# Linux背景

## Linux发展史

Linux是一种自由和开放源代码的类UNIX操作系统，该操作系统的内核由林纳斯托瓦兹在1991年首次发布，之后，在加上用户空间的应用程序之后，就成为了Linux操作系统。严格来讲，Linux只是操作系统内核本身，但通常采用“Linux内核”来表达该意思。而Linux则常用来指基于Linux内核的完整操作系统，它包括GUI组件和许多其他实用工具。

GNU通用公共许可协议（GNU General Public License，简称GNU GPL或GPL），是一个广泛被使用的自由软件许可协议条款，最初由理查德斯托曼为GNU计划而撰写，GPL给予了计算机程序自由软件的定义，任何基于GPL软件开发衍生的产品在发布时必须采用GPL许可证方式，且必须公开源代码。

Linux是自由软件和开放源代码软件发展中最著名的例子。只要遵循GNU通用公共许可证，任何个人和机构都可以自由地使用Linux的所有底层源代码，也可以自由地修改和再发布。随着Linux操作系统飞速发展，各种集成在Linux上的开源软件和实用工具也得到了应用和普及，因此，Linux也成为了开源软件的代名词。

## Linux应用现状

* + - 1. Linux在服务器领域的发展

随着开源软件在世界范围内影响力日益增强，Linux服务器操作系统在整个服务器操作系统市场格局中 占据了越来越多的市场份额，已经形成了大规模市场应用的局面。并且保持着快速的增长率。尤其在政府、金融、农业、交通、电信等国家关键领域。此外，考虑到Linux的快速成长性以及国家相关政策的扶持力度，Linux服务器产品一定能够冲击更大的服务器市场。

据权威部门统计，目前Linux在服务器领域已经占据75%的市场份额，同时，Linux在服务器市场的迅速崛起，已经引起全球IT产业的高度关注，并以强劲的势头成为服务器操作系统领域中的中坚力量。

* + - 1. Linux在桌面领域的发展

近年来，特别在国内市场，Linux桌面操作系统的发展趋势非常迅猛。国内如中标麒麟Linux、红旗Linux、深度Linux等系统软件厂商都推出的Linux桌面操作系统，目前已经在政府、企业、OEM等领域得到了广泛应用。另外SUSE、Ubuntu也相继推出了基于Linux的桌面系统，特别是Ubuntu Linux，已经积累了大量社区用户。但是，从系统的整体功能、性能来看，Linux桌面系统与Windows系列相比还有一定的差距，主要表现在系统易用性、系统管理、软硬件兼容性、软件的丰富程度等方面。

* + - 1. Linux在移动嵌入式领域的发展

Linux的低成本、强大的定制功能以及良好的移植性能，使得Linux在嵌入式系统方面也得到广泛应用，目前Linux以广泛应用于手机、平板电脑、路由器、电视和电子游戏机等领域。在移动设备上广泛使用的Android操作系统就是创建在Linux内核之上的。目前，Android已经成为全球最流行的智能手机操作系统，据2015年权威部门最新统计，Android操作系统的全球市场份额已达84.6%。

此外，思科在网络防火墙和路由器也使用了定制的Linux，阿里云也开发了一套基于Linux的操作系统“YunOS”，可用于智能手机、平板电脑和网络电视；常见的数字视频录像机、舞台灯光控制系统等都在逐渐采用定制版本的Linux来实现，而这一切均归功与Linux与开源的力量。

* + - 1. Linux在云计算/大数据领域的发展

互联网产业的迅猛发展，促使云计算、大数据产业的形成并快速发展，云计算、大数据作为一个基于开源软件的平台，Linux占据了核心优势；据Linux基金会的研究，86%的企业已经使用Linux操作系统进行云计算、大数据平台的构建，目前，Linux已开始取代Unix成为最受青睐的云计算、大数据平台操作系统。

## 发行版本

* + - 1. Debian

Debian运行起来极其稳定，这使得它非常适合用于服务器。debian这款操作系统无疑并不适合新手用户，而是适合系统管理员和高级用户。

* + - 1. Ubuntu

Ubuntu是Debian的一款衍生版，也是当今最受欢迎的免费操作系统。Ubuntu侧重于它在这个市场的应用，在服务器、云计算、甚至一些运行Ubuntu Linux的移动设备上很常见。Ubuntu是新手用户肯定 爱不释手的一款操作系统。

* + - 1. 红帽企业级Linux

这是第一款面向商业市场的Linux发行版。它有服务器版本，支持众多处理器架构，包括x86和 x86\_64。红帽公司通过课程红帽认证系统管理员/红帽认证工程师（RHCSA/RHCE），对系统管理员进行培训和认证。

* + - 1. CentOS

CentOS是一款企业级Linux发行版，它使用红帽企业级Linux中的免费源代码重新构建而成。这款重构版完全去掉了注册商标以及Binary程序包方面一个非常细微的变化。有些人不想支付一大笔钱，又能领略红帽企业级Linux；对他们来说，CentOS值得一试。此外，CentOS的外观和行为似乎与母发行版红帽企业级Linux如出一辙。CentOS使用YUM来管理软件包。

* + - 1. Fedora

小巧的Fedora适合那些人：想尝试最先进的技术，等不及程序的稳定版出来。其实，Fedora就是红帽公司的一个测试平台；产品在成为企业级发行版之前，在该平台上进行开发和测试。Fedora是一款非常好的发行版，有庞大的用户论坛，软件库中还有为数不少的软件包。

* + - 1. Kali Linux

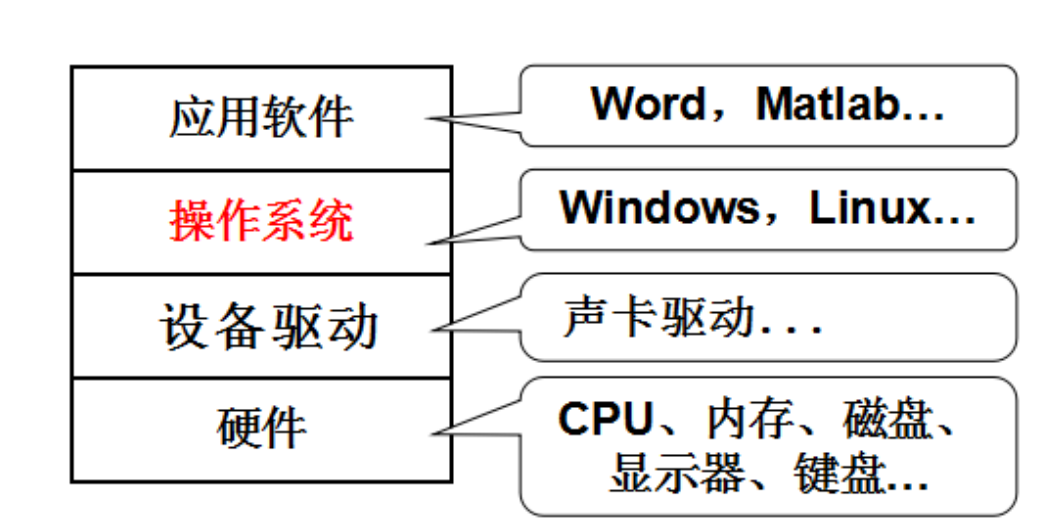
Kali Linux是Debian的一款衍生版。Kali旨在用于渗透测试。Kali的前身是Backtrack。用于Debian的所有Binary软件包都可以安装到Kali Linux上，而Kali的魅力或威力就来自于此。此外，支持Debian的用户论坛为Kali加分不少。Kali随带许多的渗透测试工具，无论是Wifi、数据库还是其他任何工具，都设计成立马可以使用。Kali使用APT来管理软件包。

毫无疑问，Kali Linux是一款渗透测试工具，或者是文明黑客青睐的操作系统。



## 定位

操作系统是一款管理软件，用来管理软硬件。对上（应用软件/用户）提供良好的运行环境，对下（设备驱动/硬件）管理好软硬件资源。使计算机更好用!这是操作系统的根本要义!



服务器为什么没有图形化界面？

进入服务器没有界面的原因是因为服务器通常是以命令行界面（CLI）的方式工作的，而不是图形用户界面（GUI）。

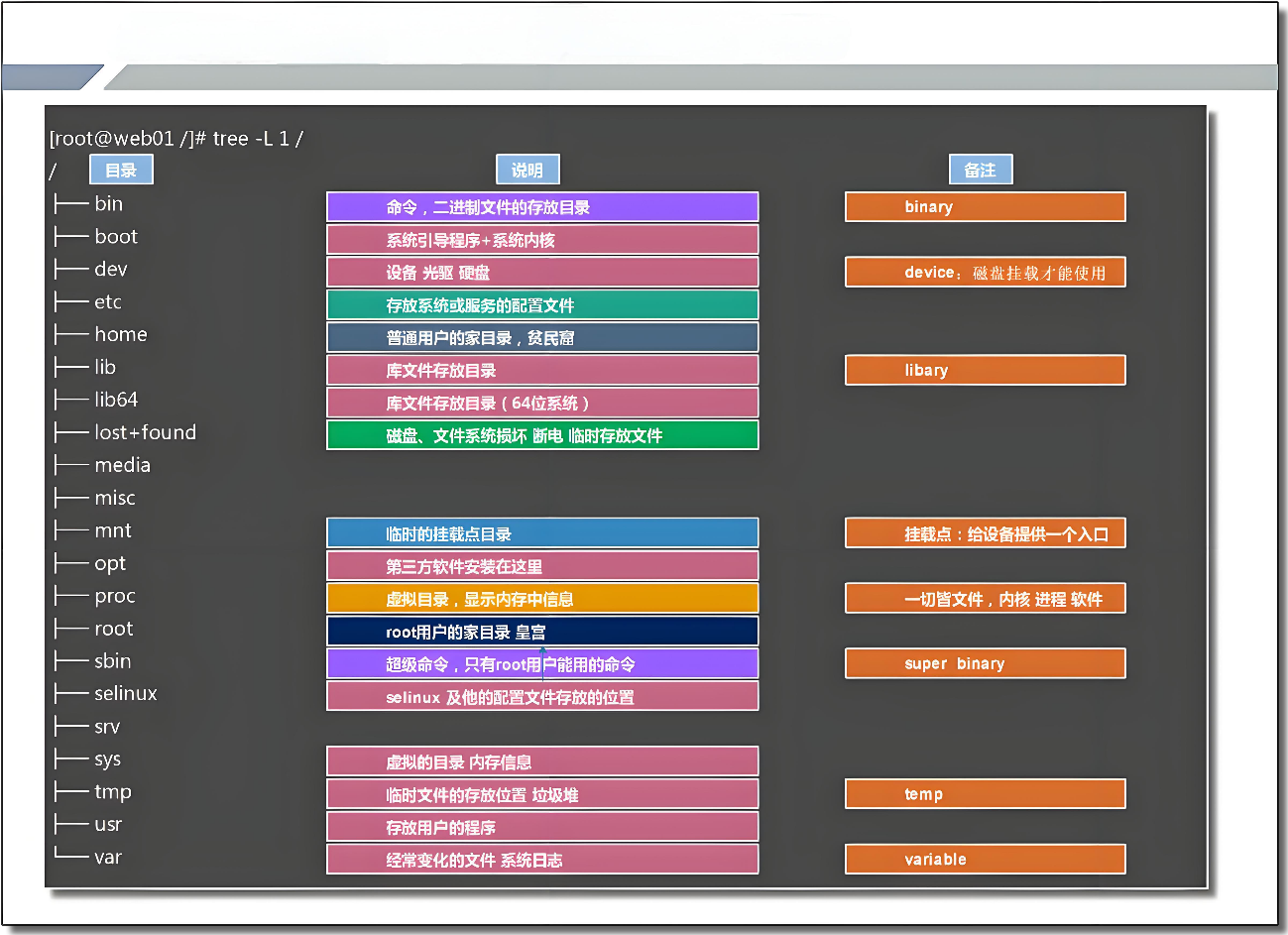
首先，服务器是为了处理大量的数据和资源，因此需要一个高效、稳定的操作环境。相比于图形界面，命令行界面更加轻量级、占用资源更少，可以提供更好的性能和可靠性。

其次，命令行界面提供了更强大的控制和管理功能。通过命令行可以执行各种操作，如安装软件、配置网络、管理用户等。命令行界面还可以自动化批量操作，提高工作效率。

另外，服务器通常是通过远程连接方式进行管理和访问的，而不是直接连接到服务器的物理界面。这意味着管理员可以通过终端或远程桌面等方式远程访问服务器，并通过命令行界面进行管理和操作。

当然，如果你希望在服务器上使用图形界面，也是可以的。可以通过安装图形界面的桌面环境，如Gnome、KDE等，来实现在服务器上使用带有界面的操作系统。但需要注意的是，这样会增加服务器的资源消耗，降低性能和可靠性。在大多数情况下，服务器管理员更倾向于使用命令行界面来管理服务器，以获得更好的性能和安全性。

总之，进入服务器没有界面是正常的，这是为了提供高效、稳定和强大的操作环境。管理员可以通过命令行界面来进行各种操作和管理任务，远程访问服务器并保持对服务器的全面控制。



# Linux常见指令及权限理解

## Linux下基本指令

注：linux下写指令操作等同于windows的基本操作。linux是命令行（CLI：Command-Line Interface）操作方式，windows是图形化用户界面（GUI：Graphical User Interface）操作方式，而这两种方式在操作系统层面来说没有区别。

* + - 1. ls指令（全称：list）

语法：ls [选项][目录或文件]

功能：对于目录，该命令列出该目录下的所有子目录与文件。对于文件，将列出文件名以及其他信息。

常用选项：

* -a 列出目录下的所有文件，包括以 . 开头的隐含文件。
* -d 将目录像文件一样显示，而不是显示其下的文件。如：ls –d 指定目录
* -i 输出文件的 i 节点的索引信息。 如 ls –ai 指定文件
* -k 以 k 字节的形式表示文件的大小。ls –alk 指定文件
* -l 列出文件的详细信息。
* -n 用数字的 UID,GID 代替名称。 （介绍 UID， GID）
* -F 在每个文件名后附上一个字符以说明该文件的类型，“\*”表示可执行的普通文件；“/”表示目录；“@”表 示符号链接；“|”表示FIFOs；“=”表示套接字(sockets)。（目录类型识别）
* -r 对目录反向排序。
* -t 以时间排序。
* -s 在I文件名后输出该文件的大小。（大小排序，如何找到目录下最大的文件）
* -R 列出所有子目录下的文件。(递归)
* -1 一行只输出一个文件

注：关于文件

* 文件=文件内容（数据）+文件属性（元数据）
* 操作文件：操作文件内容和操作文件属性
  + - 1. pwd命令（全称：print working directory）

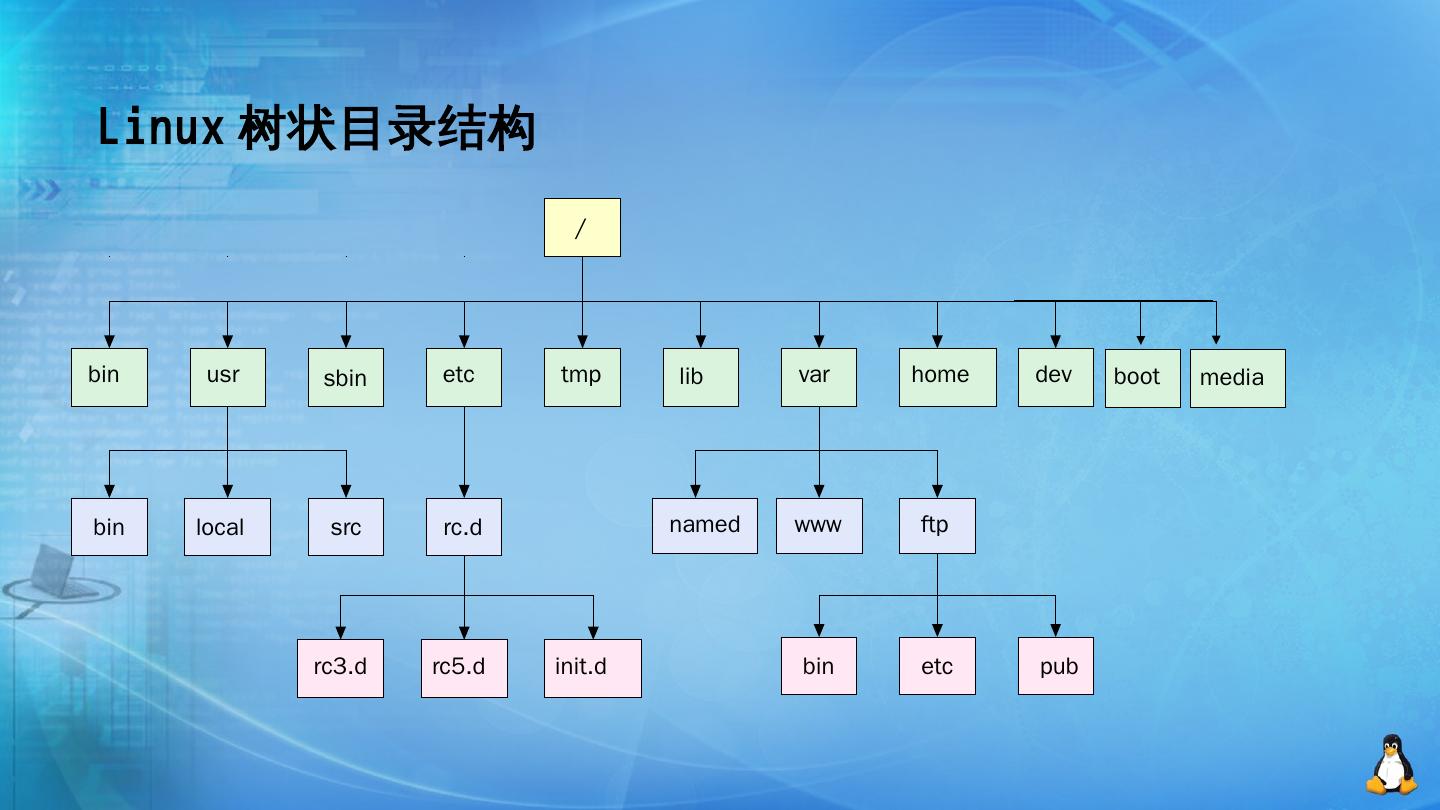
语法: pwd

功能：显示用户当前所在的目录

常用选项：无

* + - 1. cd指令（全称：change directory）

Linux系统中，磁盘上的文件和目录被组成一棵目录树，每个节点都是目录或文件。



注：几乎任何操作系统文件的目录组织都是一颗多叉树

* 叶子节点：普通文件或空目录
* 路上节点：只能是目录
* 路径+文件名 = 具有唯一性

语法:cd 目录名

功能：改变工作目录。将当前工作目录改变到指定的目录下。

举例：

|  |
| --- |
| cd .. : 返回上级目录  cd /home/litao/linux/ : 绝对路径  cd ../day02/ : 相对路径  cd ~：进入用户家目  cd -：返回最近访问目录 |

* + - 1. touch指令

语法:touch [选项] ...文件...

功能：touch命令参数可更改文档或目录的日期时间，包括存取时间和更改时间，或者新建一个不存在的文件。

常用选项：

* -a 或--time=atime或--time=access或--time=use只更改存取时间。
* -c 或--no-create 不建立任何文档。
* -d 使用指定的日期时间，而非现在的时间。
* -f 此参数将忽略不予处理，仅负责解决BSD版本touch指令的兼容性问题。
* -m 或--time=mtime或--time=modify 只更改变动时间。
* -r 把指定文档或目录的日期时间，统统设成和参考文档或目录的日期时间相同。
* -t 使用指定的日期时间，而非现在的时间。
  + - 1. mkdir指令（全称：make directory）

语法：mkdir [选项] dirname...

功能：在当前目录下创建一个名为 “dirname”的目录

常用选项：

* -p, --parents 可以是一个路径名称。此时若路径中的某些目录尚不存在,加上此选项后,系统将自动建立好那些尚不存在的目录,即一次可以建立多个目录;

举例：

|  |
| --- |
| mkdir –p test/test1 : 递归建立多个目录 |

* + - 1. rmdir指令 && rm 指令（重要）（全称：remove）

rmdir是一个与mkdir相对应的命令。mkdir是建立目录，而rmdir是删除命令。

语法：rmdir [-p][dirName]

适用对象：具有当前目录操作权限的所有使用者

功能：删除空目录

常用选项：

|  |
| --- |
| -p 当子目录被删除后如果父目录也变成空目录的话，就连带父目录一起删除。 |

rm命令可以同时删除文件或目录

语法：rm [-f-i-r-v][dirName/dir]

适用对象：所有使用者

功能：删除文件或目录

常用选项：

* -f 即使文件属性为只读(即写保护)，亦直接删除
* -i 删除前逐一询问确认
* -r 删除目录及其下所有文件
  + - 1. man指令（重要）：

Linux的命令有很多参数，我们不可能全记住，我们可以通过查看联机手册获取帮助。访问Linux手册页的命令是 man

语法: man [选项] 命令

常用选项：

* -k 根据关键字搜索联机帮助
* num 只在第num章节找
* -a 将所有章节的都显示出来，比如 man printf 它缺省从第一章开始搜索，知道就停止，用a选项，当按下q退出，他会继续往后面搜索，直到所有章节都搜索完毕。

注：解释一下,面手册分为8章。1是普通的命令；2是系统调用，如open、write之类的(通过这个，至少可以很方便的查到调用这个函数，需要加什么头文件)；3是库函数，如printf、fread；4是特殊文件，也就是/dev下的各种设备文件；5是指文件的格式，比如passwd，就会说明这个文件中各个字段的含义；6是给游戏留的，由各个游戏自己定义；7是附件还有一些变量，比如向environ这种全局变量在这里就有说明；8是系统管理用的命令，这些命令只能由root使用，如ifconfig。

* + - 1. cp指令（重要）（全称：copy）

语法：cp [选项] 源文件或目录 目标文件或目录

功能: 复制文件或目录

说明: cp指令用于复制文件或目录，如同时指定两个以上的文件或目录，且最后的目的地是一个已经存在的目录，则它会把前面指定的所有文件或目录复制到此目录中。若同时指定多个文件或目录，而最后的目的地并非一个已存在的目录，则会出现错误信息

常用选项：

* -f 或 –force 强行复制文件或目录，不论目的文件或目录是否已经存在
* -i 或 --interactive 覆盖文件之前先询问用户
* -r递归处理，将指定目录下的文件与子目录一并处理。若源文件或目录的形态，不属于目录或符号链接，则一律视为普通文件处理
* -R 或 --recursive递归处理，将指定目录下的文件及子目录一并处理
  + - 1. mv指令（重要）（全称：move）

mv命令是move的缩写，可以用来移动文件或者将文件改名，是Linux系统下常用的命令，经常用来备份文件或者目录。

语法: mv [选项] 源文件或目录 目标文件或目录

功能:

1. 视mv命令中第二个参数类型的不同（是目标文件还是目标目录），mv命令将文件重命名或将其移至一个新的目录中。
2. 当第二个参数类型是文件时，mv命令完成文件重命名，此时，源文件只能有一个（也可以是源目录名），它将所给的源文件或目录重命名为给定的目标文件名。
3. 当第二个参数是已存在的目录名称时，源文件或目录参数可以有多个，mv命令将各参数指定的源文件均移至目标目录中。

常用选项：

* -f ：force强制的意思，如果目标文件已经存在，不会询问而直接覆盖
* -i ：若目标文件(destination)已经存在时，就会询问是否覆盖！
  + - 1. cat指令（全称：concatenate）

语法：cat [选项][文件]

功能：查看目标文件内容（连接文件并打印到标准输出设备上）

常用选项：

* -b 对非空输出行编号
* -n 对输出的所有行编号
* -s 不输出多行空行
  + - 1. more指令

语法：more [选项][文件]

功能：more命令，功能类似 cat

常用选项：

* -n 对输出的所有行编号
* q 退出more
  + - 1. less指令（重要）

less工具也是对文件或其它输出进行分页显示的工具，应该说是linux正统查看文件内容的工具，功能极其强大。less的用法比起more更加的有弹性。在more的时候，我们并没有办法向前面翻，只能往后面看但若使用了less时，就可以使用[pageup][pagedown]等按键的功能来往前往后翻看文件，更容易用来查看一个文件的内容！除此之外，在less里头可以拥有更多的搜索功能，不止可以向下搜，也可以向上搜。

语法： less [参数] 文件

功能： less与more类似，但使用less可以随意浏览文件，而more仅能向前移动，却不能向后移动，而且less在查看之前不会加载整个文件。

选项：

* -i 忽略搜索时的大小写
* -N 显示每行的行号
* /字符串：向下搜索“字符串”的功能
* ?字符串：向上搜索“字符串”的功能
* n：重复前一个搜索（与/或?有关）
* N：反向重复前一个搜索（与/或?有关）
* q:quit
  + - 1. head指令

head与tail就像它的名字一样的浅显易懂，它是用来显示开头或结尾某个数量的文字区块，head用来显示档案的开头至标准输出中，而tail想当然尔就是看档案的结尾。

语法： head[参数]...[文件]...

功能： head用来显示档案的开头至标准输出中，默认head命令打印其相应文件的开头10行。

选项： -n显示的行数

* + - 1. tail指令

tail命令从指定点开始将文件写到标准输出。使用tail命令的-f选项可以方便的查阅正在改变的日志文件，tail -f filename会把filename里最尾部的内容显示在屏幕上,并且不但刷新,使你看到最新的文件内容。

语法：tail[必要参数][选择参数][文件]

功能：用于显示指定文件末尾内容，不指定文件时，作为输入信息进行处理。常用查看日志文件。

选项：

* -f 循环读取
* -n 显示行数
  + - 1. 时间相关的指令

date显示

date 指定格式显示时间： date +%Y:%m:%d

date 用法：date [OPTION]... [+FORMAT]

1. 在显示方面，使用者可以设定欲显示的格式，格式设定为一个加号后接数个标记，其中常用的标记列表如下

* %H : 小时(00..23)
* %M : 分钟(00..59)
* %S : 秒(00..60)
* %X : 相当于 %H:%M:%S
* %d : 日 (01..31)
* %m : 月份 (01..12)
* %Y : 完整年份 (0000..9999)
* %F : 相当于 %Y-%m-%d

1. 在设定时间方面

* date -s //设置当前时间，只有root权限才能设置，其他只能查看。
* date -s 20080523 //设置成20080523，这样会把具体时间设置成空00:00:00
* date -s 01:01:01 //设置具体时间，不会对日期做更改
* date -s “01:01:01 2008-05-23″ //这样可以设置全部时间
* date -s “01:01:01 20080523″ //这样可以设置全部时间
* date -s “2008-05-23 01:01:01″ //这样可以设置全部时间
* date -s “20080523 01:01:01″ //这样可以设置全部时间

1. 时间戳

* 时间->时间戳：date +%s
* 时间戳->时间：date -d@1508749502

注：Unix时间戳（英文为Unix epoch, Unix time, POSIX time 或 Unix timestamp）是从1970年1月1日（UTC/GMT的午夜）开始所经过的秒数，不考虑闰秒。

* + - 1. Cal指令（全称：Calendar）

cal命令可以用来显示公历（阳历）日历。公历是现在国际通用的历法，又称格列历，通称阳历。“阳历”又名“太阳 历”，系以地球绕行太阳一周为一年，为西方各国所通用，故又名“西历”。

命令格式：cal [参数][月份][年份]

功能：用于查看日历等时间信息，如只有一个参数，则表示年份(1-9999)，如有两个参数，则表示月份和年份

常用选项：

* -3显示系统前一个月，当前月，下一个月的月历
* -j 显示在当年中的第几天（一年日期按天算，从1月1号算起，默认显示当前月在一年中的天数）
* -y 显示当前年份的日历
  + - 1. find指令

Linux下find命令在目录结构中搜索文件，并执行指定的操作。Linux下find命令提供了相当多的查找条件，功能很强大。由于find具有强大的功能，所以它的选项也很多，其中大部分选项都值得我们花时间来了解一下。即使系统中含有网络文件系统( NFS)，find命令在该文件系统中同样有效，只你具有相应的权限。在运行一个非常消耗资源的find命令时，很多人都倾向于把它放在后台执行，因为遍历一个大的文件系统可能会花费很长的时间(这里是指30G字节以上的文件系统)。

语法：find pathname -options

功能：用于在文件树种查找文件，并作出相应的处理（可能访问磁盘）

常用选项：-name 按照文件名查找文件。

* + - 1. grep指令（全称：global regular expression）

语法： grep [选项] 搜寻字符串文件

功能： 在文件中搜索字符串，将找到的行打印出来

常用选项：

* -i ：忽略大小写的不同，所以大小写视为相同
* -n ：顺便输出行号
* -v ：反向选择，亦即显示出没有'搜寻字符串'内容的那一行
  + - 1. zip/unzip指令

语法：zip 压缩文件.zip目录或文件

功能：将目录或文件压缩成zip格式

常用选项：-r 递归处理，将指定目录下的所有文件和子目录一并处理

* + - 1. tar指令（重要）（全称：tape archive）

语法：tar [-cxtzjvf] 文件与目录....

功能：打包/解包，不打开它，直接看内容

常用选项：

* -c：建立一个压缩文件的参数指令(create 的意思)；
* -x：解开一个压缩文件的参数指令！
* -t：查看tarfile里面的文件！
* -z：是否同时具有gzip的属性？亦即是否需要用gzip压缩？
* -j：是否同时具有bzip2的属性？亦即是否需要用bzip2压缩？
* -v：压缩的过程中显示文件！这个常用，但不建议用在背景执行过程！
* -f：使用档名，请留意，在f之后要立即接档名喔！不要再加参数！
* -C：解压到指定目录
  + - 1. bc指令（全称：basic calculator）

语法：bc [options] 文件

功能：进行浮点运算

常用选项：

* -l mathlib, 使用标准数学库, 例如使用内置函数就需要使用这个参数
* -q 退出，不显示欢迎信息
  + - 1. uname -r指令（全称：unix name）

语法：uname [选项]

功能：uname用来获取电脑和操作系统的相关信息。

补充说明：uname可显示linux主机所用的操作系统的版本、硬件的名称等基本信息。

常用选项：-a或–all详细输出所有信息，依次为内核名称，主机名，内核版本号，内核版本，硬件名，处理器类型，硬件平台类型，操作系统名称

* + - 1. 重要的热键
* [Tab]按键---具有『命令补全』和『档案补齐』的功能
* [Ctrl]-c按键---让当前的程序『停掉』
* [Ctrl]-d按键---通常代表着：『键盘输入结束(End Of File, EOF 或 End OfInput)』的意思；另外，他也可以用来取代exit
  + - 1. shutdown

语法：shutdown [选项]

常见选项：

* -h：将系统的服务停掉后，立即关机
* -r：在将系统的服务停掉之后就重新启动
* -t sec：-t 后面加秒数，亦即『过几秒后关机』的意思
  + - 1. 其他扩展命令
* 安装和登录命令：login、shutdown、halt、reboot、install、mount、umount、chsh、exit、last；
* 文件处理命令：file、mkdir、grep、dd、find、mv、ls、diff、cat、ln；
* 系统管理相关命令：df、top、free、quota、at、lp、adduser、groupadd、kill、crontab；
* 网络操作命令：ifconfig、ip、ping、netstat、telnet、ftp、route、rlogin、rcp、finger、mail、 nslookup；
* 系统安全相关命令：passwd、su、umask、chgrp、chmod、chown、chattr、sudo ps、who
* 其它命令：tar、unzip、gunzip、unarj、mtools、man、unendcode、uudecode。

## Shell命令及其运行原理

Linux严格意义上说的是一个操作系统，我们称之为“核心（kernel）“ ，但我们一般用户，不能直接使用kernel。而是通过kernel的“外壳”程序，也就是所谓的shell，来与kernel沟通。如何理解？为什么不能直接使用kernel？

从技术角度，Shell的最简单定义：命令行解释器（command Interpreter）主要包含：

* 将使用者的命令翻译给核心（kernel）处理。
* 同时，将核心的处理结果翻译给使用者。

对比windows GUI，我们操作windows 不是直接操作windows内核，而是通过图形接口，点击，从而完成我们的 操作（比如进入D盘的操作，我们通常是双击D盘盘符或者运行起来一个应用程序）。

shell 对于Linux，有相同的作用，主要是对我们的指令进行解析，解析指令给Linux内核。反馈结果在通过内核运行出结果，通过shell解析给用户。

## Linux权限管理

Linux下有两种用户：超级用户（root）、普通用户。

* 超级用户：可以再linux系统下做任何事情，不受限制
* 普通用户：在linux下做有限的事情
* 超级用户的命令提示符是“#”，普通用户的命令提示符是“$”。

命令：su [用户名]

功能：切换用户。 例如，要从root用户切换到普通用户user，则使用 su user。 要从普通用户user切换到root用户则使用 su root（root可以省略），此时系统会提示输入root用户的口令。

### 文件访问者的分类（人）

文件和文件目录的所有者：u---User

文件和文件目录的所有者所在的组的用户：g---Group

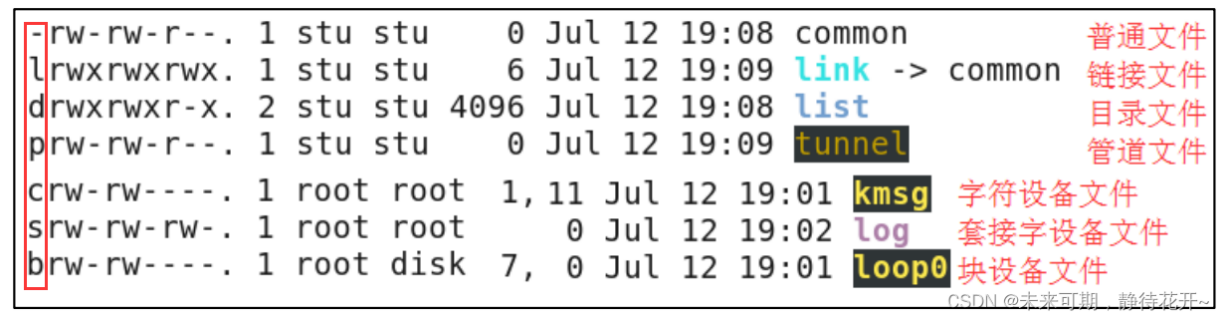
其它用户：o---Others

### 文件类型和访问权限（事务的属性）

Linux下所有的东西都可以看做文件，并且Linux下不以文件的扩展名区别文件类型，而是在文件属性中有一列专门记录文件类型。

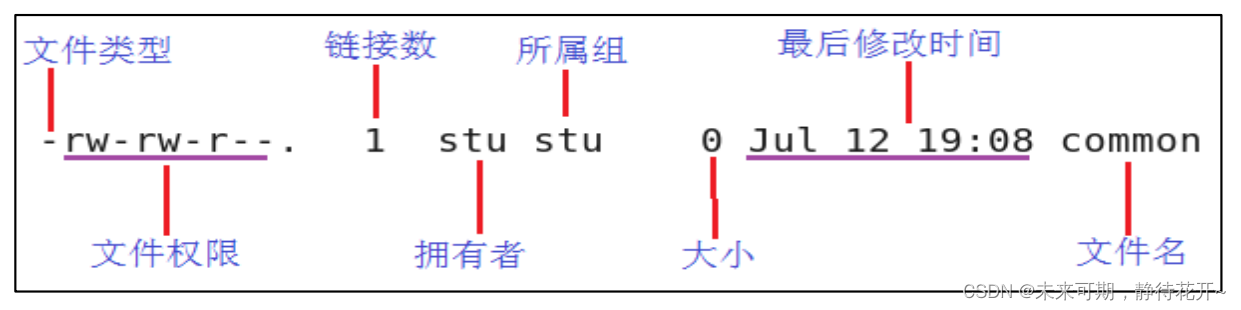
* d：目录/文件夹
* -：普通文件（源代码、库文件、可执行程序、文档压缩包等）
* l：链接文件（类似Windows的快捷方式）
* b：块设备文件（例如硬盘、光驱等）
* p：管道文件
* c：字符设备文件（键盘、显示器）
* s：套接口文件

这7种文件在Linux上呈现如下图所示：



注：Linux区分文件与后缀无关，后缀的本质是文件名的一部分

上图中列出的文件信息发别为：



文件的基本权限如下：

* 读：Read对文件而言，具有读取文件内容的权限；对目录来说，具有浏览该目录信息的权限
* 写：Write对文件而言，具有修改文件内容的权限；对目录来说具有删除移动目录内文件的权限
* 执行：execute对文件而言，具有执行文件的权限；对目录来说，具有进入目录的权限
* “—”：表示不具有该项权限

注：文件的读/写/执行权限只与文件的内容有关，与文件自己无关。删除一个文件不需要文件自身的读/写/执行权限，而需要它所在目录的读/写/执行权限。

### 文件权限值的表示方法

* + - 1. 字符表示方法

|  |  |  |  |
| --- | --- | --- | --- |
| Linux表示 | 说明 | Linux表示 | 说明 |
| r-- | 只读 | -w- | 只写 |
| --x | 仅可执行 | rw- | 可读可写 |
| -wx | 可写和可执行 | r-x | 可读可执行 |
| rwx | 可读可写可执行 | --- | 无权限 |

* + - 1. 8进制数值表示方法

|  |  |  |
| --- | --- | --- |
| 权限符号（读写执行） | 八进制 | 二进制 |
| r-- | 4 | 100 |
| -w- | 2 | 010 |
| --x | 1 | 001 |
| rw- | 6 | 110 |
| r-x | 5 | 101 |
| -wx | 3 | 011 |
| rwx | 7 | 111 |
| --- | 0 | 000 |

### 文件访问权限的相关设置方法

系统判断文件与用户之间的关系是，从user，group，other依次判别，只要能匹配上，就只能用所匹配的权限。比如一个文件的user有r权限，同时user=group，但是group有rw权限，那么这个user来操作文件时，优先匹配user的权限，就只有user的r权限，而不能用group的w权限。

* + - 1. chmod

功能：设置文件的访问权限

格式：chmod [参数] 权限 文件名

常用选项：

* R -> 递归修改目录文件的权限
* 说明：只有文件拥有者和root才能改变文件权限

chmod命令权限值的方式有两种

①用户表示符+/-=权限字符

* +:向权限范围增加权限代号所表示的权限
* -:向权限范围取消权限代号所表示的权限
* =:向权限范围赋予权限代号所表示的权限
* 用户符号：
* u：拥有者
* g：拥有者所有组
* o：其它用户
* a：所有用户

示例如下：

|  |
| --- |
| #chmod u+x /home/abc.txt //给abc.txt文件拥有者加上执行权限  #chmod o-x /home/abc.txt //给abc.txt文件其他用户取消执行权限  #chmod a=x /home/abc.txt //给abc.txt文件所有用户只设置执行权限  #chmod a=x+r=w file //给file文件所有用户先设置执行权限，再加上读权限，  //最后再只设置为w权限 |

②三位数8进制数字

举例如下：

|  |
| --- |
| #chmod 664 /home/abc.txt //给abc.txt的ugo用户分别设置权限为664 |

* + - 1. chowm

功能：修改文件的拥有者

格式：chown [参数] 用户名 文件名

常用选项：-R 递归修改文件或目录的所属组

实例：

|  |
| --- |
| # chown user1 file1 //改变file1文件所属用户为user1  # chown -R user1 filegroup1 //改变filegroup1目录的所有者为user1 |

注：在系统中，更改文件的拥有者需要root权限。

* + - 1. chgrp

功能：修改文件或目录的所属组

格式：chgrp [参数] 用户组名 文件名

常用选项：-R 递归修改文件或目录的所属组

实例：

|  |
| --- |
| #chgrp users file1 ////改变file1文件所属用户组为users |

注：更改文件的所属组也需要root权限。

* + - 1. umask（权限掩码）

功能：查看或修改文件掩码

新建文件夹默认权限=0666

新建目录默认权限=0777

但实际上你所创建的文件和目录，看到的权限往往不是上面这个值。原因就是创建文件或目录的时候还要受到umask的影响。假设默认权限是mask，则实际创建的出来的文件权限是: mask & ~umask

格式：umask 权限值

说明：将现有的存取权限减去权限掩码后，即可产生建立文件时预设权限。超级用户默认掩码值为0022，普通用户默认为0002。

注：凡是在umask中出现的权限，都必须在起始权限中去掉。

### 粘滞位

首先介绍一下目录的权限

* 可执行权限：如果没有目录的可执行权限，则无法在目录下执行权限，也无法进入目录
* 可读权限：如果没有目录的可读权限，则无法用ls等命令查看目录中的文件
* 可写权限：如果目录没有可写权限，则无法在目录中新增、删除文件、重命名等修改操作的权限

注：只要用户具有目录的写和执行权限，用户就可以随意删除目录中的文件，而不论这个用户是否有这个文件的写权限。

Linux系统中有很多人，我们需要一个公共的目录（比如tmp文件夹），进行临时文件的操作（增删查改），那么用户A在这个目录中创建的文件，用户B可以随意的删除，为了解决这个问题，Linux引入了粘滞位的概念。

|  |
| --- |
| #chmod 777 tmp //将目录的所有权限打开  #chmod +t tmp //加上粘滞位，此时的权限为：drwxrwxrwt |

所以加上粘滞位之后目录的其他人的执行权限变为rwt，所以当一个目录被设置为“粘滞位”时，该目录下的文件只能由

* root用户删除
* 目录所有者删除
* 该文件的所有者删除

# Linux环境基础开发工具

## Linux软件包管理器yum

在Linux下安装软件，一个通常的办法是下载到程序的源代码，并进行编译，得到可执行程序。但是这样太麻烦了，于是有些人把一些常用的软件提前编译好，做成软件包(可以理解成windows上的安装程序)放在一个服务器上，通过包管理器可以很方便的获取到这个编译好的软件包，直接进行安装。软件包和软件包管理器，就好比“App”和“应用商店”这样的关系。

yum(Yellow dog Updater, Modified)是Linux下非常常用的一种包管理器。主要应用在Fedora, RedHat, Centos等发行版上。

* + - 1. yum list搜索安装包命令

通过yum list命令可以罗列出当前一共有哪些软件包。由于包的数目可能非常之多，一般结合grep命令只筛选出我们关注的包。例如：

|  |
| --- |
| yum list | grep lrzsz |

结果如下：

|  |
| --- |
| lrzsz.x86\_64 0.12.20-36.el7 @base |

注：上面的含义如下

* 软件包名称:主版本号.次版本号.源程序发行号-软件包的发行号.主机平台.cpu架构.
* "x86\_64" 后缀表示64位系统的安装包, "i686" 后缀表示32位系统安装包. 选择包时要和系统匹配.
* "el7" 表示操作系统发行版的版本. "el7" 表示的是 centos7/redhat7. "el6" 表示 centos6/redhat6.
* 最后一列, base 表示的是 "软件源" 的名称, 类似于 "小米应用商店", "华为应用商店" 这样的概念.
  + - 1. yun install安装包命令

通过 yum, 我们可以通过很简单的一条命令完成包的安装

|  |
| --- |
| sudo yum install lrzsz |

yum 会自动找到都有哪些软件包需要下载, 这时候敲 "y" 确认安装.

注：

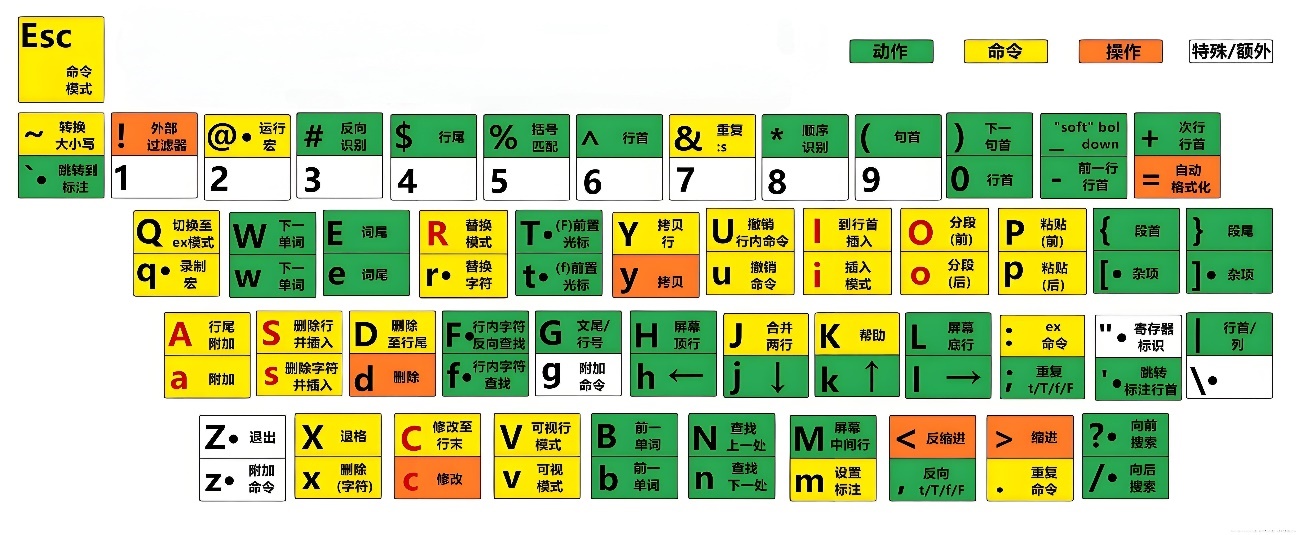
* 安装软件时由于需要向系统目录中写入内容，一般需要 sudo 或者切到 root 账户下才能完成。
* yum安装软件只能一个装完了再装另一个。正在yum安装一个软件的过程中，如果再尝试用yum安装另外一个软件, yum会报错。
  + - 1. yum remove卸载包命令

通过如下命令即可卸载已安装的包：

|  |
| --- |
| sudo yum remove lrzsz |

## Linux编辑器vim

vi/vim的区别简单点来说，它们都是多模式编辑器，不同的是vim是vi的升级版本，它不仅兼容vi的所有指令，而且还有一些新的特性在里面。例如语法加亮，可视化操作不仅可以在终端运行，也可以运行于x window、 mac os、 windows。



### vim的基本概念

vim有很多种模式，有七种基本模式：Nromal mode、Visual mode、Select mode、Insert mode、Command-line mode（Cmdline mode）、Ex mode、Terminal mode。有六种附加模式：Operator-pending mode、Replace mode、Virtual Replace mode、Insert Normal mode、Insert Visual mode、Insert Select mode。

注：vim打开默认就是命令模式（底行模式）

* + - 1. 正常/普通/命令模式(Normal mode)

控制屏幕光标的移动，字符、字或行的删除，移动复制某区段及进入Insert mode下，或者到 last line mode。

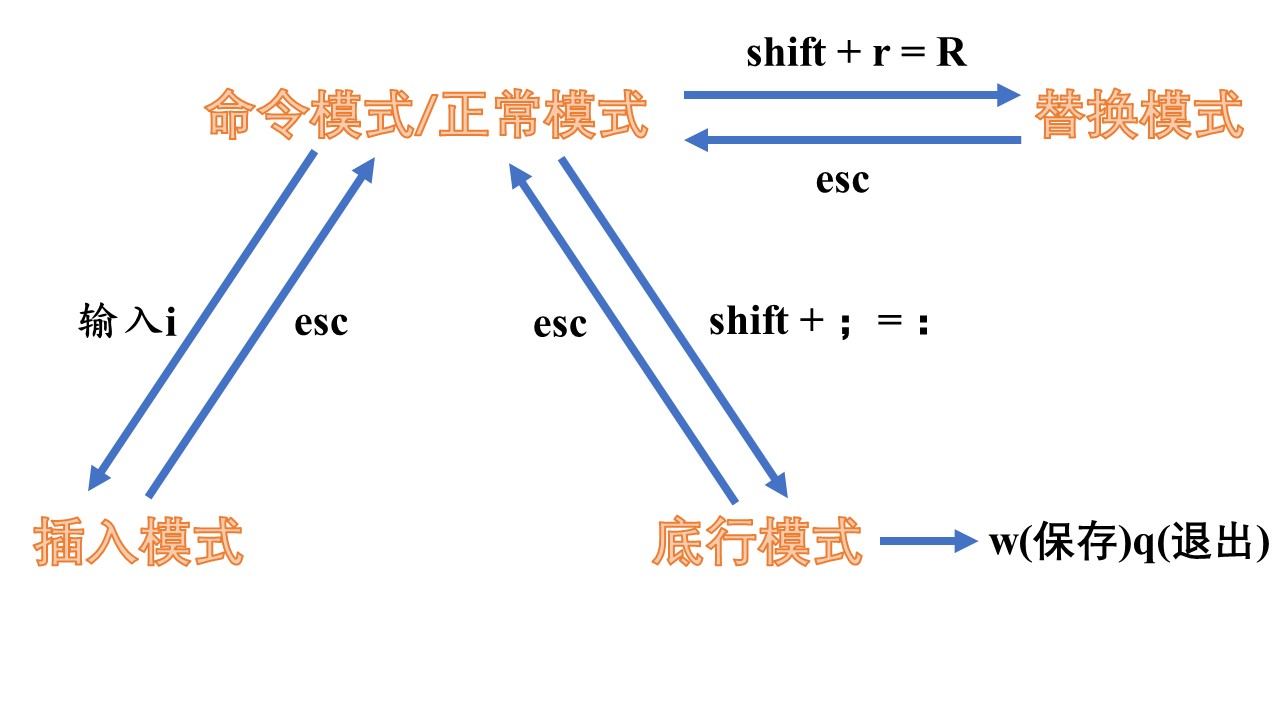
* + - 1. 插入模式(Insert mode)

只有在Insert mode下，才可以做文字输入，按「ESC」键可回到命令行模式。该模式是我们后面用的最频繁 的编辑模式。

* + - 1. 末行模式(Command mode)

文件保存或退出，也可以进行文件替换，找字符串，列出行号等操作。在命令模式下，shift+: 即可进入该模式。要查看你的所有模式：打开vim，底行模式直接输入:help vim-modes

### vim的模式切换



### vim正常模式命令集

* + - 1. 进入插入模式
* 按「i」切换进入插入模式「insert mode」，按“i”进入插入模式后是从光标当前位置开始输入文件；
* 按「a」进入插入模式后，是从目前光标所在位置的下一个位置开始输入文字；
* 按「o」进入插入模式后，是插入新的一行，从行首开始输入文字。
  + - 1. 从插入模式切换为命令模式
* 按「ESC」键
  + - 1. 移动光标
* vim可以直接用键盘上的光标来上下左右移动，但正规的vim是用小写英文字母「h」、「j」、「k」、 「l」，分别控制光标左、下、上、右移一格。
* 按「G」：移动到文章的最后。
* 按［gg］：进入到文本开始。
* 按「^」：移动到光标所在行的“行首”。
* 按「 $ 」：移动到光标所在行的“行尾”。
* 按「w」：光标跳到下个字的开头。
* 按「b」：光标回到上个字的开头。
  + - 1. 删除文字
* 「x」：每按一次，删除光标所在位置的一个字符。
* 「#x」：例如，「6x」表示删除光标所在位置的“后面（包含自己在内）”6个字符。
* 「X」：大写的X，每按一次，删除光标所在位置的“前面”一个字符。
* 「#X」：例如，「20X」表示删除光标所在位置的“前面”20个字符。
* 「dd」：剪切光标所在行。
* 「#dd」：从光标所在行开始删除#行。
  + - 1. 复制
* 「yy」：复制光标所在行到缓冲区。
* 「#yy」：例如，「6yy」表示拷贝从光标所在的该行“往下数”6行文字。
* 「p」：将缓冲区内的字符贴到光标所在位置。注意：所有与“y”有关的复制命令都必须与“p”配合才能完 成复制与粘贴功能。
  + - 1. 替换
* 「r」：替换光标所在处的字符。
* 「R」：替换光标所到之处的字符，直到按下「ESC」键为止。
  + - 1. 撤销操作
* 「u」：如果您误执行一个命令，可以马上按下「u」，回到上一个操作。按多次“u”可以执行多次恢复。
* 「ctrl + r」: 撤销的恢复。
  + - 1. 更改
* 「cw」：更改光标所在处的字到字尾处。
* 「c#w」：例如，「c3w」表示更改3个字

### vim末行模式命令集

* + - 1. 列出行号
* 「set nu」: 输入「set nu」后，会在文件中的每一行前面列出行号。
* 「set nonu」: 输入「set nonu」后，会取消文件中的行号。
  + - 1. 跳到文件某一行
* 「#」:「#」号表示一个数字，在冒号后输入一个数字，再按回车键就会跳到该行了，如输入数字15，再回车，就会跳到文章的第15行。
  + - 1. 查找字符
* 「/关键字」: 先按「/」键，再输入您想寻找的字符，如果第一次找的关键字不是您想要的，可以一直按「n」会往后寻找到您要的关键字为止。
* 「?关键字」：先按「?」键，再输入您想寻找的字符，如果第一次找的关键字不是您想要的，可以一直按「n」会往前寻找到您要的关键字为止。

注：「/」是从前往后找，「?」是从后往前找。

* + - 1. 保存文件、
* 「w」: 在冒号输入字母「w」就可以将文件保存起来
  + - 1. 退出vim
* 「q」：按「q」就是退出，如果无法离开vim，可以在「q」后跟一个「!」强制离开vim。
* 「wq」：一般建议离开时，搭配「w」一起使用，这样在退出的时候还可以保存文件。
  + - 1. 打开另一个文件
* 「vs 文件名」：在同一窗口打开另一个文件

## Linux编译器gcc/g++

### 程序翻译过程

1. 预处理（进行宏替换)

2. 编译（生成汇编)

3. 汇编（生成机器可识别代码）

4. 连接（生成可执行文件或库文件)

### gcc/g++

**命令：gcc [选项] 要编译的文件 [选项] [目标文件]**

* + - 1. 预处理(进行宏替换)
* 预处理功能主要包括宏定义,文件包含,条件编译,去注释等。
* 预处理指令是以#号开头的代码行。
* 实例: gcc –E hello.c –o hello.i
* 选项“-E”,该选项的作用是让 gcc 在预处理结束后停止编译过程。
* 选项“-o”是指目标文件,“.i”文件为已经过预处理的C原始程序。
  + - 1. 编译（生成汇编）
* 在这个阶段中,gcc 首先要检查代码的规范性、是否有语法错误等,以确定代码的实际要做的工作,在检查无误后,gcc 把代码翻译成汇编语言。
* 用户可以使用“-S”选项来进行查看,该选项只进行编译而不进行汇编,生成汇编代码。
* 实例: gcc –S hello.i –o hello.s
  + - 1. 汇编（生成机器可识别代码）
* 汇编阶段是把编译阶段生成的“.s”文件转成目标文件
* 读者在此可使用选项“-c”就可看到汇编代码已转化为“.o”的二进制目标代码了
* 实例: gcc –c hello.s –o hello.o
  + - 1. 连接（生成可执行文件或库文件）
* 在成功编译之后,就进入了链接阶段。
* 实例: gcc hello.o –o hello

助记：ESc————————————>iso（镜像）

gcc默认是动态链接，添加-static是静态链接

gcc默认是release版本，添加-g是debug版本

### 函数的动静态库

* + - 1. 静态库

**静态库 (Static Library)**是一种在编译时链接到程序中的库文件，它的格式为.a（在Linux系统中）。静态库包含了程序运行所需的所有代码和数据，当程序编译时，链接器会将静态库中的代码和数据直接复制到生成的可执行文件中。

**静态库的优点：**

* **执行速度快**：由于代码已经包含在可执行文件中，不需要运行时动态链接，可以减少启动时间。即编译成功的可执行文件可以独立执行，而不需要再向外部要求读取函数库的内容。
* **确定性**：程序的行为不会受到其他程序使用相同库的不同版本的影响。

**静态库的缺点：**

* **占用空间**：每个使用静态库的可执行文件都会包含库的一份副本，导致文件体积增大。
* **更新困难**：当静态库更新时，所有使用该库的程序都需要重新编译和链接。
* **资源浪费**：如果多个程序使用相同的静态库，那么每个程序中都会有一份库的副本，浪费内存和磁盘空间。
  + - 1. 动态库

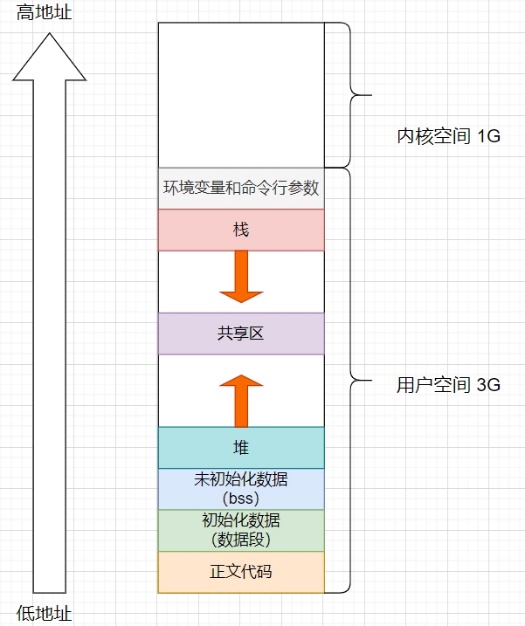
动态库(Dynamic Library)是一种在程序运行时可以动态加载的库。它的格式为.so (Shared Object)在Linux系统中，动态库也称为共享库。动态库是一种在程序运行时可以动态加载的库。这意味着动态库的内容（包括函数、变量和类等）并不在编译时被包含进程序本身，而是在程序运行时才根据需要被加载。因此，多个程序可以共享同一个动态库，从而节省内存空间。

**动态库的优点：**

* **代码重用**：动态库允许多个程序共享使用相同的代码和数据，从而减少了内存的使用并提高了代码的重用
* **资源节约**：由于多个程序可以共享使用同一个动态库，因此不需要在每个程序中都包含相同的代码和数据，从而节省了磁盘空间。
* **更新和维护方便**：当动态库更新时，所有使用该库的程序都可以立即受益，而无需重新编译或重新分发程序本身。这大大简化了软件的更新和维护过程。

**动态库的缺点：**

* **加载时间**：由于动态库在程序运行时需要被加载到内存中，因此相对于静态链接的程序来说，动态链接的程序在启动时可能会有一定的延迟。
* **依赖问题**：动态链接的程序依赖于外部的动态库文件。如果这些库文件不存在、版本不匹配或路径不正确，程序将无法正常运行。这增加了程序部署和管理的复杂性。



当程序使用动态库时，操作系统会在内存中为动态库分配一块共享区域（即上图中的共享区），并将该区域映射到所有使用该动态库的进程的地址空间中。这样，当多个进程同时访问动态库中的函数或数据时，它们实际上是在访问同一块内存区域。这种共享机制使得动态库成为了一种非常高效的代码和数据复用方式。

**动态库加载过程：**

* 加载到进程地址空间：当程序尝试调用动态库中的函数时，动态链接器会将该库加载到进程的地址空间中，即上图的“共享区”。
* 符号解析：动态链接器会解析程序引用的动态库中的符号（如函数和数据）。它会在动态库的符号表中查找这些符号的地址，并在程序的地址空间中建立相应的映射。
* 调用库函数：当程序调用动态库中的函数时，它会跳转到地址空间中的共享区，执行库中的函数代码。执行完成后，程序会跳转回原来的位置，继续执行。
* 页表映射：为了使得程序能够正确地访问这个共享区中的代码和数据，操作系统会在页表中建立相应的条目，将动态库的虚拟地址映射到这块共享区的物理地址。并在每个需要使用该动态库的进程的页表中建立相应的映射条目。这样，当程序访问动态库中的函数或数据时，处理器就可以通过页表找到对应的物理地址，从而实现对动态库的访问。

注：两个命令

* ldd命令查看程序所依赖的动态链接库
* file命令查看文件类型

## Linux调试器gdb

### 背景

* 程序的发布方式有两种，debug模式和release模式
* Linux gcc/g++出来的二进制程序，默认是release模式
* 要使用gdb调试，必须在源代码生成二进制程序的时候，加上 -g 选项

### 使用细节

语法：gdb 二进制程序

退出：ctrl+d或者q+enter

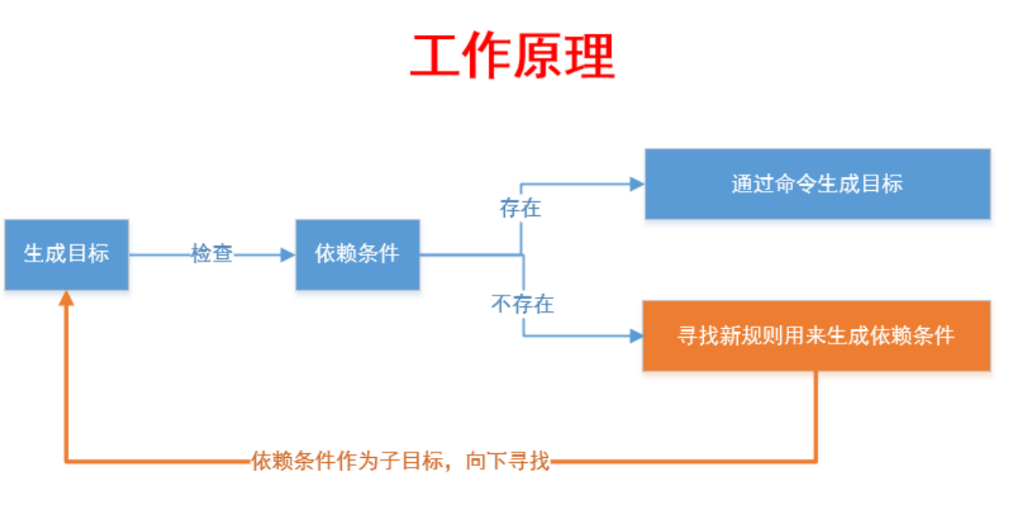
* list或l 行号：显示二进制程序的源代码，接着上次的位置往下列，每次显示10行
* list或l 函数名：列出这个函数的源代码
* r或run：运行程序，断点处停下来0
* n或next：逐过程，单条执行，不会进入函数
* s或step：逐语句，进入函数调 用
* b或break 行号：在某一行设置断点
* b或break 函数名：在某个函数开头设置断点
* info break：查看断点信息
* finish：执行到当前函数返回，然后接下来等待命令
* p或print：打印表达式的值，通过表达式可以修改变量或者调用函数
* set var：修改变量的值
* c或continue：从当前位置开始连续而非单步执行程序，运行至下一断点处
* delete breakpoint (n)：不加n删除所有断点，加n删除序号为n的断点
* disable breakpoint：禁用断点
* enable breakpoint：启用断点
* info breakpoint：查看当前设置哪些断点
* display 变量名：跟踪查看一个变量，每次停下来都显示这个值
* undisplay：取消已跟踪变量
* until X行号：跳至X行
* beraktrace或bt：查看各级函数调用及参数
* info locals：查看当前栈帧局部变量的值
* quit：退出gdb

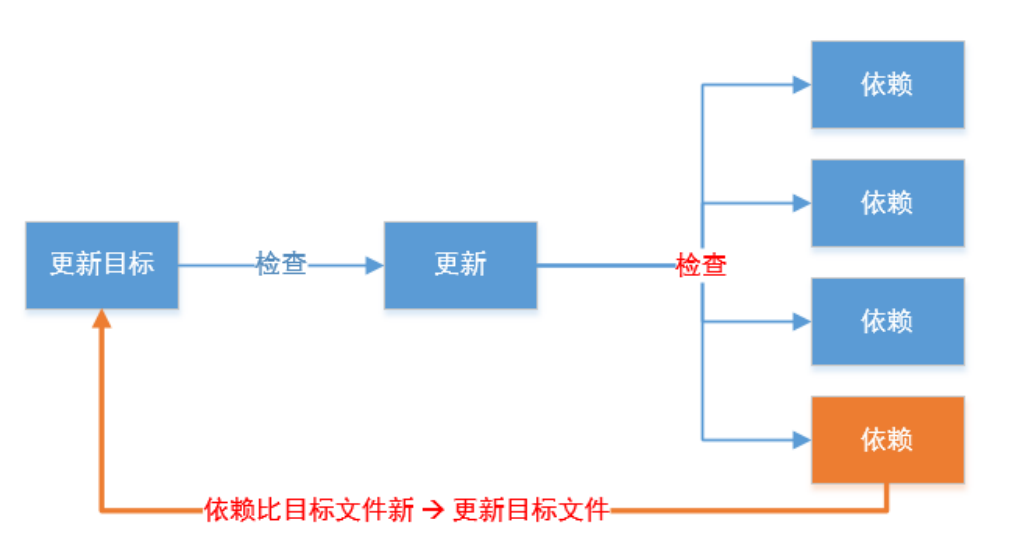
## Linux项目自动化构建工具make/Makefile

### makefile规则

makefile的规则由目标（targets）、先决条件（prerequisites）以及命令（commands）所组成的，语法如下所示：

|  |
| --- |
| targets : prerequisites  command  ... |





### 文件的三个时间

在linux系统中，文件有三个时间，分别是Access、Modify、Change。

* Access：最近一次访问的时间
* Modify：最近一次内容修改时间
* Change：最近一次属性修改时间

注：1.在最开始的Linux内核中，只要访问一次文件，文件的Access的时间就会改变，文件打开或者更改都会使Access时间发生改变，那么这个时间就会频繁发生变化，就要进行更多次的IO。新的内核改变Access更新的策略，在访问一定次数之后才会发生改变。2.文件内容修改之后，文件的属性（大小）一定会发生变化。3.touch一个存在的文件会更新这个文件的时间到当前时间。

### makefile执行策略

当先决条件的Modify时间比目标的Modify时间更早时，make不会重新再执行命令。用.PHONY修饰的目标就不用遵循这个策略。

当makefile文件中有多个命令时，make总是执行第一个命令。如果缺少先决条件，则会依次向下寻找。例如：

|  |
| --- |
| mycode:mycode.o  gcc mycode.o -o mycode  mycode.o:mycode.s  gcc -c mycode.s -o mycode.o  mycode.s:mycode.i  gcc -S mycode.i -o mycode.s  mycode.i:mycode.c  gcc -E mycode.c -o mycode.i |

## Linux第一个小程序-进度条

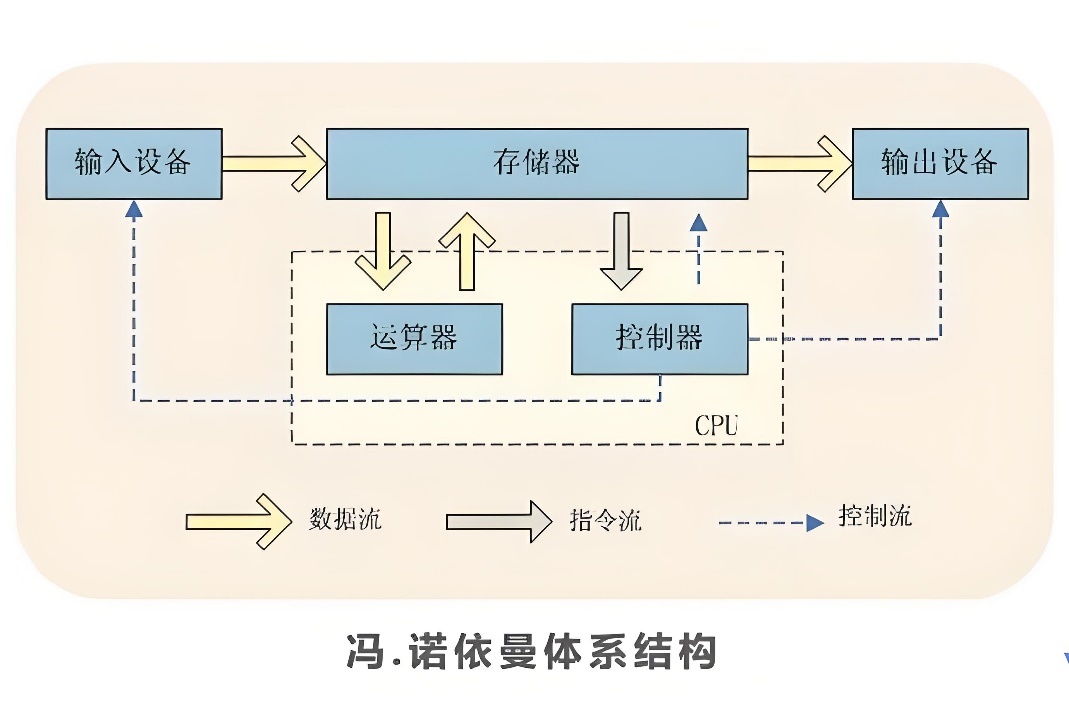
回车：光标回到一行的开始————>\r

换行：光标到下一行————————>\n

在语言层面，\n就是回车+换行！！！

# Linux进程概念

## 冯·诺依曼体系结构



我们常见的计算机，如笔记本。我们不常见的计算机，如服务器，大部分都遵守冯诺依曼体系。计算机共有五大组成部件，分别是输入设备、输出设备、运算器、控制器与存储器。其中各部分的作用如下：

* 输入单元：用于将数据输入到计算机中，如键盘、鼠标等。
* 输出单元：用于将计算机处理后的数据输出，如显示器、打印机等。
* 中央处理器（CPU）：计算机的核心，运算器负责执行计算，控制器负责空则计算机的操作。
* 存储器：用于存储数据和指令，包括内存和高速缓存。

注：不考虑缓存情况，这里的CPU能且只能对内存进行读写，不能访问外设(输入或输出设备)，同样地，外设（输入或输出设备）要输入或者输出数据，也只能写入内存或者从内存中读取。总结就是，所有设备都只能直接和内存打交道。

CPU有自己的指令集（发为精简指令集和复杂指令集），而编译器的任务就是将文本程序翻译成CPU认识的指令集。

DMA（直接内存访问）：允许数据在外设和内存之间直接访问，无需CPU的干预。

网卡和磁盘既是输入设备也是输出设备。

## 操作系统

* + - 1. 概念

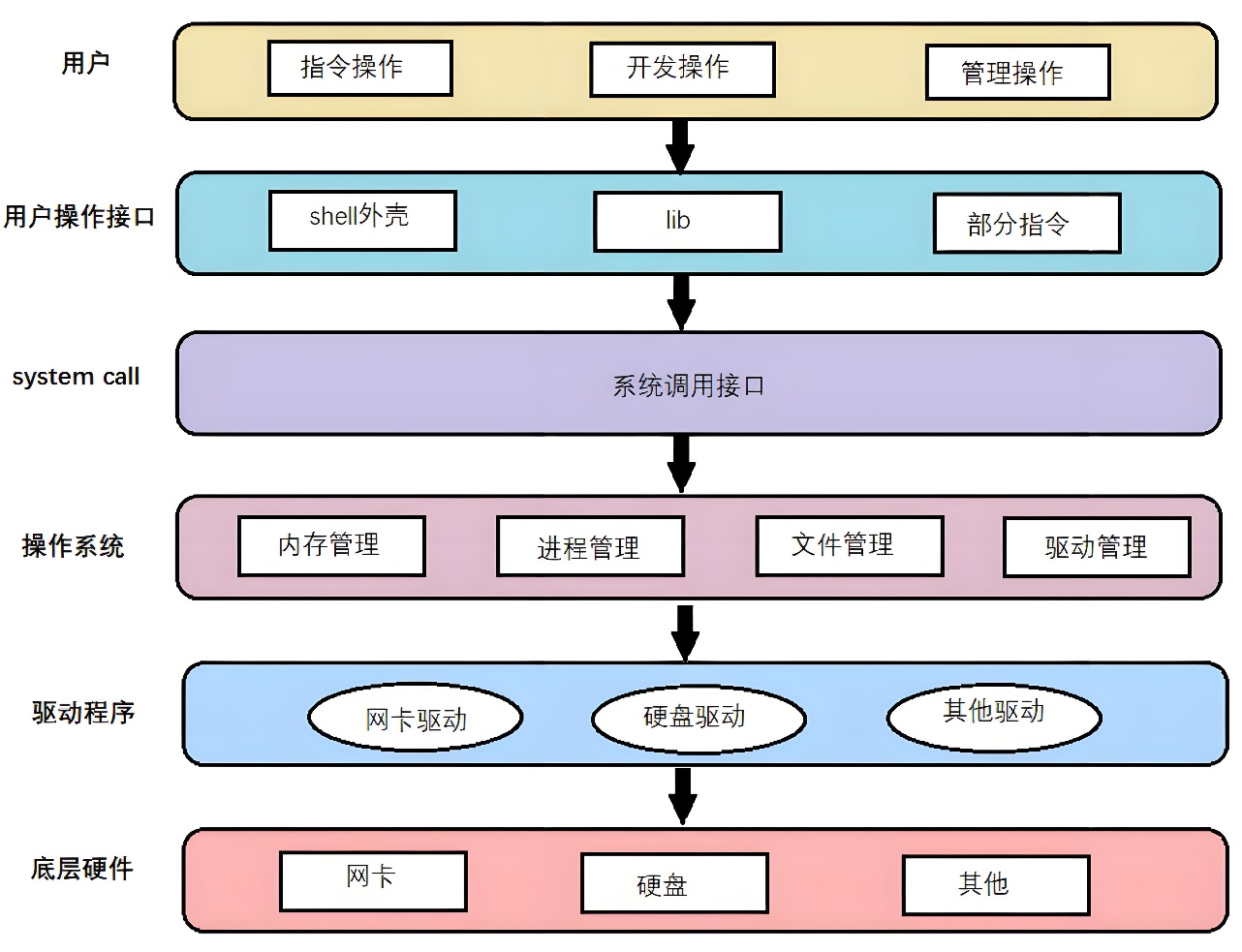
任何计算机系统都包含一个基本的程序集合，称为操作系统(OS)。笼统的理解，操作系统包括：

* 内核（进程管理，内存管理，文件管理，驱动管理）
* 其他程序（例如函数库，shell程序等等）
  + - 1. 目的

设计操作系统的目的可以概况为两个：

* 对上为用户程序提供一个良好的执行环境
* 对下与硬件交互，管理所有的软硬件资源
  + - 1. 定位

在整个计算机软硬件架构中，操作系统的定位就是：一款纯正的管理软件！



* + - 1. 计算机管理硬件

先描述，用struct结构体

后组织，用链表或者其他高效的数据结构

* + - 1. 系统调用和库函数的概念

在开发角度，操作系统对外会表现为一个整体，但是会暴露自己的部分接口，供上层开发使用，这部分由操作系统提供的接口，叫做系统调用。

系统调用在使用上，功能比较基础，对用户的要求相对也比较高，有心的开发者可以对部分系统调用进行适度封装，从而形成库，有了库，就很有利于更上层用户或者开发者进行二次开发。

## 进程

* + - 1. 基本概念

一个运行起来的（加载到内存）程序就叫做进程。

进程=内核数据结构（task\_struct）+进程对应的磁盘代码

* + - 1. 描述进程-PCB

PCB全称叫做进程控制块（process control block），是进程属性的集合，Linux操作系统下的PCB是: task\_struct。task\_struct是Linux内核的一种数据结构，它会被装载到（RAM）中，具体有以下这些信息：

* 标示符：描述本进程的唯一标示符，用来区别其他进程。
* 状态：任务状态，退出代码，退出信号等。
* 优先级：相对于其他进程的优先级。
* 程序计数器：程序中即将被执行的下一条指令的地址。
* 内存指针：包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针
* 上下文数据：进程执行时处理器的寄存器中的数据。
* I／O状态信息：包括显示的I/O请求,分配给进程的I／O设备和被进程使用的文件列表。
* 记账信息：可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。
* 其他信息
  + - 1. 查看进程的方式

|  |
| --- |
| ps ajx | head -1 && ps ajx | grep “myproc” |

进程也是一种文件，所以进入/proc文件夹也可以查看

|  |
| --- |
| ls /proc/pid -d |

一般命令上启动的进程，父进程都是bash

* + - 1. fork函数
* fork()之后，会有父进程+子进程两个进程执行后续代码
* fork后续的代码，被父子进程共享
* 通过不同的返回值。让父子进程执行后续共享的代码的一部分
  + - 1. 进程状态

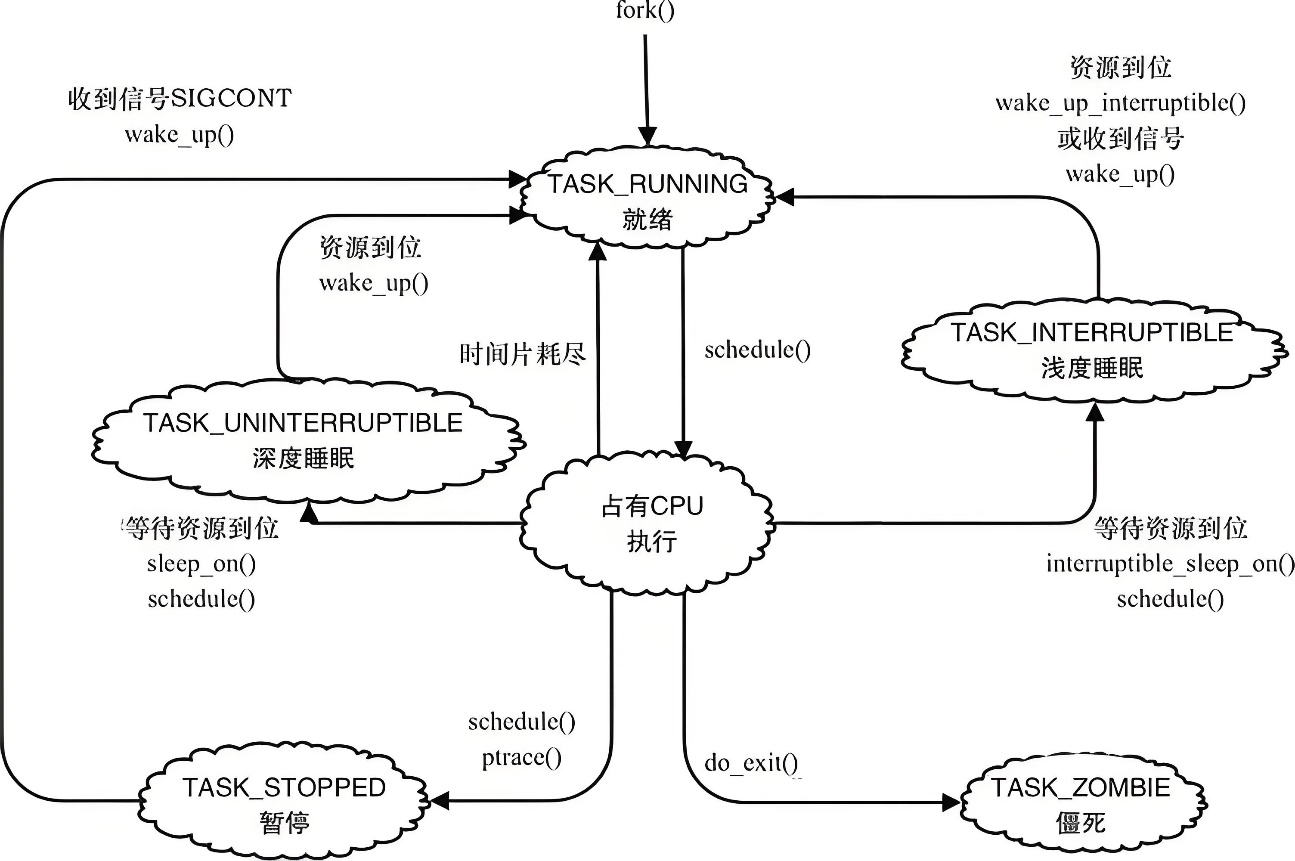
在Linux内核源代码中，一个进程有几个状态，进程有时候也叫任务。

|  |
| --- |
| /\*  \* The task state array is a strange "bitmap" of  \* reasons to sleep. Thus "running" is zero, and  \* you can test for combinations of others with  \* simple bit tests.  \*/  static const char \* const task\_state\_array[] = {  "R (running)", /\* 0 \*/  "S (sleeping)", /\* 1 \*/  "D (disk sleep)", /\* 2 \*/  "T (stopped)", /\* 4 \*/  "t (tracing stop)", /\* 8 \*/  "X (dead)", /\* 16 \*/  "Z (zombie)", /\* 32 \*/  }; |

* R运行状态（running）: 并不意味着进程一定在运行中，它表明进程要么是在运行中要么在运行队列里。
* S睡眠状态（sleeping): 意味着进程在等待事件完成（这里的睡眠有时候也叫做可中断睡眠 （interruptible sleep）
* D磁盘休眠状态（Disk sleep）有时候也叫不可中断睡眠状态（uninterruptible sleep），在这个状态的进程通常会等待IO的结束。该状态无法被操作系统杀死，只能通过断点，或者进程自己醒来解决。
* T停止状态（stopped）： 可以通过发送 SIGSTOP 信号给进程来停止（T）进程。这个被暂停的进程可 以通过发送 SIGCONT 信号让进程继续运行。
* X死亡状态（dead）：这个状态只是一个返回状态，你不会在任务列表里看到这个状态。

要查看一个进程状态，需要用到以下命令：

|  |
| --- |
| ps aux / ps ajx |



阻塞：正在执行的进程由于发生某时间（如I/O请求、申请缓冲区失败等）暂时无法继续执行。此时引起进程调度，OS把处理机分配给另一个就绪进程，而让受阻进程处于暂停状态，一般将这种状态称为阻塞状态。

挂起：由于系统和用户的需要引入了挂起的操作，进程被挂起意味着该进程处于静止状态。如果进程正在执行，它将暂停执行，若原本处于就绪状态，则该进程此时暂不接受调度。

共同点：1. 进程都暂停执行。2. 进程都释放CPU，即两个过程都会涉及上下文切换。

不同点：

1. 对系统资源占用不同：虽然都释放了CPU，但阻塞的进程仍处于内存中，而挂起的进程通过“对换”技术被换出到外存（磁盘）中。

2. 发生时机不同：阻塞一般在进程等待资源（IO资源、信号量等）时发生；而挂起是由于用户和系统的需要，例如，终端用户需要暂停程序研究其执行情况或对其进行修改、OS为了提高内存利用率需要将暂时不能运行的进程（处于就绪或阻塞队列的进程）调出到磁盘

3. 恢复时机不同：阻塞要在等待的资源得到满足（例如获得了锁）后，才会进入就绪状态，等待被调度而执行；被挂起的进程由将其挂起的对象（如用户、系统）在时机符合时（调试结束、被调度进程选中需要重新执行）将其主动激活。

前台进程（有+号），不能接收shell命令，可按ctrl+c终止

后台进程（无+号），可以接收shell命令，但是不能按ctrl+c终止，只能

kill -9杀掉进程。

* + - 1. Z(zombie)僵尸进程

僵死状态（Zombies）是一个比较特殊的状态。当进程退出并且父进程没有读取到子进程退出的返回代码时就会产生僵死(尸)进程。僵死进程会以终止状态保持在进程表中，并且会一直在等待父进程读取退出状态代码。所以，只要子进程退出，父进程还在运行，但父进程没有读取子进程状态，子进程进入Z状态。

如果父进程一直不读取子进程的退出状态，那么子进程就一直维持在Z状态，同时，子进程的PCB也一直在内存中，会造成内存资源的浪费，也就是内存泄漏。

注：僵尸状态不可以被kill命令杀死。

* + - 1. 孤儿进程

父进程如果先退出，子进程就称之为“孤儿进程”，孤儿进程被1号init进程领取，由init进程回收。（1号进程就是操作系统）。

如果是前台进程创建的子进程，如果孤儿了，会自动变成后台进程。

* + - 1. 进程优先级

权限是能和不能的问题，而优先级是在能做，考虑的是先做还是后做的问题。因为硬件资源太少，从而会存在优先级。而优先级的本质就是pcb里面的一个整数数字，Linux中就是由PRI（priority）和NI（nice）这两个代表优先级。

* PRI就是进程的优先级，此值越小则进程的优先级越高
* NI表示进程可被执行的优先级的修正数值，取值范围为-20~19，一共40个级别。
* 优先级的公式：PRI（new）=PRI（old）+nice。PRI值越小优先级越高，加入nice值后，优先级会改变，当nice值为负时，其优先级会变高，当nice值为正时，其优先级会变低。

top工具可以改变优先级，但是需要root权限来改，具体步骤如下：

sudo top->按“r”->输入进程PID->输入nice值

* + - 1. 其他概念

竞争性：系统进程数目众多，而CPU资源只有少量，甚至1个，所以进程之间是具有竞争属性的。为了高效完成任务，更合理竞争相关资源，便具有了优先级。

独立性：多进程运行，需要独享各种资源，多进程运行期间互不干扰。

并行：多个进程在多个CPU下分别，同时进行运行，这称之为并行。

并发：多个进程在一个CPU下采用进程切换的方式，在一段时间之内，让多个进程都得以推进，称之为并发。

## 环境变量

* + - 1. 基本概念

环境变量一般是指在操作系统中用来指定操作系统运行环境的一些默认参数，比入在编译C/C++代码过程中，在链接的时候，从来都不知道链接的动静态库在哪里，但是都可以链接成功，生成可执行程序，原因就是由相关环境变量帮助编译器进程查找。环境变量通常具有某些特殊用途，还有在系统当中通常具有全局特性。

环境变量是操作系统为了满足不同应用场景，而预先在系统内设置的一大批全局变量。这些变量在整个系统当中，可以被所有的进程访问，每个环境变量都有不同的用途。

* + - 1. 常见的环境变量
* PATH：指定命令的搜索路径
* HOME：指定用户的主工作目录（即用户登录Linux系统中，默认的目录）
* SHELL：当前Shell，它的值通常是/bin/bash
  + - 1. 查看环境变量的方法
* env命令可以查看所有的环境变量
* echo $NAME 可以查看某一个环境变量的值
* C语言有一个系统调用getenv(NAME)函数可以获取环境变量
* 全局二级指针变量：extern char \*\*environ;也可以访问环境变量。libc中定义的全局变量environ指向环境变量表，environ没有包含在任何头文件中，所以在使用时要用extern声明。
  + - 1. 和环境变量相关的命令

1.echo：显示某个环境变量值

2.export：设置一个新的环境变量

3.env：显示所有环境变量

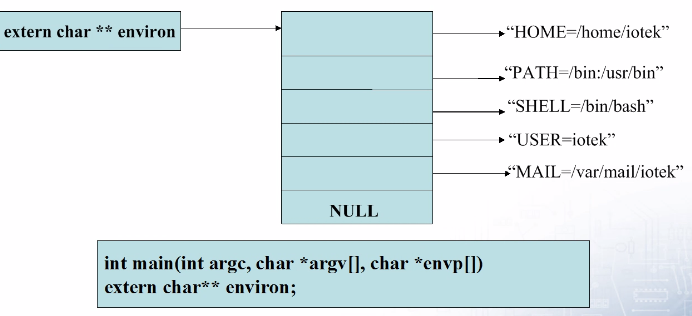
4.unset：清除环境变量

5.set：显示本地定义的shell变量和环境变量

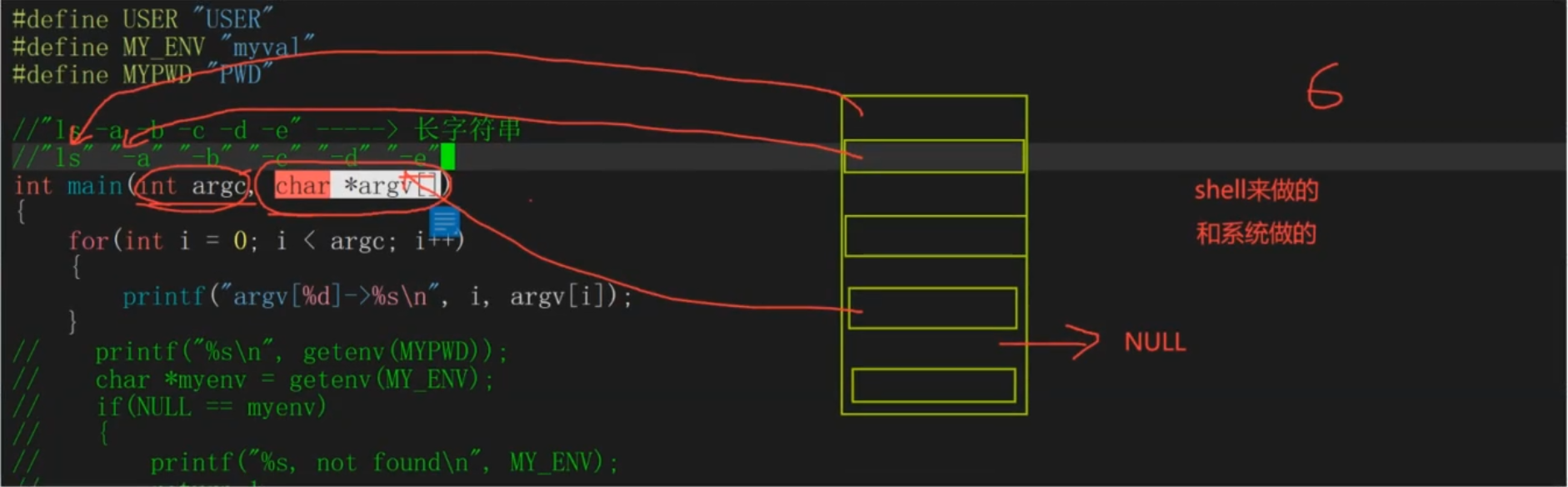
注：本地变量—只在当前进程（bash）内有效

* + - 1. 环境变量的组织方式

环境变量是一个字符指针数组，每个指针指向一个以‘\0’结尾的环境字符表。



* 每个进程都有一个独立的环境表，初始的环境表继承自父进程



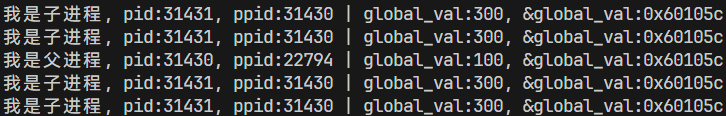
* 主函数可以带三个参数，第三个参数则是环境表，可以通过第三个参数获取环境参数，也可以通过外部全局变量来定义环境表，extern char \*\*environ。argc是命令行参数个数，argv[]是命令行指针数组



* 设置的环境变量（本地变量）只影响当前进程的，不为全局的，不能被子进程继承，只能用set命令查看。而export 命令可以把本地变量暂时添加到系统变量里。shell一旦退出，下次再在登录时，自己添加的环境变量就不存在了（根本原因是我们自己添加的环境变量在内存中）。如果想要将自己添加的环境变量变成永久的，就要保存在.bash\_profile配置文件中。

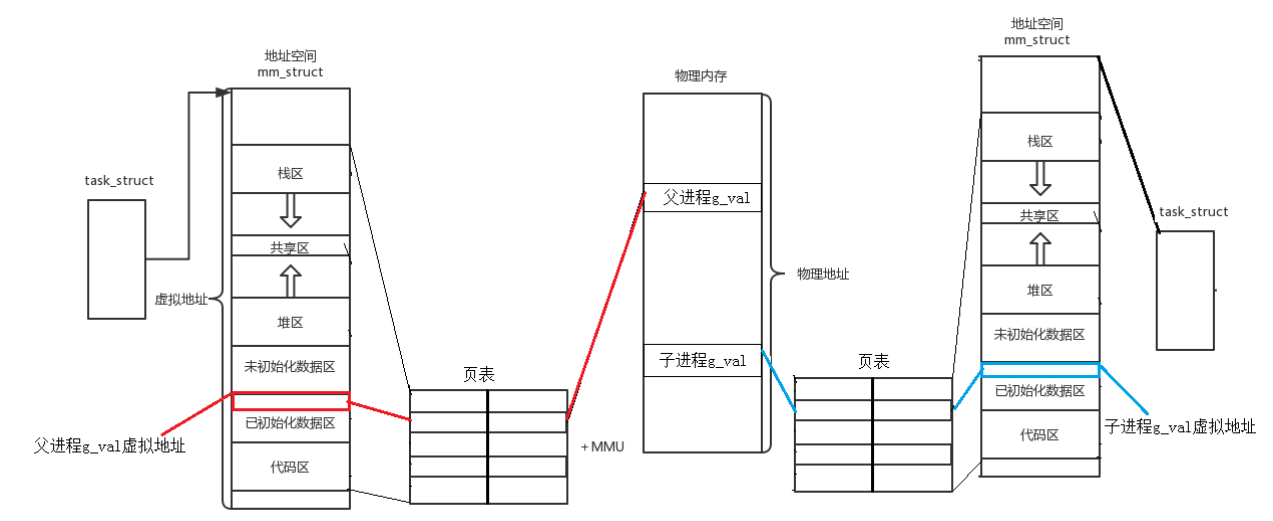
## 进程地址空间

程序地址空间不是内存，不是物理地址，而是一个虚拟的地址（线性地址/逻辑地址），如下所示：父子进程对于同一个变量global\_val，而且是同一个地址，但是在父进程中是100，但是在子进程中是300。



* 变量内容不一样,所以父子进程输出的变量绝对不是同一个变量
* 但地址值是一样的，说明，该地址绝对不是物理地址！
* 在Linux地址下，这种地址叫做虚拟地址
* 我们在用C/C++语言所看到的地址，全部都是虚拟地址！物理地址，用户一概看不到，由OS统一管理
* OS必须负责将 虚拟地址 转化成 物理地址

所以之前说“程序的地址空间”是不准确的，准确的应该说成“进程地址空间”，如图所示：



为什么要存在地址空间？

1.如果让进程之间访问物理内存，万一进程越界非法操作，非常不安全

2.地址空间的存在，可以更加方便的进行进程和进程的数据代码解耦，保证了进程的独立性。（进程=内核数据结构+进程对应的代码和数据）

3.让进程以统一的视角，来看待进程对应的代码和数据的各个区域，方便使用，编译器也以统一的视角来进行编译。

# 进程控制

## 进程创建

* + - 1. fork函数初识

在linux中fork函数时非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程，而原进程为父进程。

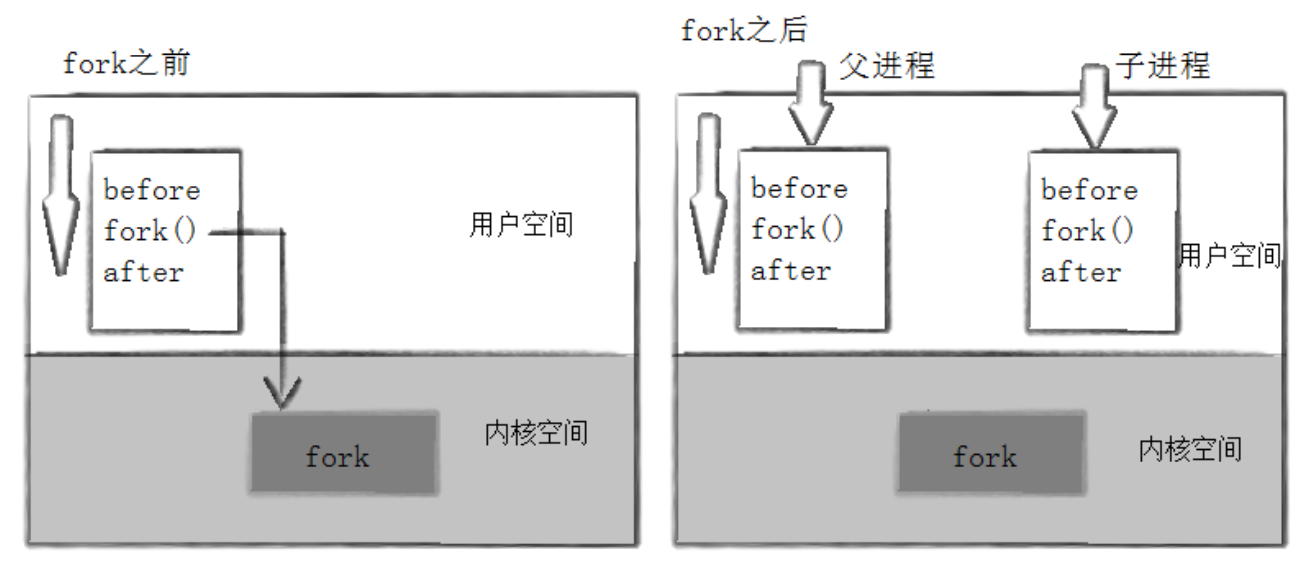
|  |
| --- |
| #include <unistd.h>  pid\_t fork(void); |

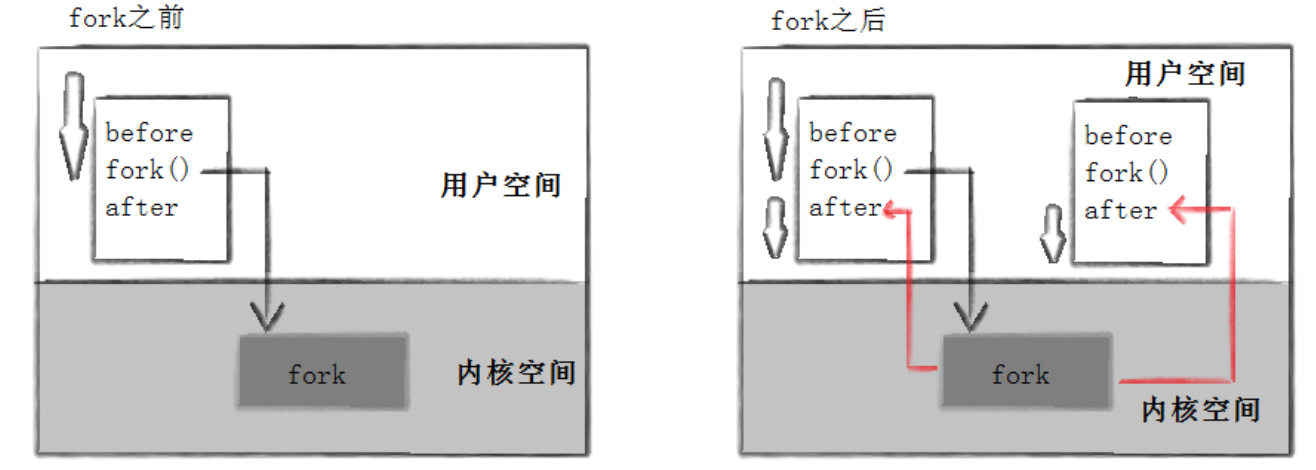
注：子进程返回0，创建成功父进程返回子进程id，失败返回-1。

进程调用fork，当控制转移到内核中的fork代码后，内核做：

* 分配新的内存块和内核数据结构给子进程
* 将父进程部分数据结构内容拷贝至子进程
* 添加子进程到系统进程列表当中
* fork返回，开始调度器调度

当一个进程调用fork之后，就有两个二进制代码相同的进程。而且它们都运行到相同的地方。fork之前父进程独立执行，fork之后，父子两个执行流分别执行。注意，fork之后，谁先执行完全由调度器决定





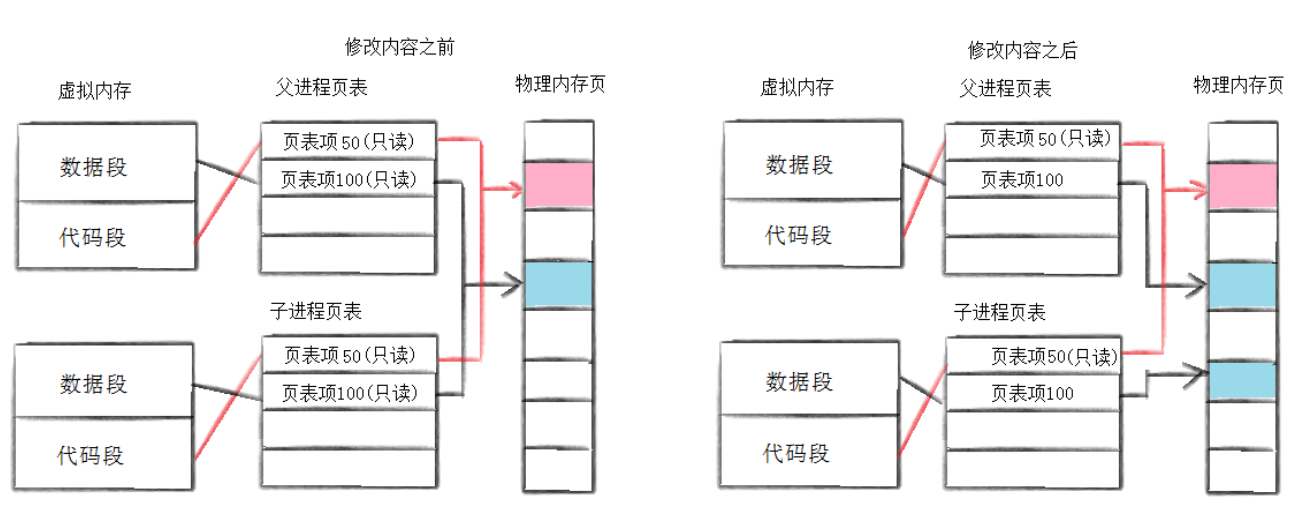
* + - 1. fork函数返回值

子进程返回0，父进程返回的是子进程的pid。

fork为什么会有两个返回值，因为fork函数返回之前，子进程已经创建好了，父子进程各自返回各自的返回值。

* + - 1. 写时拷贝

通常，父子代码共享，父子再不写入时，数据也是共享的，当任意一方试图写入，便以写时拷贝的方式各自一份副本。如下所示：



* + - 1. fork常规用法
* 一个父进程希望复制自己，使父子进程同时执行不同的代码段。例如，父进程等待客户端请求，生成子进程来处理请求。
* 一个进程要执行一个不同的程序。例如子进程从fork返回后，调用exec函数。
  + - 1. fork调用失败的原因
* 系统中有太多的进程
* 实际用户的进程数超过了限制
  + - 1. fork函数流程

创建子进程的PCB->赋值->创建子进程的地址空间->赋值->创建并设置页表->子进程放入进程list->...->返回子进程的pid给父进程

## 进程终止

* + - 1. 进程退出场景进程退出场景
* 代码运行完毕，结果正确--return 0
* 代码运行完毕，结果不正确--return !0
* 代码异常终止，退出码无意义
  + - 1. 进程常见退出方法

正常终止（可以通过 echo $? 查看进程退出码）：

* 从main返回
* 调用exit，一旦调用exit进程就终止
* \_exit

注：\_exit是系统调用，exit是库函数。exit的底层原理就是\_exit，但是exit会主动刷新缓冲区，而\_exit不会主动刷新缓冲区。

异常退出：ctrl + c，信号终止

* + - 1. \_exit函数

|  |
| --- |
| #include <unistd.h>  void \_exit(int status);  参数：status 定义了进程的终止状态，父进程通过wait来获取该值 |

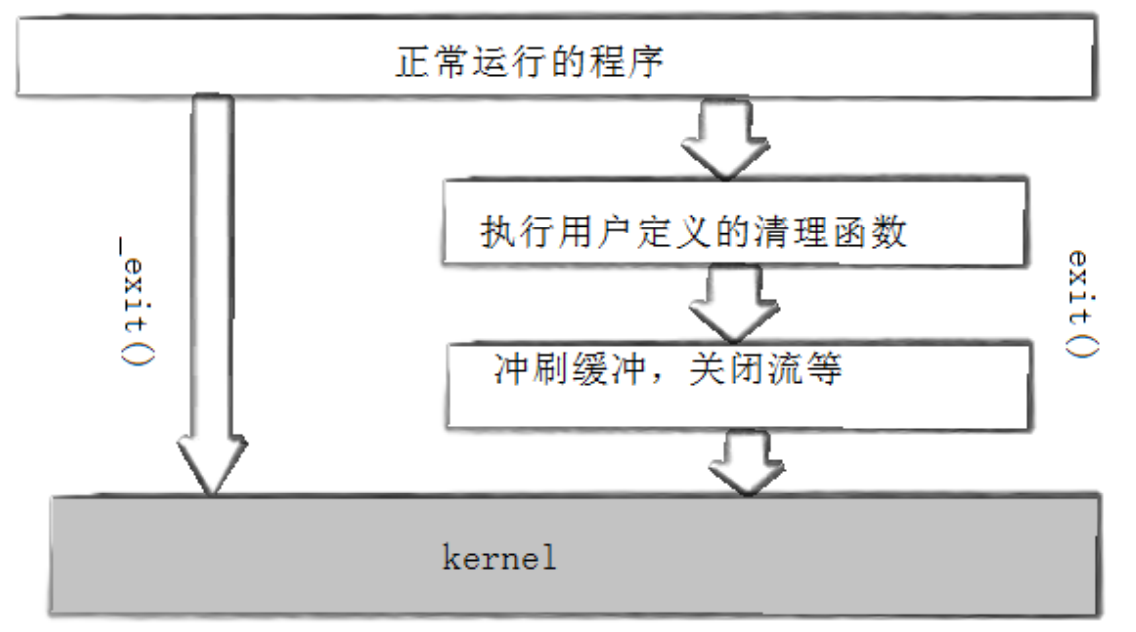
说明：虽然status是int，但是仅有低8位可以被父进程所用。所以\_exit(-1)时，在终端执行echo $?发现返回值是255。$?永远记录最近一个进程在命令行执行完毕时对应的退出码（main->return ?;）

* + - 1. exit函数

|  |
| --- |
| #include <unistd.h>  void exit(int status); |

exit最后也会调用\_exit, 但在调用exit之前，还做了其他工作：

* 执行用户通过atexit或on\_exit定义的清理函数。
* 关闭所有打开的流，所有的缓存数据均被写入
* 调用\_exit



* + - 1. return退出

return是一种更常见的退出进程方法。执行return n等同于执行exit(n),因为调用main的运行时函数会将main的返回值当做exit的参数。

## 进程等待

* + - 1. 进程等待必要性
* 子进程退出，父进程如果不管不顾，就可能造成‘僵尸进程’的问题，进而造成内存泄漏。
* 进程一旦变成僵尸状态，那就刀枪不入，“杀人不眨眼”的kill -9 也无能为力，因为谁也没有办法杀死一个已经死去的进程。
* 父进程派给子进程的任务完成的如何，我们需要知道。如，子进程运行完成，结果对还是不对，或者是否正常退出。
* 父进程通过进程等待的方式，回收子进程资源，获取子进程退出信息。
  + - 1. 进程等待的方法

wait方法

|  |
| --- |
| #include <sys/types.h>  #include <sys/wait.h>  pid\_t wait(int\*status);  返回值：  成功返回被等待进程pid，失败返回-1。  参数：  输出型参数，获取子进程退出状态，不关心则可以设置成为NULL。 |

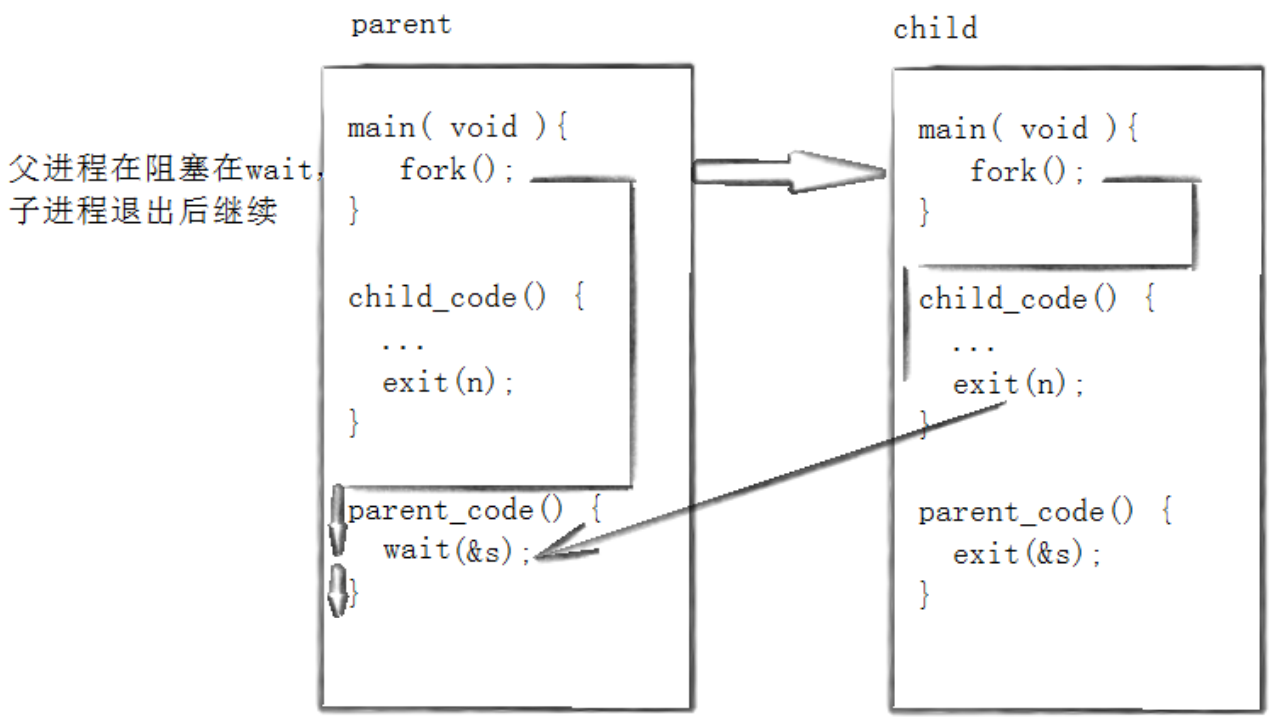
waitpid方法

|  |
| --- |
| pid\_ t waitpid(pid\_t pid, int \*status, int options);  返回值：   * 当正常返回的时候waitpid返回收集到的子进程的进程ID； * 如果设置了选项WNOHANG,而调用中waitpid发现没有已退出的子进程可收集,则返回0； * 如果调用中出错,则返回-1,这时errno会被设置成相应的值以指示错误所在；   参数：   * pid：   Pid=-1,等待任一个子进程。与wait等效。  Pid>0.等待其进程ID与pid相等的子进程。   * status:   WIFEXITED(status)：若为正常终止子进程返回的状态，则为真。（查看进程是否是正常退出）  WEXITSTATUS(status)：若WIFEXITED非零，提取子进程退出码。（查看进程的退出码）   * options:   WNOHANG：若pid指定的子进程没有结束，则waitpid()函数返回0，不予以等待。若正常结束，则返回该子进程的ID。 |

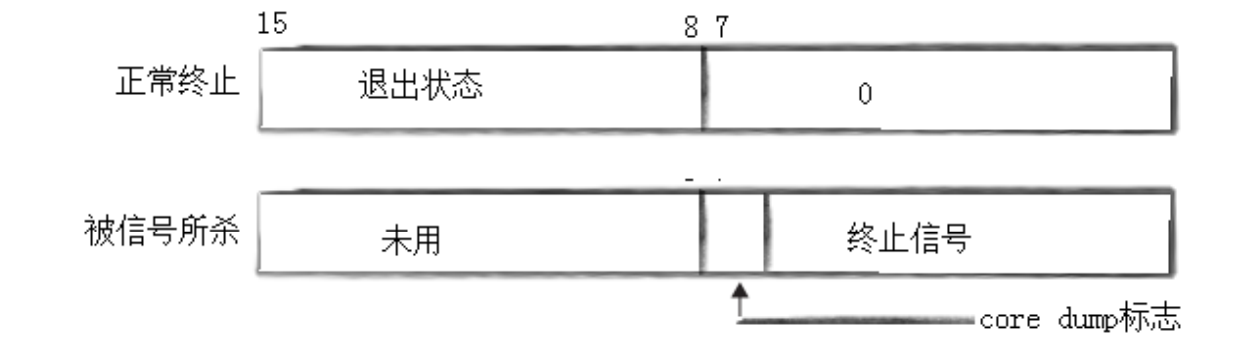
如果子进程已经退出，调用wait/waitpid时，wait/waitpid会立即返回，并且释放资源，获得子进程退出信息。

如果在任意时刻调用wait/waitpid，子进程存在且正常运行，则进程可能阻塞。

如果不存在该子进程，则立即出错返回。



* + - 1. 获取子进程status
* wait和waitpid，都有一个status参数，该参数是一个输出型参数，由操作系统填充。
* 如果传递NULL，表示不关心子进程的退出状态信息。
* 否则，操作系统会根据该参数，将子进程的退出信息反馈给父进程。
* status不能简单的当作整形来看待，可以当作位图来看待，具体细节如下图（只研究status低16比特位）：



* + - 1. waitpid的阻塞等待与非阻塞等待

阻塞等待：父进程希望一直等待子进程的结束，并且不需要急切执行其他任务时使用，这和wait函数的行为一样，等待子进程执行完毕之后，才继续执行父进程。阻塞等待适用于没有高响应需求并且子进程的任务量比较小，或者必须拿到子进程结果父进程才能继续的的程序，父进程可以接受等待子进程执行完再继续执行自己。

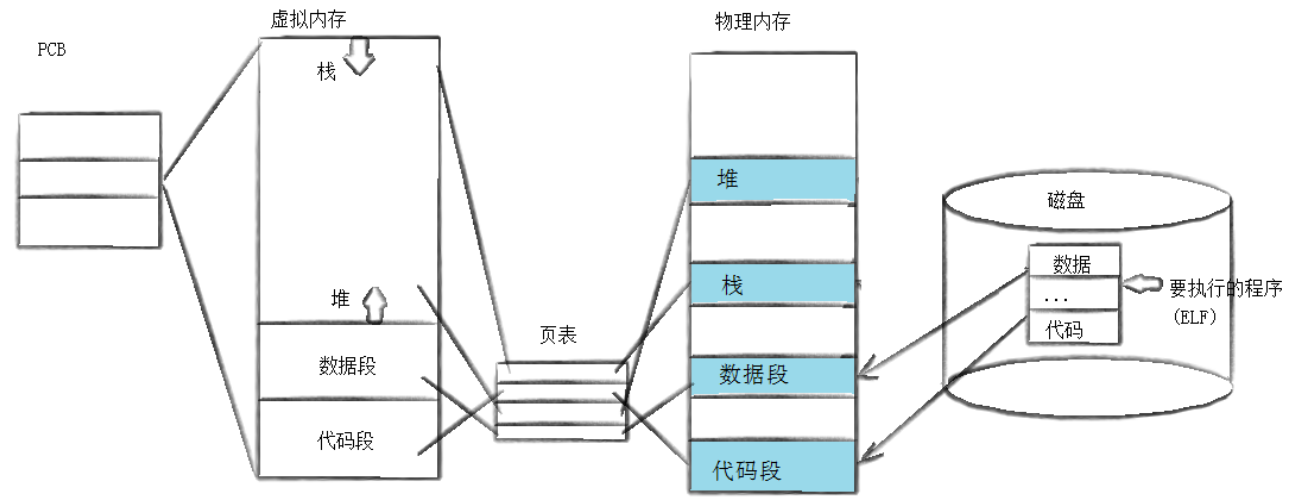
非阻塞等待（WNOHANG）：在父进程不希望被子进程阻塞，可以继续执行其他任务的场景中使用。适用于**高响应需求的程序或者子进程的任务量较大的**程序。麻烦点：需要父进程周期性地调用waitpid（轮询）以检查子进程是否已经终止。

## 程序替换exec

* + - 1. 替换原理

用fork创建子进程后执行的是和父进程相同的程序(但有可能执行不同的代码分支)，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用exec并不创建新进程，所以调用exec前后该进程的id并未改变。

注：exec也是一种程序加载器，一般在main函数之前调用。



* + - 1. 替换函数

其实有六种以exec开头的函数,统称exec函数:

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg, ...,char \*const envp[]);

int execv(const char \*path, char \*const argv[]);

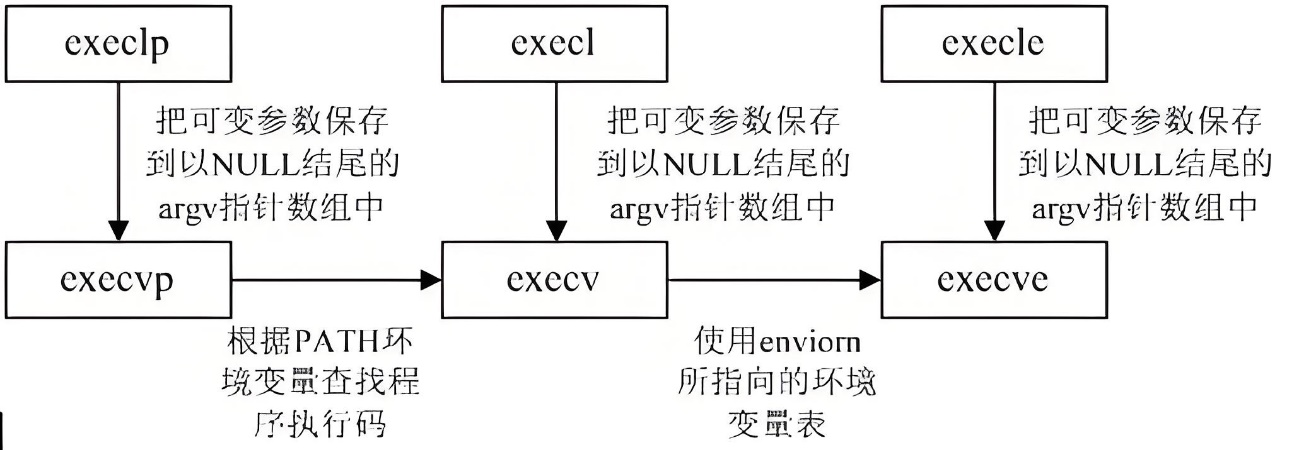
int execvp(const char \*file, char \*const argv[]);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

* + - 1. 函数解释
* 这些函数如果调用成功则加载新的程序从启动代码开始执行，不再返回。
* 如果调用出错则返回-1
* 所以exec函数只有出错的返回值而没有成功的返回值。
  + - 1. 命名理解
* l(list) : 表示参数采用列表
* v(vector) : 参数用数组
* p(path) : 有p自动搜索环境变量PATH
* e(env) : 表示自己维护环境变量

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名** | **参数格式** | **是否带路径** | **是否使用当前环境变量** |
| execl | 列表 | 不是 | 是 |
| execlp | 列表 | 是 | 是 |
| execle | 列表 | 不是 | 不是，自己组装环境 |
| execv | 数组 | 不是 | 是 |
| execvp | 数组 | 是 | 是 |
| execve | 数组 | 不是 | 不是，自己组装环境 |

事实上，只有execve是真正的系统调用，其它五个函数最终都调用execve，所以execve在man手册第2节，其它函数在man手册第3节。这些函数之间的关系如下图所示。

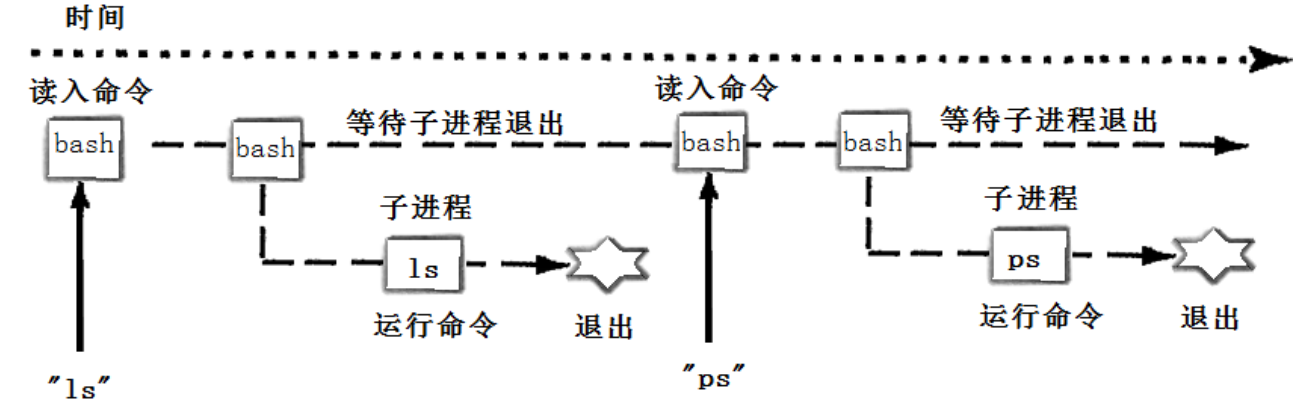


注：man手册分8章，具体细节如下：

|  |  |
| --- | --- |
| 章节号 | 手册类型 |
| 1 | 可执行程序或shell命令 |
| 2 | 系统调用（内核提供的函数） |
| 3 | 库调用和库函数 |
| 4 | 设备文件说明（通常位于/dev） |
| 5 | 文件格式和规范（如/etc/psaawd） |
| 6 | 游戏（games） |
| 7 | 杂项，惯例和协议等，如Linux文件系统、网络协议、ASCII代码等的说明。 |
| 8 | 系统管理命令（针对root用户） |
| 9 | 内核例程 |

* + - 1. 基于程序替换的简易shell

用下图的时间轴来表示事件的发生次序。其中时间从左向右。shell由标识为sh的方块代表，它随着时间的流逝从左向右移动。shell从用户读入字符串"ls"。shell建立一个新的进程，然后在那个进程中运行ls程序并等待那个进程结束。



然后shell读取新的一行输入，建立一个新的进程，在这个进程中运行程序 并等待这个进程结束。所以要写一个shell，需要循环以下过程:

1.获取命令行

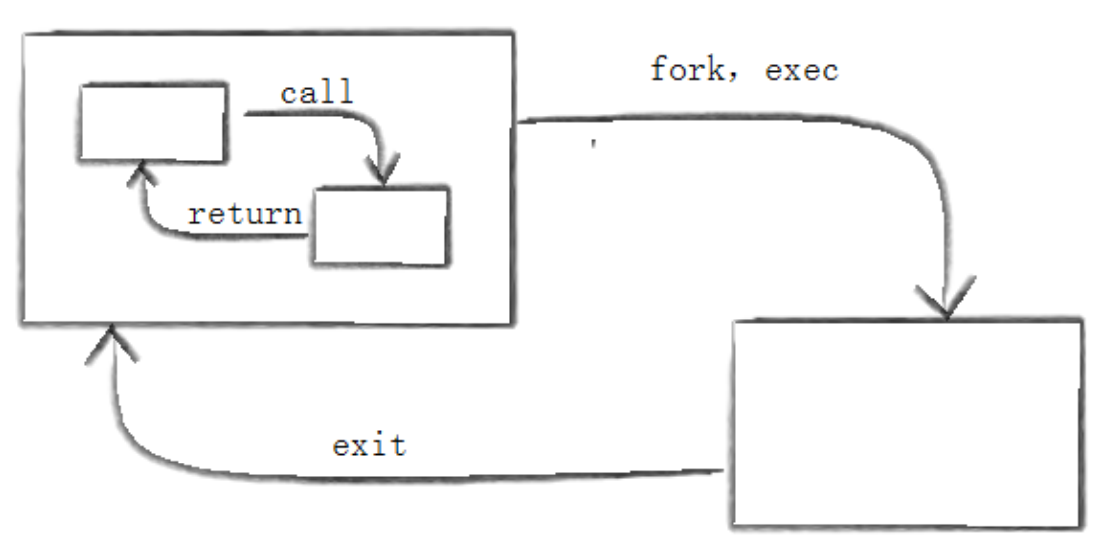
2.解析命令行

3.建立一个子进程（fork）

4.替换子进程（execvp）

5.父进程等待子进程退出（wait/waitpid）

一个C程序有很多函数组成。一个函数可以调用另外一个函数，同时传递给它一些参数。被调用的函数执行一定的操作，然后返回一个值。每个函数都有他的局部变量，不同的函数通过call/return系统进行通信。这种通过参数和返回值在拥有私有数据的函数间通信的模式是结构化程序设计的基础。Linux鼓励将这种应用于程序之内的模式扩展到程序之间。如下图



一个C程序可以fork/exec另一个程序，并传给它一些参数。这个被调用的程序执行一定的操作，然后通过exit(n)来返回值。调用它的进程可以通过wait（&ret）来获取exit的返回值。

# 文件系统

## C文件函数和系统调用接口

|  |  |
| --- | --- |
| C接口 | 系统调用 |
| fopen | open |
| fclose | close |
| fwrite | write |
| fread | read |
| fseek | lseek |

f#系列的函数，都是对系统调用的封装，方便二次开发。

## 文件描述符fd

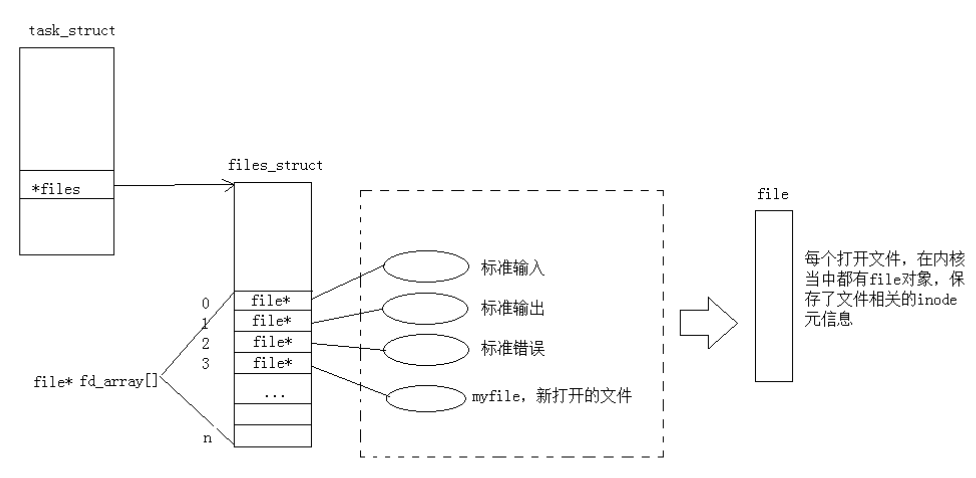
通过对open函数的学习，我们知道了文件描述符就是一个小整数.

### 3个默认打开文件

Linux进程默认情况下会有3个缺省打开的文件描述符，分别是标准输入0， 标准输出1，标准错误2.

0,1,2对应的物理设备一般是：键盘，显示器，显示器。

进程PCB与打开的文件之间是通过文件描述符表联系起来的，如下所示：



文件描述符就是从0开始的小整数。当我们打开文件时，操作系统在内存中要创建相应的数据结构来描述目标文件。于是就有了file结构体。表示一个已经打开的文件对象。而进程执行open系统调用，所以必须让进程和文件关联起来。每个进程都有一个指针\*files，指向一张表files\_struct,该表最重要的部分就是包涵一个指针数组，每个元素都是一个指向打开文件的指针！所以，本质上，文件描述符就是该数组的下标。所以，只要拿着文件描述符，就可以找到对应的文件。

### 文件描述符的分配规则

直接打开一个文件时，它的文件描述符是fd：3。如果关闭0或者2，而此时打开的文件描述符是fd：0或者fd：2。所以文件描述符的分配规则：在files\_struct数组当中，找到当前没有被使用的最小的一个下标，作为新的文件描述符。

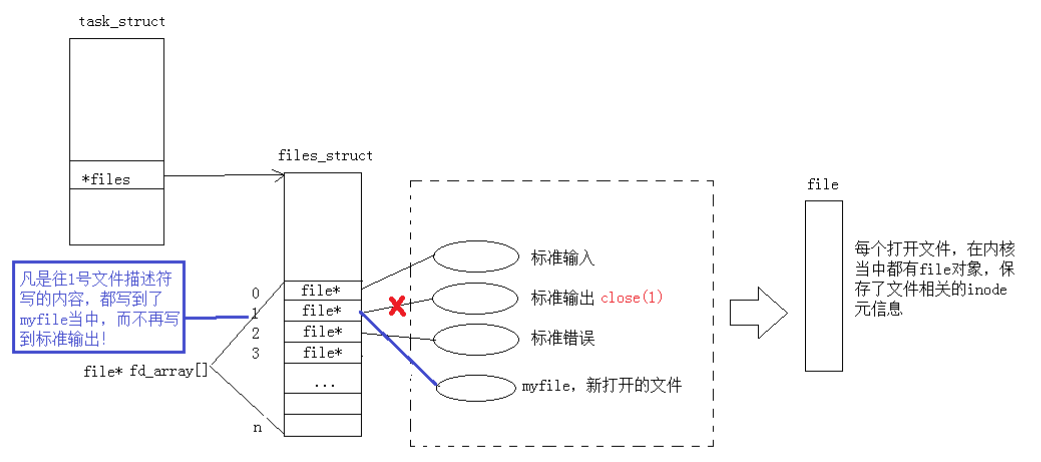
## 重定向

当标准输出1号文件描述符被关闭时，此时新创建的文件描述符就会是1号。有如下代码：



本来应该输出到显示器上的内容，输出到了文件 myfile 当中，其中，fd＝1。这种现象叫做**输出重定向**。

重定向的本质是：上层的fd不变，在内核中更改fd对应的struct file\*的地址！常见的重定向有：①输入重定向>；②追加重定向>>；③输入重定向<



## dup2 系统调用

dup2() makes newfd be the copy of oldfd, closing newfd first if necessary.

int dup2(int oldfd, int newfd) //调用后以oldfd为准

如果不关闭1号文件描述符，让原先打印到显示器的字符打印到新打开的文件myfile中，它的文件描述符是fd，那么用命令dup2（fd，1）；即可。

注：关于文件读写，第一步操作就是建立进程PCB的文件描述符表与文件之间的关系，所以open就是有这个作用的，在files\_struct数组当中，找到当前没有被使用的最小的一个下标，再在数组中存下所打开文件的地址，支持后续的读写操作。默认有三个文件描述符被打开，就是0号（stdin）、1号（stdout）、2号（stderr），所以对这三个文件读写时不需要open操作，可以直接进行读（cin/scanf）和写（cout/printf）操作。读和写都是文件与内存之间进行交互的操作，读就是从文件向内存输出内容，而写就是从内存向文件输出内容。

## FILE结构体及缓冲区

因为IO相关函数与系统调用接口对应，并且库函数封装系统调用，所以本质上，访问文件都是通过fd访问的。所以C库当中的FILE结构体内部，必定封装了fd。首先看如下代码



当代码直接完毕之后输出的代码是

|  |
| --- |
| hello printf  hello fprintf  hello fputs  hello write |

但如果对进程实现输出重定向时， ./hello > myfile ，结果变成了：

|  |
| --- |
| hello write  hello printf  hello fprintf  hello fputs  hello printf  hello fprintf  hello fputs |

结果就是printf、fwrite以及fgets（C库函数）都输出了2次，而write（系统调用）只输出了一次。与fork函数有关，具体原因如下：

一般C库函数写入文件时是全缓冲（缓冲区满了之后再刷新），而写入显示器是行缓冲。printf、fwrite以及fgets等C库函数会自带缓冲区，缓冲区的本质就是一段内存。当发生重定向到普通文件时，数据的缓冲方式由行缓冲变成了全缓冲。所以此时放在缓冲区的数据，没有被立刻刷新，fork之后还是没有刷新。但是进程退出之后，会统一刷新，写入文件当中。但是fork的时候，父子数据会发生写时拷贝，所以当你父进程准备刷新的时候，子进程也就有了同样的一份数据，随即产生两份数据。write没有变化，说明没有所谓的缓冲。

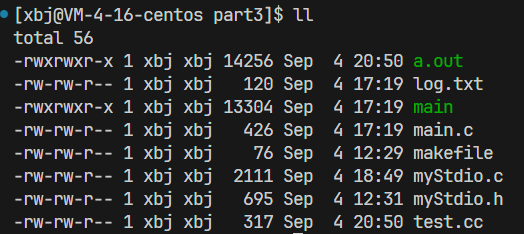
缓冲区一般有三种刷新策略，一般结合具体的设备：

* 立即刷新，无缓冲
* 行刷新，行缓存：显示器
* 缓冲区满，全缓冲：磁盘文件

一般用户也可以使用fflush函数强制刷新，或者进程退出时也会进行缓冲区刷新，如果使用系统调用\_exit()退出程序，则缓冲区的内容全部丢失。

## 理解文件系统/软硬链接

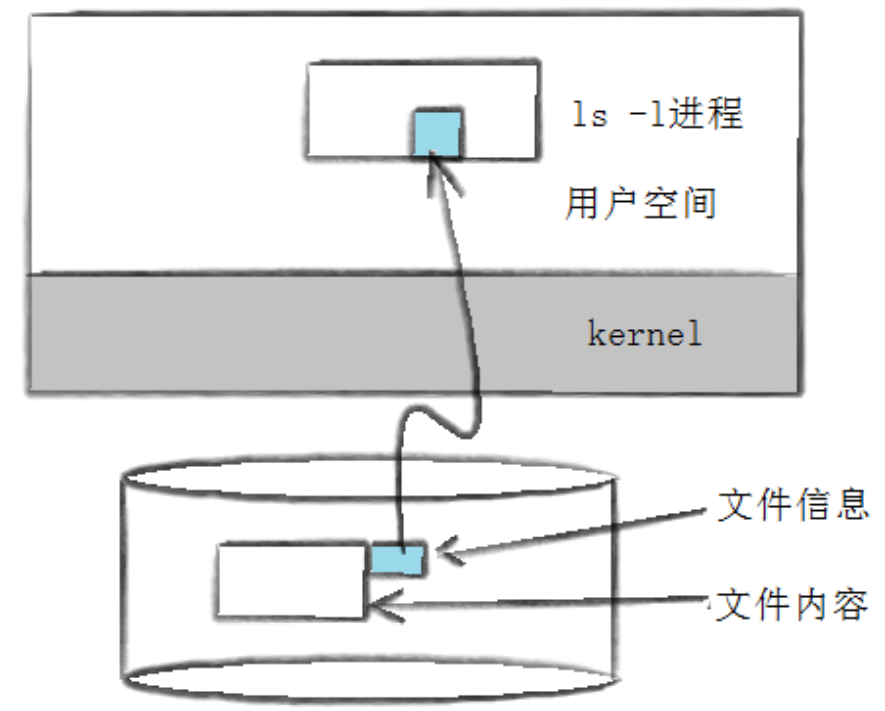
我们使用ls -l的时候看到的除了看到文件名，还看到了文件元数据。



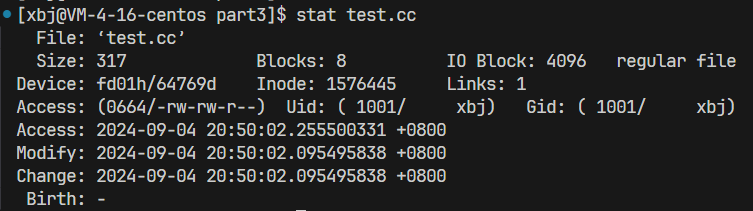
每一行包含7列，每一列代表的意思如下：

* 模式
* 硬链接数
* 文件所有者
* 文件所有组
* 大小
* 最后修改时间
* 文件名

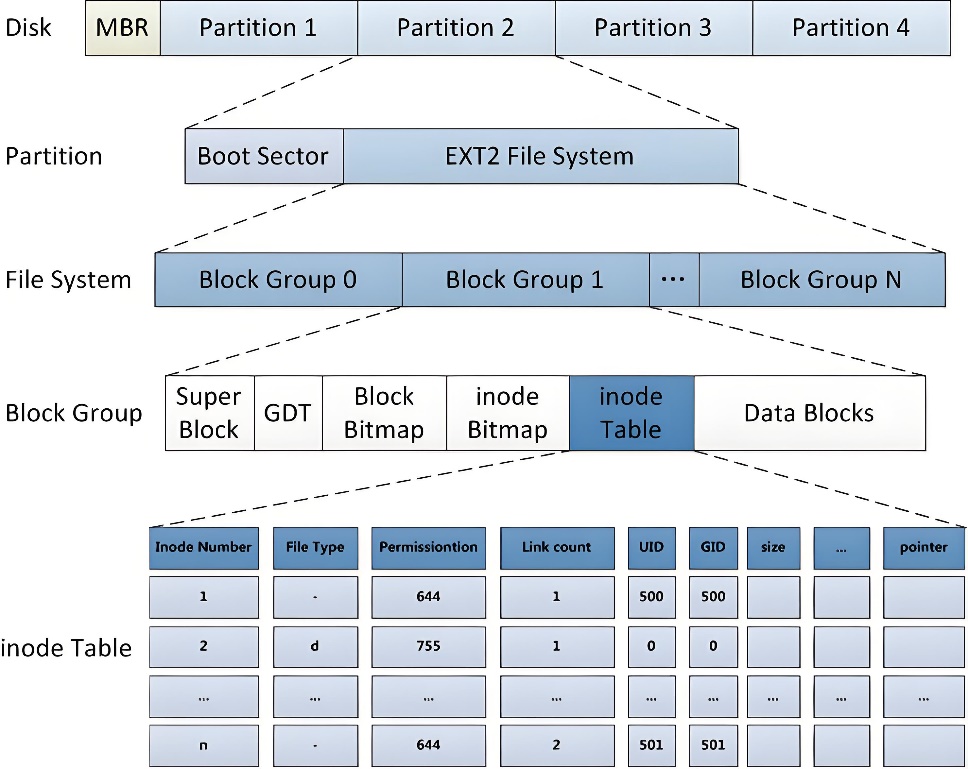
ls -l读取存储在磁盘上的文件信息，然后显示出来，如下图所示：



其实这个信息除了通过这种方式来读取，还有一个stat命令能够看到更多信息：stat - display file or file system status.



* Size：文件的大小是317字节
* Blocks: 8：文件占用了8个块（block）的存储空间。注意，块的大小可能大于文件的大小，因此文件可能不会完全占用所有块的全部空间。
* IO Block : 4096：文件系统的I/O块大小是4096字节。这是文件系统进行读写操作时的最小单位。
* regulal file：表明这是一个普通文件，而不是目录、设备文件等其他类型的文件
* Device: fd01h/64769d：文件所在的设备的主要和次要设备号。fd01h是十六进制表示的主要设备号，/64769d可能是系统特定或格式化后的表示方式，但通常我们只关注fd01h部分。
* Inode: 1576445：文件的inode号。inode是文件系统中用于存储文件元数据（如权限、链接数、大小等）的数据结构。
* Links: 1：有多少文件名链接到这个inode。对于普通文件来说，这个值通常是1，因为只有一个文件名指向它。对于目录来说，这个值会更高，因为目录中的每个子目录和.、..链接都会增加这个值。
* Access: (0664/-rw-rw-r--)：文件的访问权限，以八进制（0664）和符号形式（-rw-rw-r--）表示。这里表示文件所有者和所属组有读写权限，而其他用户只有读权限。
* Uid: ( 1001/ username)：文件所有者的用户ID（Uid）和用户名。
* Gid: ( 1001/ groupname)：文件所属组的组ID（Gid）和组名。
* Access、Modify、Change分别显示了文件的最后访问时间（atime）、最后修改时间（mtime）和最后状态改变时间（ctime）。这些时间戳通常包含日期、时间和时区信息（在这个例子中是UTC时间）。
* Birth时间通常表示文件被创建的时间，但在大多数Unix/Linux文件系统中（如ext4），这个信息并不总是可用的。在这个例子中，-表示没有可用的创建时间信息。



Disk（磁盘）=MBR（主引导记录）+ Partition（分区）

Partition（分区）= Boot Sector（引导扇区）+EXT2 File System**（**第二代扩展文件系统）

File System（文件系统）=N × Block Group(块组)

Block Group= Super Block（超级块）+GDT（块组描述符表）+Block Bitmap（块位图）+inode Bitmap(inode位图)+ inode Table（inode表）+ Data Blocks(数据块)

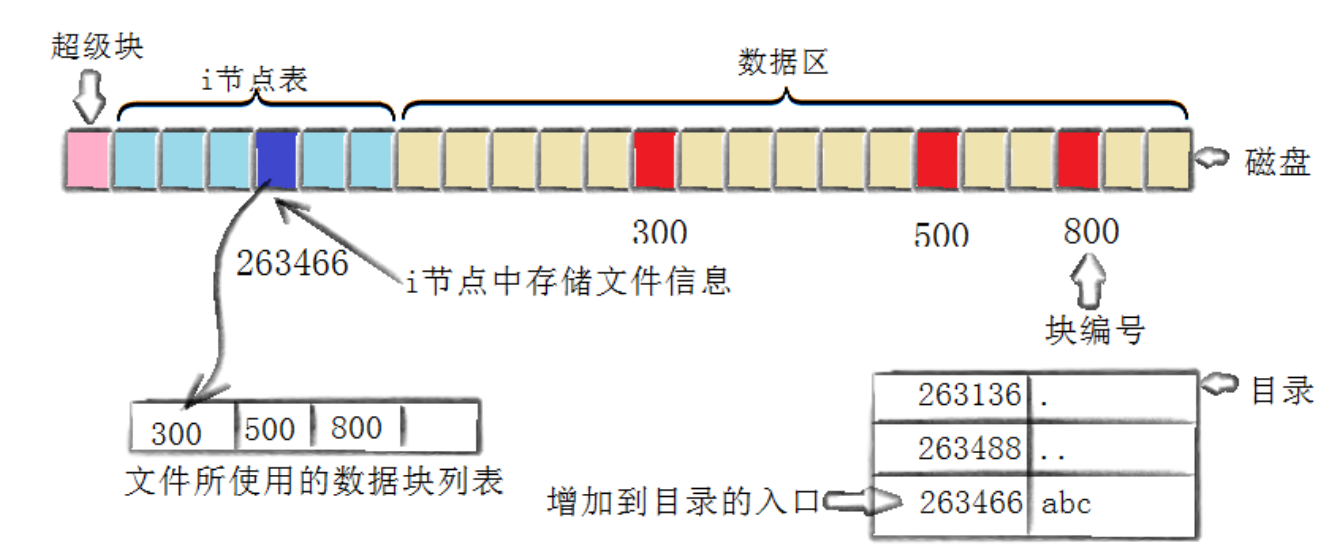
Linux ext2文件系统，上图为磁盘文件系统图（内核内存映像肯定有所不同），磁盘是典型的块设备，硬盘分区被划分为一个个的block。一个block的大小是由格式化的时候确定的，并且不可以更改。例如mke2fs的-b选项可以设定block大小为1024、2048或4096字节。而上图中启动块（Boot Sector）的大小是确定的。

* Block Group：ext2文件系统会根据分区的大小划分为数个Block Group。而每个Block Group都有着相同的结构组成。
* 超级块（Super Block）：存放文件系统本身的结构信息。记录的信息主要有：bolck和inode的总量，未使用的block和inode的数量，一个block和inode的大小，最近一次挂载的时间，最近一次写入数据的时间，最近一次检验磁盘的时间等其他文件系统的相关信息。Super Block的信息被破坏，可以说整个文件系统结构就被破坏了，一个分区中有多个超级块，互相属于备份。
* GDT，Group Descriptor Table：块组描述符，描述块组属性信息。
* 块位图（Block Bitmap）：Block Bitmap中记录着Data Block中哪个数据块已经被占用，哪个数据块没有被占用。
* inode位图（inode Bitmap）：每个bit表示一个inode是否空闲可用。
* inode表:存放文件属性，如文件大小，所有者，最近修改时间等，inode是跨组的，但是不能跨分区，每一个文件都有一个inode编号。
* 数据区：存放文件内容。

注：删除文件时，把对应的inode Bitmap位置和Block Bitmap位置置零即删除，并没有真正地把Data Blocks中的数据删除，这是一种惰性删除。

目录也是一个文件，也有inode和数据块，它的数据块中存放的是该目录下所有文件名和对应inode之间的关系。

将属性和数据分开存放的想法看起来很简单，但实际上是如何工作的呢？我们通过touch一个新文件来看看如何工作。



所以创建一个文件主要有以下四个操作：

* 存储属性：内核先找到一个空闲的i节点（这里是263466）。内核把文件信息记录到其中。
* 存储数据：该文件需要存储在三个磁盘块，内核找到了三个空闲块：300,500，800。将内核缓冲区的第一块数据复制到300，下一块复制到500，以此类推。
* 记录分配情况：文件内容按顺序300,500,800存放。内核在inode上的磁盘分布区记录了上述块列表。
* 添加文件名到目录。

新的文件名abc。linux如何在当前的目录中记录这个文件？内核将入口（263466，abc）添加到目录文件。文件名和inode之间的对应关系将文件名和文件的内容及属性连接起来。

软硬链接的区别：是否具有独立的inode

* 软链接具有独立的inode--可以被当作独立的文件看待
* 硬链接没有独立的inode--建立硬链接，没有新增文件，因为没有给硬链接分配独立的inode。创建硬链接时，只是在指定路径下，新增文件名和inode编号的映射关系，在inode表的硬链接数（引用计数）中增加1。

问：那么什么时候一个文件算被真正删除？删除一个文件时操作步骤有哪些？

答：当一个文件的硬链接数变为0时，这个文件才被真正删除。删除文件时一般步骤如下：1.在目录中将对应的记录删除，2.将硬连接数-1，如果为0，则将对应的磁盘释放（inode Table中删除对应的inode信息，inode Bitmap对应位置置零，Block Bitmap对应位置置零。

软链接相当于快捷方式，硬链接最常用于目录中两个文件：.(当前目录)和..（上级目录）。Linux不允许普通用户给目录建立硬链接！

## 动静态库

静态库（.a）：程序在编译链接的时候把库的代码链接到可执行文件中。程序运行的时候将不再需要静态库。

动态库（.so）：程序在运行的时候才去链接动态库的代码，多个程序共享使用库的代码。

一个与动态库链接的可执行文件仅仅包含它用到的函数入口地址的一个表，而不是外部函数所在目标文件的整个机器码。

在可执行文件开始运行以前，外部函数的机器码由操作系统从磁盘上的该动态库中复制到内存中，这个过程称为动态链接（dynamic linking）。

动态库可以在多个程序间共享，所以动态链接使得可执行文件更小，节省了磁盘空间。操作系统采用虚拟内存机制允许物理内存中的一份动态库被要用到该库的所有进程共用，节省了内存和磁盘空间。

注：库的本质就是.o文件的集合，不同的打包方式形成动态库和静态库。安装库的本质就是拷贝。

* + - 1. 静态库和静态链接

**生成静态库命令**：利用gcc -c生成.o文件，再用ar打包到一起成为库

|  |
| --- |
| [root@localhost linux]# gcc -c add.c -o add.o  [root@localhost linux]# gcc -c sub.c -o sub.o  [root@localhost linux]#ar -rc libmymath.a my\_add.o my\_sub.o |

* ar是gnu归档工具
* rc表示(replace and create)

**查看静态库命令：**

|  |
| --- |
| [root@localhost linux]# ar -tv libmymath.a |

* t:列出静态库中的文件
* v:verbose 详细信息

**使用静态库命令：**

|  |
| --- |
| [root@localhost linux]# gcc main.c -I . -L . -lmymath |

* -I：指定头文件路径，不指定优先当前路径
* -L：指定库路径，
* -l：指定库名
* 以上三个选项带不带空格都可以

注：形成一个可执行程序，可能不仅仅依赖一个库。gcc是默认动态链接的，对于特定的一个库，取决于所提供的是静态或者动态库。而gcc所链接的所有库中，只要有一个动态库，那链接属性就是动态链接。

* + - 1. 库搜索路径
* 从左到右搜索-L指定的目录。
* 由环境变量指定的目录 （LIBRARY\_PATH）
* 由系统指定的目录：/usr/lib、/usr/local/lib /usr/lib
  + - 1. 动态库和动态链接

**生成动态库：**利用gcc -c-fPIC生成.o文件，再用gcc -shared打包成库。

|  |
| --- |
| [root@localhost linux]# gcc -fPIC -c sub.c add.c  [root@localhost linux]# gcc -shared -o libmymath.so \*.o |

* fPIC：产生位置无关码(position independent code)
* shared: 表示生成共享库格式
* 库名规则：libxxx.so

**使用动态库：**使用动态链接库生成可执行程序，需要给gcc指定头文件、库文件路径以及库文件名称。

|  |
| --- |
| [root@localhost linux]#gcc main.o -o main –L. -lhello |

* -I：指定头文件路径
* -L：指定库文件路径
* -l：库文件名称

**运行链接动态库的可执行程序：**运行可执行程序时也需要给操作系统指定库文件名称和库文件路径。有以下几种解决方法：

* 1.拷贝.so文件到系统共享库路径下, 一般指/usr/lib
* 2.将当前库文件的路径添加到环境变量LD\_LIBRARY\_PATH中

|  |
| --- |
| [root@localhost linux]# export LD\_LIBRARY\_PATH=. |

* 3. ldconfig 配置/etc/ld.so.conf.d/，ldconfig更新

|  |
| --- |
| [root@localhost linux]# cat /etc/ld.so.conf.d/bit.conf /root/tools/linux  [root@localhost linux]# ldconfig |

* 4.给库文件新增软连接到当前路径/系统路径下
  + - 1. 动静态库的加载

静态库在编译时被链接到可执行文件中。编译器会将静态库中的所有必要的代码直接复制到生成的可执行文件中。当程序运行时，不再需要静态库文件，因为所有相关代码已经嵌入到了可执行文件中。

优点是运行时不需要依赖库文件，程序可以独立运行。链接时会将库的相关部分复制到可执行文件中，减少了运行时加载开销。

缺点是可执行文件会变得较大，因为每个使用静态库的程序都会有自己独立的库代码副本。如果库更新，必须重新编译整个程序才能使用更新后的库。

动态库在程序运行时加载，通常通过动态链接加载（由操作系统在加载可执行文件时自动完成）或通过显式加载（使用函数如dlopen或LoadLibrary）。可执行文件本身只包含对动态库的引用，而不包含库的实际代码。运行时，系统会找到相应的动态库，并将其加载到进程地址空间的（共享区）中。

优点是可执行文件体积较小，因为库代码在运行时共享，不需要嵌入到每个可执行文件中。动态库可以独立更新，程序无需重新编译即可使用库的更新版本。

缺点是运行时需要动态库的存在，否则程序无法运行。因为动态链接在运行时发生，加载库会引入一些额外的性能开销。版本不兼容问题：不同的程序可能需要同一个动态库的不同版本。

|  |  |  |
| --- | --- | --- |
| 特性 | 静态库 | 动态库 |
| 加载时间 | 编译时加载并链接 | 运行时动态加载 |
| 可执行文件大小 | 较大，因为库代码嵌入到可执行文件中 | 较小，库代码独立于可执行文件 |
| 运行时依赖 | 无需依赖库文件 | 运行时必须依赖库文件 |
| 更新库的方式 | 更新库后需重新编译程序 | 更新库后无需重新编译程序，直接使用 |
| 性能 | 无需运行时加载，速度快 | 运行时加载库，可能有性能开销 |
| 代码共享 | 每个可执行文件有独立的库代码 | 多个程序可以共享相同的库实例 |

选择静态库还是动态库，取决于具体的项目需求和使用场景。如果需要独立分发程序且不依赖外部库文件，静态库更合适；而如果需要减少可执行文件体积，或者库可能会经常更新，则动态库是更好的选择。

# 进程通信

## 进程通信的简介

### 进程间通信目的

* 数据传输：一个进程需要将它的数据发送给另一个进程
* 资源共享：多个进程之间共享同样的资源。
* 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
* 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另 一个进程的所有陷入和异常，并能够及时知道它的状态改变。

### 进程间通信的发展

* 管道
* System V进程间通信
* POSIX进程间通信

### 进程间通信分类

管道：匿名管道pipe、命名管道

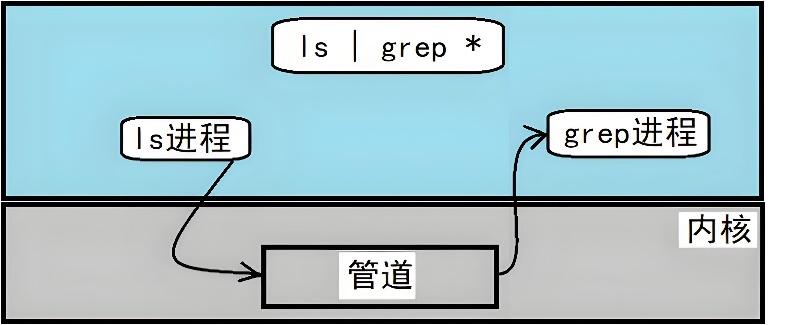
System V IPC：消息队列、共享内存、信号量

POSIX IPC：消息队列、共享内存、信号量、互斥量、条件变量、读写锁

## 管道

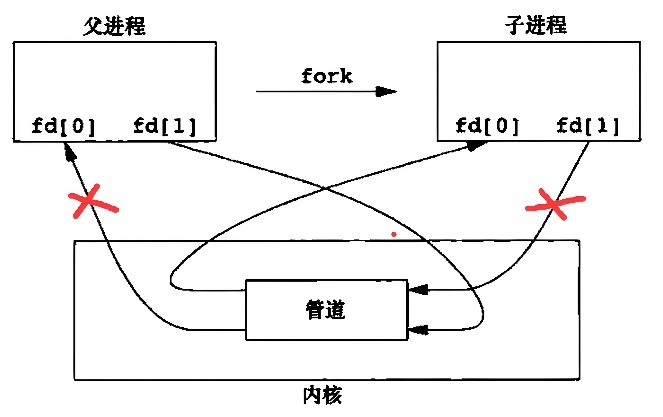
### 什么是管道

管道是Unix中最古老的进程间通信的形式。我们把从一个进程连接到另一个进程的一个数据流称为一个“管道”。

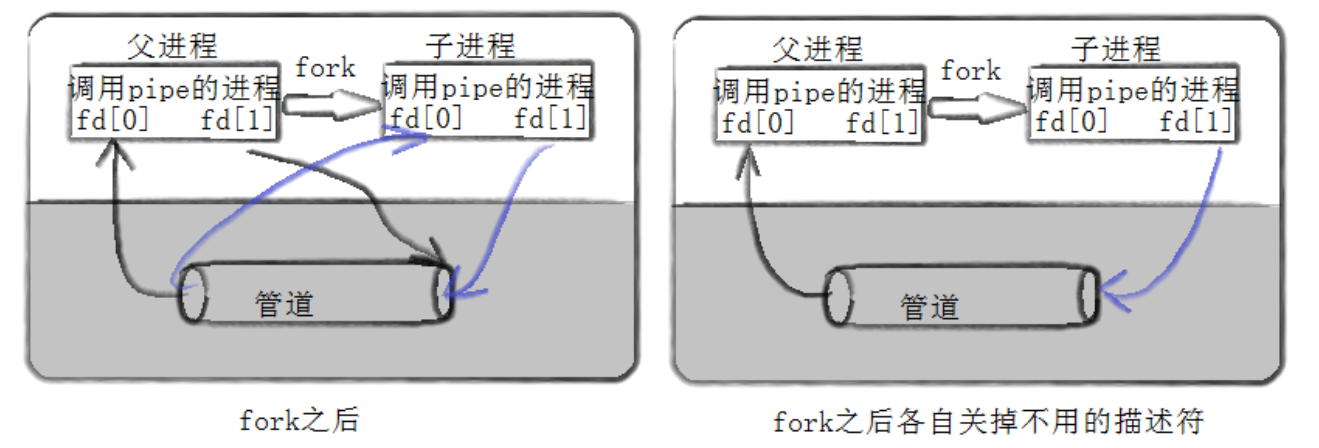


### 匿名管道

* + - 1. 用pipe命令创建无名管道，函数原型：int pipe(int fd[2])
* fd[0]:read
* fd[1]:write
* 返回值：成功返回0，失败返回-1



* + - 1. 一般用fork来共享管道，原理如下：



* + - 1. 文件描述符角度

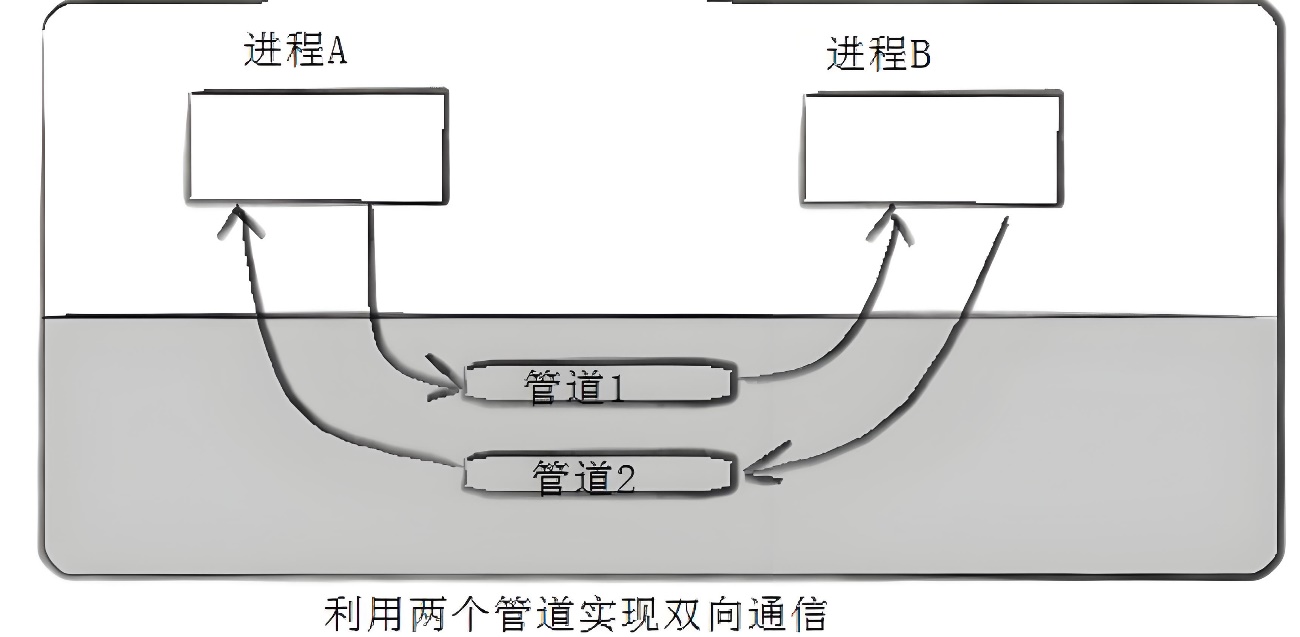
本质就是OS给通信进程双方提供的一段共享空间！！看待管道就和文件一样。



* + - 1. 管道读写规则
* 读慢，写快：写满管道时，写进程会阻塞等待
* 读快，写慢：读进程调用阻塞，进程暂停，直到数据来为止
* 写关闭，读到0：read返回0
* 读关闭：write操作产生SIGPIPE，进而write进程退出

注：Linux系统中管道的大小是64KB；管道被写满时，写进程要等待读进程取数据，管道写满以后，读进程每次至少要读取4KB，即4096个字符，写进程才会继续写入；当管道里没有数据可读的时候，读进程会等待子进程写入数据。读进程退出以后，写进程不会继续写入；写进程推出后，读进程在读完数据后进程退出。

* + - 1. 管道特点
* 只能用于具有共同祖先的进程（具有亲缘关系的进程）之间进行通信；通常，一个管道由一个进程创建，然后该进程调用fork，此后父、子进程之间就可应用该管道。
* 管道提供流式服务（网络）
* 一般而言，进程退出，管道释放，所以管道的生命周期随进程
* 一般而言，内核会对管道操作进行同步与互斥
* 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道



### 命名管道

管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。命名管道是一种特殊类型的文件。

* + - 1. 创建一个命名管道

命名管道可以从命令行上创建，命令行方法是使用下面这个命令：

**mkfifo filename**

命名管道也可以从程序里创建，相关函数有：

**int mkfifo(const char \*filename, mode\_t mode);**

* filename：创建管道的名字
* mode:设置管道文件的权限
* return value：若成功则返回0，否则返回-1，错误原因存于errno中。
  + - 1. 匿名管道与命名管道的区别
* 匿名管道由pipe函数创建并打开，命名管道由mkfifo函数创建，打开用open。
* FIFO（命名管道）与pipe（匿名管道）之间唯一的区别在它们创建与打开的方式不同，一但这些工作完成之后，它们具有相同的语义。
  + - 1. 命名管道的打开规则

如果当前打开操作是为读而打开FIFO时

* O\_NONBLOCK disable：阻塞直到有相应进程为写而打开该FIFO
* O\_NONBLOCK enable：立刻返回成功

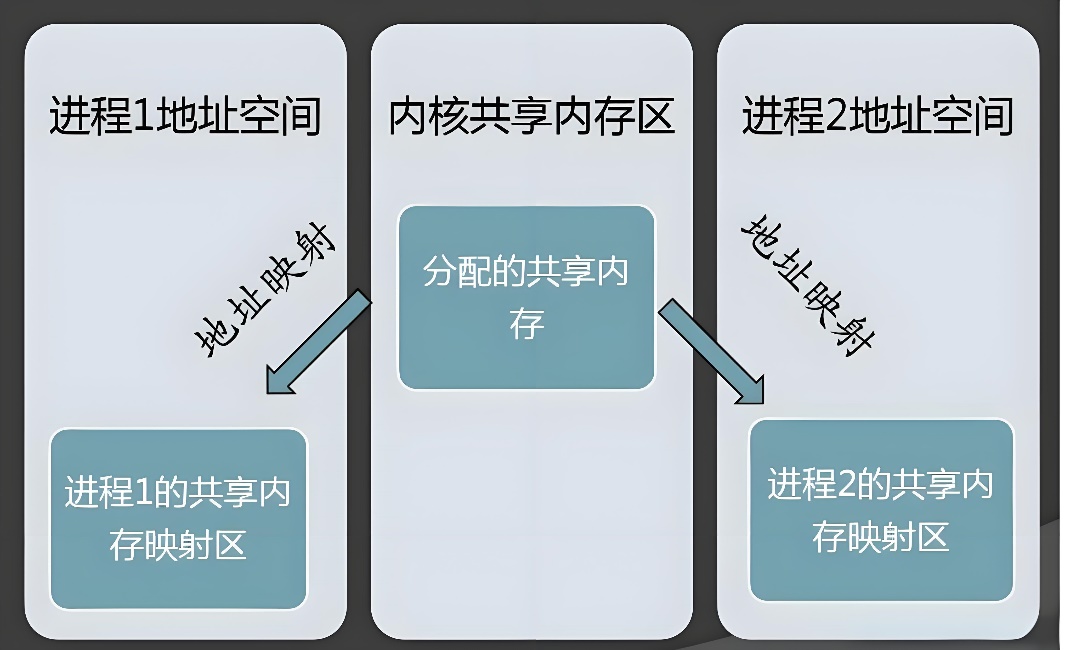
如果当前打开操作是为写而打开FIFO时

* O\_NONBLOCK disable：阻塞直到有相应进程为读而打开该FIFO
* O\_NONBLOCK enable：立刻返回失败，错误码为ENXIO

## system V共享内存

共享内存区是最快的IPC形式。一旦这样的内存映射到共享它的进程的地址空间，这些进程间数据传递不再涉及到内核，换句话说是进程不再通过执行进入内核的系统调用来传递彼此的数据。

共享内存的示意图如下：



共享内存的数据结构



共享内存函数

* + - 1. shmget函数

功能：用来创建共享内存

原型：int shmget(key\_t key, size\_t size, int shmflg);

参数：

* key:这个共享内存段名字
* size:共享内存大小
* shmflg:由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的

返回值：成功返回一个非负整数，即该共享内存段的标识码；失败返回-1

* + - 1. shmat函数

功能：将共享内存段连接到进程地址空间

原型：void \*shmat(int shmid, const void \*shmaddr, int shmflg);

参数

* shmid: 共享内存标识
* shmaddr:指定连接的地址。当设置为NULL时，核心自动选择一个地址；不为NULL且shmflg无SHM\_RND标记，则以shmaddr为连接地址；shmaddr不为NULL且shmflg设置了SHM\_RND标记，则连接的地址会自动向下调整为SHMLBA的整数倍。
* shmflg:它的两个可能取值是SHM\_RND和SHM\_RDONLY。shmflg=SHM\_RDONLY，表示连接操作用来只读共享内存

返回值：成功返回一个指针，指向共享内存第一个节；失败返回-1

* + - 1. shmdt函数

功能：将共享内存段与当前进程脱离

原型：int shmdt(const void \*shmaddr);

参数：

* shmaddr: 由shmat所返回的指针

返回值：成功返回0；失败返回-1

注意：将共享内存段与当前进程脱离不等于删除共享内存段

* + - 1. shmctl函数

功能：用于控制共享内存（删除等）

原型：int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

参数：

* shmid:由shmget返回的共享内存标识码
* cmd:将要采取的动作（有三个可取值）
* buf:指向一个保存着共享内存的模式状态和访问权限的数据结构

返回值：成功返回0；失败返回-1

|  |  |
| --- | --- |
| 命令 | 说明 |
| IPC\_STAT | 把shmid\_ds结构中的数据设置为共享内存的当前关联值 |
| IPC\_SET | 在进程有足够权限的情况下，把共享内存的当前关联值设置为shmid\_ds数据结构中的值 |
| IPC\_RMID | 删除共享内存片段 |

## system V消息队列

* 消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法
* 每个数据块都被认为是有一个类型，接收者进程接收的数据块可以有不同的类型值
* IPC资源必须删除，否则不会自动清除，除非重启，所以system V IPC资源的生命周期随内核

## system V信号量

信号量主要用于同步和互斥的。

进程互斥：

* 由于各进程要求共享资源，而且有些资源需要互斥使用，因此各进程间竞争使用这些资源，进程的这种关系为进程的互斥。
* 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源。
* 在进程中涉及到互斥资源的程序段叫临界区。

# 进程信号

## 什么是信号

* + - 1. 生活中的信号

发令枪、闹钟、红绿灯等等，这些信号是人能够识别的，并且产生对应的行为。

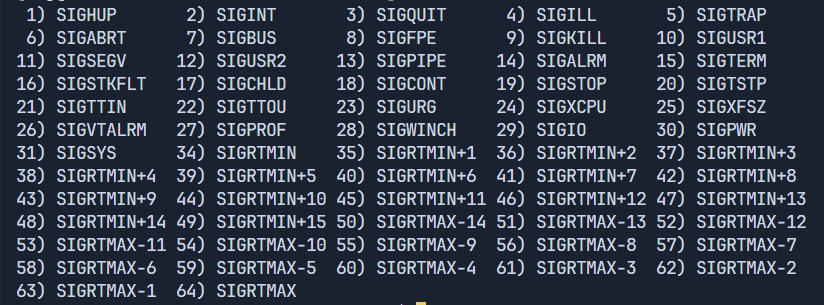
* + - 1. 技术应用角度的信号

在Shell下启动一个前台进程，键盘按下ctrl+c就可以终止进程。因为键盘输入产生一个硬件中断，被OS截获，解释成信号，并且将这个信号发送给目标进程。

* + - 1. 前台进程与后台进程
* ctrl+c产生的信号只能发给前台进程。一个命令后面加个&可以放到后台运行，这样Shell不必等待进程结束就可以接受新的命令，启动新的进程。
* Shell可以同时运行一个前台进程和任意多个后台进程，只有前台进程才能接到像ctrl+c这种控制键产生的信号。
* 前台进程在运行过程中用户随时可能按下ctrl+c而产生一个信号，也就是说该进程的用户空间代码执行到任何地方都有可能收到SIGINT信号而终止，所以信号相对于进程的控制流程来说是异步(Asynchronous)的。
  + - 1. 信号的概念

信号是进程之间事件异步通知的一种方式，属于软中断。

* + - 1. 用kill -l命令可以察看系统定义的信号列表



* 每个信号都有一个编号和一个宏定义名称,这些宏定义可以在signal.h中找到,例如其中有定义 #define SIGINT 2
* 编号34以上的是实时信号，编号1-31是普通信号。这些信号各自在什么条件下产生，默认的处理动作是什么，在signal(7)中都有详细说明: man 7 signal
  + - 1. 信号处理常见方式概览

可选的处理动作有以下三种:

* 忽略此信号。
* 执行该信号的默认处理动作。
* 提供一个信号处理函数,要求内核在处理该信号时切换到用户态执行这个处理函数,这种方式称为捕捉(Catch)一个信号。

## 产生信号

* + - 1. 通过终端按键产生信号

SIGINT的默认处理动作是终止进程，SIGQUIT的默认处理动作是终止进程并且Core Dump。

注：当一个进程要异常终止时，可以选择把进程的用户空间内存数据全部保存到磁盘上,文件名通常是core,这叫做Core Dump。进程异常终止通常是因为有Bug，比如非法内存访问导致段错误，事后可以用调试器检查core文件以查清错误原因，这叫做Post-mortem Debug（事后调试）。一个进程允许产生多大的core文件取决于进程的Resource Limit(这个信息保存 在PCB中)。默认是不允许产生core文件的，因为core文件中可能包含用户密码等敏感信息，不安全。在开发调试阶段可以用ulimit命令改变这个限制,允许产生core文件。

* + - 1. 终端调用kill命令向进程发送信号

例如在Shell中输入kill -9 pid，就可以杀死进程id为pid的进程。

* + - 1. 系统调用kill函数向进程发送信号

#include <signal.h>

// kill函数可以给一个指定的进程发送指定的信号

int kill(pid\_t pid, int signo);

// raise函数可以给当前进程发送指定的信号(自己给自己发信号)。

// raise(signo)=kill(getpid(), signo)

int raise(int signo); //

// abort函数使当前进程接收到信号而异常终止。

// abort()=kill(getpid(), SIGABRT)

void abort(void);

* + - 1. 由软件条件产生信号

alarm、SIGPIPE等产生的信号

* + - 1. 硬件异常产生信号

硬件异常被硬件以某种方式被硬件检测到并通知内核，然后内核向当前进程发送适当的信号。例如当前进程执行了除以0的指令，CPU的运算单元会产生异常，内核将这个异常解释为SIGFPE信号发送给进程。再比如当前进程访问了非法内存地址，MMU会产生异常，内核将这个异常解释为SIGSEGV信号发送给进程。

## 信号捕捉

从上面介绍可以知道，编号1-32的信号为普通信号，每种信号都有自己的默认动作，我们可以用signal系统调用函数捕捉到信号，并且修改信号的默认动作。如下所示：

void handler(int sig)

{

printf("catch a sig : %d\n", sig);

}

sianal(2, handler);

此时再向进程发送信号，就不会跟之前默认的终止进程一样，而是调用handler函数。

注：9号信号不能被捕捉！！！