第一个参数 -20，会告诉 head 应用程序打印文件的前 20 行，而不是默认的 10 行。控制命令操作或者指定可选值的参数称为标志(flag)，按照惯例标志应该使用 - 来表示。这个符号是必要的，比如

head 20 file

是一个完全合法的命令，它会告诉 head 程序输出文件名为 20 的文件的前 10 行，然后输出文件名为 file 文件的前 10 行。Linux 操作系统可以接受一个或多个参数。

为了更容易的指定多个文件名，shell 支持 魔法字符(magic character)，也被称为通配符(wild cards)。比如，\* 可以匹配一个或者多个可能的字符串

ls \*.c

告诉 ls 列举出所有文件名以 .c 结束的文件。如果同时存在多个文件，则会在后面进行并列。

另一个通配符是问号，负责匹配任意一个字符。一组在中括号中的字符可以表示其中任意一个，因此

ls [abc]\*

会列举出所有以 a、b 或者 c 开头的文件。

shell 应用程序不一定通过终端进行输入和输出。shell 启动时，就会获取 「标准输入、标准输出、标准错误」文件进行访问的能力。

标准输出是从键盘输入的，标准输出或者标准错误是输出到显示器的。许多 Linux 程序默认是从标准输入进行输入并从标准输出进行输出。比如

sort

会调用 sort 程序，会从终端读取数据(直到用户输入 ctrl-d 结束)，根据字母顺序进行排序，然后将结果输出到屏幕上。

通常还可以重定向标准输入和标准输出，重定向标准输入使用 < 后面跟文件名。标准输出可以通过一个大于号 > 进行重定向。允许一个命令中重定向标准输入和输出。例如命令

sort <in >out

会使 sort 从文件 in 中得到输入，并把结果输出到 out 文件中。由于标准错误没有重定向，所以错误信息会直接打印到屏幕上。从标准输入读入，对其进行处理并将其写入到标准输出的程序称为 过滤器。

考虑下面由三个分开的命令组成的指令

sort <in >temp;head -30 <temp;rm temp

首先会调用 sort 应用程序，从标准输入 in 中进行读取，并通过标准输出到 temp。当程序运行完毕后，shell 会运行 head ，告诉它打印前 30 行，并在标准输出(默认为终端)上打印。最后，temp 临时文件被删除。「轻轻的，你走了，你挥一挥衣袖，不带走一片云彩」。

命令行中的第一个程序通常会产生输出，在上面的例子中，产生的输出都不 temp 文件接收。然而，Linux 还提供了一个简单的命令来做这件事，例如下面

sort <in | head -30

上面 | 称为竖线符号，它的意思是从 sort 应用程序产生的排序输出会直接作为输入显示，无需创建、使用和移除临时文件。由管道符号连接的命令集合称为管道(pipeline)。例如如下

grep cxuan \*.c | sort | head -30 | tail -5 >f00

对任意以 .c 结尾的文件中包含 cxuan 的行被写到标准输出中，然后进行排序。这些内容中的前 30 行被 head 出来并传给 tail ，它又将最后 5 行传递给 foo。这个例子提供了一个管道将多个命令连接起来。

可以把一系列 shell 命令放在一个文件中，然后将此文件作为输入来运行。shell 会按照顺序对他们进行处理，就像在键盘上键入命令一样。包含 shell 命令的文件被称为 shell 脚本(shell scripts)。

❝

推荐一个 shell 命令的学习网站：https://www.shellscript.sh/

❞

shell 脚本其实也是一段程序，shell 脚本中可以对变量进行赋值，也包含循环控制语句比如 「if、for、while」 等，shell 的设计目标是让其看起来和 C 相似(There is no doubt that C is father)。由于 shell 也是一个用户程序，所以用户可以选择不同的 shell。

Linux 应用程序

Linux 的命令行也就是 shell，它由大量标准应用程序组成。这些应用程序主要有下面六种

* 文件和目录操作命令
* 过滤器
* 文本程序
* 系统管理
* 程序开发工具，例如编辑器和编译器
* 其他

除了这些标准应用程序外，还有其他应用程序比如 「Web 浏览器、多媒体播放器、图片浏览器、办公软件和游戏程序等」。

我们在上面的例子中已经见过了几个 Linux 的应用程序，比如 sort、cp、ls、head，下面我们再来认识一下其他 Linux 的应用程序。

我们先从几个例子开始讲起，比如

cp a b

是将 a 复制一个副本为 b ，而

mv a b

是将 a 移动到 b ，但是删除原文件。

上面这两个命令有一些区别，cp 是将文件进行复制，复制完成后会有两个文件 a 和 b；而 mv 相当于是文件的移动，移动完成后就不再有 a 文件。cat 命令可以把多个文件内容进行连接。使用 rm 可以删除文件；使用 chmod 可以允许所有者改变访问权限；文件目录的的创建和删除可以使用 mkdir 和 rmdir 命令；使用 ls 可以查看目录文件，ls 可以显示很多属性，比如大小、用户、创建日期等；sort 决定文件的显示顺序

Linux 应用程序还包括过滤器 grep，grep 从标准输入或者一个或多个输入文件中提取特定模式的行；sort 将输入进行排序并输出到标准输出；head 提取输入的前几行；tail 提取输入的后面几行；除此之外的过滤器还有 cut 和 paste，允许对文本行的剪切和复制；od 将输入转换为 ASCII ；tr 实现字符大小写转换；pr 为格式化打印输出等。

程序编译工具使用 gcc；

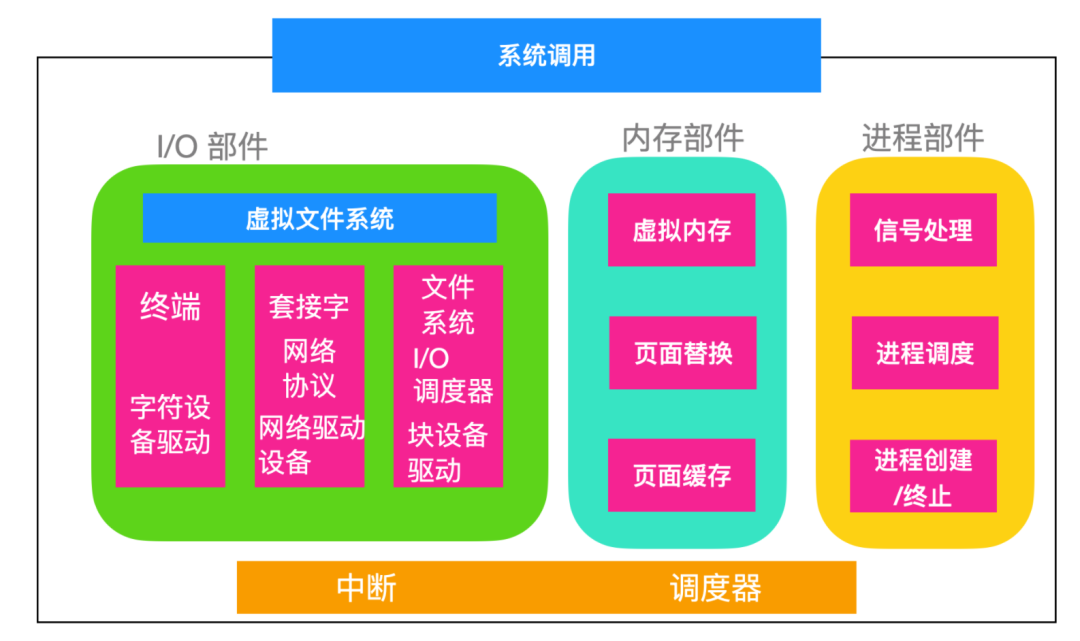
make 命令用于自动编译，这是一个很强大的命令，它用于维护一个大的程序，往往这类程序的源码由许多文件构成。典型的，有一些是 header files 头文件，源文件通常使用 include 指令包含这些文件，make 的作用就是跟踪哪些文件属于头文件，然后安排自动编译的过程。

下面列出了 POSIX 的标准应用程序

| 程序 | 应用 |
| --- | --- |
| ls | 列出目录 |
| cp | 复制文件 |
| head | 显示文件的前几行 |
| make | 编译文件生成二进制文件 |
| cd | 切换目录 |
| mkdir | 创建目录 |
| chmod | 修改文件访问权限 |
| ps | 列出文件进程 |
| pr | 格式化打印 |
| rm | 删除一个文件 |
| rmdir | 删除文件目录 |
| tail | 提取文件最后几行 |
| tr | 字符集转换 |
| grep | 分组 |
| cat | 将多个文件连续标准输出 |
| od | 以八进制显示文件 |
| cut | 从文件中剪切 |
| paste | 从文件中粘贴 |

Linux 内核结构

在上面我们看到了 Linux 的整体结构，下面我们从整体的角度来看一下 Linux 的内核结构



内核直接坐落在硬件上，内核的主要作用就是 I/O 交互、内存管理和控制 CPU 访问。上图中还包括了 中断 和 调度器，中断是与设备交互的主要方式。中断出现时调度器就会发挥作用。这里的低级代码停止正在运行的进程，将其状态保存在内核进程结构中，并启动驱动程序。进程调度也会发生在内核完成一些操作并且启动用户进程的时候。图中的调度器是 dispatcher。

❝

注意这里的调度器是 dispatcher 而不是 scheduler，这两者是有区别的

scheduler 和 dispatcher 都是和进程调度相关的概念，不同的是 scheduler 会从几个进程中随意选取一个进程；而 dispatcher 会给 scheduler 选择的进程分配 CPU。

❞

然后，我们把内核系统分为三部分。

* I/O 部分负责与设备进行交互以及执行网络和存储 I/O 操作的所有内核部分。

从图中可以看出 I/O 层次的关系，最高层是一个虚拟文件系统，也就是说不管文件是来自内存还是磁盘中，都是经过虚拟文件系统中的。从底层看，所有的驱动都是字符驱动或者块设备驱动。二者的主要区别就是是否允许随机访问。网络驱动设备并不是一种独立的驱动设备，它实际上是一种字符设备，不过网络设备的处理方式和字符设备不同。

上面的设备驱动程序中，每个设备类型的内核代码都不同。字符设备有两种使用方式，有一键式的比如 vi 或者 emacs ，需要每一个键盘输入。其他的比如 shell ，是需要输入一行按回车键将字符串发送给程序进行编辑。

网络软件通常是模块化的，由不同的设备和协议来支持。大多数 Linux 系统在内核中包含一个完整的硬件路由器的功能，但是这个不能和外部路由器相比，路由器上面是协议栈，包括 TCP/IP 协议，协议栈上面是 socket 接口，socket 负责与外部进行通信，充当了门的作用。

磁盘驱动上面是 I/O 调度器，它负责排序和分配磁盘读写操作，以尽可能减少磁头的无用移动。

* I/O 右边的是内存部件，程序被装载进内存，由 CPU 执行，这里会涉及到虚拟内存的部件，页面的换入和换出是如何进行的，坏页面的替换和经常使用的页面会进行缓存。
* 进程模块负责进程的创建和终止、进程的调度、Linux 把进程和线程看作是可运行的实体，并使用统一的调度策略来进行调度。

在内核最顶层的是系统调用接口，所有的系统调用都是经过这里，系统调用会触发一个 trap，将系统从用户态转换为内核态，然后将控制权移交给上面的内核部件。

Linux 进程和线程

下面我们就深入理解一下 Linux 内核来理解 Linux 的基本概念之进程和线程。系统调用是操作系统本身的接口，它对于创建进程和线程，内存分配，共享文件和 I/O 来说都很重要。

我们将从各个版本的共性出发来进行探讨。

基本概念

每个进程都会运行一段独立的程序，并且在初始化的时候拥有一个独立的控制线程。换句话说，每个进程都会有一个自己的程序计数器，这个程序计数器用来记录下一个需要被执行的指令。Linux 允许进程在运行时创建额外的线程。

Linux 是一个多道程序设计系统，因此系统中存在彼此相互独立的进程同时运行。此外，每个用户都会同时有几个活动的进程。因为如果是一个大型系统，可能有数百上千的进程在同时运行。

在某些用户空间中，即使用户退出登录，仍然会有一些后台进程在运行，这些进程被称为 守护进程(daemon)。

Linux 中有一种特殊的守护进程被称为 计划守护进程(Cron daemon) ，计划守护进程可以每分钟醒来一次检查是否有工作要做，做完会继续回到睡眠状态等待下一次唤醒。

❝

Cron 是一个守护程序，可以做任何你想做的事情，比如说你可以定期进行系统维护、定期进行系统备份等。在其他操作系统上也有类似的程序，比如 Mac OS X 上 Cron 守护程序被称为 launchd 的守护进程。在 Windows 上可以被称为 计划任务(Task Scheduler)。

❞

在 Linux 系统中，进程通过非常简单的方式来创建，fork 系统调用会创建一个源进程的拷贝(副本)。调用 fork 函数的进程被称为 父进程(parent process)，使用 fork 函数创建出来的进程被称为 子进程(child process)。父进程和子进程都有自己的内存映像。如果在子进程创建出来后，父进程修改了一些变量等，那么子进程是看不到这些变化的，也就是 fork 后，父进程和子进程相互独立。

虽然父进程和子进程保持相互独立，但是它们却能够共享相同的文件，如果在 fork 之前，父进程已经打开了某个文件，那么 fork 后，父进程和子进程仍然共享这个打开的文件。对共享文件的修改会对父进程和子进程同时可见。

那么该如何区分父进程和子进程呢？子进程只是父进程的拷贝，所以它们几乎所有的情况都一样，包括内存映像、变量、寄存器等。区分的关键在于 fork 函数调用后的返回值，如果 fork 后返回一个非零值，这个非零值即是子进程的 进程标识符(Process Identiier, PID)，而会给子进程返回一个零值，可以用下面代码来进行表示

pid = fork();    // 调用 fork 函数创建进程  
if(pid < 0){  
  error()     // pid < 0,创建失败  
}  
else if(pid > 0){  
  parent\_handle() // 父进程代码  
}  
else {  
  child\_handle()  // 子进程代码  
}

父进程在 fork 后会得到子进程的 PID，这个 PID 即能代表这个子进程的唯一标识符也就是 PID。如果子进程想要知道自己的 PID，可以调用 getpid 方法。当子进程结束运行时，父进程会得到子进程的 PID，因为一个进程会 fork 很多子进程，子进程也会 fork 子进程，所以 PID 是非常重要的。我们把第一次调用 fork 后的进程称为 原始进程，一个原始进程可以生成一颗继承树

Linux 进程间通信

Linux 进程间的通信机制通常被称为 Internel-Process communication,IPC下面我们来说一说 Linux 进程间通信的机制，大致来说，Linux 进程间的通信机制可以分为 6 种

下面我们分别对其进行概述

信号 signal

信号是 UNIX 系统最先开始使用的进程间通信机制，因为 Linux 是继承于 UNIX 的，所以 Linux 也支持信号机制，通过向一个或多个进程发送异步事件信号来实现，信号可以从键盘或者访问不存在的位置等地方产生；信号通过 shell 将任务发送给子进程。

你可以在 Linux 系统上输入 kill -l 来列出系统使用的信号，下面是我提供的一些信号

进程可以选择忽略发送过来的信号，但是有两个是不能忽略的：SIGSTOP 和 SIGKILL 信号。SIGSTOP 信号会通知当前正在运行的进程执行关闭操作，SIGKILL 信号会通知当前进程应该被杀死。除此之外，进程可以选择它想要处理的信号，进程也可以选择阻止信号，如果不阻止，可以选择自行处理，也可以选择进行内核处理。如果选择交给内核进行处理，那么就执行默认处理。

操作系统会中断目标程序的进程来向其发送信号、在任何非原子指令中，执行都可以中断，如果进程已经注册了新号处理程序，那么就执行进程，如果没有注册，将采用默认处理的方式。

例如：当进程收到 SIGFPE 浮点异常的信号后，默认操作是对其进行 dump(转储)和退出。信号没有优先级的说法。如果同时为某个进程产生了两个信号，则可以将它们呈现给进程或者以任意的顺序进行处理。

下面我们就来看一下这些信号是干什么用的

* SIGABRT 和 SIGIOT

SIGABRT 和 SIGIOT 信号发送给进程，告诉其进行终止，这个 信号通常在调用 C标准库的abort()函数时由进程本身启动

* SIGALRM 、 SIGVTALRM、SIGPROF

当设置的时钟功能超时时会将 SIGALRM 、 SIGVTALRM、SIGPROF 发送给进程。当实际时间或时钟时间超时时，发送 SIGALRM。当进程使用的 CPU 时间超时时，将发送 SIGVTALRM。当进程和系统代表进程使用的CPU 时间超时时，将发送 SIGPROF。

* SIGBUS

SIGBUS 将造成总线中断错误时发送给进程

* SIGCHLD

当子进程终止、被中断或者被中断恢复，将 SIGCHLD 发送给进程。此信号的一种常见用法是指示操作系统在子进程终止后清除其使用的资源。

* SIGCONT

SIGCONT 信号指示操作系统继续执行先前由 SIGSTOP 或 SIGTSTP 信号暂停的进程。该信号的一个重要用途是在 Unix shell 中的作业控制中。

* SIGFPE

SIGFPE 信号在执行错误的算术运算（例如除以零）时将被发送到进程。

* SIGUP

当 SIGUP 信号控制的终端关闭时，会发送给进程。许多守护程序将重新加载其配置文件并重新打开其日志文件，而不是在收到此信号时退出。

* SIGILL

SIGILL 信号在尝试执行非法、格式错误、未知或者特权指令时发出

* SIGINT

当用户希望中断进程时，操作系统会向进程发送 SIGINT 信号。用户输入 ctrl - c 就是希望中断进程。

* SIGKILL

SIGKILL 信号发送到进程以使其马上进行终止。与 SIGTERM 和 SIGINT 相比，这个信号无法捕获和忽略执行，并且进程在接收到此信号后无法执行任何清理操作，下面是一些例外情况

僵尸进程无法杀死，因为僵尸进程已经死了，它在等待父进程对其进行捕获

处于阻塞状态的进程只有再次唤醒后才会被 kill 掉

init 进程是 Linux 的初始化进程，这个进程会忽略任何信号。

SIGKILL 通常是作为最后杀死进程的信号、它通常作用于 SIGTERM 没有响应时发送给进程。

* SIGPIPE

SIGPIPE 尝试写入进程管道时发现管道未连接无法写入时发送到进程

* SIGPOLL

当在明确监视的文件描述符上发生事件时，将发送 SIGPOLL 信号。

* SIGRTMIN 至 SIGRTMAX

SIGRTMIN 至 SIGRTMAX 是实时信号

* SIGQUIT

当用户请求退出进程并执行核心转储时，SIGQUIT 信号将由其控制终端发送给进程。

* SIGSEGV

当 SIGSEGV 信号做出无效的虚拟内存引用或分段错误时，即在执行分段违规时，将其发送到进程。

* SIGSTOP

SIGSTOP 指示操作系统终止以便以后进行恢复时

* SIGSYS

当 SIGSYS 信号将错误参数传递给系统调用时，该信号将发送到进程。

* SYSTERM

我们上面简单提到过了 SYSTERM 这个名词，这个信号发送给进程以请求终止。与 SIGKILL 信号不同，该信号可以被过程捕获或忽略。这允许进程执行良好的终止，从而释放资源并在适当时保存状态。SIGINT 与SIGTERM 几乎相同。

* SIGTSIP

SIGTSTP 信号由其控制终端发送到进程，以请求终端停止。

* SIGTTIN 和 SIGTTOU

当 SIGTTIN 和SIGTTOU 信号分别在后台尝试从 tty 读取或写入时，信号将发送到该进程。

* SIGTRAP

在发生异常或者 trap 时，将 SIGTRAP 信号发送到进程

* SIGURG

当套接字具有可读取的紧急或带外数据时，将 SIGURG 信号发送到进程。

* SIGUSR1 和 SIGUSR2

SIGUSR1 和 SIGUSR2 信号被发送到进程以指示用户定义的条件。

* SIGXCPU

当 SIGXCPU 信号耗尽 CPU 的时间超过某个用户可设置的预定值时，将其发送到进程

* SIGXFSZ

当 SIGXFSZ 信号增长超过最大允许大小的文件时，该信号将发送到该进程。

* SIGWINCH

SIGWINCH 信号在其控制终端更改其大小（窗口更改）时发送给进程。

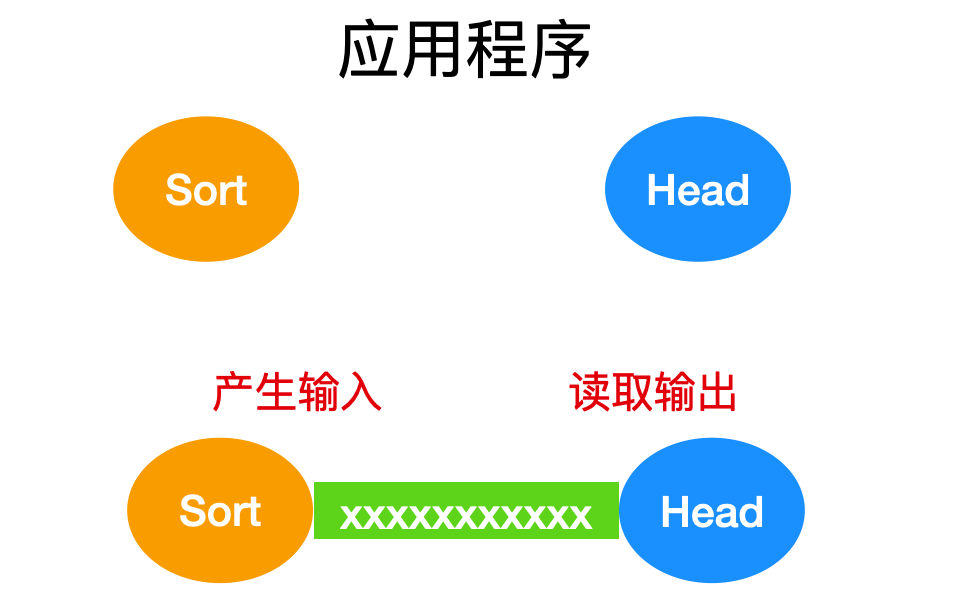
管道 pipe

Linux 系统中的进程可以通过建立管道 pipe 进行通信。

在两个进程之间，可以建立一个通道，一个进程向这个通道里写入字节流，另一个进程从这个管道中读取字节流。管道是同步的，当进程尝试从空管道读取数据时，该进程会被阻塞，直到有可用数据为止。shell 中的管线 pipelines 就是用管道实现的，当 shell 发现输出

sort <f | head

它会创建两个进程，一个是 sort，一个是 head，sort，会在这两个应用程序之间建立一个管道使得 sort 进程的标准输出作为 head 程序的标准输入。sort 进程产生的输出就不用写到文件中了，如果管道满了系统会停止 sort 以等待 head 读出数据



管道实际上就是 |，两个应用程序不知道有管道的存在，一切都是由 shell 管理和控制的。

共享内存 shared memory

两个进程之间还可以通过共享内存进行进程间通信，其中两个或者多个进程可以访问公共内存空间。两个进程的共享工作是通过共享内存完成的，一个进程所作的修改可以对另一个进程可见(很像线程间的通信)。

在使用共享内存前，需要经过一系列的调用流程，流程如下

* 创建共享内存段或者使用已创建的共享内存段(shmget())
* 将进程附加到已经创建的内存段中(shmat())
* 从已连接的共享内存段分离进程(shmdt())
* 对共享内存段执行控制操作(shmctl())

先入先出队列 FIFO

先入先出队列 FIFO 通常被称为 命名管道(Named Pipes)，命名管道的工作方式与常规管道非常相似，但是确实有一些明显的区别。未命名的管道没有备份文件：操作系统负责维护内存中的缓冲区，用来将字节从写入器传输到读取器。一旦写入或者输出终止的话，缓冲区将被回收，传输的数据会丢失。相比之下，命名管道具有支持文件和独特 API ，命名管道在文件系统中作为设备的专用文件存在。当所有的进程通信完成后，命名管道将保留在文件系统中以备后用。命名管道具有严格的 FIFO 行为

写入的第一个字节是读取的第一个字节，写入的第二个字节是读取的第二个字节，依此类推。

消息队列 Message Queue

一听到消息队列这个名词你可能不知道是什么意思，消息队列是用来描述内核寻址空间内的内部链接列表。可以按几种不同的方式将消息按顺序发送到队列并从队列中检索消息。每个消息队列由 IPC 标识符唯一标识。消息队列有两种模式，一种是严格模式， 严格模式就像是 FIFO 先入先出队列似的，消息顺序发送，顺序读取。还有一种模式是 非严格模式，消息的顺序性不是非常重要。

套接字 Socket

还有一种管理两个进程间通信的是使用 socket，socket 提供端到端的双相通信。一个套接字可以与一个或多个进程关联。就像管道有命令管道和未命名管道一样，套接字也有两种模式，套接字一般用于两个进程之间的网络通信，网络套接字需要来自诸如TCP（传输控制协议）或较低级别UDP（用户数据报协议）等基础协议的支持。

套接字有以下几种分类

* 顺序包套接字(Sequential Packet Socket)：此类套接字为最大长度固定的数据报提供可靠的连接。此连接是双向的并且是顺序的。
* 数据报套接字(Datagram Socket)：数据包套接字支持双向数据流。数据包套接字接受消息的顺序与发送者可能不同。
* 流式套接字(Stream Socket)：流套接字的工作方式类似于电话对话，提供双向可靠的数据流。
* 原始套接字(Raw Socket)：可以使用原始套接字访问基础通信协议。

Linux 中进程管理系统调用

现在关注一下 Linux 系统中与进程管理相关的系统调用。在了解之前你需要先知道一下什么是系统调用。

操作系统为我们屏蔽了硬件和软件的差异，它的最主要功能就是为用户提供一种抽象，隐藏内部实现，让用户只关心在 GUI 图形界面下如何使用即可。操作系统可以分为两种模式

* 内核态：操作系统内核使用的模式
* 用户态：用户应用程序所使用的模式

我们常说的上下文切换 指的就是内核态模式和用户态模式的频繁切换。而系统调用指的就是引起内核态和用户态切换的一种方式，系统调用通常在后台静默运行，表示计算机程序向其操作系统内核请求服务。

系统调用指令有很多，下面是一些与进程管理相关的最主要的系统调用

fork

fork 调用用于创建一个与父进程相同的子进程，创建完进程后的子进程拥有和父进程一样的程序计数器、相同的 CPU 寄存器、相同的打开文件。

exec

exec 系统调用用于执行驻留在活动进程中的文件，调用 exec 后，新的可执行文件会替换先前的可执行文件并获得执行。也就是说，调用 exec 后，会将旧文件或程序替换为新文件或执行，然后执行文件或程序。新的执行程序被加载到相同的执行空间中，因此进程的 PID不会修改，因为我们「没有创建新进程，只是替换旧进程」。但是进程的数据、代码、堆栈都已经被修改。如果当前要被替换的进程包含多个线程，那么所有的线程将被终止，新的进程映像被加载执行。

这里需要解释一下进程映像(Process image) 的概念

「什么是进程映像呢」？进程映像是执行程序时所需要的可执行文件，通常会包括下面这些东西

* 「代码段（codesegment/textsegment）」

又称文本段，用来存放指令，运行代码的一块内存空间

此空间大小在代码运行前就已经确定

内存空间一般属于只读，某些架构的代码也允许可写

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

* 「数据段（datasegment）」

可读可写

存储初始化的全局变量和初始化的 static 变量

数据段中数据的生存期是随程序持续性（随进程持续性） 随进程持续性：进程创建就存在，进程死亡就消失

* 「bss 段（bsssegment）：」

可读可写

存储未初始化的全局变量和未初始化的 static 变量

bss 段中的数据一般默认为 0

* 「Data 段」

是可读写的，因为变量的值可以在运行时更改。此段的大小也固定。

* 「栈（stack）：」

可读可写

存储的是函数或代码中的局部变量(非 static 变量)

栈的生存期随代码块持续性，代码块运行就给你分配空间，代码块结束，就自动回收空间

* 「堆（heap）：」

可读可写

存储的是程序运行期间动态分配的 malloc/realloc 的空间

堆的生存期随进程持续性，从 malloc/realloc 到 free 一直存在

下面是这些区域的构成图

exec 系统调用是一些函数的集合，这些函数是

* execl
* execle
* execlp
* execv
* execve
* execvp

下面来看一下 exec 的工作原理

1. 当前进程映像被替换为新的进程映像
2. 新的进程映像是你做为 exec 传递的灿睡
3. 结束当前正在运行的进程
4. 新的进程映像有 PID，相同的环境和一些文件描述符(因为未替换进程，只是替换了进程映像)
5. CPU 状态和虚拟内存受到影响，当前进程映像的虚拟内存映射被新进程映像的虚拟内存代替。

waitpid

等待子进程结束或终止

exit

在许多计算机操作系统上，计算机进程的终止是通过执行 exit 系统调用命令执行的。0 表示进程能够正常结束，其他值表示进程以非正常的行为结束。

其他一些常见的系统调用如下

| 系统调用指令 | 描述 |
| --- | --- |
| pause | 挂起信号 |
| nice | 改变分时进程的优先级 |
| ptrace | 进程跟踪 |
| kill | 向进程发送信号 |
| pipe | 创建管道 |
| mkfifo | 创建 fifo 的特殊文件（命名管道） |
| sigaction | 设置对指定信号的处理方法 |
| msgctl | 消息控制操作 |
| semctl | 信号量控制 |

Linux 进程和线程的实现

Linux 进程

在 Linux 内核结构中，进程会被表示为 任务，通过结构体 structure 来创建。不像其他的操作系统会区分进程、轻量级进程和线程，Linux 统一使用任务结构来代表执行上下文。因此，对于每个单线程进程来说，单线程进程将用一个任务结构表示，对于多线程进程来说，将为每一个用户级线程分配一个任务结构。Linux 内核是多线程的，并且内核级线程不与任何用户级线程相关联。

对于每个进程来说，在内存中都会有一个 task\_struct 进程描述符与之对应。进程描述符包含了内核管理进程所有有用的信息，包括 「调度参数、打开文件描述符等等」。进程描述符从进程创建开始就一直存在于内核堆栈中。

Linux 和 Unix 一样，都是通过 PID 来区分不同的进程，内核会将所有进程的任务结构组成为一个双向链表。PID 能够直接被映射称为进程的任务结构所在的地址，从而不需要遍历双向链表直接访问。

我们上面提到了进程描述符，这是一个非常重要的概念，我们上面还提到了进程描述符是位于内存中的，这里我们省略了一句话，那就是进程描述符是存在用户的任务结构中，当进程位于内存并开始运行时，进程描述符才会被调入内存。

❝

进程位于内存被称为 PIM(Process In Memory) ，这是冯诺伊曼体系架构的一种体现，加载到内存中并执行的程序称为进程。简单来说，一个进程就是正在执行的程序。

❞

进程描述符可以归为下面这几类

* 调度参数(scheduling parameters)：进程优先级、最近消耗 CPU 的时间、最近睡眠时间一起决定了下一个需要运行的进程
* 内存映像(memory image)：我们上面说到，进程映像是执行程序时所需要的可执行文件，它由数据和代码组成。
* 信号(signals)：显示哪些信号被捕获、哪些信号被执行
* 寄存器：当发生内核陷入 (trap) 时，寄存器的内容会被保存下来。
* 系统调用状态(system call state)：当前系统调用的信息，包括参数和结果
* 文件描述符表(file descriptor table)：有关文件描述符的系统被调用时，文件描述符作为索引在文件描述符表中定位相关文件的 i-node 数据结构
* 统计数据(accounting)：记录用户、进程占用系统 CPU 时间表的指针，一些操作系统还保存进程最多占用的 CPU 时间、进程拥有的最大堆栈空间、进程可以消耗的页面数等。
* 内核堆栈(kernel stack)：进程的内核部分可以使用的固定堆栈
* 其他：当前进程状态、事件等待时间、距离警报的超时时间、PID、父进程的 PID 以及用户标识符等

有了上面这些信息，现在就很容易描述在 Linux 中是如何创建这些进程的了，创建新流程实际上非常简单。「为子进程开辟一块新的用户空间的进程描述符，然后从父进程复制大量的内容。为这个子进程分配一个 PID，设置其内存映射，赋予它访问父进程文件的权限，注册并启动」。

当执行 fork 系统调用时，调用进程会陷入内核并创建一些和任务相关的数据结构，比如内核堆栈(kernel stack) 和 thread\_info 结构。

❝

关于 thread\_info 结构可以参考

https://docs.huihoo.com/doxygen/linux/kernel/3.7/arch\_2avr32\_2include\_2asm\_2thread\_\_info\_8h\_source.html

❞

这个结构中包含进程描述符，进程描述符位于固定的位置，使得 Linux 系统只需要很小的开销就可以定位到一个运行中进程的数据结构。

进程描述符的主要内容是根据父进程的描述符来填充。Linux 操作系统会寻找一个可用的 PID，并且此 PID 没有被任何进程使用，更新进程标示符使其指向一个新的数据结构即可。为了减少 hash table 的碰撞，进程描述符会形成链表。它还将 task\_struct 的字段设置为指向任务数组上相应的上一个/下一个进程。

❝

task\_struct ：Linux 进程描述符，内部涉及到众多 C++ 源码，我们会在后面进行讲解。

❞

从原则上来说，为子进程开辟内存区域并为子进程分配数据段、堆栈段，并且对父进程的内容进行复制，但是实际上 fork 完成后，子进程和父进程没有共享内存，所以需要复制技术来实现同步，但是复制开销比较大，因此 Linux 操作系统使用了一种 欺骗 方式。即为子进程分配页表，然后新分配的页表指向父进程的页面，同时这些页面是只读的。当进程向这些页面进行写入的时候，会开启保护错误。内核发现写入操作后，会为进程分配一个副本，使得写入时把数据复制到这个副本上，这个副本是共享的，这种方式称为 写入时复制(copy on write)，这种方式避免了在同一块内存区域维护两个副本的必要，节省内存空间。

在子进程开始运行后，操作系统会调用 exec 系统调用，内核会进行查找验证可执行文件，把参数和环境变量复制到内核，释放旧的地址空间。

现在新的地址空间需要被创建和填充。如果系统支持映射文件，就像 Unix 系统一样，那么新的页表就会创建，表明内存中没有任何页，除非所使用的页面是堆栈页，其地址空间由磁盘上的可执行文件支持。新进程开始运行时，立刻会收到一个缺页异常(page fault)，这会使具有代码的页面加载进入内存。最后，参数和环境变量被复制到新的堆栈中，重置信号，寄存器全部清零。新的命令开始运行。

下面是一个示例，用户输出 ls，shell 会调用 fork 函数复制一个新进程，shell 进程会调用 exec 函数用可执行文件 ls 的内容覆盖它的内存。

Linux 线程

现在我们来讨论一下 Linux 中的线程，线程是轻量级的进程，想必这句话你已经听过很多次了，轻量级体现在所有的进程切换都需要清除所有的表、进程间的共享信息也比较麻烦，一般来说通过管道或者共享内存，如果是 fork 函数后的父子进程则使用共享文件，然而线程切换不需要像进程一样具有昂贵的开销，而且线程通信起来也更方便。线程分为两种：用户级线程和内核级线程

用户级线程

用户级线程避免使用内核，通常，每个线程会显示调用开关，发送信号或者执行某种切换操作来放弃 CPU，同样，计时器可以强制进行开关，用户线程的切换速度通常比内核线程快很多。在用户级别实现线程会有一个问题，即单个线程可能会垄断 CPU 时间片，导致其他线程无法执行从而 饿死。如果执行一个 I/O 操作，那么 I/O 会阻塞，其他线程也无法运行。

一种解决方案是，一些用户级的线程包解决了这个问题。可以使用时钟周期的监视器来控制第一时间时间片独占。然后，一些库通过特殊的包装来解决系统调用的 I/O 阻塞问题，或者可以为非阻塞 I/O 编写任务。

内核级线程

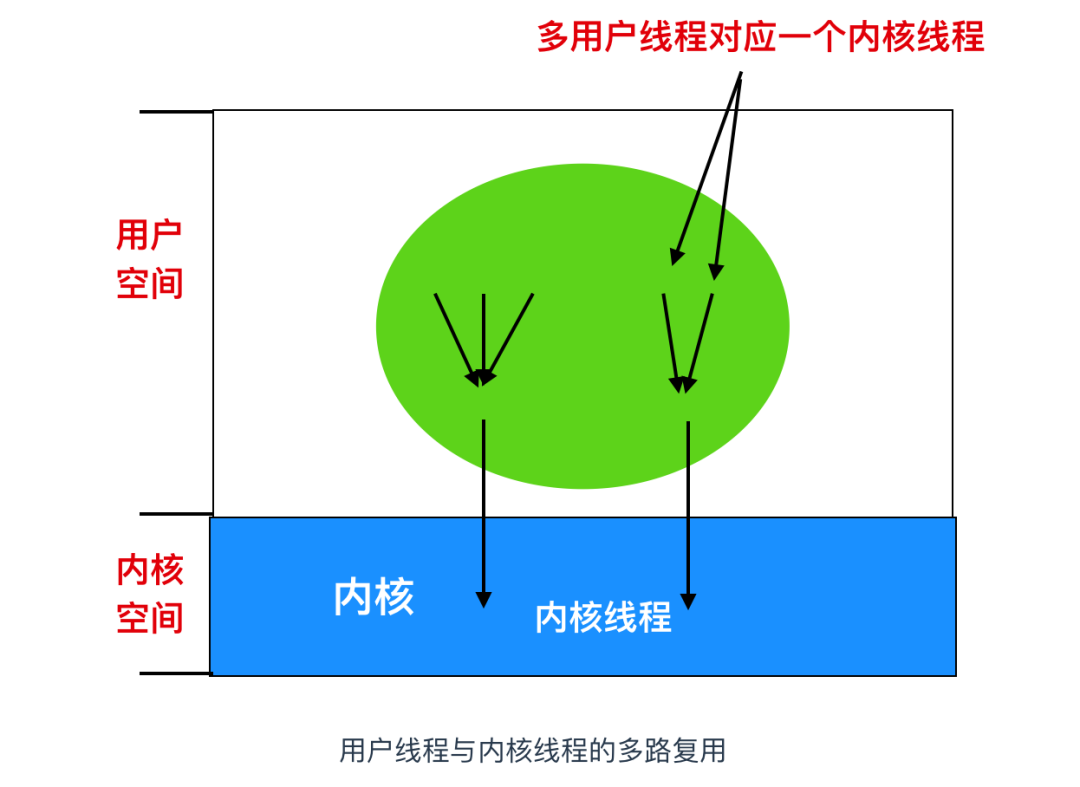
内核级线程通常使用几个进程表在内核中实现，每个任务都会对应一个进程表。在这种情况下，内核会在每个进程的时间片内调度每个线程。

所有能够阻塞的调用都会通过系统调用的方式来实现，当一个线程阻塞时，内核可以进行选择，是运行在同一个进程中的另一个线程（如果有就绪线程的话）还是运行一个另一个进程中的线程。

从用户空间 -> 内核空间 -> 用户空间的开销比较大，但是线程初始化的时间损耗可以忽略不计。这种实现的好处是由时钟决定线程切换时间，因此不太可能将时间片与任务中的其他线程占用时间绑定到一起。同样，I/O 阻塞也不是问题。

混合实现

结合用户空间和内核空间的优点，设计人员采用了一种内核级线程的方式，然后将用户级线程与某些或者全部内核线程多路复用起来



在这种模型中，编程人员可以自由控制用户线程和内核线程的数量，具有很大的灵活度。采用这种方法，内核只识别内核级线程，并对其进行调度。其中一些内核级线程会被多个用户级线程多路复用。

Linux 调度

下面我们来关注一下 Linux 系统的调度算法，首先需要认识到，Linux 系统的线程是内核线程，所以 Linux 系统是基于线程的，而不是基于进程的。

为了进行调度，Linux 系统将线程分为三类

* 实时先入先出
* 实时轮询
* 分时

实时先入先出线程具有最高优先级，它不会被其他线程所抢占，除非那是一个刚刚准备好的，拥有更高优先级的线程进入。实时轮转线程与实时先入先出线程基本相同，只是每个实时轮转线程都有一个时间量，时间到了之后就可以被抢占。如果多个实时线程准备完毕，那么每个线程运行它时间量所规定的时间，然后插入到实时轮转线程末尾。

❝

注意这个实时只是相对的，无法做到绝对的实时，因为线程的运行时间无法确定。它们相对分时系统来说，更加具有实时性

❞

Linux 系统会给每个线程分配一个 nice 值，这个值代表了优先级的概念。nice 值默认值是 0 ，但是可以通过系统调用 nice 值来修改。修改值的范围从 -20 - +19。nice 值决定了线程的静态优先级。一般系统管理员的 nice 值会比一般线程的优先级高，它的范围是 -20 - -1。

下面我们更详细的讨论一下 Linux 系统的两个调度算法，它们的内部与调度队列(runqueue) 的设计很相似。运行队列有一个数据结构用来监视系统中所有可运行的任务并选择下一个可以运行的任务。每个运行队列和系统中的每个 CPU 有关。

Linux O(1) 调度器是历史上很流行的一个调度器。这个名字的由来是因为它能够在常数时间内执行任务调度。在 O(1) 调度器里，调度队列被组织成两个数组，一个是任务「正在活动」的数组，一个是任务「过期失效」的数组。如下图所示，每个数组都包含了 140 个链表头，每个链表头具有不同的优先级。

大致流程如下：

调度器从正在活动数组中选择一个优先级最高的任务。如果这个任务的时间片过期失效了，就把它移动到过期失效数组中。如果这个任务阻塞了，比如说正在等待 I/O 事件，那么在它的时间片过期失效之前，一旦 I/O 操作完成，那么这个任务将会继续运行，它将被放回到之前正在活动的数组中，因为这个任务之前已经消耗一部分 CPU 时间片，所以它将运行剩下的时间片。当这个任务运行完它的时间片后，它就会被放到过期失效数组中。一旦正在活动的任务数组中没有其他任务后，调度器将会交换指针，使得正在活动的数组变为过期失效数组，过期失效数组变为正在活动的数组。使用这种方式可以保证每个优先级的任务都能够得到执行，不会导致线程饥饿。

在这种调度方式中，不同优先级的任务所得到 CPU 分配的时间片也是不同的，高优先级进程往往能得到较长的时间片，低优先级的任务得到较少的时间片。

这种方式为了保证能够更好的提供服务，通常会为 交互式进程 赋予较高的优先级，交互式进程就是用户进程。

Linux 系统不知道一个任务究竟是 I/O 密集型的还是 CPU 密集型的，它只是依赖于交互式的方式，Linux 系统会区分是静态优先级 还是 动态优先级。动态优先级是采用一种奖励机制来实现的。奖励机制有两种方式：「奖励交互式线程、惩罚占用 CPU 的线程」。在 Linux O(1) 调度器中，最高的优先级奖励是 -5，注意这个优先级越低越容易被线程调度器接受，所以最高惩罚的优先级是 +5。具体体现就是操作系统维护一个名为 sleep\_avg 的变量，任务唤醒会增加 sleep\_avg 变量的值，当任务被抢占或者时间量过期会减少这个变量的值，反映在奖励机制上。

❝

O(1) 调度算法是 2.6 内核版本的调度器，最初引入这个调度算法的是不稳定的 2.5 版本。早期的调度算法在多处理器环境中说明了通过访问正在活动数组就可以做出调度的决定。使调度可以在固定的时间 O(1) 完成。

❞

O(1) 调度器使用了一种 启发式 的方式，这是什么意思？

❝

在计算机科学中，启发式是一种当传统方式解决问题很慢时用来快速解决问题的方式，或者找到一个在传统方法无法找到任何精确解的情况下找到近似解。

❞

O(1) 使用启发式的这种方式，会使任务的优先级变得复杂并且不完善，从而导致在处理交互任务时性能很糟糕。

为了改进这个缺点，O(1) 调度器的开发者又提出了一个新的方案，即 公平调度器(Completely Fair Scheduler, CFS)。CFS 的主要思想是使用一颗红黑树作为调度队列。

❝

数据结构太重要了。

❞

CFS 会根据任务在 CPU 上的运行时间长短而将其有序地排列在树中，时间精确到纳秒级。下面是 CFS 的构造模型

CFS 的调度过程如下：

CFS 算法总是优先调度哪些使用 CPU 时间最少的任务。最小的任务一般都是在最左边的位置。当有一个新的任务需要运行时，CFS 会把这个任务和最左边的数值进行对比，如果此任务具有最小时间值，那么它将进行运行，否则它会进行比较，找到合适的位置进行插入。然后 CPU 运行红黑树上当前比较的最左边的任务。

在红黑树中选择一个节点来运行的时间可以是常数时间，但是插入一个任务的时间是 O(loog(N))，其中 N 是系统中的任务数。考虑到当前系统的负载水平，这是可以接受的。

调度器只需要考虑可运行的任务即可。这些任务被放在适当的调度队列中。不可运行的任务和正在等待的各种 I/O 操作或内核事件的任务被放入一个等待队列中。等待队列头包含一个指向任务链表的指针和一个自旋锁。自旋锁对于并发处理场景下用处很大。

Linux 系统中的同步

下面来聊一下 Linux 中的同步机制。早期的 Linux 内核只有一个 大内核锁(Big Kernel Lock,BKL) 。它阻止了不同处理器并发处理的能力。因此，需要引入一些粒度更细的锁机制。

Linux 提供了若干不同类型的同步变量，这些变量既能够在内核中使用，也能够在用户应用程序中使用。在地层中，Linux 通过使用 atomic\_set 和 atomic\_read 这样的操作为硬件支持的原子指令提供封装。硬件提供内存重排序，这是 Linux 屏障的机制。

具有高级别的同步像是自旋锁的描述是这样的，当两个进程同时对资源进行访问，在一个进程获得资源后，另一个进程不想被阻塞，所以它就会自旋，等待一会儿再对资源进行访问。Linux 也提供互斥量或信号量这样的机制，也支持像是 mutex\_tryLock 和 mutex\_tryWait 这样的非阻塞调用。也支持中断处理事务，也可以通过动态禁用和启用相应的中断来实现。

Linux 启动

下面来聊一聊 Linux 是如何启动的。

当计算机电源通电后，BIOS会进行开机自检(Power-On-Self-Test, POST)，对硬件进行检测和初始化。因为操作系统的启动会使用到磁盘、屏幕、键盘、鼠标等设备。下一步，磁盘中的第一个分区，也被称为 MBR(Master Boot Record) 主引导记录，被读入到一个固定的内存区域并执行。这个分区中有一个非常小的，只有 512 字节的程序。程序从磁盘中调入 boot 独立程序，boot 程序将自身复制到高位地址的内存从而为操作系统释放低位地址的内存。

复制完成后，boot 程序读取启动设备的根目录。boot 程序要理解文件系统和目录格式。然后 boot 程序被调入内核，把控制权移交给内核。直到这里，boot 完成了它的工作。系统内核开始运行。

内核启动代码是使用汇编语言完成的，主要包括创建内核堆栈、识别 CPU 类型、计算内存、禁用中断、启动内存管理单元等，然后调用 C 语言的 main 函数执行操作系统部分。

这部分也会做很多事情，首先会分配一个消息缓冲区来存放调试出现的问题，调试信息会写入缓冲区。如果调试出现错误，这些信息可以通过诊断程序调出来。

然后操作系统会进行自动配置，检测设备，加载配置文件，被检测设备如果做出响应，就会被添加到已链接的设备表中，如果没有相应，就归为未连接直接忽略。

配置完所有硬件后，接下来要做的就是仔细手工处理进程0，设置其堆栈，然后运行它，执行初始化、配置时钟、挂载文件系统。创建 init 进程(进程 1 ) 和 守护进程(进程 2)。

init 进程会检测它的标志以确定它是否为单用户还是多用户服务。在前一种情况中，它会调用 fork 函数创建一个 shell 进程，并且等待这个进程结束。后一种情况调用 fork 函数创建一个运行系统初始化的 shell 脚本（即 /etc/rc）的进程，这个进程可以进行文件系统一致性检测、挂载文件系统、开启守护进程等。

然后 /etc/rc 这个进程会从 /etc/ttys 中读取数据，/etc/ttys 列出了所有的终端和属性。对于每一个启用的终端，这个进程调用 fork 函数创建一个自身的副本，进行内部处理并运行一个名为 getty 的程序。

getty 程序会在终端上输入

login:

等待用户输入用户名，在输入用户名后，getty 程序结束，登陆程序 /bin/login开始运行。login 程序需要输入密码，并与保存在 /etc/passwd 中的密码进行对比，如果输入正确，login 程序以用户 shell 程序替换自身，等待第一个命令。如果不正确，login 程序要求输入另一个用户名。

整个系统启动过程如下

Linux 内存管理

Linux 内存管理模型非常直接明了，因为 Linux 的这种机制使其具有可移植性并且能够在内存管理单元相差不大的机器下实现 Linux，下面我们就来认识一下 Linux 内存管理是如何实现的。

基本概念

每个 Linux 进程都会有地址空间，这些地址空间由三个段区域组成：「text 段、data 段、stack 段」。下面是进程地址空间的示例。

数据段(data segment) 包含了程序的变量、字符串、数组和其他数据的存储。数据段分为两部分，已经初始化的数据和尚未初始化的数据。其中尚未初始化的数据就是我们说的 BSS。数据段部分的初始化需要编译就期确定的常量以及程序启动就需要一个初始值的变量。所有 BSS 部分中的变量在加载后被初始化为 0 。

和 代码段(Text segment) 不一样，data segment 数据段可以改变。程序总是修改它的变量。而且，许多程序需要在执行时动态分配空间。Linux 允许数据段随着内存的分配和回收从而增大或者减小。为了分配内存，程序可以增加数据段的大小。在 C 语言中有一套标准库 malloc 经常用于分配内存。进程地址空间描述符包含动态分配的内存区域称为 堆(heap)。

第三部分段是 栈段(stack segment)。在大部分机器上，栈段会在虚拟内存地址顶部地址位置处，并向低位置处（向地址空间为 0 处）拓展。举个例子来说，在 32 位 x86 架构的机器上，栈开始于 0xC0000000，这是用户模式下进程允许可见的 3GB 虚拟地址限制。如果栈一直增大到超过栈段后，就会发生硬件故障并把页面下降一个页面。

当程序启动时，栈区域并不是空的，相反，它会包含所有的 shell 环境变量以及为了调用它而向 shell 输入的命令行。举个例子，当你输入

cp cxuan lx

时，cp 程序会运行并在栈中带着字符串 cp cxuan lx ，这样就能够找出源文件和目标文件的名称。

当两个用户运行在相同程序中，例如编辑器(editor)，那么就会在内存中保持编辑器程序代码的两个副本，但是这种方式并不高效。Linux 系统支持共享文本段作为替代。下面图中我们会看到 A 和 B 两个进程，它们有着相同的文本区域。

数据段和栈段只有在 fork 之后才会共享，共享也是共享未修改过的页面。如果任何一个都需要变大但是没有相邻空间容纳的话，也不会有问题，因为相邻的虚拟页面不必映射到相邻的物理页面上。

除了动态分配更多的内存，Linux 中的进程可以通过内存映射文件来访问文件数据。这个特性可以使我们把一个文件映射到进程空间的一部分而该文件就可以像位于内存中的字节数组一样被读写。把一个文件映射进来使得随机读写比使用 read 和 write 之类的 I/O 系统调用要容易得多。共享库的访问就是使用了这种机制。如下所示

我们可以看到两个相同文件会被映射到相同的物理地址上，但是它们属于不同的地址空间。

映射文件的优点是，两个或多个进程可以同时映射到同一文件中，任意一个进程对文件的写操作对其他文件可见。通过使用映射临时文件的方式，可以为多线程共享内存提供高带宽，临时文件在进程退出后消失。但是实际上，并没有两个相同的地址空间，因为每个进程维护的打开文件和信号不同。

Linux 内存管理系统调用

下面我们探讨一下关于内存管理的系统调用方式。事实上，POSIX 并没有给内存管理指定任何的系统调用。然而，Linux 却有自己的内存系统调用，主要系统调用如下

| 系统调用 | 描述 |
| --- | --- |
| s = brk(addr) | 改变数据段大小 |
| a = mmap(addr,len,prot,flags,fd,offset) | 进行映射 |
| s = unmap(addr,len) | 取消映射 |

如果遇到错误，那么 s 的返回值是 -1，a 和 addr 是内存地址，len 表示的是长度，prot 表示的是控制保护位，flags 是其他标志位，fd 是文件描述符，offset 是文件偏移量。

brk 通过给出超过数据段之外的第一个字节地址来指定数据段的大小。如果新的值要比原来的大，那么数据区会变得越来越大，反之会越来越小。

mmap 和 unmap 系统调用会控制映射文件。mmp 的第一个参数 addr 决定了文件映射的地址。它必须是页面大小的倍数。如果参数是 0，系统会分配地址并返回 a。第二个参数是长度，它告诉了需要映射多少字节。它也是页面大小的倍数。prot 决定了映射文件的保护位，保护位可以标记为 「可读、可写、可执行或者这些的结合」。第四个参数 flags 能够控制文件是私有的还是可读的以及 addr 是必须的还是只是进行提示。第五个参数 fd 是要映射的文件描述符。只有打开的文件是可以被映射的，因此如果想要进行文件映射，必须打开文件；最后一个参数 offset 会指示文件从什么时候开始，并不一定每次都要从零开始。

Linux 内存管理实现

内存管理系统是操作系统最重要的部分之一。从计算机早期开始，我们实际使用的内存都要比系统中实际存在的内存多。内存分配策略克服了这一限制，并且其中最有名的就是 虚拟内存(virtual memory)。通过在多个竞争的进程之间共享虚拟内存，虚拟内存得以让系统有更多的内存。虚拟内存子系统主要包括下面这些概念。

「大地址空间」

操作系统使系统使用起来好像比实际的物理内存要大很多，那是因为虚拟内存要比物理内存大很多倍。

「保护」

系统中的每个进程都会有自己的虚拟地址空间。这些虚拟地址空间彼此完全分开，因此运行一个应用程序的进程不会影响另一个。并且，硬件虚拟内存机制允许内存保护关键内存区域。

「内存映射」

内存映射用来向进程地址空间映射图像和数据文件。在内存映射中，文件的内容直接映射到进程的虚拟空间中。

「公平的物理内存分配」

内存管理子系统允许系统中的每个正在运行的进程公平分配系统的物理内存。

「共享虚拟内存」

尽管虚拟内存让进程有自己的内存空间，但是有的时候你是需要共享内存的。例如几个进程同时在 shell 中运行，这会涉及到 IPC 的进程间通信问题，这个时候你需要的是共享内存来进行信息传递而不是通过拷贝每个进程的副本独立运行。

下面我们就正式探讨一下什么是 虚拟内存

虚拟内存的抽象模型

在考虑 Linux 用于支持虚拟内存的方法之前，考虑一个不会被太多细节困扰的抽象模型是很有用的。

处理器在执行指令时，会从内存中读取指令并将其解码(decode)，在指令解码时会获取某个位置的内容并将他存到内存中。然后处理器继续执行下一条指令。这样，处理器总是在访问存储器以获取指令和存储数据。

在虚拟内存系统中，所有的地址空间都是虚拟的而不是物理的。但是实际存储和提取指令的是物理地址，所以需要让处理器根据操作系统维护的一张表将虚拟地址转换为物理地址。

为了简单的完成转换，虚拟地址和物理地址会被分为固定大小的块，称为 页(page)。这些页有相同大小，如果页面大小不一样的话，那么操作系统将很难管理。Alpha AXP系统上的 Linux 使用 8 KB 页面，而 Intel x86 系统上的 Linux 使用 4 KB 页面。每个页面都有一个唯一的编号，即页面框架号（PFN）。

上面就是 Linux 内存映射模型了，在这个页模型中，虚拟地址由两部分组成：「偏移量和虚拟页框号」。每次处理器遇到虚拟地址时都会提取偏移量和虚拟页框号。处理器必须将虚拟页框号转换为物理页号，然后以正确的偏移量的位置访问物理页。

上图中展示了两个进程 A 和 B 的虚拟地址空间，每个进程都有自己的页表。这些页表将进程中的虚拟页映射到内存中的物理页中。页表中每一项均包含

* 有效标志(valid flag)：表明此页表条目是否有效
* 该条目描述的物理页框号
* 访问控制信息，页面使用方式，是否可写以及是否可以执行代码

要将处理器的虚拟地址映射为内存的物理地址，首先需要计算虚拟地址的页框号和偏移量。页面大小为 2 的次幂，可以通过移位完成操作。

如果当前进程尝试访问虚拟地址，但是访问不到的话，这种情况称为 缺页异常，此时虚拟操作系统的错误地址和页面错误的原因将通知操作系统。

通过以这种方式将虚拟地址映射到物理地址，虚拟内存可以以任何顺序映射到系统的物理页面。

按需分页

由于物理内存要比虚拟内存少很多，因此操作系统需要注意尽量避免直接使用低效的物理内存。节省物理内存的一种方式是仅加载执行程序当前使用的页面（这何尝不是一种懒加载的思想呢？）。例如，可以运行数据库来查询数据库，在这种情况下，不是所有的数据都装入内存，只装载需要检查的数据。这种仅仅在需要时才将虚拟页面加载进内中的技术称为按需分页。

交换

如果某个进程需要将虚拟页面传入内存，但是此时没有可用的物理页面，那么操作系统必须丢弃物理内存中的另一个页面来为该页面腾出空间。

如果页面已经修改过，那么操作系统必须保留该页面的内容，以便以后可以访问它。这种类型的页面被称为脏页，当将其从内存中移除时，它会保存在称为交换文件的特殊文件中。相对于处理器和物理内存的速度，对交换文件的访问非常慢，并且操作系统需要兼顾将页面写到磁盘的以及将它们保留在内存中以便再次使用。

Linux 使用最近最少使用(LRU)页面老化技术来公平的选择可能会从系统中删除的页面，这个方案涉及系统中的每个页面，页面的年龄随着访问次数的变化而变化，如果某个页面访问次数多，那么该页就表示越 年轻，如果某个呃页面访问次数太少，那么该页越容易被换出。

物理和虚拟寻址模式

大多数多功能处理器都支持 物理地址模式和虚拟地址模式的概念。物理寻址模式不需要页表，并且处理器不会在此模式下尝试执行任何地址转换。Linux 内核被链接在物理地址空间中运行。

Alpha AXP 处理器没有物理寻址模式。相反，它将内存空间划分为几个区域，并将其中两个指定为物理映射的地址。此内核地址空间称为 KSEG 地址空间，它包含从 0xfffffc0000000000 向上的所有地址。为了从 KSEG 中链接的代码（按照定义，内核代码）执行或访问其中的数据，该代码必须在内核模式下执行。链接到 Alpha 上的 Linux内核以从地址 0xfffffc0000310000 执行。

访问控制

页面表的每一项还包含访问控制信息，访问控制信息主要检查进程是否应该访问内存。

必要时需要对内存进行访问限制。例如包含可执行代码的内存，自然是只读内存；操作系统不应允许进程通过其可执行代码写入数据。相比之下，包含数据的页面可以被写入，但是尝试执行该内存的指令将失败。大多数处理器至少具有两种执行模式：内核态和用户态。你不希望访问用户执行内核代码或内核数据结构，除非处理器以内核模式运行。

访问控制信息被保存在上面的 Page Table Entry ，页表项中，上面这幅图是 Alpha AXP的 PTE。位字段具有以下含义

* V

表示 valid ，是否有效位

* FOR

读取时故障，在尝试读取此页面时出现故障

* FOW

写入时错误，在尝试写入时发生错误

* FOE

执行时发生错误，在尝试执行此页面中的指令时，处理器都会报告页面错误并将控制权传递给操作系统，

* ASM

地址空间匹配，当操作系统希望清除转换缓冲区中的某些条目时，将使用此选项。

* GH

当在使用单个转换缓冲区条目而不是多个转换缓冲区条目映射整个块时使用的提示。

* KRE

内核模式运行下的代码可以读取页面

* URE

用户模式下的代码可以读取页面

* KWE

以内核模式运行的代码可以写入页面

* UWE

以用户模式运行的代码可以写入页面

* 页框号

对于设置了 V 位的 PTE，此字段包含此 PTE 的物理页面帧号（页面帧号）。对于无效的 PTE，如果此字段不为零，则包含有关页面在交换文件中的位置的信息。

除此之外，Linux 还使用了两个位

* \_PAGE\_DIRTY

如果已设置，则需要将页面写出到交换文件中

* \_PAGE\_ACCESSED

Linux 用来将页面标记为已访问。

缓存

上面的虚拟内存抽象模型可以用来实施，但是效率不会太高。操作系统和处理器设计人员都尝试提高性能。但是除了提高处理器，内存等的速度之外，最好的方法就是维护有用信息和数据的高速缓存，从而使某些操作更快。在 Linux 中，使用很多和内存管理有关的缓冲区，使用缓冲区来提高效率。

缓冲区缓存

缓冲区高速缓存包含块设备驱动程序使用的数据缓冲区。

还记得什么是块设备么？这里回顾下

块设备是一个能存储固定大小块信息的设备，它支持「以固定大小的块，扇区或群集读取和（可选）写入数据」。每个块都有自己的物理地址。通常块的大小在 512 - 65536 之间。所有传输的信息都会以连续的块为单位。块设备的基本特征是每个块都较为对立，能够独立的进行读写。常见的块设备有 「硬盘、蓝光光盘、USB 盘」

与字符设备相比，块设备通常需要较少的引脚。

缓冲区高速缓存通过设备标识符和块编号用于快速查找数据块。如果可以在缓冲区高速缓存中找到数据，则无需从物理块设备中读取数据，这种访问方式要快得多。

页缓存

页缓存用于加快对磁盘上图像和数据的访问

它用于一次一页地缓存文件中的内容，并且可以通过文件和文件中的偏移量进行访问。当页面从磁盘读入内存时，它们被缓存在页面缓存中。

交换区缓存

仅仅已修改（脏页）被保存在交换文件中

只要这些页面在写入交换文件后没有修改，则下次交换该页面时，无需将其写入交换文件，因为该页面已在交换文件中。可以直接丢弃。在大量交换的系统中，这节省了许多不必要的和昂贵的磁盘操作。

硬件缓存

处理器中通常使用一种硬件缓存。页表条目的缓存。在这种情况下，处理器并不总是直接读取页表，而是根据需要缓存页的翻译。这些是转换后备缓冲区 也被称为 TLB，包含来自系统中一个或多个进程的页表项的缓存副本。

引用虚拟地址后，处理器将尝试查找匹配的 TLB 条目。如果找到，则可以将虚拟地址直接转换为物理地址，并对数据执行正确的操作。如果处理器找不到匹配的 TLB 条目， 它通过向操作系统发信号通知已发生 TLB 丢失获得操作系统的支持和帮助。系统特定的机制用于将该异常传递给可以修复问题的操作系统代码。操作系统为地址映射生成一个新的 TLB 条目。清除异常后，处理器将再次尝试转换虚拟地址。这次能够执行成功。

使用缓存也存在缺点，为了节省精力，Linux 必须使用更多的时间和空间来维护这些缓存，并且如果缓存损坏，系统将会崩溃。

Linux 页表

Linux 假定页表分为三个级别。访问的每个页表都包含下一级页表

图中的 PDG 表示全局页表，当创建一个新的进程时，都要为新进程创建一个新的页面目录，即 PGD。

要将虚拟地址转换为物理地址，处理器必须获取每个级别字段的内容，将其转换为包含页表的物理页的偏移量，并读取下一级页表的页框号。这样重复三次，直到找到包含虚拟地址的物理页面的页框号为止。

Linux 运行的每个平台都必须提供翻译宏，这些宏允许内核遍历特定进程的页表。这样，内核无需知道页表条目的格式或它们的排列方式。

页分配和取消分配

对系统中物理页面有很多需求。例如，当图像加载到内存中时，操作系统需要分配页面。

系统中所有物理页面均由 mem\_map 数据结构描述，这个数据结构是 mem\_map\_t的列表。它包括一些重要的属性

* count ：这是页面的用户数计数，当页面在多个进程之间共享时，计数大于 1
* age：这是描述页面的年龄，用于确定页面是否适合丢弃或交换
* map\_nr ：这是此mem\_map\_t描述的物理页框号。

页面分配代码使用 free\_area向量查找和释放页面，free\_area 的每个元素都包含有关页面块的信息。

页面分配

Linux 的页面分配使用一种著名的伙伴算法来进行页面的分配和取消分配。页面以 2 的幂为单位进行块分配。这就意味着它可以分配 1页、2 页、4页等等，只要系统中有足够可用的页面来满足需求就可以。判断的标准是「nr\_free\_pages> min\_free\_pages」，如果满足，就会在 free\_area 中搜索所需大小的页面块完成分配。free\_area 的每个元素都有该大小的块的已分配页面和空闲页面块的映射。

分配算法会搜索请求大小的页面块。如果没有任何请求大小的页面块可用的话，会搜寻一个是请求大小二倍的页面块，然后重复，直到一直搜寻完 free\_area 找到一个页面块为止。如果找到的页面块要比请求的页面块大，就会对找到的页面块进行细分，直到找到合适的大小块为止。

因为每个块都是 2 的次幂，所以拆分过程很容易，因为你只需将块分成两半即可。空闲块在适当的队列中排队，分配的页面块返回给调用者。

如果请求一个 2 个页的块，则 4 页的第一个块（从第 4 页的框架开始）将被分成两个 2 页的块。第一个页面（从第 4 页的帧开始）将作为分配的页面返回给调用方，第二个块（从第 6 页的页面开始）将作为 2 页的空闲块排队到 free\_area 数组的元素 1 上。

页面取消分配

上面的这种内存方式最造成一种后果，那就是内存的碎片化，会将较大的空闲页面分成较小的页面。页面解除分配代码会尽可能将页面重新组合成为更大的空闲块。每释放一个页面，都会检查相同大小的相邻的块，以查看是否空闲。如果是，则将其与新释放的页面块组合以形成下一个页面大小块的新的自由页面块。每次将两个页面块重新组合为更大的空闲页面块时，页面释放代码就会尝试将该页面块重新组合为更大的空闲页面。通过这种方式，可用页面的块将尽可能多地使用内存。

例如上图，如果要释放第 1 页的页面，则将其与已经空闲的第 0 页页面框架组合在一起，并作为大小为 2页的空闲块排队到 free\_area 的元素 1 中

内存映射

内核有两种类型的内存映射：共享型(shared) 和私有型(private)。私有型是当进程为了只读文件，而不写文件时使用，这时，私有映射更加高效。但是，任何对私有映射页的写操作都会导致内核停止映射该文件中的页。所以，写操作既不会改变磁盘上的文件，对访问该文件的其它进程也是不可见的。

按需分页

一旦可执行映像被内存映射到虚拟内存后，它就可以被执行了。因为只将映像的开头部分物理的拉入到内存中，因此它将很快访问物理内存尚未存在的虚拟内存区域。当进程访问没有有效页表的虚拟地址时，操作系统会报告这项错误。

页面错误描述页面出错的虚拟地址和引起的内存访问（RAM）类型。

Linux 必须找到代表发生页面错误的内存区域的 vm\_area\_struct 结构。由于搜索 vm\_area\_struct 数据结构对于有效处理页面错误至关重要，因此它们以 AVL（Adelson-Velskii和Landis）树结构链接在一起。如果引起故障的虚拟地址没有 vm\_area\_struct 结构，则此进程已经访问了非法地址，Linux 会向进程发出 SIGSEGV 信号，如果进程没有用于该信号的处理程序，那么进程将会终止。

然后，Linux 会针对此虚拟内存区域所允许的访问类型，检查发生的页面错误类型。如果该进程以非法方式访问内存，例如写入仅允许读的区域，则还会发出内存访问错误信号。

现在，Linux 已确定页面错误是合法的，因此必须对其进行处理。

文件系统

在 Linux 中，最直观、最可见的部分就是 文件系统(file system)。下面我们就来一起探讨一下关于 Linux 中国的文件系统，系统调用以及文件系统实现背后的原理和思想。这些思想中有一些来源于 MULTICS，现在已经被 Windows 等其他操作系统使用。Linux 的设计理念就是 小的就是好的(Small is Beautiful) 。虽然 Linux 只是使用了最简单的机制和少量的系统调用，但是 Linux 却提供了强大而优雅的文件系统。

Linux 文件系统基本概念

Linux 在最初的设计是 MINIX1 文件系统，它只支持 14 字节的文件名，它的最大文件只支持到 64 MB。在 MINIX 1 之后的文件系统是 ext 文件系统。ext 系统相较于 MINIX 1 来说，在支持字节大小和文件大小上均有很大提升，但是 ext 的速度仍没有 MINIX 1 快，于是，ext 2 被开发出来，它能够支持长文件名和大文件，而且具有比 MINIX 1 更好的性能。这使他成为 Linux 的主要文件系统。只不过 Linux 会使用 VFS 曾支持多种文件系统。在 Linux 链接时，用户可以动态的将不同的文件系统挂载倒 VFS 上。

Linux 中的文件是一个任意长度的字节序列，Linux 中的文件可以包含任意信息，比如 ASCII 码、二进制文件和其他类型的文件是不加区分的。

为了方便起见，文件可以被组织在一个目录中，目录存储成文件的形式在很大程度上可以作为文件处理。目录可以有子目录，这样形成有层次的文件系统，Linux 系统下面的根目录是 / ，它通常包含了多个子目录。字符 / 还用于对目录名进行区分，例如 「/usr/cxuan」 表示的就是根目录下面的 usr 目录，其中有一个叫做 cxuan 的子目录。

下面我们介绍一下 Linux 系统根目录下面的目录名

* /bin，它是重要的二进制应用程序，包含二进制文件，系统的所有用户使用的命令都在这里
* /boot，启动包含引导加载程序的相关文件
* /dev，包含设备文件，终端文件，USB 或者连接到系统的任何设备
* /etc，配置文件，启动脚本等，包含所有程序所需要的配置文件，也包含了启动/停止单个应用程序的启动和关闭 shell 脚本
* /home，本地主要路径，所有用户用 home 目录存储个人信息
* /lib，系统库文件，包含支持位于 /bin 和 /sbin 下的二进制库文件
* /lost+found，在根目录下提供一个遗失+查找系统，必须在 root 用户下才能查看当前目录下的内容
* /media，挂载可移动介质
* /mnt，挂载文件系统
* /opt，提供一个可选的应用程序安装目录
* /proc，特殊的动态目录，用于维护系统信息和状态，包括当前运行中进程信息
* /root，root 用户的主要目录文件夹
* /sbin，重要的二进制系统文件
* /tmp， 系统和用户创建的临时文件，系统重启时，这个目录下的文件都会被删除
* /usr，包含绝大多数用户都能访问的应用程序和文件
* /var，经常变化的文件，诸如日志文件或数据库等

在 Linux 中，有两种路径，一种是 绝对路径(absolute path) ，绝对路径告诉你从根目录下查找文件，绝对路径的缺点是太长而且不太方便。还有一种是 相对路径(relative path) ，相对路径所在的目录也叫做工作目录(working directory)。

如果 /usr/local/books 是工作目录，那么 shell 命令

cp books books-replica

就表示的是相对路径，而

cp /usr/local/books/books /usr/local/books/books-replica

则表示的是绝对路径。

在 Linux 中经常出现一个用户使用另一个用户的文件或者使用文件树结构中的文件。两个用户共享同一个文件，这个文件位于某个用户的目录结构中，另一个用户需要使用这个文件时，必须通过绝对路径才能引用到他。如果绝对路径很长，那么每次输入起来会变的非常麻烦，所以 Linux 提供了一种 链接(link) 机制。

举个例子，下面是一个使用链接之前的图

以上所示，比如有两个工作账户 jianshe 和 cxuan，jianshe 想要使用 cxuan 账户下的 A 目录，那么它可能会输入 /usr/cxuan/A ，这是一种未使用链接之后的图。

使用链接后的示意如下

现在，jianshe 可以创建一个链接来使用 cxuan 下面的目录了。‘

当一个目录被创建出来后，有两个目录项也同时被创建出来，它们就是 . 和 ..，前者代表工作目录自身，后者代表该目录的父目录，也就是该目录所在的目录。这样一来，在 /usr/jianshe 中访问 cxuan 中的目录就是 ../cxuan/xxx

Linux 文件系统不区分磁盘的，这是什么意思呢？一般来说，一个磁盘中的文件系统相互之间保持独立，如果一个文件系统目录想要访问另一个磁盘中的文件系统，在 Windows 中你可以像下面这样。

两个文件系统分别在不同的磁盘中，彼此保持独立。

而在 Linux 中，是支持挂载的，它允许一个磁盘挂在到另外一个磁盘上，那么上面的关系会变成下面这样

挂在之后，两个文件系统就不再需要关心文件系统在哪个磁盘上了，两个文件系统彼此可见。

Linux 文件系统的另外一个特性是支持 加锁(locking)。在一些应用中会出现两个或者更多的进程同时使用同一个文件的情况，这样很可能会导致竞争条件(race condition)。一种解决方法是对其进行加不同粒度的锁，就是为了防止某一个进程只修改某一行记录从而导致整个文件都不能使用的情况。

POSIX 提供了一种灵活的、不同粒度级别的锁机制，允许一个进程使用一个不可分割的操作对一个字节或者整个文件进行加锁。加锁机制要求尝试加锁的进程指定其 「要加锁的文件，开始位置以及要加锁的字节」

Linux 系统提供了两种锁：「共享锁和互斥锁」。如果文件的一部分已经加上了共享锁，那么再加排他锁是不会成功的；如果文件系统的一部分已经被加了互斥锁，那么在互斥锁解除之前的任何加锁都不会成功。为了成功加锁、请求加锁的部分的所有字节都必须是可用的。

在加锁阶段，进程需要设计好加锁失败后的情况，也就是判断加锁失败后是否选择阻塞，如果选择阻塞式，那么当已经加锁的进程中的锁被删除时，这个进程会解除阻塞并替换锁。如果进程选择非阻塞式的，那么就不会替换这个锁，会立刻从系统调用中返回，标记状态码表示是否加锁成功，然后进程会选择下一个时间再次尝试。

加锁区域是可以重叠的。下面我们演示了三种不同条件的加锁区域。

如上图所示，A 的共享锁在第四字节到第八字节进行加锁

如上图所示，进程在 A 和 B 上同时加了共享锁，其中 6 - 8 字节是重叠锁

如上图所示，进程 A 和 B 和 C 同时加了共享锁，那么第六字节和第七字节是共享锁。

如果此时一个进程尝试在第 6 个字节处加锁，此时会设置失败并阻塞，由于该区域被 A B C 同时加锁，那么只有等到 A B C 都释放锁后，进程才能加锁成功。

Linux 文件系统调用

许多系统调用都会和文件与文件系统有关。我们首先先看一下对单个文件的系统调用，然后再来看一下对整个目录和文件的系统调用。

为了创建一个新的文件，会使用到 creat 方法，注意没有 e。

❝

这里说一个小插曲，曾经有人问 UNIX 创始人 Ken Thompson，如果有机会重新写 UNIX ，你会怎么办，他回答自己要把 creat 改成 create ，哈哈哈哈。

❞

这个系统调用的两个参数是文件名和保护模式

fd = creat("aaa",mode);

这段命令会创建一个名为 aaa 的文件，并根据 mode 设置文件的保护位。这些位决定了哪个用户可能访问文件、如何访问。

creat 系统调用不仅仅创建了一个名为 aaa 的文件，还会打开这个文件。为了允许后续的系统调用访问这个文件，这个 creat 系统调用会返回一个 非负整数， 这个就叫做 文件描述符(file descriptor)，也就是上面的 fd。

如果在已经存在的文件上调用了 creat 系统调用，那么该文件中的内容会被清除，从 0 开始。通过设置合适的参数，open 系统调用也能够创建文件。

下面让我们看一看主要的系统调用，如下表所示

| 系统调用 | 描述 |
| --- | --- |
| fd = creat(name,mode) | 一种创建一个新文件的方式 |
| fd = open(file, ...) | 打开文件读、写或者读写 |
| s = close(fd) | 关闭一个打开的文件 |
| n = read(fd, buffer, nbytes) | 从文件中向缓存中读入数据 |
| n = write(fd, buffer, nbytes) | 从缓存中向文件中写入数据 |
| position = lseek(fd, offset, whence) | 移动文件指针 |
| s = stat(name, &buf) | 获取文件信息 |
| s = fstat(fd, &buf) | 获取文件信息 |
| s = pipe(&fd[0]) | 创建一个管道 |
| s = fcntl(fd,...) | 文件加锁等其他操作 |

为了对一个文件进行读写的前提是先需要打开文件，必须使用 creat 或者 open 打开，参数是打开文件的方式，是只读、可读写还是只写。open 系统调用也会返回文件描述符。打开文件后，需要使用 close 系统调用进行关闭。close 和 open 返回的 fd 总是未被使用的最小数量。

❝

什么是文件描述符？文件描述符就是一个数字，这个数字标示了计算机操作系统中打开的文件。它描述了数据资源，以及访问资源的方式。

❞

当程序要求打开一个文件时，内核会进行如下操作

* 授予访问权限
* 在全局文件表(global file table)中创建一个条目(entry)
* 向软件提供条目的位置

文件描述符由唯一的非负整数组成，系统上每个打开的文件至少存在一个文件描述符。文件描述符最初在 Unix 中使用，并且被包括 Linux，macOS 和 BSD 在内的现代操作系统所使用。

当一个进程成功访问一个打开的文件时，内核会返回一个文件描述符，这个文件描述符指向全局文件表的 entry 项。这个文件表项包含文件的 inode 信息，字节位移，访问限制等。例如下图所示

默认情况下，前三个文件描述符为 STDIN(标准输入)、STDOUT(标准输出)、STDERR(标准错误)。

标准输入的文件描述符是 0 ，在终端中，默认为用户的键盘输入

标准输出的文件描述符是 1 ，在终端中，默认为用户的屏幕

与错误有关的默认数据流是 2，在终端中，默认为用户的屏幕。

在简单聊了一下文件描述符后，我们继续回到文件系统调用的探讨。

在文件系统调用中，开销最大的就是 read 和 write 了。read 和 write 都有三个参数

* 文件描述符：告诉需要对哪一个打开文件进行读取和写入
* 缓冲区地址：告诉数据需要从哪里读取和写入哪里
* 统计：告诉需要传输多少字节

这就是所有的参数了，这个设计非常简单轻巧。

虽然几乎所有程序都按顺序读取和写入文件，但是某些程序需要能够随机访问文件的任何部分。与每个文件相关联的是一个指针，该指针指示文件中的当前位置。顺序读取（或写入）时，它通常指向要读取（写入）的下一个字节。如果指针在读取 1024 个字节之前位于 4096 的位置，则它将在成功读取系统调用后自动移至 5120 的位置。

Lseek 系统调用会更改指针位置的值，以便后续对 read 或 write 的调用可以在文件中的任何位置开始，甚至可以超出文件末尾。

❝

lseek = Lseek ，段首大写。

❞

lseek 避免叫做 seek 的原因就是 seek 已经在之前 16 位的计算机上用于搜素功能了。

Lseek 有三个参数：第一个是文件的文件描述符，第二个是文件的位置；第三个告诉文件位置是相对于文件的开头，当前位置还是文件的结尾

lseek(int fildes, off\_t offset, int whence);

lseek 的返回值是更改文件指针后文件中的绝对位置。lseek 是唯一从来不会造成真正磁盘查找的系统调用，它只是更新当前的文件位置，这个文件位置就是内存中的数字。

对于每个文件，Linux 都会跟踪文件模式（常规，目录，特殊文件），大小，最后修改时间以及其他信息。程序能够通过 stat 系统调用看到这些信息。第一个参数就是文件名，第二个是指向要放置请求信息结构的指针。这些结构的属性如下图所示。

| 存储文件的设备 |
| --- |
| 存储文件的设备 |
| i-node 编号 |
| 文件模式(包括保护位信息) |
| 文件链接的数量 |
| 文件所有者标识 |
| 文件所属的组 |
| 文件大小(字节) |
| 创建时间 |
| 最后一个修改/访问时间 |

fstat 调用和 stat 相同，只有一点区别，fstat 可以对打开文件进行操作，而 stat 只能对路径进行操作。

pipe 文件系统调用被用来创建 shell 管道。它会创建一系列的伪文件，来缓冲和管道组件之间的数据，并且返回读取或者写入缓冲区的文件描述符。在管道中，像是如下操作

sort <in | head –40

sort 进程将会输出到文件描述符1，也就是标准输出，写入管道中，而 head 进程将从管道中读入。在这种方式中，sort 只是从文件描述符 0 中读取并写入到文件描述符 1 （管道）中，甚至不知道它们已经被重定向了。如果没有重定向的话，sort 会自动的从键盘读入并输出到屏幕中。

最后一个系统调用是 fcntl，它用来锁定和解锁文件，应用共享锁和互斥锁，或者是执行一些文件相关的其他操作。

现在我们来关心一下和整体目录和文件系统相关的系统调用，而不是把精力放在单个的文件上，下面列出了这些系统调用，我们一起来看一下。

| 系统调用 | 描述 |
| --- | --- |
| s = mkdir(path,mode) | 创建一个新的目录 |
| s = rmdir(path) | 移除一个目录 |
| s = link(oldpath,newpath) | 创建指向已有文件的链接 |
| s = unlink(path) | 取消文件的链接 |
| s = chdir(path) | 改变工作目录 |
| dir = opendir(path) | 打开一个目录读取 |
| s = closedir(dir) | 关闭一个目录 |
| dirent = readdir(dir) | 读取一个目录项 |
| rewinddir(dir) | 回转目录使其在此使用 |

可以使用 mkdir 和 rmdir 创建和删除目录。但是需要注意，只有目录为空时才可以删除。

创建一个指向已有文件的链接时会创建一个目录项(directory entry)。系统调用 link 来创建链接，oldpath 代表已有的路径，newpath 代表需要链接的路径，使用 unlink 可以删除目录项。当文件的最后一个链接被删除时，这个文件会被自动删除。

使用 chdir 系统调用可以改变工作目录。

最后四个系统调用是用于读取目录的。和普通文件类似，他们可以被打开、关闭和读取。每次调用 readdir 都会以固定的格式返回一个目录项。用户不能对目录执行写操作，但是可以使用 creat 或者 link 在文件夹中创建一个目录，或使用 unlink 删除一个目录。用户不能在目录中查找某个特定文件，但是可以使用 rewindir 作用于一个打开的目录，使他能在此从头开始读取。

Linux 文件系统的实现

下面我们主要讨论一下 虚拟文件系统(Virtual File System)。VFS 对高层进程和应用程序隐藏了 Linux 支持的所有文件系统的区别，以及文件系统是存储在本地设备，还是需要通过网络访问远程设备。设备和其他特殊文件和 VFS 层相关联。接下来，我们就会探讨一下第一个 Linux 广泛传播的文件系统：ext2。随后，我们就会探讨 ext4 文件系统所做的改进。各种各样的其他文件系统也正在使用中。所有 Linux 系统都可以处理多个磁盘分区，每个磁盘分区上都有不同的文件系统。

Linux 虚拟文件系统

为了能够使应用程序能够在不同类型的本地或者远程设备上的文件系统进行交互，因为在 Linux 当中文件系统千奇百种，比较常见的有 EXT3、EXT4，还有基于内存的 ramfs、tmpfs 和基于网络的 nfs，和基于用户态的 fuse，当然 fuse 应该不能完全的文件系统，只能算是一个能把文件系统实现放到用户态的模块，满足了内核文件系统的接口，他们都是文件系统的一种实现。对于这些文件系统，Linux 做了一层抽象就是 VFS虚拟文件系统，

下表总结了 VFS 支持的四个主要的文件系统结构。

| 对象 | 描述 |
| --- | --- |
| 超级块 | 特定的文件系统 |
| Dentry | 目录项，路径的一个组成部分 |
| I-node | 特定的文件 |
| File | 跟一个进程相关联的打开文件 |

超级块(superblock) 包含了有关文件系统布局的重要信息，超级块如果遭到破坏那么就会导致整个文件系统不可读。

i-node 索引节点，包含了每一个文件的描述符。

❝

在 Linux 中，目录和设备也表示为文件，因为它们具有对应的 i-node

❞

超级块和索引块所在的文件系统都在磁盘上有对应的结构。

为了便于某些目录操作和路径遍历，比如 /usr/local/cxuan，VFS 支持一个 dentry 数据结构，该数据结构代表着目录项。这个 dentry 数据结构有很多东西（http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec7.html）这个数据结构由文件系统动态创建。

目录项被缓存在 dentry\_cache 缓存中。例如，缓存条目会缓存 /usr 、 /usr/local 等条目。如果多个进程通过硬连接访问相同的文件，他们的文件对象将指向此缓存中的相同条目。

最后，文件数据结构是代表着打开的文件，也代表着内存表示，它根据 open 系统调用创建。它支持 「read、write、sendfile、lock」 和其他在我们之前描述的系统调用中。

在 VFS 下实现的实际文件系统不需要在内部使用完全相同的抽象和操作。但是，它们必须在语义上实现与 VFS 对象指定的文件系统操作相同的文件系统操作。四个 VFS 对象中每个对象的操作数据结构的元素都是指向基础文件系统中功能的指针。

Linux Ext2 文件系统

现在我们一起看一下 Linux 中最流行的一个磁盘文件系统，那就是 ext2 。Linux 的第一个版本用于 MINIX1 文件系统，它的文件名大小被限制为最大 64 MB。MINIX 1 文件系统被永远的被它的扩展系统 ext 取代，因为 ext 允许更长的文件名和文件大小。由于 ext 的性能低下，ext 被其替代者 ext2 取代，ext2 目前仍在广泛使用。

一个 ext2 Linux 磁盘分区包含了一个文件系统，这个文件系统的布局如下所示

Boot 块也就是第 0 块不是让 Linux 使用的，而是用来加载和引导计算机启动代码的。在块 0 之后，磁盘分区被分成多个组，这些组与磁盘柱面边界所处的位置无关。

第一个块是 超级块(superblock)。它包含有关文件系统布局的信息，包括 i-node、磁盘块数量和以及空闲磁盘块列表的开始。下一个是 组描述符(group descriptor)，其中包含有关位图的位置，组中空闲块和 i-node 的数量以及组中的目录数量的信息。这些信息很重要，因为 ext2 会在磁盘上均匀分布目录。

图中的两个位图用来记录空闲块和空闲 i-node，这是从 MINIX 1文件系统继承的选择，大多数 UNIX 文件系统使用位图而不是空闲列表。每个位图的大小是一个块。如果一个块的大小是 1 KB，那么就限制了块组的数量是 8192 个块和 8192 个 i-node。块的大小是一个严格的限制，块组的数量不固定，在 4KB 的块中，块组的数量增大四倍。

在超级块之后分布的是 i-node 它们自己，i-node 取值范围是 1 - 某些最大值。每个 i-node 是 128 字节的 long ，这些字节恰好能够描述一个文件。i-node 包含了统计信息(包含了 stat 系统调用能获得的所有者信息，实际上 stat 就是从 i-node 中读取信息的)，以及足够的信息来查找保存文件数据的所有磁盘块。

在 i-node 之后的是 数据块(data blocks)。所有的文件和目录都保存在这。如果一个文件或者目录包含多个块，那么这些块在磁盘中的分布不一定是连续的，也有可能不连续。事实上，大文件块可能会被拆分成很多小块散布在整个磁盘上。

对应于目录的 i-node 分散在整个磁盘组上。如果有足够的空间，ext2 会把普通文件组织到与父目录相同的块组中，而把同一块上的数据文件组织成初始 i-node节点。位图用来快速确定新文件系统数据的分配位置。在分配新的文件块时，ext2 也会给该文件预分配许多额外的数据块，这样可以减少将来向文件写入数据时产生的文件碎片。这种策略在整个磁盘上实现了文件系统的 负载，后续还有对文件碎片的排列和整理，而且性能也比较好。

为了达到访问的目的，需要首先使用 Linux 系统调用，例如 open，这个系统调用会确定打开文件的路径。路径分为两种，相对路径 和 绝对路径。如果使用相对路径，那么就会从当前目录开始查找，否则就会从根目录进行查找。

目录文件的文件名最高不能超过 255 个字符，它的分配如下图所示

每一个目录都由整数个磁盘块组成，这样目录就可以整体的写入磁盘。在一个目录中，文件和子目录的目录项都是未经排序的，并且一个挨着一个。目录项不能跨越磁盘块，所以通常在每个磁盘块的尾部会有部分未使用的字节。

上图中每个目录项都由四个固定长度的属性和一个长度可变的属性组成。第一个属性是 i-node 节点数量，文件 first 的 i-node 编号是 19 ，文件 second 的编号是 42，目录 third 的 i-node 编号是 88。紧随其后的是 rec\_len 域，表明目录项大小是多少字节，名称后面会有一些扩展，当名字以未知长度填充时，这个域被用来寻找下一个目录项，直至最后的未使用。这也是图中箭头的含义。紧随其后的是 类型域：F 表示的是文件，D 表示的是目录，最后是固定长度的文件名，上面的文件名的长度依次是 5、6、5，最后以文件名结束。

rec\_len 域是如何扩展的呢？如下图所示

我们可以看到，中间的 second 被移除了，所以将其所在的域变为第一个目录项的填充。当然，这个填充可以作为后续的目录项。

由于目录是按照线性的顺序进行查找的，因此可能需要很长时间才能在大文件末尾找到目录项。因此，系统会为近期的访问目录维护一个缓存。这个缓存用文件名来查找，如果缓存命中，那么就会避免线程搜索这样昂贵的开销。组成路径的每个部分都在目录缓存中保存一个 dentry 对象，并且通过 i-node 找到后续的路径元素的目录项，直到找到真正的文件 i - node。

比如说要使用绝对路径来寻找一个文件，我们暂定这个路径是 /usr/local/file，那么需要经过如下几个步骤：

* 首先，系统会确定根目录，它通常使用 2 号 i -node ，也就是索引 2 节点，因为索引节点 1 是 ext2 /3/4 文件系统上的坏块索引节点。系统会将一项放在 dentry 缓存中，以应对将来对根目录的查找。
* 然后，在根目录中查找字符串 usr，得到 /usr 目录的 i - node 节点号。/usr 的 i - node 同样也进入 dentry 缓存。然后节点被取出，并从中解析出磁盘块，这样就可以读取 /usr 目录并查找字符串 local 了。一旦找到这个目录项，目录 /usr/local 的 i - node 节点就可以从中获得。有了 /usr/local 的 i - node 节点号，就可以读取 i - node 并确定目录所在的磁盘块。最后，从 /usr/local 目录查找 file 并确定其 i - node 节点呢号。

如果文件存在，那么系统会提取 i - node 节点号并把它作为索引在 i - node 节点表中定位相应的 i - node 节点并装入内存。i - node 被存放在 i - node 节点表(i-node table) 中，节点表是一个内核数据结构，它会持有当前打开文件和目录的 i - node 节点号。下面是一些 Linux 文件系统支持的 i - node 数据结构。

| 属性 | 字节 | 描述 |
| --- | --- | --- |
| Mode | 2 | 文件属性、保护位、setuid 和 setgid 位 |
| Nlinks | 2 | 指向 i - node 节点目录项的数目 |
| Uid | 2 | 文件所有者的 UID |
| Gid | 2 | 文件所有者的 GID |
| Size | 4 | 文件字节大小 |
| Addr | 60 | 12 个磁盘块以及后面 3 个间接块的地址 |
| Gen | 1 | 每次重复使用 i - node 时增加的代号 |
| Atime | 4 | 最近访问文件的时间 |
| Mtime | 4 | 最近修改文件的时间 |
| Ctime | 4 | 最近更改 i - node 的时间 |

现在我们来一起探讨一下文件读取过程，还记得 read 函数是如何调用的吗？

n = read(fd,buffer,nbytes);

当内核接管后，它会从这三个参数以及内部表与用户有关的信息开始。内部表的其中一项是文件描述符数组。文件描述符数组用文件描述符 作为索引并为每一个打开文件保存一个表项。

文件是和 i - node 节点号相关的。那么如何通过一个文件描述符找到文件对应的 i - node 节点呢？

这里使用的一种设计思想是在文件描述符表和 i - node 节点表之间插入一个新的表，叫做 打开文件描述符(open-file-description table)。文件的读写位置会在打开文件描述符表中存在，如下图所示

我们使用 shell 、P1 和 P2 来描述一下父进程、子进程、子进程的关系。Shell 首先生成 P1，P1 的数据结构就是 Shell 的一个副本，因此两者都指向相同的打开文件描述符的表项。当 P1 运行完成后，Shell 的文件描述符仍会指向 P1 文件位置的打开文件描述。然后 Shell 生成了 P2，新的子进程自动继承文件的读写位置，甚至 P2 和 Shell 都不知道文件具体的读写位置。

上面描述的是父进程和子进程这两个 相关 进程，如果是一个不相关进程打开文件时，它将得到自己的打开文件描述符表项，以及自己的文件读写位置，这是我们需要的。

❝

因此，打开文件描述符相当于是给相关进程提供同一个读写位置，而给不相关进程提供各自私有的位置。

❞

i - node 包含三个间接块的磁盘地址，它们每个指向磁盘块的地址所能够存储的大小不一样。

Linux Ext4 文件系统

为了防止由于系统崩溃和电源故障造成的数据丢失，ext2 系统必须在每个数据块创建之后立即将其写入到磁盘上，磁盘磁头寻道操作导致的延迟是无法让人忍受的。为了增强文件系统的健壮性，Linux 依靠日志文件系统，ext3 是一个日志文件系统，它在 ext2 文件系统的基础之上做了改进，ext4 也是 ext3 的改进，ext4 也是一个日志文件系统。ext4 改变了 ext3 的块寻址方案，从而支持更大的文件和更大的文件系统大小。下面我们就来描述一下 ext4 文件系统的特性。

具有记录的文件系统最基本的功能就是记录日志，这个日志记录了按照顺序描述所有文件系统的操作。通过顺序写出文件系统数据或元数据的更改，操作不受磁盘访问期间磁盘头移动的开销。最终，这个变更会写入并提交到合适的磁盘位置上。如果这个变更在提交到磁盘前文件系统宕机了，那么在重启期间，系统会检测到文件系统未正确卸载，那么就会遍历日志并应用日志的记录来对文件系统进行更改。

Ext4 文件系统被设计用来高度匹配 ext2 和 ext3 文件系统的，尽管 ext4 文件系统在内核数据结构和磁盘布局上都做了变更。尽管如此，一个文件系统能够从 ext2 文件系统上卸载后成功的挂载到 ext4 文件系统上，并提供合适的日志记录。

日志是作为循环缓冲区管理的文件。日志可以存储在与主文件系统相同或者不同的设备上。日志记录的读写操作会由单独的 JBD(Journaling Block Device) 来扮演。

JBD 中有三个主要的数据结构，分别是 「log record(日志记录)、原子操作和事务」。一个日志记录描述了一个低级别的文件系统操作，这个操作通常导致块内的变化。因为像是 write 这种系统调用会包含多个地方的改动 --- i - node 节点，现有的文件块，新的文件块和空闲列表等。相关的日志记录会以原子性的方式分组。ext4 会通知系统调用进程的开始和结束，以此使 JBD 能够确保原子操作的记录都能被应用，或者一个也不被应用。最后，主要从效率方面考虑，JBD 会视原子操作的集合为事务。一个事务中的日志记录是连续存储的。只有在所有的变更一起应用到磁盘后，日志记录才能够被丢弃。

由于为每个磁盘写出日志的开销会很大，所以 ext4 可以配置为保留所有磁盘更改的日志，或者仅仅保留与文件系统元数据相关的日志更改。仅仅记录元数据可以减少系统开销，提升性能，但不能保证不会损坏文件数据。其他的几个日志系统维护着一系列元数据操作的日志，例如 SGI 的 XFS。

/proc 文件系统

另外一个 Linux 文件系统是 /proc (process) 文件系统

❝

它的主要思想来源于贝尔实验室开发的第 8 版的 UNIX，后来被 BSD 和 System V 采用。

❞

然而，Linux 在一些方面上对这个想法进行了扩充。它的基本概念是为系统中的每个进程在 /proc 中创建一个目录。目录的名字就是进程 PID，以十进制数进行表示。例如，/proc/1024 就是一个进程号为 1024 的目录。在该目录下是进程信息相关的文件，比如进程的命令行、环境变量和信号掩码等。事实上，这些文件在磁盘上并不存在磁盘中。当需要这些信息的时候，系统会按需从进程中读取，并以标准格式返回给用户。

许多 Linux 扩展与 /proc 中的其他文件和目录有关。它们包含各种各样的关于 CPU、磁盘分区、设备、中断向量、内核计数器、文件系统、已加载模块等信息。非特权用户可以读取很多这样的信息，于是就可以通过一种安全的方式了解系统情况。

NFS 网络文件系统

从一开始，网络就在 Linux 中扮演了很重要的作用。下面我们会探讨一下 NFS(Network File System) 网络文件系统，它在现代 Linux 操作系统的作用是将不同计算机上的不同文件系统链接成一个逻辑整体。

NFS 架构

NFS 最基本的思想是允许任意选定的一些客户端和服务器共享一个公共文件系统。在许多情况下，所有的客户端和服务器都会在同一个 LAN(Local Area Network)  局域网内共享，但是这并不是必须的。也可能是下面这样的情况：如果客户端和服务器距离较远，那么它们也可以在广域网上运行。客户端可以是服务器，服务器可以是客户端，但是为了简单起见，我们说的客户端就是消费服务，而服务器就是提供服务的角度来聊。

每一个 NFS 服务都会导出一个或者多个目录供远程客户端访问。当一个目录可用时，它的所有子目录也可用。因此，通常整个目录树都会作为一个整体导出。服务器导出的目录列表会用一个文件来维护，这个文件是 /etc/exports，当服务器启动后，这些目录可以自动的被导出。客户端通过挂载这些导出的目录来访问它们。当一个客户端挂载了一个远程目录，这个目录就成为客户端目录层次的一部分，如下图所示。

在这个示例中，一号客户机挂载到服务器的 bin 目录下，因此它现在可以使用 shell 访问 /bin/cat 或者其他任何一个目录。同样，客户机 1 也可以挂载到 二号服务器上从而访问 /usr/local/projects/proj1 或者其他目录。二号客户机同样可以挂载到二号服务器上，访问路径是 /mnt/projects/proj2。

从上面可以看到，由于不同的客户端将文件挂载到各自目录树的不同位置，同一个文件在不同的客户端有不同的访问路径和不同的名字。挂载点一般通常在客户端本地，服务器不知道任何一个挂载点的存在。

NFS 协议

由于 NFS 的协议之一是支持 异构 系统，客户端和服务器可能在不同的硬件上运行不同的操作系统，因此有必要在服务器和客户端之间进行接口定义。这样才能让任何写一个新客户端能够和现有的服务器一起正常工作，反之亦然。

NFS 就通过定义两个客户端 - 服务器协议从而实现了这个目标。协议就是客户端发送给服务器的一连串的请求，以及服务器发送回客户端的相应答复。

第一个 NFS 协议是处理挂载。客户端可以向服务器发送路径名并且请求服务器是否能够将服务器的目录挂载到自己目录层次上。因为服务器不关心挂载到哪里，因此请求不会包含挂载地址。如果路径名是合法的并且指定的目录已经被导出，那么服务器会将文件 句柄 返回给客户端。

❝

文件句柄包含唯一标识文件系统类型，磁盘，目录的i节点号和安全性信息的字段。

❞

随后调用读取和写入已安装目录或其任何子目录中的文件，都将使用文件句柄。

当 Linux 启动时会在多用户之前运行 shell 脚本 /etc/rc 。可以将挂载远程文件系统的命令写入该脚本中，这样就可以在允许用户登陆之前自动挂载必要的远程文件系统。大部分 Linux 版本是支持自动挂载的。这个特性会支持将远程目录和本地目录进行关联。

相对于手动挂载到 /etc/rc 目录下，自动挂载具有以下优势

* 如果列出的 /etc/rc 目录下出现了某种故障，那么客户端将无法启动，或者启动会很困难、延迟或者伴随一些出错信息，如果客户根本不需要这个服务器，那么手动做了这些工作就白费了。
* 允许客户端并行的尝试一组服务器，可以实现一定程度的容错率，并且性能也可以得到提高。

另一方面，我们默认在自动挂载时所有可选的文件系统都是相同的。由于 NFS 不提供对文件或目录复制的支持，用户需要自己确保这些所有的文件系统都是相同的。因此，大部分的自动挂载都只应用于二进制文件和很少改动的只读的文件系统。

第二个 NFS 协议是为文件和目录的访问而设计的。客户端能够通过向服务器发送消息来操作目录和读写文件。客户端也可以访问文件属性，比如文件模式、大小、上次修改时间。NFS 支持大多数的 Linux 系统调用，但是 open 和 close 系统调用却不支持。

❝

不支持 open 和 close 并不是一种疏忽，而是一种刻意的设计，完全没有必要在读一个文件之前对其进行打开，也没有必要在读完时对其进行关闭。

❞

NFS 使用了标准的 UNIX 保护机制，使用 rwx 位来标示所有者(owner)、组(groups)、其他用户 。最初，每个请求消息都会携带调用者的 groupId 和 userId，NFS 会对其进行验证。事实上，它会信任客户端不会发生欺骗行为。可以使用公钥密码来创建一个安全密钥，在每次请求和应答中使用它验证客户端和服务器。

NFS 实现

即使客户端和服务器的代码实现是独立于 NFS 协议的，大部分的 Linux 系统会使用一个下图的三层实现，顶层是系统调用层，系统调用层能够处理 open 、 read 、 close 这类的系统调用。在解析和参数检查结束后调用第二层，虚拟文件系统 (VFS) 层。

VFS 层的任务是维护一个表，每个已经打开的文件都在表中有一个表项。VFS 层为每一个打开的文件维护着一个虚拟i节点，简称为 v - node。v 节点用来说明文件是本地文件还是远程文件。如果是远程文件的话，那么 v - node 会提供足够的信息使客户端能够访问它们。对于本地文件，会记录其所在的文件系统和文件的 i-node ，因为现代操作系统能够支持多文件系统。虽然 VFS 是为了支持 NFS 而设计的，但是现代操作系统都会使用 VFS，而不管有没有 NFS。

Linux IO

我们之前了解过了 Linux 的进程和线程、Linux 内存管理，那么下面我们就来认识一下 Linux 中的 I/O 管理。

Linux 系统和其他 UNIX 系统一样，IO 管理比较直接和简洁。所有 IO 设备都被当作文件，通过在系统内部使用相同的 read 和 write 一样进行读写。

Linux IO 基本概念

Linux 中也有磁盘、打印机、网络等 I/O 设备，Linux 把这些设备当作一种 特殊文件 整合到文件系统中，一般通常位于 /dev 目录下。可以使用与普通文件相同的方式来对待这些特殊文件。

特殊文件一般分为两种：

块特殊文件是一个能存储固定大小块信息的设备，它支持「以固定大小的块，扇区或群集读取和（可选）写入数据」。每个块都有自己的物理地址。通常块的大小在 512 - 65536 之间。所有传输的信息都会以连续的块为单位。块设备的基本特征是每个块都较为对立，能够独立的进行读写。常见的块设备有 「硬盘、蓝光光盘、USB 盘」与字符设备相比，块设备通常需要较少的引脚。

块特殊文件的缺点基于给定固态存储器的块设备比基于相同类型的存储器的字节寻址要慢一些，因为必须在块的开头开始读取或写入。所以，要读取该块的任何部分，必须寻找到该块的开始，读取整个块，如果不使用该块，则将其丢弃。要写入块的一部分，必须寻找到块的开始，将整个块读入内存，修改数据，再次寻找到块的开头处，然后将整个块写回设备。

另一类 I/O 设备是字符特殊文件。字符设备以字符为单位发送或接收一个字符流，而不考虑任何块结构。字符设备是不可寻址的，也没有任何寻道操作。常见的字符设备有 「打印机、网络设备、鼠标、以及大多数与磁盘不同的设备」。

每个设备特殊文件都会和 设备驱动 相关联。每个驱动程序都通过一个 主设备号来标识。如果一个驱动支持多个设备的话，此时会在主设备的后面新加一个 次设备号 来标识。主设备号和次设备号共同确定了唯一的驱动设备。

我们知道，在计算机系统中，CPU 并不直接和设备打交道，它们中间有一个叫作 设备控制器（Device Control Unit）的组件，例如硬盘有磁盘控制器、USB 有 USB 控制器、显示器有视频控制器等。这些控制器就像代理商一样，它们知道如何应对硬盘、鼠标、键盘、显示器的行为。

绝大多数字符特殊文件都不能随机访问，因为他们需要使用和块特殊文件不同的方式来控制。比如，你在键盘上输入了一些字符，但是你发现输错了一个，这时有一些人喜欢使用 backspace 来删除，有人喜欢用 del 来删除。为了中断正在运行的设备，一些系统使用 ctrl-u 来结束，但是现在一般使用 ctrl-c 来结束。

网络

I/O 的另外一个概念是网络， 也是由 UNIX 引入，网络中一个很关键的概念就是 套接字(socket)。套接字允许用户连接到网络，正如邮筒允许用户连接到邮政系统，套接字的示意图如下

套接字的位置如上图所示，套接字可以动态创建和销毁。成功创建一个套接字后，系统会返回一个文件描述符(file descriptor)，在后面的创建链接、读数据、写数据、解除连接时都需要使用到这个文件描述符。每个套接字都支持一种特定类型的网络类型，在创建时指定。一般最常用的几种

* 可靠的面向连接的字节流
* 可靠的面向连接的数据包
* 不可靠的数据包传输

可靠的面向连接的字节流会使用管道 在两台机器之间建立连接。能够保证字节从一台机器按照顺序到达另一台机器，系统能够保证所有字节都能到达。

除了数据包之间的分界之外，第二种类型和第一种类型是类似的。如果发送了 3 次写操作，那么使用第一种方式的接受者会直接接收到所有字节；第二种方式的接受者会分 3 次接受所有字节。除此之外，用户还可以使用第三种即不可靠的数据包来传输，使用这种传输方式的优点在于高性能，有的时候它比可靠性更加重要，比如在流媒体中，性能就尤其重要。

以上涉及两种形式的传输协议，即 TCP 和 UDP，TCP 是 传输控制协议，它能够传输可靠的字节流。UDP 是 用户数据报协议，它只能够传输不可靠的字节流。它们都属于 TCP/IP 协议簇中的协议，下面是网络协议分层

可以看到，TCP 、UDP 都位于网络层上，可见它们都把 IP 协议 即 互联网协议作为基础。

一旦套接字在源计算机和目的计算机建立成功，那么两个计算机之间就可以建立一个链接。通信一方在本地套接字上使用 listen 系统调用，它就会创建一个缓冲区，然后阻塞直到数据到来。另一方使用 connect 系统调用，如果另一方接受 connect 系统调用后，则系统会在两个套接字之间建立连接。

socket 连接建立成功后就像是一个管道，一个进程可以使用本地套接字的文件描述符从中读写数据，当连接不再需要的时候使用 close 系统调用来关闭。

Linux I/O 系统调用

Linux 系统中的每个 I/O 设备都有一个特殊文件(special file)与之关联，什么是特殊文件呢？

❝

在操作系统中，特殊文件是一种在文件系统中与硬件设备相关联的文件。特殊文件也被称为 设备文件(device file)。特殊文件的目的是将设备作为文件系统中的文件进行公开。特殊文件为硬件设备提供了借口，用于文件 I/O 的工具可以进行访问。因为设备有两种类型，同样特殊文件也有两种，即字符特殊文件和块特殊文件

❞

对于大部分 I/O 操作来说，只用合适的文件就可以完成，并不需要特殊的系统调用。然后，有时需要一些设备专用的处理。在 POSIX 之前，大多数 UNIX 系统会有一个叫做 ioctl 的系统调用，它用于执行大量的系统调用。随着时间的发展，POSIX 对其进行了整理，把 ioctl 的功能划分为面向终端设备的独立功能调用，现在已经变成独立的系统调用了。

下面是几个管理终端的系统调用

| 系统调用 | 描述 |
| --- | --- |
| tcgetattr | 获取属性 |
| tcsetattr | 设置属性 |
| cfgetispeed | 获取输入速率 |
| cfgetospeed | 获取输出速率 |
| cfsetispeed | 设置输入速率 |
| cfsetospeed | 设置输出速率 |

Linux IO 实现

Linux 中的 IO 是通过一系列设备驱动实现的，每个设备类型对应一个设备驱动。设备驱动为操作系统和硬件分别预留接口，通过设备驱动来屏蔽操作系统和硬件的差异。

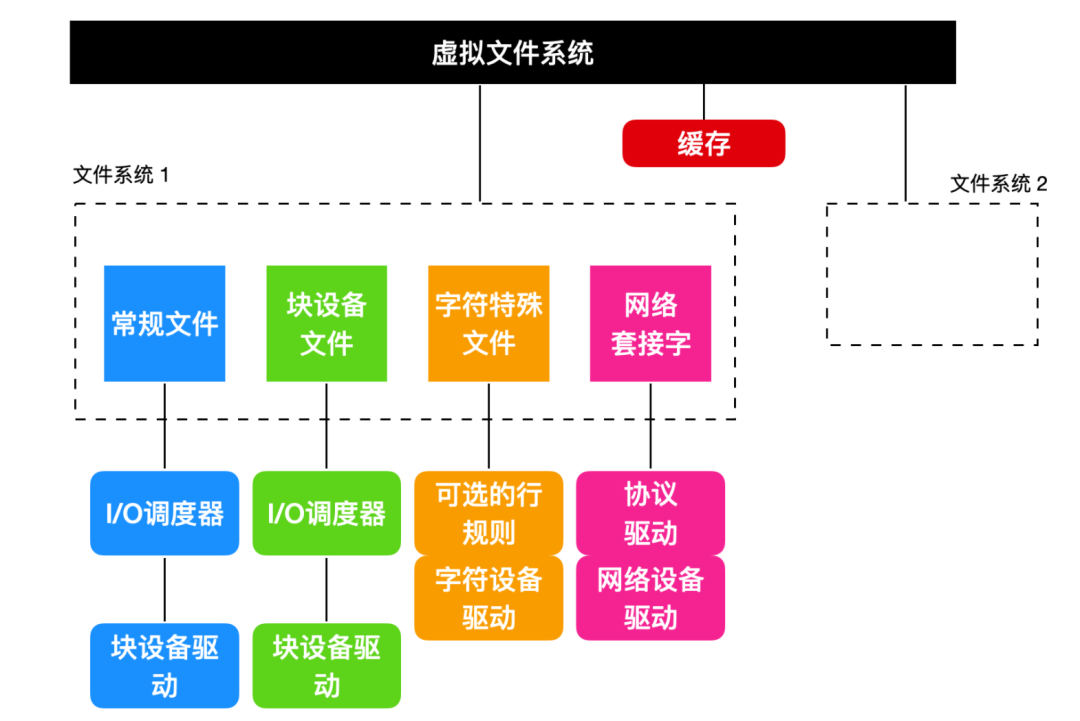
当用户访问一个特殊的文件时，由文件系统提供此特殊文件的主设备号和次设备号，并判断它是一个块特殊文件还是字符特殊文件。主设备号用于标识字符设备还是块设备，次设备号用于参数传递。

每个驱动程序 都有两部分：这两部分都是属于 Linux 内核，也都运行在内核态下。上半部分运行在调用者上下文并且与 Linux 其他部分交互。下半部分运行在内核上下文并且与设备进行交互。驱动程序可以调用内存分配、定时器管理、DMA 控制等内核过程。可被调用的内核功能都位于 驱动程序 - 内核接口 的文档中。

I/O 实现指的就是对字符设备和块设备的实现

块设备实现

系统中处理块特殊文件 I/O 部分的目标是为了使传输次数尽可能的小。为了实现这个目标，Linux 系统在磁盘驱动程序和文件系统之间设置了一个 高速缓存(cache) ，如下图所示



在 Linux 内核 2.2 之前，Linux 系统维护着两个缓存：页面缓存(page cache) 和 缓冲区缓存(buffer cache)，因此，存储在一个磁盘块中的文件可能会在两个缓存中。2.2 版本以后 Linux 内核只有一个统一的缓存一个 通用数据块层(generic block layer) 把这些融合在一起，实现了磁盘、数据块、缓冲区和数据页之间必要的转换。那么什么是通用数据块层？

❝

通用数据块层是一个内核的组成部分，用于处理对系统中所有块设备的请求。通用数据块主要有以下几个功能

将数据缓冲区放在内存高位处，当 CPU 访问数据时，页面才会映射到内核线性地址中，并且此后取消映射

实现 零拷贝机制，磁盘数据可以直接放入用户模式的地址空间，而无需先复制到内核内存中

管理磁盘卷，会把不同块设备上的多个磁盘分区视为一个分区。

利用最新的磁盘控制器的高级功能，例如 DMA 等。

❞

cache 是提升性能的利器，不管以什么样的目的需要一个数据块，都会先从 cache 中查找，如果找到直接返回，避免一次磁盘访问，能够极大的提升系统性能。

如果页面 cache 中没有这个块，操作系统就会把页面从磁盘中调入内存，然后读入 cache 进行缓存。

cache 除了支持读操作外，也支持写操作，一个程序要写回一个块，首先把它写到 cache 中，而不是直接写入到磁盘中，等到磁盘中缓存达到一定数量值时再被写入到 cache 中。

Linux 系统中使用 IO 调度器 来保证减少磁头的反复移动从而减少损失。I/O 调度器的作用是对块设备的读写操作进行排序，对读写请求进行合并。Linux 有许多调度器的变体，从而满足不同的工作需要。最基本的 Linux 调度器是基于传统的 Linux 电梯调度器(Linux elevator scheduler)。Linux 电梯调度器的主要工作流程就是按照磁盘扇区的地址排序并存储在一个双向链表 中。新的请求将会以链表的形式插入。这种方法可以有效的防止磁头重复移动。因为电梯调度器会容易产生饥饿现象。因此，Linux 在原基础上进行了修改，维护了两个链表，在 最后日期(deadline) 内维护了排序后的读写操作。默认的读操作耗时 0.5s，默认写操作耗时 5s。如果在最后期限内等待时间最长的链表没有获得服务，那么它将优先获得服务。

字符设备实现

和字符设备的交互是比较简单的。由于字符设备会产生并使用字符流、字节数据，因此对随机访问的支持意义不大。一个例外是使用 行规则(line disciplines)。一个行规可以和终端设备相关联，使用 tty\_struct 结构来表示，它表示与终端设备交换数据的解释器，当然这也属于内核的一部分。例如：行规可以对行进行编辑，映射回车为换行等一系列其他操作。

❝

什么是行规则？

行规是某些类 UNIX 系统中的一层，终端子系统通常由三层组成：上层提供字符设备接口，下层硬件驱动程序与硬件或伪终端进行交互，中层规则用于实现终端设备共有的行为。

❞



网络设备实现

网络设备的交互是不一样的，虽然 网络设备(network devices) 也会产生字符流，因为它们的异步(asynchronous) 特性是他们不易与其他字符设备在同一接口下集成。网络设备驱动程序会产生很多数据包，经由网络协议到达用户应用程序中。

Linux 中的模块

UNIX 设备驱动程序是被静态加载到内核中的。因此，只要系统启动后，设备驱动程序都会被加载到内存中。随着个人电脑 Linux 的出现，这种静态链接完成后会使用一段时间的模式被打破。相对于小型机上的 I/O 设备，PC 上可用的 I/O 设备有了数量级的增长。绝大多数用户没有能力去添加一个新的应用程序、更新设备驱动、重新连接内核，然后进行安装。

Linux 为了解决这个问题，引入了 可加载(loadable module) 机制。可加载是在系统运行时添加到内核中的代码块。

当一个模块被加载到内核时，会发生下面几件事情：第一，在加载的过程中，模块会被动态的重新部署。第二，系统会检查程序程序所需的资源是否可用。如果可用，则把这些资源标记为正在使用。第三步，设置所需的中断向量。第四，更新驱动转换表使其能够处理新的主设备类型。最后再来运行设备驱动程序。

在完成上述工作后，驱动程序就会安装完成，其他现代 UNIX 系统也支持可加载机制。

Linux 安全

Linux 作为 MINIX 和 UNIX 的衍生操作系统，从一开始就是一个多用户系统。这意味着 Linux 从早期开始就建立了安全和信息访问控制机制。下面我们主要探讨的就是 Linux 安全性的一些内容

Linux 安全基本概念

一个 Linux 系统的用户群里由一系列注册用户组成，他们每一个都有一个唯一的 UID (User ID)。一个 UID 是一个位于 0 到 65535 之间的整数。文件（进程或者是其他资源）都标记了它的所有者的 UID。默认情况下，文件的所有者是创建文件的人，文件的所有者是创建文件的用户。

用户可以被分成许多组，每个组都会由一个 16 位的整数标记，这个组叫做 GID(组 ID)。给用户分组是手动完成的，它由系统管理员执行，分组就是在数据库中添加一条记录指明哪个用户属于哪个组。一个用户可以属于不同组。

Linux 中的基本安全机制比较容易理解，每个进程都会记录它所有者的 UID 和 GID。当文件创建后，它会获取创建进程的 UID 和 GID。当一个文件被创建时，它的 UID 和 GID 就会被标记为进程的 UID 和 GID。这个文件同时会获取由该进程决定的一些权限。这些权限会指定所有者、所有者所在组的其他用户及其他用户对文件具有什么样的访问权限。对于这三类用户而言，潜在的访问权限是 「读、写和执行」，分别由 r、w 和 x 标记。当然，执行文件的权限仅当文件时可逆二进制程序时才有意义。试图执行一个拥有执行权限的非可执行文件，系统会报错。

「Linux 用户分为三种」

* root（超级管理员），它的 UID 为 0，这个用户有极大的权限，可以直接无视很多的限制 ，包括读写执行的权限。
* 系统用户，UID 为 1～499。
* 普通用户，UID 范围一般是 500～65534。这类用户的权限会受到基本权限的限制，也会受到来自管理员的限制。不过要注意 nobody 这个特殊的帐号，UID 为 65534，这个用户的权限会进一步的受到限制，一般用于实现来宾帐号。

Linux 中的每类用户由 3 个比特为来标记，所以 9 个比特位就能够表示所有的权限。

下面来看一下一些基本的用户和权限例子

| 二进制 | 标记 | 准许的文件访问权限 |
| --- | --- | --- |
| 111000000 | rwx------ | 所有者可读、写和执行 |
| 111111000 | rwxrwx--- | 所有者和组可以读、写和执行 |
| 111111111 | rwxrwxrwx | 所有人可以读、写和执行 |
| 000000000 | --------- | 任何人不拥有任何权限 |
| 000000111 | ------rwx | 只有组以外的其他用户拥有所有权 |
| 110100100 | rw-r--r-- | 所有者可以读和写，其他人可以读 |
| 110100100 | rw-r----- | 所有者可以读和写，组可以读 |

我们上面提到，UID 为 0 的是一个特殊用户，称为 超级用户(或者根用户)。超级用户能够读和写系统中的任何文件，不管这个文件由谁所有，也不管这个文件的保护模式如何。UID 为 0 的进程还具有少数调用受保护系统调用的权限，而普通用户是不可能有这些功能的。通常情况下，只有系统管理员知道超级用户的密码。

在 Linux 系统下，目录也是一种文件，并且具有和普通文件一样的保护模式。不同的是，目录的 x 比特位表示查找权限而不是执行权限。因此，如果一个目录的保护模式是 rwxr-xr-x，那么它允许所有者读、写和查找目录，而其他人只可以读和查找，而不允许从中添加或者删除目录中的文件。

与 I/O 有关的特殊文件拥有和普通文件一样的保护位。这种机制可以用来限制对 I/O 设备的访问权限。举个例子，打印机是特殊文件，它的目录是 /dev/lp，它可以被根用户或者一个叫守护进程的特殊用户拥有，具有保护模式 rw-------，从而阻止其他所有人对打印机的访问。毕竟每个人都使用打印机的话会发生混乱。

当然，如果 /dev/lp 的保护模式是 rw-------，那就意味着其他任何人都不能使用打印机。

这个问题通过增加一个保护位 SETUID 到之前的 9 个比特位来解决。当一个进程的 SETUID 位打开，它的 有效 UID 将变成相应可执行文件的所有者 UID，而不是当前使用该进程的用户的 UID。将访问打印机的程序设置为守护进程所有，同时打开 SETUID 位，这样任何用户都可以执行此程序，而且拥有守护进程的权限。

除了 SETUID 之外，还有一个 SETGID 位，SETGID 的工作原理和 SETUID 类似。但是这个位一般很不常用。

Linux 安全相关的系统调用

Linux 中关于安全的系统调用不是很多，只有几个，如下列表所示

| 系统调用 | 描述 |
| --- | --- |
| chmod | 改变文件的保护模式 |
| access | 使用真实的 UID 和 GID 测试访问权限 |
| chown | 改变所有者和组 |
| setuid | 设置 UID |
| setgid | 设置 GID |
| getuid | 获取真实的 UID |
| getgid | 获取真实的 GID |
| geteuid | 获取有效的 UID |
| getegid | 获取有效的 GID |

我们在日常开发中用到最多的就是 chmod了，没想到我们日常开发过程中也能用到系统调用啊，chmod 之前我们一直认为是改变权限，现在专业一点是改变文件的保护模式。它的具体函数如下

s = chmod("路径名","值");

例如

s = chmod("/usr/local/cxuan",777);

他就是会把 /usr/local/cxuan 这个路径的保护模式改为 rwxrwxrwx，任何组和人都可以操作这个路径。只有该文件的所有者和超级用户才有权利更改保护模式。

access 系统调用用来检验实际的 UID 和 GID 对某文件是否拥有特定的权限。下面就是四个 getxxx 的系统调用，这些用来获取 uid 和 gid 的。

❝

注意：其中的 chown、setuid 和 setgid 是超级用户才能使用，用来改变所有者进程的 UID 和 GID。

❞

Linux 安全实现

当用户登录时，登录程序，也被称为 login，会要求输入用户名和密码。它会对密码进行哈希处理，然后在 /etc/passwd 中进行查找，看看是否有匹配的项。使用哈希的原因是防止密码在系统中以非加密的方式存在。如果密码正确，登录程序会在 /etc/passwd 中读取用户选择的 shell 程序的名称，有可能是 bash，有可能是 shell 或者其他的 csh 或 ksh。然后登录程序使用 setuid 和 setgid 这两个系统调用来把自己的 UID 和 GID 变为用户的 UID 和 GID，然后它打开键盘作为标准输入、标准输入的文件描述符是 0 ，屏幕作为标准输出，文件描述符是 1 ，屏幕也作为标准错误输出，文件描述符为 2。最后，执行用户选择的 shell 程序，终止。

当任何进程想要打开一个文件，系统首先将文件的 i - node 所记录的保护位与用户有效 UID 和 有效 GID 进行对比，来检查访问是否允许。如果访问允许，就打开文件并返回文件描述符；否则不打开文件，返回 - 1。

Linux 安全模型和实现在本质上与大多数传统的 UNIX 系统相同。

后记

这篇文章从 Linux 进程线程、内存管理、文件系统、IO 管理和安全来为你呈现了一幅 Linux 蓝图，文中涉及大量的系统调用和解释，是你了解 Linux 操作系统需要仔细研读的一篇文章。

希望这篇文章能带你了解更多关于 Linux 的基本知识，如果对你有帮助，希望小伙伴们不要吝啬的为我

在看、点赞、留言、转发

拒绝白嫖、拒绝下次一定、这次一定要四连～

END