Linux 软件体系结构

LINUX 系统是多进程、多用户和交互式的计算环境。

退出系统 文本界面下按<Ctrl-D>键或 logout 命令 shutdown shutdown –h 8:00 shutdown –h +3

Shell 是Linux系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行 Shell 也被称为 Linux 的命令解释器(command interpreter)

(1)Bourne Shell 是 AT&T Bell 实验室的 Steven Bourne 为 AT&T 的 Unix 开发的，它是 Unix 的默认 Shell，也是其它 Shell 的开发基础。Bourne Shell 在编程方面相当优秀，但在处理与用户的交互方面不如其它几种 Shell。

(2)C Shell 是加州伯克利大学的 Bill Joy 为 BSD Unix 开发的，与 sh 不同，它的语法与 C 语言很相似。它提供了 Bourne Shell 所不能处理的用户交互特征，如命令补全、命令别名、历史命令替换等。但是，C Shell 与 BourneShell 并不兼容。

(3)Korn Shell 是 AT&T Bell 实验室的 David Korn 开发的，它集合了 C Shell 和 Bourne Shell 的优点，并且与 Bourne Shell 向下完全兼容。Korn Shell 的效率很高，其命令交互界面和编程交互界面都很好。

(4)Bourne Again Shell (即 bash)是自由软件基金会(GNU)开发的一个 Shell，它是 Linux 系统中一个默认的 Shell。Bash 不但与 Bourne Shell 兼容，还继承了 C Shell、Korn Shell 等优点。

Shell 命令可以被分为**内部命令和外部命令**。

1.内部命令是 shell 本身包含的一些命令，这些内部命令的代码是整个 shell 代码的一个组成部分；
2.内部命令，shell 是通过执行自己代码中相应的部分来完成的
3.外部命令的代码则存放在一些二进制的可执行文件或者 shell 脚本中
4.外部命令，shell 会到文件系统结构（file system structure）中的一些目录去搜索那些文件名与外部命令的名字相同的文件，因为 shell 认为这些文件中就存放了将要执行的代码。

Shell 命令搜索路径

1.Shell 搜索的目录的名字都保存在一个 shell 变量 PATH(在 TC shell 中是 path)中。
2.变量 PATH(或者 path)中的目录名用一些特定的符号分开。在 bash shell 中，目录名用**冒号**分开。
3. $ echo $PATH
4.变量 PATH（或者 path）保存在主目录中的隐藏文件(hidden file).profile 或者.login 中

Bash 常用内部命令：

alias 用于设置或显示系统或用户定义的命令别名
unalias 撤销已经设定的命令别名
bg 把指定的作业置入后台运行模式 如果未指定作业号 则把当前的作业置如后台运行
break 用于退出 for while 和 until 等循环语句
continue 用于结束 for while until 等循环语句中的当前循环并执行下一轮循环
test 计算条件表达式
cd 切换目录
declare 用于申明变量或定义变量的属性 如果未指定变量名 **declare** 将会列出所有定义的变量 “-p” 选项用于显示给点变量的值和属性，下列选项用于声明变量并限定变量的属性
-a 把指定的变量定义为数组变量 **-f** 用于声明一个函数或显示函数的定义 **-i** 把指定的变量定义为整数变量当给变量赋值时按算术扩展的要求执行算术运算 **-r** 把指定的变量定义为只读变量，之后就不能给它赋值了 **-x** 公布指定的变量以便其他命令或 Shell 脚本能够继续使用
echo 用于输出字符串 变量值或表达式的计算结果 通常会在输出的信息之后加一个换行字符
enable 启用和禁用 shell 内置命令
eval 用于读取并组合随后的参数命令 然后提交 shell 执行
exec 如果 exec 命令带有参数则使用命令参数替代当前的 shell 进程，而不是像通常那样创建一个新进程执行给定的命令，如果在 shell 脚本中使用 exec ，当 exec 命令执行完后 将会强制终止 shell 脚本的运行 。因此在脚本中使用 exec 通常做为最后一条命令
let 用于实现算术运算 **kill** 杀进程
pwd 显示当前的工作目录 **return** 这个不多说 你懂的
source 类似 "."命令 **logout** 退出注册的 shell
read 从标准输入中读取数据 并把数据赋值给指定的变量， -r 选项 忽略转义字符"\" 使之作为普通字符
readonly 把指定的变量设置为只读 之后要修改它将会产生错误信息
export 导出设置的变量使变量能够用于正常运行的脚本或 shell 的所有子进程 用的最多的还是在 profile 文件中设置环境变量
set set 命令用于改变内部变量或脚本变量的值 set 命令的一个终于用途是重置位置参数 set 的部分选项
-a 使定义的所有变量和函数设置能够自动导入 shell 运行环境 **-e** 如果其中任何一条结束后返回非 0 的状态值 立即终止 shell 脚本的执行 **-f** 禁用文件名生成机制 **-m** 监控模式 在交换是 shell 中 这个选项默认设置为 on **-n** 读取 shell 脚本中的命令 检查是否存在语法错误 但不执行 **-t** 读入并在执行任何一条语句之后立即终止 shell 脚本的运行 **-v** 显示 shell 读入的命令语句 **-x** 显示执行的命令及其参数
unset 用于清楚 Shell 变量 把变量的值设为 null
shift 把位置参数 $2,$3,...$n 依次重新命名为$1,$2,$3....$n-1 原来的$1 和$n 就不存在了位置参数总是相应的减 1
type 用于显示解释的命令的完整路径名

在 linux 系统的终端有几个有用的**终端变量**

HISTORY ： 用于存储历史命令的文件
HISTSIZE：历史命令列表的大小
HOME：当前用户的目录
OLDPWD ： 前一个工作目录
PATH ： Bash 寻找可执行文件的搜索路径
PS1 ： 命令行的一级提示符
PS2 ： 命令行的二级提示符
PWD ： 当前工作的目录
SECONDS ： 当前 shell 开始到目前为止的秒数

Bash 脚本变量：

$# 脚本的参数个数
$* 以一个单字符串显示所有向脚本传递的参数。与位置变量不同，此选项参数可超过 9 个
$$ 脚本运行的当前进程 ID 号
$! 后台运行的最后一个进程的进程 ID 号
$@ 与$*相同，但是使用时加引号，并在引号中返回每个参数
$- 显示 shell 使用的当前选项，与 set 命令功能相同
$? 显示最后命令的退出状态。0 表示没有错误，其他任何值表明有错误。
$0 脚本名称 $1..$9 第 N 个参数

passwd 修改密码

/etc/passwd 记录每一个用户的 shell 程序
root:x:0:0:root:/root:/bin/bash
[用户名]:[密码]:[UID]:[GID]:[身份描述]:[主目录]:[登录 shell]

man

man -S2 open#选择第二个 section
1 用户命令，2 系统调用，3 语言函数库调用，4 设备和网络界面，5 文件格式，6 游戏和示范，troff 的环境、7 表格和宏，8 关于系统维护的命令

info <Q>退出 <Space>滚屏
whoami:显示用户名
who:显示正在使用系统的用户信息
-a 显示你的用户名和同一系统其他用户的列表
hostname 显示登录上的主机的名字
uname:显示关于运行在计算机上的操作系统的信息
－n 显示系统域名－p 显示系统的 CPU 名称
pwd print working directory，显示工作目录
date 显示时间，后面跟时间，更改时间
uptime 是显示时间，和一些比如有多少登陆用户等统计数据
cal [[month]year]显示日历 cal 4 2007
alias name=com 给命令指定一个名字；unalias 删除别名 -a 删除所有的；"\!*"会被实际的参数所替换
su [-][-c<CMD>][username]
- 改变身份，同时改变工作目录，及 HOME、SHELL、PATH 变量
-c 执行完命令之后返回原来用户
vi [option][filename]
vi 的退出，":wq"保存，":q!"不保存
+n：从第 n 行开始编辑文件/在 emacs 中，还有-nw，表不开启新窗口
+/exp：从文件中匹配字符串 exp 的第一行开始编辑
<A>在当前行末尾后添加新文本，<a>光标当前下一位置，<i>从当前光标所在位置开始，<o>从当前光标下一行位置
<ESC>或<CTRL+I>退出到指令模式

文件类型

普通文件、目录文件、符号链接文件、设备（特殊）文件（块设备，字符设备）、管道文件、socket 文件

Linux 继承了 UNIX，把文件名和文件控制信息分开管理， 文件控制信息单独组成一个称为 i 节点(inode)。inode 实质上是一个由系统管理的“目录项”。每个文件对应一个 inode，它们有唯一的编号，称为 inode 号。
Linux 的目录项主要由**文件名**和 **inode 号**组成。

每个连到计算机的设备都至少有一个设备文件。格式：/dev/xxyN
/dev :保存所有设备文件的目录
xx :设备类型，如 IDE 硬盘为 hd、SCSI 硬盘和 usb 盘为 sd、软盘为 fd
y :同种设备的顺序号，如第一个硬盘为 a
N :同一个设备编号，如硬盘的第一个分区为 1，硬盘 1-4 为前面四个主分区，5 开始为逻辑分区。
iso9660 :cd-rom 使用的标准文件系统
ntfs : windows 的 NTFS 文件系统，NT
vfat : windows 的 fat32 文件系统,2000/xp
msdos :MS-DOS 的 fat 文件系统

Linux 下目录组织结构

/bin: 该目录存放最常用的基本命令，比如拷贝命令 cp、编辑命令 vi、删除命令 rm 等。
/boot:该目录包含了系统启动需要的配置文件、内核（vmliuxz）和系统镜像(initrd…img)等。
/dev: 该目录下存放的是 Linux 中使用或未使用的外部设备文件（fd 代表软盘，hd 代表硬盘等），使用这些设备文件可以用操作文件的方式操作设备。
/etc:该目录下包含了所有系统服务和系统管理使用的配置文件；比如系统日志服务的配置文件 syslog.conf，系统用户密码文件 passwd 等
/home:该目录下包含了除系统管理员外的所有用户的主目录，用户主目录一般以用户登陆帐号命名。
/lib:该目录下包含了系统使用的动态连接库（*.so）和内核模块(在 modules 下)。
/lost+found:该目录包含了磁盘扫描检测到的文件碎片，如果你非法关机，那么下次启动时系统会进行磁盘扫描，将损坏的碎片存到该目录下。
/mnt:该目录下包含用户动态挂载的文件系统。如果要使用光盘，U 盘都一般应该将它们安装到该目录下的特定位置。
/proc: 该目录属于内存影射的一个虚拟目录，其中包含了许多系统现场数据，比如进程序数，中断情况，cpu 信息等等，它其中的信息都是动态生成的，不在磁盘中存储。
/root:该目录是系统管理员（root 用户）的主目录。
/sbin:该目录下包含系统管理员使用的系统管理命令,比如防火墙设置命令 iptable，系统停机命令 halt 等
/tmp: 该目录下包含一些临时文件。
/usr: 该目录下一般来说包含系统发布时自带的程序（但具体放什么东西，并没有明确的要求），其中最值得说明的有三个子目录
/usr/src ： Linux 内核源代码就存在这个目录
/usr/man ： Linux 中命令的帮助文件
/usr/local ： 新安装的应用软件一般默认在该目录下
/var：该目录中存放着在不断扩充着的信息，比如日志文件内核代码的结构

主目录(登录目录) Home Directories ~

mount [-t fstype] [-o options] device dirname

mount -t iso9660 /dev/hdc /media/cdrom
－a 命令挂载在/etc/fsta 中列出的所有设备

umount /mnt/floppy/

echo

-E 不解析转义字符 -e 解析转义字符 -n 不输出行尾的换行符

ls

-R 递归列出文件 -i 确定文件的 inode 号 -a (all)显示所有，包括隐藏文件 -l 长格式显示 -r 文件名逆序显示
－l 选项显示详细信息，包括访问权限、连接数、所有者 、组、文件大小（以字节计）和修改时间
-rwxr--r-- 1 sarwar faculty163 Apr 11 14:34 mid2

cd 不加任何参数，返回到主目录

mkdir [options] dirnames

-p 若上层目录目前尚未建立，则创建父目录
-m 建立目录时，同时设置目录的权限
mkdir –m 777 m02

rmdir [options] dirnames

-p 若上层目录为空，一并删除

touch [Option] FILE

修改 FILE 的 access 和 modify 时间，如果文件不存在,就会创建一个新文件

file [options] file-list 显示文件内容类型命令
cat 同时显示一个或多个文件的内容

-n 给每一行编号 -b 给每个非空行编号
cat file1 file2 > file3 将 file1 与 file2 合并成 file3
此时 file3 会被覆盖，避免可用>>代替>
tac 逆序显示一个文件

nl [options] [file-list]

显示文本文件的内容，同时显示行号
nl s_records #与 cat －n st_records 等价

more [Option] [-num] [+/ PATTERN] [+num] FILE

-s 多行空格只显示一行
+/str 从包含 str 那行的前两行开始显示
-nN 每屏/页显示 N 行
+N 从第 N 行开始显示文件内容

more 可将文件内容显示于屏幕上，每次只显示一页，可以往下浏览，但无法向上浏览。less 指令可以上下浏览

head [options] [file-list] -N 显示开始的 N 行
tail [options] [file-list]

-n N 若 N 前加“+”号表示显示从文件第 N 行开始的所有行；否则显示文件的最后 N 行

cp [option] file1 dir/file2 -f 强制 -i 覆盖先询问 -r 递归复制文件 -u 只有在新时复制
mv [option] file1 file2
mv [option] file-list directory
rm [option] file-list

wc [option] file-list 显示文件的大小，包括行数、单词数和字符数
-c 字符 -l 行 -w 单词数

diff [option] [file1][file2] 比较文件，显示区别

uniq [option][+N][input-file][output-file] 删除已经排序好的文件 input-file 中的所有重复行

lpr [options] file-list 打印文件列表中的文件

符号扩展

~代表主目录，**echo ~user1 显示 user1 的主目录**
{}花括号扩展 mkdir dir{1,2,3}会建立 3 个目录

id [opt][username]

-g 所属群组 -G 所有群组 -u 显示用户 ID
系统中的所有用户组的信息以及该组的用户都记录 在/etc/group 文件中.
Linux 有一个特殊用户，称为超级用户或根用户(Superuser or root user)可以访问所有文件。
用户名：root 用户 ID：0

chmod [opt] octal/symbolic file-list
chmod 700 file1(rwx 的顺序)
chmod u/g/o/a=/+/-rwx file1
-R 递归更改访问特权

目录设置权限：读特权对目录而言意味着可以读出目录的内容，写特权意味着可以在目录下创建或 删除一个文件，执行特权意味着可以检索这个目录

chgrp [opt] group file-list 改变文件目录群组
chown [opt] username file-list 更改所有者

unmask 文件访问权限=默认的访问权限 － mask 默认的访问权限:执行文件为 777 文本(text)文件为 666 ex: umask 013

特殊访问位

允许普通用户运行某个可执行文件，其权限是文件拥 有者的权限，通过设置有效用户标识(Effective user id)实现

set-user-ID(SUID) -如果对一个可执行文件设置了这个标志位，那么该可执行文件就以这个文件的拥有者的权限运行。
chmod 4xxx file-list
chmod u+s file-list

当设置 SUID 位为 1 时，如果用户对该文件有执行权限，那么 执行位被设置位‘s’，否则执行位变为‘S’

set-group-ID(SGID)与上面类似
chmod 2xxx file-list
chmod g+s file-list

sticky - sticky 位被设置，可以保证只有文件拥有者可以删除或重命名某个目录下的文件，即使其他用户 有写权限也不能。如果 sticky 位为 1，并且其他用户对目录有可执行的权 限，那么该权限位变为小写的‘t’，如果没有可执行的 权限，那么该权限位就变为大写的‘T’。
chmod 1xxx file-list
chmod +t file-list

gzip [opt][filename-list]
-d 解压缩文件
gzip 1.txt 得到 1.txt.gz 文件

gunzip 执行解压缩
zcat [opt][filename-list]解压文件输出到标准输出设备

tar
-c 建立备份文件
-z 压缩/解压一个存档文件
-v 详细地显示文件处理过程：用功能字母 x 解压文件的过程或存档文件的过程
-f Arch 用 Arch 作为存档或恢复文件的档案文件
-x 从磁带中解压（恢复）文件；如果没有指定，默认对整条磁带解压
tar –cvf bash.help.tar *.help
tar –zxvf linux-2.6.15.tar.gz

pr
命令可以用来将文本转换成适合打印的文件。这个工具的一个基本用途就是将较大的文件分割成多个页面，并为每个页面添加标题。

ln 使用方式 : ln [options] source dist，其中 option 的格式为 : [-bdfinsvF] [-S backup-suffix] [-V {numbered,existing,simple}] [--help] [--version] [--]

说明 : Linux/Unix 档案系统中，有所谓的连结(link)，我们可以将其视为档案的别名，而连结又可分为两种 : 硬连结(hard link)与软连结(symbolic link)，硬连结的意思是一个档案可以有多个名称，而软连结的方式则是产生一个特殊的档案，该档案的内容是指向另一个档案的位置。硬连结是存在**同一个档案系统**中，而软连结却可以**跨越不同的档案系统**。

ln source dist 是产生一个连结(dist)到 source，至于使用硬连结或软链结则由参数决定。

不论是硬连结或软链结都不会将原本的档案复制一份，只会占用非常少量的磁碟空间。

-f : 链结时先将与 dist 同档名的档案删除
-d : 允许系统管理者硬链结自己的目录
-i : 在删除与 dist 同档名的档案时先进行询问
-n : 在进行软连结时，将 dist 视为一般的档案
-s : 进行软链结(symbolic link)
-v : 在连结之前显示其档名
-b : 将在链结时会被覆写或删除的档案进行备份

rpm [options] [rpm-filename]
rpm –ivh RPM 包的全路径文件名
-i：安装
-v：代表 verbose，设置在安装过程中将显示详细的信息
-h：代表 hash 设置在安装过程中将显示“#”来表示安装的进度
-e: 删除 -U: 升级 -q: 查询 -v：校验
rpm –Uvh FC5-2.6.15-0.rr.10.4.i686.rpm

sort [opt] [file-list]
-b 忽略空格 -d 忽略除字母、数字、空格外的字符 -f 忽略大小写 -k n1[,n2] 按照 n1~n2 字段排序 -o filename 结果输出到文件
-r 逆序排 -u 重复只输出一次
sort -k 1 -k 5 students

find [目录列表][表达式]
find 查找目录列表中匹配表达式标准的文件
-name pattern 搜索文件名匹配 pattern 的文件 -newer file 搜索修改时间在 file 之后的文件（即比 file 新的文件） -user name 搜索所有权为 name 的文件 -print 显示符合要求的文件路径和文件名 \(expr \)当表达式为真结果为真；表达式可以用 OR 和 AND 组合 ! expr 取反，当表达式为假时结果为真
find ~\(-name sample -o -name '*.old'\) -print -exec rm{}\;

whereis [options] [file-list]
whereis 命令查明你系统上是否存在特定的一个命令，如果存在，给出该命令的路径

which [command-list] 如果一个系统中有多个版本命令，which 命令告诉当输入某个命令执行时，shell 到底调用了哪个版本的命令

grep [选项] 模式 [文件列表]
作用：显示文件中匹配特定模式的行
-c 仅输出匹配的行的个数
-i 在匹配的过程中忽略字母的大小写
-l 仅输出有匹配行的文件名
-n 匹配时同时输出行号
-s 对 shell 脚本有用，避免出现错误信息(如果成功返回 0，失败返回非零值)
-v 打印出不匹配的行
-w 全字匹配
grep -E [a-z]\{7\} student

history [options] [filename]
显示或操作历史命令列表
历史命令保存在~/.bash_history 文件中
!! 执行最近一个命令
!n 执行历史命令列表中的第 n 个命令
!-n 当前之前的第 n 个命令

!string 最近用到的以 string 开始的命令
!?string[?] 最近用到的包含 string 的命令

Crtl 的快捷键
<Backspace>或<Ctrl-H>删除前一个字符
<Ctrl-U>删除当前行 <Ctrl-C>终止现在的命令，终止一个前台进程 使用<Ctrl-Z>挂起一个前台进程 <Ctrl+D>退出当前的 shell，eof，必须从登陆 shell 退出，必须关闭所有的 shell <Ctrl-K>删除一行光标后字符 <Ctrl-P>上一次执行的命令，扫描过的不会再次出现

Shell 命令可以是内部或者外部命令。

内部（内置）命令（internal (built-in) command）的代码本身就是 shell 进程的一部分。LINUX shell 中的一些内部命令如.（点命令）、alias、bg、cd 、continue、echo 、exec 、exit 、fg 、jobs 、pwd 、set 、shift 、test 、time 、umask 、unset 和 wait。

外部命令是(external command)命令代码以文件的形式出现的称为： 文件内容可以是二进制代码或者 shell 脚本。通常使用的一些外部命 令如 grep、more 、cat 、mkdir 、rmdir 、ls 、sort 、ftp 、 telnet 、lp 和 ps 。

ps -a 显示所有终端上执行的进程信息，包括其他用户的进程信息。 -e/-A 显示所有系统中运行的进程的信息 -j 采用作业控制格式显示所有信息（包括父进程的 PID、组 ID、会话 ID 等） -l 用长列表来显示状态报告信息 -r 显示运行状态的进程 –f 显示进程间的层进关系
STAT 字段含义：
D 不可中断睡眠（IO）；N 低优先级进程；R ready queue 进程；S 处于睡眠进程；T 跟踪或被停止；Z 僵死进程；W 完全交换到磁盘

top 命令：实时监视 CPU 的活动状态。该命令显示系统中 CPU 密集型任务的状态并且允许你交互地控制这些进程。

在命令后面加上一个“与”操作符号（&），使该命令在后台操作。
jobs 显示所有挂起的和后台进程的作业号
jobs [option] [%jobID] -l 显示作业 PID

kill [-sig_num] proc-list
在后台运行一个进程[1] 23467 括号中 shell 返回的数字是该进程的作业号（job number）,另外一个数字是进程 PID.
1 挂断，退出系统;2 中断（ctrl-c）;9 强制终止；15 终止进程（默认）

fg [%jobid]使作业号为 jobid 的前台进程(之前被挂起)继续执行，或者把后台进程转到前台执行.
bg [%jobid]通过将暂挂的作业作为后台作业运行，可在当前环境中恢复执行这些作业.
%+或%% 指代当前作业 %- 指代前一个作业
%N 作业号 N %String 以字符串开头的作业

守护进程
守护进程（daemon）是运行于后台的系统进程。
守护进程在 LINUX 中经常用于向用户提供各种类型的服务和执行系统管理任务。
smtpd (e-mail service) httpd (web browsing)
inetd (internet related services)

P1;P2; 顺序执行
P1&P2& 并发执行
P1&&P2 运行 P1;若成功,再运行 P2
P1||P2 运行 P1;若不成功,再运行 P2 直至有一个成功为止

终止进程
按<Ctrl-C>来终止一个前台进程。
终止后台进程可用两种方法中的一种：
1.先使用 fg 命令把进程转向前台，然后按<Ctrl-C> 2.使用 kill 命令

ps –e f 命令或者 pstree 命令可以用图的形式显示当前系统中执行进程的进程树，勾勒出进程间的父子关系

标准输入、标准输出和标准出错能够分别用 0< 、1>和 2>操作符来重定向。
m>&n 将 m 指定的输出与 n 指定的输出合并,所有输出送到 n 指定的输出设备。

Linux 允许一条命令的标准输出成为另外一条命令的标准输入
command1 | command2 | … | commandN
一些经常用到的过滤器是：cat、compress、crypt、grep、gzip、lp、pr、sort、tr、uniq 和 wc

tee [options] file-list 从标准输入中得到输入然后送到标准输出和 file-list 中

{cmd1；cmd2}：当前 shell 中执行或实现扩展
(cmd1；cmd2)：在子 shell 中执行

gcc [选项] 文件列表
-c 编译成目标（.o）文件
-l 库文件名，连接库文件
-o 指定输出文件 g
-E preprocess only, 结果输出到 stdout
-S compile only 生成.s 文件
gcc -o hello hello.c /usr/lib/libm.a 或者-lm 或者-L/usr/……
如果 lib 不在标准位置，那么用-L
将生成的可执行文件保存到指定文件:
gcc driver.o stack.o misc.o –o polish
使用 gcc 的-l 选项可以连接已有的程序库:
gcc power.o -lm –o power
文件名中“lib”以后，扩展名以前的部分
gcc -o mypro1 –I（大写 i）/usr/openwin/include mypro1.c 生成了 mypro1 可执行文件
gcc –o power power.o -lm
gcc –o power power.o /usr/lib/libm.a
ld -s -o hello hello.o（连接程序）
gas -o hello.o hello.s（汇编程序）
gcc -o hello hello.s（汇编和连接一步执行）

make [选项] [目标] [宏定义]
makefile 的默认文件名为 GNUmakefile、makefile 或 Makefile
-d 显示调试信息
-f 文件 指定文件的依赖关系文件
-n 只显示 makefile 中的指令，不执行
-s 执行，但是不显示
Makefile 规则的语法格式目标文件列表，依赖文件列表，<tab>命令列表
target 是 Makefile 文件中定义的目标之一，如果省略 target，make 就将生成 Makefile 文件中定义的第一个目标，VARNAME=string makefile 变量定义 $(VARNAME) makefile 变量使用

标准系统库文件在 linux 的/lib 和/usr/lib 目录下。

du [option] .. [FILE](计算文件或目录的磁盘使用情况)
-a all; –c total
-b bytes; –h human_readable

df [option] … [FILE] 显示文件系统相关信息
常用选项: -a all; -h human_readable; -i inode; -t TYPE

系统调用篇：

进程控制部分：

fork	创建一个新进程	clone	按指定条件创建子进程
execve	运行可执行文件	exit	中止进程
_exit	立即中止当前进程	getdtablesize	进程所能打开的最大文件数
getpgid	设置指定进程组标志号	getpgrp	获取当前进程组标识号
setpgid		setpgrp	设置当前进程组标志号
getpid	获取进程标识号	getppid	获取父进程标识号
getpriority	获取调度优先级	setpriority	设置调度优先级
nice	改变分时进程的优先级		
pause	挂起进程，等待信号		
sched_yield	进程主动让出处理器,并将自己等候调度队列队尾		
vfork	创建一个子进程，以供执行新程序，常与 execve 等同时使用		
wait	等待子进程终止	waitpid	等待指定子进程终止
capget	获取进程权限	capset	设置进程权限
getsid	获取会晤标识号	setsid	设置会晤标识号

文件读写操作

fcntl	文件控制	open	打开文件
creat	创建新文件	close	关闭文件描述字
read	读文件	write	写文件
readv	从文件读入数据到缓冲数组中		
writev	将缓冲数组里的数据写入文件		
pread	对文件随机读	pwrite	对文件随机写
lseek	移动文件指针		
dup	复制已打开的文件描述字		
dup2	按指定条件复制文件描述字		
flock	文件加/解锁	poll	I/O 多路转换
truncate	截断文件	umask	设置文件权限掩码
fsync	把文件在内存中的部分写回磁盘		

共享内存

shmctl	控制共享内存	shmget	获取共享内存
shmat	连接共享内存	shmdt	拆卸共享内存

消息

msgctl	消息控制操作	msgget	获取消息队列
msgsnd	发消息	msgrcv	取消息

内核模块
什么是内核模块(动态可加载模块 LKM)
1.linux 操作系统的内核是单一体系结构的
2.用户根据需要，在不对内核重新编译的情况下，模块动态地装入内核或移出
3.模块在内核空间运行，实际上是一种目标对象文件，没有链接，不能对立运行，但是其代码可以再运行时链接到系统中作为内核的一部分运行或从内核中取下，从而可以动态扩充内核的功能
4.这种目标代码通常由一组函数和数据结构组成，如用来实现一种文件系统、一个驱动程序或其他内核上层功能
5.**优点**：使得内核更加紧凑和灵活；修改内核时，不必全部重新编译整个内核。系统如果需要使用新模块，只要编译相应的模块，然后使用 insmod 将模块插入即可；模块不依赖于某个固定的硬件平台；模块的目标代码一旦被链接到内核，它的作用域和静态链接的内核目标代码完全等价。**缺点**：由于内核所占用的内存是不会被换出的，所以链接近内核的模块会给系统带来一定的性能和内存利用方面的损失；装入内核的模块就成为内核的一部分，可以修改内核中的其他部分，因此，模块的使用不当会导致系统崩溃；为了让内核模块能访问所有内核资源，内核必须维护符号表，并在装入和卸载模块时修改符号表；模块会要求利用其他模块的功能，所以，内核要维护模块之间的依赖性。
模块指令 insmod 加载 lsmod 查 rmmod 卸载
Insmod: 调用 insmod 撑血将把需要插入的模块以目标代码的形式插入到内核中；在插入的时候，insmod 会自动运行 init_module()函数中定义的过程(需超级用户权限)。
insmod [path]modulename.**ko**
insmod 程序完成下面一系列工作：1.从命令行中读入要链接的模块名，通常是扩展名为".ko",elf 格式的目标文件 2.确定模块对象代码所在文件的位置。通常这个文件都是在 lib/modules 的某个子目录中；计算存放模块代码、模块名和 module 对象所需要的内存大小 4.在用户空间分配一个内存区，把 module 对象、模块名以及正在运行的内核所重定位的模块代码拷贝到这个内存里。其中 module 对象中的 init 域指向这个模块的入口函数重新分配到的地址； exit 域指向出口函数所重新非配的地址 5.调用 init_module()，向它传递上米娜所创建的用户态的内存区的地址 6.释放用户态内存，整个过程结束。
lsmod(等价于 cat /proc/modules)显示当前系统中正在使用的模块信息

ksyms(等价于 cat /proc/kallsyms)显示内核符号和模块符号表的信息
rmmod 自动运行在 cleanup_module()函数中的过程：rmmod [path]modulename
modprobe 自动加载所依赖的模块

内核模块机制

内核符号表:.用来存放所有模块可以访问的那些符号以及相应地址的特殊的表；模块所声明的任何全局符号都成为内核符号表的一部分；内核符号表出于内核代码段的_ksymtab,其开始地址和结束地址由 C 编译器所产生的两个符号来指定:_start_ksymtab 和 _stop_ksymtab
从文件/proc/ksyms 中以文本的方式读取
内存地址 符号名称 【所属模块】
模块引用计数：计数器存放在 module 对象的 uc.usecount 域；当开始执行模块操作时，递增计数器；在操作结束时，递减这个计数器；维护三个宏__MOD_INC_USE_COUNT 模块计数+1
__MOD_DEC_USE_COUNT 模块计数-1
__MOD_IN_USE 计数非 0 时返回真；计数器的值为 0 时，可以卸载这个模块；计数器的当前值可以在/proc/modules(lsmod)中每一项的第三个域找到 **模块依赖**：一个模块 A 引用另一个模块 B 所到处的符号

存储管理

保护模式下 i386 提供虚拟存储器的硬件机制
i386 的地址转换机制：地址总线 32(36)位，物理内存 4(64)GB;指令系统提供的逻辑地址为 48 位，虚地址空间 64T
虚拟内存(4G),内核空间(最高的 1G 字节 由所有进程共享，存放内核代码和数据)和用户空间(较低的 3G 字节 存放用户程序的代码和数据)，每个进程最大拥有 3G 字节私有虚存空间；地址转换(通过页表把虚存空间的一个地址转换为物理空间中的实际地址)
进程用户空间的管理 每个程序经编译、链接后形成的二进制映像文件有一个代码段和数据段 进程运行是须有独占的堆栈空间

进程用户空间

linux 把进程的用户空间划分为一个个区间，便于管理；一个进程的用户地址空间按主要由 mm_struct 和 vm_area_structts 结构来描述；
mm_struct 结构对进程整个用户空间进行描述;
vm_area_structs 结构对用户空间中各个区间(简称虚存区)进行描述
mm_struct 结构首地址在 task_struct 成员项 mm 中：struct mm_struct*mm
include/linux/sched.c
count(对 mm_struct 结构的引用进行计数。为了在 linux 中实现线程，内核调用 clone 派生一个线程，线程和调用进程共享用户空间，即 mm_struct 结构，派生后系统就会累加 mm_struct 中的引用计数);pgd(进程的页目录基地址，当调度程序调查一个进程运行时，就将这个地址转成物理地址，并写入控制寄存器(cr3)) map_count(在进程的整个用户空间中虚存区的个数) semaphore(对 mm_struct 结构进行串行访问所使用的信号量)start_code end_code start_data end_data(代码段和数据段起始地址和终止地址)start_brk brk start_stack
(每个进程都有一个特殊的地址区间，这个区间就是所谓的堆，也就是前图中的空洞。前两个域分别描述堆的起始地址和终止地址，最后一个域描述堆栈段的起始地址) rss(进程驻留内存中的页面数)total_vm(进程所需的总页数) locked_vm(被锁定在物理内存中的页数) mmap
(vm_area_struct 虚区结构形成一个单链表，其基址由小到大排列) mmap_avl(vm_area_struct 虚区形成一棵 avl 平衡树) mmap_cache(最近一次用到的虚存区很可能下一次还要用到，因此，把最近用到的虚存区结构放入高速缓存，这个徐存取就由 mmap_cache 指向)
vm_area_struct 结构：虚存空间中一个连续的区域，在这个区域中的信息具有相同的操作和访问特性；各区间互不重叠，按线性地址的次序链接在一起。当区间的数目较多时，将建立 AVL 树以保证搜索速度。
include/linux/mm_types.h
vm_mm 指针指向进程的 mm_struct 结构体
vm_start 和 vm_end 虚拟区域的开始和终止地址 vm_flags 指出了操作特性
VM_EXEC 允许执行 VM_SHARED 允许共享
VM_LOCKED 可以加锁 VM_STACK_FLAGS 做为堆栈使用
vm_page_prot 虚存区域的页面的保护特性

VMA

所有 vm_area_struct 结构体按地址递增链接成一个单向链表，vm_next 指向下一个 vm_area_struct 结构体。链表的首地址由 mm_struct 中成员项 mmap 指出。
vm_ops：是指向 vm_operations_struct 结构体的指针。该结构体中包含着指向各种操作的函数的指针。
vm_operations_struct 结构体在 include/linux/mm.h 文件中定义。
vm_next_share 和 vm_prev_share，把有关的 vm_area_struct 结合成一个共享内存时使用的双向链表。

虚存段的建立

Linux 使用 do_mmap()函数完成可执行映像向虚存段的映射，由它建立有关的虚存段。
do_mmap()函数定义在 include/linux/mm.h 文件中：unsigned long do_mmap(struct file * file, unsigned long addr, unsigned long len,unsigned long prot, unsigned long flags, unsigned long off)
do_mmap()函数参数含义：
file 是指向该文件结构体的指针，若 file 为 NULL，称为匿名映射(anonymous mapping)。
off 是相对于文件起始位置的偏移量。
addr 虚存段在虚拟内存空间的开始地址。
len 是这个虚存段的长度。
prot 指定了虚存段的访问特性：
PROT_READ 0x1 对虚存段允许读取
PROT_WEITE 0x2 对虚存段允许写入
PROT_EXEC 0x4 虚存段(代码)允许执行
PROT_NONE 0x0 不允许访问该虚存段
flag 指定了虚存段的属性：
MAP_FIXE 指定虚存段固定在 addr 的位置上
MAP_SHARED 指定对虚存段的操作是作用在共享页面上
MAP_PRIVATE 指定了对虚存段的写入操作将引起页面拷贝

创建进程用户空间

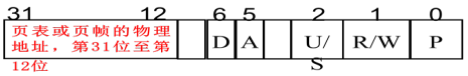
fork()系统调用在创建新进程时也为该进程创建完整的用户空间
fork()是通过拷贝或共享父进程的用户空间来实现的，即内核调用 copy_mm()函数，为新进程建立所有页表和 mm_struct 结构
Linux 利用“写时复制 copy on write”技术来快速创建进程

Linux 的页表机制

三级分页管理把虚拟地址分成四个位段：
页目录、页中间目录、页表、页内偏移。
系统设置三级页表：
页目录 PGD（Page Directory）
页中间目录 PMD（Page Middle Directory）
页表 PTE（Page Table）
2.6.11 以后 Linux 内核采用四级页表模型来使用硬件分页机制(支持 64 位 CPU 架构) ，分别是：Page Global Directory(pgd_t), Page Upper Directory(pud_t),Page Middle Directory(pmd_t) 和 Page Table(pte_t)。
当硬件分页机制实际是两级页表时，Linux 内核把 Page Upper Directory 和 Page Middle Direcotry 跳过了
对 i386，提供了把三级分页管理转换成两级分页机制的方法。其中一个重要的方面就是把 PGD 与 PMD 合二为一，使所有关于 PMD 的操作变为对 PGD 的操作。

MMU 地址映射过程：

每个进程都有一个页目录,当进程运行时,寄存器 CR3 指向该页目录的基址。地址映射过程：
1.从 CR3 取得页目录的基地址。页目录用一个物理页帧存储,用来保存页表的基址。
2.以逻辑地址的页目录段为索引,在页目录中找到页表的基址。页表也是用一个物理页帧存储，用来保存物理页帧号。
3.以逻辑地址的页表段为索引，在页表中找相应的物理页帧号。
4.物理页帧号加上逻辑地址的偏移段即得到了对应的物理地址。



R/W=1则该页可写，可读，且可执行；
R/W=0则该页可读，可执行，但不可写
U/S=1则该页可在任何特权级下访问；
U/S=0则该页只能在特权级0、1和2下访问
A：访问位
D：已写标志位

缺页异常处理

导致缺页异常（缺页中断）的原因有:
编程错误。可分为内核程序错误和用户程序错误。
操作系统故意引发的异常。操作系统合理利用硬件机制，在适当时间触发异常， 使得该异常的处理程序被调用，以达到预期目的。
do_page_fault()
页面异常的处理程序是 do_page_fault( )函数，该函数有两个参数：
一个是指针,指向异常发生时寄存器值存放的地址。
另一个错误码,由三位二进制信息组成：
第 0 位——访问的物理页帧是否存在；
第 1 位——写错误还是读错误或执行错误；
第 2 位——程序运行在核心态还是用户态。
do_page_fault( )函数定义在 arch/i386/mm/fault.c 文件中
do_page_fault()函数的执行过程如下：
1.得到导致异常发生的线性地址,对于 X86 该地址放在 CR2 寄存器中 2.检查异常是否发生在中断或内核线程中,如是,则进行出错处理。3 检查该线性地址属于进程的某个 vm_area_struct 区间。如果不属于任何一个区间,则需要进一步检查该地址是否属于栈的合理可扩展区间。一但是用户态产生异常的线性地址正好位于栈区间的 vm_start 前面的合理位置,则调用 expand_stack( )函数扩展该区间,通常是扩充一个页面,但此时还未分配物理页帧。
至此,线性地址必属于某个区间。
根据错误码的值确定下一个步骤:
如果错误码的值表示为写错误,则检查该区间是否允许写,不允许则进行出错处理。
如果允许写就是属于写时拷贝(COW:copy on write)。如果错误码的值表示为页面不存在,这就是所谓的按需调页(demand paging)
写时拷贝的处理过程：
1.改写对应页表项的访问标志位，表明其刚被访问过,这样在页面调度时该页面就不会被优先考虑。2.如果该页帧目前只为一个进程单独使用,则只需把页表项置为可写。3.如果该页帧为多个进程共享,则申请一个新的物理页面并标记为可写,复制原来物理页面的内容,更改当前进程相应的页表项,同时原来的物理页帧的共享计数减一。
按需调页的处理过程:
第一种情况页面从未被进程访问,这种情况页表项的值全为 0。(1)如果所属区间的 vm_ops->nopage 不为空,表示该区间映射到一个文件,并且 vm_ops->nopage 指向装入页面的函数,此时调用该函数装入该页面。(2)如果 vm_ops 或 vm_ops->nopage 为空,则该调用 do_anonymous_page( )申请一个页面;
另一种情况是该页面被进程访问过,但是目前已被写到交换分区,页表项的存在标志位为 0,但其他位被用来记录该页面在交换分区中的信息。调用 do_swap_page( )函数从交换分区调入该页面。

选择被换出的页面策略
1.只有与用户空间建立了映射关系的物理页面才会被换出，内核空间中内核所占的页面则常驻内存 2.进程映像所占的页面 ，其代码段、数据段可被换入换出，但堆栈段一般不换出 3.通过系统调用 mmap()把文件内容映射到用户空间时，页面所使用的交换区就是被映射的文件本身 4.进程间共享内存区其页面的换入换出比较复杂 5.映射到内核空间中的页面都不会被换出 6.内核在执行过程中使用的页面要经过动态分配，但永驻内存
在交换区中存放页面
1.交换区也被划分为块，每个块的大小恰好等于一页，一块叫做一个页插槽 2.换出时，内核尽可能把换出的页放在相邻的插槽中，从而减少访问交换区时磁盘的寻道时间 3.若系统使用了多个交换区，快速交换区可以获得比较高的优先级 4.当查找一个空闲插槽时，要从优先级最高的交换区中开始搜索 5.如果优先级最高的交换区不止一个，应该循环选择相同优先级的交换区
页面交换策略
策略 1：需要时才交换
策略 2：系统空闲时交换
策略 3：换出但并不立即释放
策略 4：把页面换出推迟到不能再推迟为止
页面换入 / 换出及回收的基本思想
1.**释放页面**：如果一个页面变为空闲可用，就把该页面的 page 结构链入某个空闲队列 free_area，同时页面的使用计数 count 减 1。
2.**分配页面**：调用___get_free_page()从某个空闲队列分配内存页面，并将其页面的使用计数 count 置为 1。
3.**活跃状态**：已分配的页面处于活跃状态，该页面的数据结构 page 通过其队列头结构 lru 链入活跃页面队列 active_list，并且在进程地址空间中至少有一个页与该页面之间建立了映射关系。
4.**不活跃“脏”状态**：处于该状态的页面其 page 结构通过其队列头结构 lru 链入不活跃“脏”页面队列 inactive_dirty_list，并且原则是任何进程的页面表项不再指向该页面，也就是说，断开页面的映射，同时把页面的使用计数 count 减 1。将不活跃“脏”页面的内容写入交换区，并将该页面的 page 结构从不活跃“脏”页面队列 inactive_dirty_list 转移到不活跃“干净”页面队列，准备被回收。
5.**不活跃“干净”状态**：页面 page 结构通过其队列头结构 lru 链入某个不活跃“干净”页面队列。如果在转入不活跃状态以后的一段时间内，页面又受到访问，则又转入活跃状态并恢复映射。当需要时，就从“干净”页面队列中回收页面，也就是说或者把页面链入到空闲队列，或者直接进行分配。
页面交换守护进程 kswapd
Linux 内核利用守护进程 kswapd 定期地检查系统内的空闲页面数是否小于预定义的极限，一旦发现空闲页面数太少，就预先将若干页面换出
kswapd 相当于一个进程，它有自己的进程控制块 task_struct 结构，与其它进程一样受内核调度，但没有独立的地址空间

页帧与区域
1.页帧：物理内存是以页帧(page frame)为基本单位，页帧的大小固定，i386 默认为 4KB。
2.结点：访问速度相同的一个内存区域称为一个结点（Node）。
3.区：每个结点的物理内存因为用途不同又分成不同的区(zone)。例如 x86，分成如下三个区:
DMA ZONE 低于 16MB 的内存，是 DMA 方式能够访问的物理内存。在内存分配时，尽可能保留这部分内存以供 DMA 方式使用。
NORMAL ZONE 介于 16MB 与 896MB 之间，直接被内核映射。
HIGHMEM ZONE 高端内存，超过 896MB 以上的部分，不能被内核直接映射。
4.区的划分没有任何物理意义，是内核内核为了管理页帧而采取逻辑上的分组。include/linux/mmzone.h 定义了区结构
5.Linux 对不同 zone 的内存使用单独的伙伴系统(buddy system)管理,而且独立地监控空闲页帧。
6.Linux 设置了一个 mem_map[]数组管理内存页帧。
7.mem_map[]在系统初始化时由 free_area_init()函数创建，它存放在物理内存的低地址部分
8.mem_map[]数组的元素是一个个的 page 结构体，每一个 page 结构体它对应一个物理页帧。
9.page 结构进一步被定义为 mem_map_t 类型，32 字节，其定义在 include/linux/mm_types.h 中
bitmap 表：在物理内存低端，紧跟 mem_map 表的 bitmap 表以位示图方式记录了所有物理内存的空闲状况。与 mem_map 一样，bitmap 在系统初始化时由 free_area_init() 函数创建(mm/page_alloc.c)。
Linux 对内存页帧块的每种划分都对应一个位图 map（bitmap），free_area[]各个元素中的指针 map 指向相应页帧块的位图，free_area[0]中的 map 指向内存按照 1 个页帧划分时的位图，free_area[1]中的 map 指向内存按照 2 个页帧划分时的位图…。位图中每一位（bit）表示页帧的使用情况。当 bit 所指物理页帧块都是空闲时，该位置 0；全部或部分占用时， 对应的位置 1。free_area[]数组指向的 10 个位图， 放在内存 mem_map 的上方 bitmap 区域内。
用来记录页块组使用情况的位图的长度不同，页块越小位图越长。当内存区域的开始地址为 start_mem，结束地址为 end_mem，页帧尺寸为 PAGE_SIZE 时， 每种页块组的位图的长度计算公式为：
(end_mem-start_mem)/PAGE_SIZE/2i+3（字节）
其中 i=0～1024，分别对应 1～1024 十一种页块组的 map 长度
Buddy 算法
1.两组连续页帧被认为是一对“伙伴”必须满足如下条件:
大小相同,比如说都有 b 个页帧；
物理空间上连续；
位于后面那个块的最后页帧号必须是 2×b 的倍数。
2.伙伴系统的操作：
申请空间的函数为 alloc_pages( )；

释放函数为 free_pages( )；
当在申请内存发现页面短缺时，还会唤醒 **kswapd 内核线程**运行，该线程会腾出一些空间以满足要求。

物理页面的分配

函数 alloc_pages( )用于分配物理页

该函数所做的工作如下：

1.检查所请求的页块大小是否能够被满足

2.检查系统中空闲物理页的总数是否已低于允许的下界

3.正常分配。从 free_area 数组的第 order 项开始，这是一个 mem_map_t 链表。

4.换页。通过下列语句调用函数 try_to_free_pages（），启动换页进程

1) 如果该链表中有满足要求的页块，则：

将其从链表中摘下；将 free_area 数组的位图中该页块所对应的位取反,表示页块已用;修改全局变量 nr_free_pages（减去分配出去的页数）；根据该页块在 mem_map 数组中的位置，算出其起始物理地址,返回。

2) 如果该链表中没有满足要求的页块，则在 free_area 数组中顺序向上查找.其结果有二：

a) 整个 free_area 数组中都没有满足要求的页块,此次无法分配,返回。

b) 找到一个满足要求的页块，则：

将其从链表中摘下；将 free_area 数组的位图中该页块所对应的位取反，表示页块已用；修改全局变量 nr_free_pages（减去分配出去的页数）；因为页块比申请的页块要大，所以要将它分成适当大小的块。因为所有的页块都由 2 的幂次的页数组成，所以这个分割的过程比较简单，只需要将它平分就可以：

I. 将其平分为两个伙伴，将小伙伴加入 free_area 数组中相应的链表，修改位图中相应的位；

II.如果大伙伴仍比申请的页块大，则转 I，继续划分；

III.大伙伴的大小正是所要的大小，修改位图中相应的位,根据其在 mem_map 数组中的位置，算出它的起始物理地址,返回。

物理页面的回收

函数 free_pages()用于页块的回收

根据页块的首地址 addr 算出该页块的第一页在 mem_map 数组的索引；

如果该页是保留的（内核在使用），则不允许回收；

将页块第一页对应的 mem_map_t 结构中的 count 域减 1，表示引用该页的进程数减了 1 个。若 count 域的值不为 0，有别的进程在使用该页块，不能回收，仅简单返回

清除页块第一页对应的 mem_map_t 结构中 flags 域的 PG_referenced 位，表示该页块不再被引用；

将全局变量 nr_free_pages 的值加上回收的物理页数

将页块加入到数组 free_area 的相应链表中

slab 分配器

伙伴系统是以页帧为基本分配单位，因而对于小对象容易造成内部碎片。

不同的数据类型用不同的方法分配内存可能提高效率。比如需要初始化的数据结构，释放后可以暂存着，再分配时就不必初始化了。内核的函数常常重复地使用同一类型的内存区，缓存最近释放的对象可以加速分配和释放。

解决办法：基于伙伴系统的 slab 分配器。

slab 分配器的基本思想：为经常使用的小对象建立缓冲,小对象的申请与释放都通过 slab 分配器来管理。slab 分配器再与伙伴系统打交道。

好处：其一是充分利用了空间,减小了内部碎片；其二是管理局部化,尽可能少地与伙伴系统打交道,从而提高了效率。

VFS 文件系统的结构

VFS 根据不同的文件系统抽象出了一个通用的文件模型。通用的文件模型由四种数据对象组成：

超级块对象 superblock :存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统超级块或控制块。

索引节点对象 inode object ：存储某个文件的信息。通常对应磁盘文件系统的文件控制块

目录项对象 dentry object ： dentry 对象主要是描述一个目录项，是路径的组成部分。

文件对象 file object：存储一个打开文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在与内存。

VFS 的超级块

超级块 superblock 是文件系统中描述整体组织和结构的信息体。

VFS 把不同文件系统中的整体组织和结构信息，进行抽象后形成了兼顾不同文件系统的统一的超级块结构。

VFS 超级块是各种具体文件系统在安装时建立的，并在卸载时被自动删除。

Linux 中对于每种已安装的文件系统，在内存中都有与其对应的超级块。VFS 超级块中的数据主要来自该文件系统的超级块。

VFS 超级块的数据结构定义是 super_block 结构。

super_block

s_list: 指向了超级块链表中前一个超级块和后一个超级块的指针。

s_dev：超级块所在的设备的描述符。

s_blocksize 和 s_blocksize_bits：指定了磁盘文件系统的块的大小。

s_dirty：超级块的“脏”位。

s_maxbytes: 文件最大的大小。

s_type：指向文件系统的类型的指针。

s_op: 指向超级块操作的指针。

s_root：指向目录的 dentry 项。

s_dirt：表示“脏”（内容被修改了，但尚未被刷新到磁盘上）的 inode 节点的链表，分别指向前一个节点和后一个节点。

s_fs_info：指向各个文件系统私有数据，一般是各文件系统对应的超级块信息。以 ext2 文件系统为例，当 ext2 文件系统的超级块装入到内存，即装入到 super_block 的时候，会调用 ext2_fill_super() 函数，在这个函数中填写 ext2 对应的 ext2_sb_info，然后挂在这个指针上

VFS 超级块的操作

在系统运行中， VFS 要建立、撤消一些 VFS inode，还要对 VFS 超级块进行一些必要的操作。这些操作由一系列操作函数实现。不同文件系统的组织和结构不同，完成同样功能的操作函数的代码不同，每种文件系统都有自己的操作函数。

如何在对某文件系统进行操作时就能调用该文件系统的操作函数呢？这是由 VFS 接口通过转换实现的。

在 VFS 超级块中 s_op 是一个指向 super_operations 结构的指针，super_operations 中包含着一系列的操作函数指针，即这些操作函数的入口地址。

每种文件系统 VFS 超级块指向的 super_operations 中记载的是该文件系统的操作函数的入口地址。只需使用它们各自的超级块成员项 s_op，以统一的函数调用形式：s_op->read_inode()就可以分别调用它们各自的读 inode 操作函数

read_inode()：用磁盘上读取的信息来填充 inode 对象的内容，读取的 inode 结构中的 i_ino 对象可以用来在磁盘上定位对应的 inode 节点。

dirty_inode()：表示一个 inode 对象已经“脏”。

write_inode()：更新 inode 的信息，将其转换为磁盘相关的信息并写回。

put_inode()：当有人释放 inode 对象引用的时候被调用，但是并不一定表示这个 inode 没人使用了，只是使用者减少了一个。

delete_inode():当 inode 的引用计数到达 0 的时候被调用，表明这个 inode 对应的对象可以被删除。删除磁盘的数据块，磁盘的 inode 以及 VFS 的 inode。

put_super()：由于当前的文件系统的卸载而释放当前的超级块对象。

write_super()：更新当前的超级块对象的内容。

statfs()：返回当前 mount 的文件系统的一些统计信息

remount_fs()：按照一定的选项重新 mount 文件系统

clear_inode()：和 put_inode 类似，但是也删除包含数据在内的内存对应 inode 中的结构。

umount_begin()：开始 umount 操作，并中断其它的 mount 操作，用于网络文件系统

VFS 的 inode 对象

Linux 以 ext2/ext3 做为基本的文件系统，所以它的虚拟文件系统 VFS 中也设置了 inode 结构

物理文件系统的 inode 在外存中并且是长期存在的，VFS 的 inode 对象在内存中，它仅在需要时才建立，不再需要时撤消。

物理文件系统的 inode 是静态的，而 VFS 的 inode 是一种动态结构。

VFS 的 inode 与某个文件的对应关系是通过设备号 i_dev 与 inode 号 i_ino 建立的，它们唯一地指定了某个设备上的一个文件或目录。

VFS 的 inode 是物理设备上的文件或目录的 inode 在内存中的统一映像。这些特有信息是各种文件系统的 inode 在内存中的映像。如 EXT2 的 ext2_inode_info 结构。

i_lock 表示该 inode 被锁定，禁止对它的访问。i_flock 表示该 inode 对应的文件被锁定。i_flock 是个指向 file_lock 结构链表的指针，该链表指出了一系列被锁定的文件。

VFS 的 inode 组成一个双向链表，全局变量 first_inode 指向链表的表头。在这个链表中，空闲的 inode 总是从表头加入，而占用的 inode 总是从表尾加入。

系统还设置了一些管理 inode 对象的全局变量， 如：

max_inodes 给定了 inode 的最大数量，

nr_inodes 表示当前使用的 inode 数量，

nr_free_inodes 表示空闲的 inode 数量。

目录项对象 dentry object

每个文件除了有一个索引节点 inode 数据结构外，还有一个目录项 dentry 数据结构。

每个 dentry 代表路径中的一个特定部分。如：/、bin、vi 都属于目录项对象。

目录项也可包括安装点，如：/mnt/cdrom/foo，/、mnt、cdrom、foo 都属于目录项对象。

inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统，其 inode 结构在磁盘上就有对应的映像

一个索引节点对象可能对应多个目录项对象

目录项对象作用是帮助实现文件的快速定位，还起到缓冲作用

VFS 的 dentry cache 与 inode cache

为了加速对经常使用的目录的访问， VFS 文件系统维护着一个目录项的缓存。

为了加快文件的查找速度 VFS 文件系统维护一个 inode 节点的缓存以加速对所有装配的文件系统的访问。

用 hash 表将缓存对象组织起来。

file 对象

文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。

该对象由相应的 open()系统调用创建，由 close()系统调用销毁。

文件名不同，但是同一个文件，有相同的 inode

实现了“符号连接”（symbolic links）的方式，使得连接文件只需要存放 inode 的空间。

第一次 mount 的时候会调用 ext2_read_super()

每个注册的文件系统登记在 file_system_type 结构体中，file_system_type 结构体组成一个链表，称为注册链表，链表的表头由全局变量 file_system 给出。Mount 的时候，先调用 sys_mount()，得到相应的 file_system_type 结构。**已安装的文件系统用一个 vfsmount 结构**

已安装的文件系统用一个 vfsmount 结构

超级块(super block):文件系统中最重要的结构，描述整个文件系统的信息。

组描述符(group descriptor):记录所有块组的信息，如块组中的空闲块数、空闲节点数等。

数据的块位图（block bitmap）:每一个块组有一个对应的块位图。大小为一个块,位图的每一位顺序对应组中的一个块,0 表示可用,1 表示已用。

inode 位图（inode bitmap）：每一个块组有一个对应的 inode 位图。用来表示对应的 inode 表的空间是否已被占用。

inode 表(inode table):用来存放文件及目录的 inode 数据。每个文件用一个 inode 表示（每个 inode 是 128B）。

__u32 i_block[EXT2_N_BLOCKS];/*指向磁盘块的指针 */

i_block[EXT2_N_BLOCKS]：一般用于放置文件的数据所在的磁盘块编号,EXT2_N_BLOCKS 的默认值为 15。

不同类型文件的 i_block 的含义不同：

设备文件：用 ext2_inode 就足以包含所有信息,不需要另外的数据块。

目录：数据块包含了所有属于这个目录的文件信息。数据块的数据项是个 ext2_dir_entry_2 结构,用来描述属于这个目录的文件。目录的各项在数据块中依次放置。

普通文件：指向数据块，数据在磁盘上并不一定连续,需要保存各个磁盘块号。i_block[]的前 12 项可看成一级指针,直接存放文件数据所在的磁盘块号。第 13 项是个二级指针（一级索引），第 14,15 项分别是三级指针和四级指针（二级、三级索引）

根目录的 inode 号总为 2

已知 inode 号，读 inode 信息：

因为每组的 inode 数目固定,所以很容易计算出该文件属于那个组并且得到在组中 inode 表的下标,继而可得到 ext2_inode 信息。

当一个进程通过 fork()创建一个子进程后，子进程共享父进程的打开文件表，父子进程的打开文件表中下标相同的两个元素指向同一个 file 结构。 这时 file 的 f_count 计数值增 1。

一个文件可以被某个进程多次打开，每次都分配一个 file 结构，并占用该进程打开文件表 fd[]的一项，得到一个文件描述符。但它们的 file 结构中的 f_inode 都指向同一个 inode。

各种对象的操作接口实现

每个对象都有一套函数操作集合,属于不同文件系统的对象的操作也就不同，VFS 只负责提供接口（操作集合）。

一般**日志文件系统**通常不把所有的块都拷贝到日志中。事实上，每个文件系统都由两种块组成：包含所谓元数据（metadata）的块和包含普通数据的块。在 Ext2 和 Ext3 的情形中，有六种元数据：超级块、块组描述符、索引节点、用于间接寻址的块（间接块）、数据位图块和索引节点块。其他的文件系统可能使用不同的元数据。

```
struct super_block {
  struct list_head s_list;/*  将所有的超级块链接起来 */
  kdev_t s_dev; /*  所在设备号*/
  unsigned long s_blocksize; /*该文件系统磁盘块的大小(字节数)*/
  unsigned char s_blocksize_bits;
  struct file_system_type *s_type; /*文件系统类型*/
  struct super_operations *s_op;/*指向超级块操作的指针*/
  struct dentry *s_root;     /*根目录 dentry 对象*/
  struct list_head s_dirty; /*  修改过的 inodes 队列  */ …
}
```

super_operations

read_inode：当 VFS 在系统中建立一个新的 VFSinode 时，则调用该函数从外存中读取一个文件或目录的 inode 的相关值来填充它。

dirty_inode：表示 inode 对象已经“脏”。该函数主要是对 NFS（网络文件系统）执行的操作。 inode 的属性改变时，调用该函数通知外部的计算机系统。

write_inode：当 VFSinode 的内容发生变动时，调用此函数把它写回到外存中对应的 inode 中。

put_inode(inode)：调用该函数撤消某个 VFSinode。

put_super：当某个文件系统卸载时，调用该函数撤消其超级块，同时释放与超级块有关的高速缓存空间。然后，把该超级块的成员项 s_dev 置 0，表明该超级块已撤消。以后建立新超级块时可以再次使用它。

write_super：当 VFS 超级块的内容发生变化时，调用该函数把 VFS 超级块的内容写回外存中保存。

statfs：调用该函数可以得到文件系统的某些统计信息。参数 statsbuf 指向一个 statfs 结构体，函数执行中把文件系统的统计信息填入该结构体中。当函数返回时，从 statfs 结构体中可以有关统计信息。statfs 结构与硬件有关.

remount_fs：在重新安装文件系统时需要调用该函数

```
struct inode {
        struct list_head        i_hash; /* inode hash 链表指针  */
        struct list_head        i_list;      /* inode 链表指针  */
          struct list_head      i_dentry; /*dentry 链表*/
           kdev_t               i_dev;       /*  主设备号*/
           unsigned long        i_ino; /*  外存的 inode 号  */
           umode_t i_mode;      /*  文件类型和访问权限  */
           nlink_t              i_nlink;     /*  该文件的链接数  */
           uid_t                i_uid;       /*  文件所有者的用户标识  */
           gid_t                i_gid;       /*  文件的用户组标识  */
           kdev_t               i_rdev;      /*  次设备号  */
           off_t                i_size;      /*  文件长度，以字节为单位  */
           time_t               i_atime;     /*  文件最后一次访问时间  */
           time_t               i_mtime;     /*  文件最后一次修改时间  */
           time_t               i_ctime;     /*  文件创建时间  */
           unsigned long        i_blksize; /*  块尺寸，以字节为单位  */
           unsigned long        i_blocks;   /*  文件的块数  */
           unsigned long        i_version; /*  文件版本号  */
           unsigned long        i_nrpages; /*  文件在内存中占用的页面数  */
           struct semaphore     i_sem;   /*  文件同步操作用的信号量  */
           struct inode_operations      *i_op; /*  指向 inode 操作函数入口表的指针  */
           struct super_block           *i_sb;    /*  指向该文件系统的 VFS 超级块  */
           struct wait_queue            *i_wait;   /*  文件同步操作用等待队列  */
           struct file_lock     *i_flock;   /*  指向文件锁定链表的指针  */
           struct vm_area_struct *i_mmap; /*  文件使用的虚存区域  */
           struct page *i_pages;      /*  指向文件占用内存页面 page 结构体链表  */
           struct dquot *i_dquot[MAXQUOTAS];
           struct inode *i_bound_to, *i_bound_by;
           struct  inode  *i_mount;    /*  指向该文件系统根目录 inode 的指针  */
           unsigned long i_count;       /*  使用该 inode 的进程计数  */
           unsigned short i_flags;   /*  该文件系统的超级块标志  */
           unsigned short i_writecount;/*  写计数  */
           unsigned char i_lock;      /*  对该 inode 的锁定标志  */
```

```
        unsigned char i_dirt;      /* 该 inode 的修改标志 */
        unsigned char i_pipe;      /* 该 inode 表示管道文件 */
        unsigned char i_sock;      /* 该 inode 表示套接字 */
        unsigned char i_seek;      /* 未使用 */
        unsigned char i_update;    /* inode 更新标志 */
        unsigned char i_condemned;
}
```

create：只适用于目录 inode，当 VFS 需要在“inode”里面创建一个文件（文件名在 dentry 里面给出）的时候被调用。VFS 必须已经检查过文件名在这个目录里面不存在。
lookup:用于检查一个文件（文件名在 dentry 里面给出）是否在一个 inode 目录里面。
link：在 inode 所给出的目录里面创建一个从第一个参数 dentry 文件到第三个参数 dentry 文件的硬链接（hard link）。
unlink：从 inode 目录里面删除 dentry 所代表的文件。
symlink：用于在 inode 目录里面创建软链接（soft link）。
mkdir：用于在 inode 目录里面创建子目录。
rmdir：用于在 inode 目录里面删除子目录。
mknod：用于在 inode 目录里面创建设备文件。
rename：把第一个和第二个参数（inode，dentry）所定位的文件改名为第三个和第四个参数所定位的文件。
readlink：读取一个软链接所指向的文件名。
follow_link：VFS 调用这个函数跟踪一个软链接到它所指向的 inode。
put_link：VFS 调用这个函数释放 follow_link 分配的一些资源。
truncate：VFS 调用这个函数改变一个文件的大小。
permission：VFS 调用这个函数得到对一个文件的访问权限。
setattr：VFS 调用这个函数设置一个文件的属性。比如 chmod 系统调用就是调用这个函数。
getattr：查看一个文件的属性。比如 stat 系统调用就是调用这个函数。
setxattr：设置一个文件的某项特殊属性。详细情况请查看 setxattr 系统调用帮助。
getxattr：查看一个文件的某项特殊属性。详细情况请查看 getxattr 系统调用帮助。
listxattr：查看一个文件的所有特殊属性。详细情况请查看 listxattr 系统调用帮助。
removexattr：删除一个文件的特殊属性。详细情况请查看 removexattr 系统调用帮助。

```
struct dentry {
    atomic_t d_count;            /* 目录项引用计数器 */
    unsigned int d_flags;      /* 目录项标志 */
    struct inode   * d_inode;    /* 与文件名关联的索引节点 */
    struct dentry * d_parent;          /* 父目录的目录项 */
    struct list_head d_hash;           /* 目录项形成的哈希表 */
    struct list_head d_lru;           /*未使用的 LRU 链表 */
    struct list_head d_child;         /*父目录的子目录项所形成的链表 */
    struct list_head d_subdirs;       /* 该目录项的子目录所形成的链表*/
    struct list_head d_alias;         /* 索引节点别名的链表*/
    int d_mounted;                     /* 目录项的安装点 */
    struct qstr d_name;               /* 目录项名（可快速查找） */
    struct dentry_operations   *d_op;/* 操作目录项的函数*/
    struct super_block * d_sb;      /* 目录项树的根（即文件的超级块）*/
    unsigned long d_vfs_flags;
    void * d_fsdata;                   /* 具体文件系统的数据 */
unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
……
}
```

d_revalidate（）：判定目录项是否有效。
d_hash（）：生成一个哈希值。
d_compare（）：比较两个文件名
d_delete（）：删除 d_count 域为 0 的目录项对象
d_release（）释放一个目录项对象。
d_iput（）： 调用该方法丢弃目录项对应的索引节点

```
struct file {
 struct list_head f_list;   /*file 结构链表*/
 struct dentry     *f_dentry;/*指向与文件对象关联的 dentry 对象*/
 struct vfsmount *f_vfsmnt; /*文件相应的 vfsmount 结构*/
 struct file_operations *f_op; /*文件对象的操作集合*/
 atomic_t f_count ; /*文件打开的引用计数*/
 unsigned int f_flags; /*使用 open（）时设定的标志*/
 mode_t   f_mode;   /*文件读写权限*/
 loff_t   f_pos;    /*对文件读写操作的当前位置*/
 struct fown_struct          f_owner;
     ……
 };
```

llseek：用于移动文件内部偏移量。
read：读文件。
aio_read：异步读，被 io_submit 和其他的异步 IO 函数调用。
write：写文件。
aio_write：异步写，被 io_submit 和其他的异步 IO 函数调用。
readdir：当 VFS 需要读目录内容的时候调用这个函数。
poll：当一个进程想检查一个文件是否有内容可读写的时候，VFS 调用这个函数；一般来说，调用这个函数之后进程进入睡眠，直到文件中有内容读写就绪时被唤醒。详情请参考 select 和 poll 系统调用。
ioctl：被系统调用 ioctl 调用。
unlocked_ioctl：被系统调用 ioctl 调用；不需要 BKL（内核锁）的文件系统应该使用这个函数，而不是上面那个 ioctl。
compat_ioctl：被系统调用 ioctl 调用；当在 64 位内核上使用 32 位系统调用的时候使用这个 ioctl 函数。
mmap：被系统调用 mmap 调用。
open：通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象。
flush：被系统调用 close 调用，把一个文件内容写回磁盘。
release：当对一个打开文件的最后引用关闭的时候，VFS 调用这个函数释放文件。
fsync：被系统调用 fsync 调用。
fasync：当对一个文件启用异步读写（非阻塞读写）的时候，被系统调用 fcntl 调用。
lock：fcntl 系统调用使用命令 F_GETLK，F_SETLK 和 F_SETLKW 的时候，调用这个函数。

```
 struct file_system_type {//文件系统注册链表
        const char *name;//Filesystem name
        int fs_flags;//Filesystem type flags
        struct super_block *(*get_sb) (struct file_system_type *, int,
                                       const char *, void *);//Method for
reading a superblock
        void (*kill_sb) (struct super_block *);//Method for removing a
superblock
        struct module *owner;//Pointer to the module implementing the
filesystem
        struct file_system_type * next;//Pointer to the next element in the list
of filesystem types
        struct list_head fs_supers;//Head of a list of superblock objects having
the same filesystem type
};
struct vfsmount {//已安装的文件系统用一个 vfsmount 结构
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;        /* fs we are mounted on */
        struct dentry *mnt_mountpoint;      /* dentry of mountpoint */
        struct dentry *mnt_root;            /* root of the mounted tree */
        struct super_block *mnt_sb;                 /* pointer to
superblock */
        struct list_head mnt_mounts;        /* list of children, anchored here
*/
        struct list_head mnt_child;         /* and going through their
mnt_child */
        atomic_t mnt_count;
        int mnt_flags;
        int mnt_expiry_mark;                /* true if marked for expiry */
        char *mnt_devname;                  /* Name of device e.g.
/dev/dsk/hda1 */
        struct list_head mnt_list;
        struct list_head mnt_expire;        /* link in fs-specific expiry list */
        struct list_head mnt_share;         /* circular list of shared mounts
*/
        struct list_head mnt_slave_list;    /* list of slave mounts */
        struct list_head mnt_slave;         /* slave list entry */
        struct vfsmount *mnt_master;        /* slave is on
master->mnt_slave_list */
        struct namespace *mnt_namespace; /* containing namespace */
        int mnt_pinned;
};
```

do_page_fault()工作原理: compares the linear address that caused the Page Fault against the memory regions of the current process; it can thus determine the proper way to handle the exception
Do_page_fault（）:

```
fastcall void __kprobes do_page_fault(struct pt_regs *regs,unsigned long
error_code)
{       //开始变量定义
    address = read_cr2();//从 CR2 寄存器中读取引起缺页的地址
        tsk = current; //当前进程描述符
        si_code = SEGV_MAPERR;//接下来,检查引起缺页的地址是否属于当
前进程地址空间
        if (unlikely(address >= TASK_SIZE)) {
                if(!(error_code&0x0000000d)&&   malloc_fault(address)   >=
0)//内核访问不存在的 page frame
                        return;
                ……
        if (regs->eflags & (X86_EFLAGS_IF|VM_MASK))
        local_irq_enable();//将当前进程设为可以接受中断信号
        mm = tsk->mm;//内存管理描述符
if (in_atomic()||!mm)//检查内核是否在进行处理中断,在临界区
                goto bad_area_nosemaphore;
        if (!down_read_trylock(&mm->mmap_sem)) {
        if((error_code&4)=0&& !search_exception_tables(regs->eip))
                        goto bad_area_nosemaphore;
                down_read(&mm->mmap_sem);
        }//下面:查找一个包含 faulty address 的内存区域
vma = find_vma(mm, address);
        if (!vma)//没有找到包含 faulty address 的内在区域
                goto bad_area;
if (vma->vm_start <= address)//找到了.且在当前进程的地址空间
                goto good_area;
if (!(vma->vm_flags & VM_GROWSDOWN))//缺页不发生在用用户空间
                goto bad_area;
        ……两个 if……
good_area://正常的缺页处理程序由此开始
        si_code = SEGV_ACCERR;
        write = 0;
        switch (error_code & 3) {
                default:  /* 3: write, present */
                        /* fall through */
                case 2:         /* write, not present */
                        if (!(vma->vm_flags & VM_WRITE))
                                goto bad_area;
                        write++;
                        break;
                case 1:         /* read, present */
                        goto bad_area;
                case 0:         /* read, not present */
                        if  (!(vma->vm_flags  &  (VM_READ  |  VM_EXEC  |
VM_WRITE)))
                                goto bad_area;
        }
 survive://这里为真正的缺页处理的地方
        fault = handle_mm_fault(mm, vma, address, write);//处理缺页
        ……各种 if……
bad_area://faulty address 不属于当前进程空间
        up_read(&mm->mmap_sem);
bad_area_nosemaphore://user mode accesses just cause a SIGSEGV
        ……各种 if……
                tsk->thread.cr2 = address;
                /* Kernel addresses are always protection faults */
                tsk->thread.error_code  =  error_code  |  (address  >=
TASK_SIZE);
                tsk->thread.trap_no = 14;
                force_sig_info_fault(SIGSEGV, si_code, address, tsk);
                return;//向进程发送 SIGSEGV 信号,且要保证不能被阻塞
        }
no_context://缺页是在 kernel mode 下发生的
                ……几层 if 中……
                if ((page >> PAGE_SHIFT) < max_low_pfn
                    && (page & _PAGE_PRESENT)) {
                        page &= PAGE_MASK;
                        page    =    ((__typeof__(page)    *)
__va(page))[(address >> PAGE_SHIFT)
                                                                    &
(PTRS_PER_PTE - 1)];
                        printk(KERN_ALERT  "*pte  =  %0*Lx\n",
sizeof(page)*2, (u64)page);//这是主要是处理在 kernel mode 下发生 fault, 首先是报告
                }//内核有 BUG,打出出错信息,然后直接中止进程
        ……一些结构复制还有个 die……
        do_exit(SIGKILL);
//在 kernel 状态下,引起 fault 有两种情况: (1).系统调用的参数中含有地址//,处理时发生错误; (2). Kernel 存在 BUG
out_of_memory://内存溢出,或不能正解处理缺页
do_sigbus://总线出错
        up_read(&mm->mmap_sem);
        /* Kernel mode? Handle exceptions or die */
        if (!(error_code & 4))
                goto no_context;
        /* User space => ok to do another page fault */
        if (is_prefetch(regs, address, error_code))
                return;
        ……
}
```

进程

存放在磁盘上的可执行文件的代码和数据的集合称为可执行映象(Executable Image)。
当一个程序装入系统中运行时，它就形成了一个进程。
进程是由正文段(text)、用户数据段(user segment)和系统数据段、堆栈段(system segment)组成的一个动态实体。
正文段中存放着进程要执行的指令代码，具有只读的属性。
用户数据段是进程在运行过程中处理数据的集合，它们是进程直接进行操作的所有数据，以及进程使用的进程堆栈。
系统数据段存放着进程的控制信息。其中包括进程控制块 PCB。
task_struct 结构描述
Linux 中的进程主要有如下状态
TASK_RUNNING：正在运行的进程即系统的当前进程或准备运行的进程即在 Running 队列中的进程。只有处于该状态的进程才实际参与进程调度。
TASK_INTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，也可以被其他进程或内核用信号、中断唤醒后进入就绪状态。
TASK_UNINTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，不可以被其它进程或内核通过信号、中断唤醒。
TASK_STOPPED：进程被暂停，一般当进程收到下列信号之一时进入这个状态：SIGSTOP，SIGTSTP，SIGTTIN 或者 SIGTTOU。通过其它进程的信号才能唤醒。
TASK_TRACED：进程被跟踪，一般在调试的时候用到。
EXIT_ZOMBIE：正在终止的进程，等待父进程调用 wait4()或者 waitpid()回收信息。是进程结束运行前的一个过度状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在内核中仍然保留一些这个进程的数据结构（比如 task_struct）等待父进程回收。
EXIT_DEAD：进程消亡前的最后一个状态，表示父进程已经获得了该进程的记账信息，该进程可以被销毁了。
TASK_NONINTERACTIVE：表明这个进程不是一个交互式进程，在调度器的设计中，对交互式进程的运行时间片会有一定奖励或者惩罚。是一个辅助标记

进程创建的原理

系统创建的第 1 个真正进程是 init 进程，pid=1。
系统中所有的进程都是由进程使用系统调用 fork()创建的。
子进程被创建后继承了父进程的资源。子进程共享父进程的虚存空间（只读方式）。
写时拷贝(copy on write)：子进程在创建后共享父进程的虚存内存空间，只是在两个进程中某一个进程需要向虚拟内存写入数据时才拷贝相应部分的虚拟内存。
子进程在创建后执行的是父进程的程序代码。子进程通过调用 exec 系列函数执行真正的任务。

进程创建的过程:

1.为新进程分配 task_struct 内存空间;
2.把父进程 task_struct 拷贝到子进程的 task_struct;
3.为新进程在其虚拟内存建立内核堆栈;
4.对子进程 task_struct 中部分进行初始化设置
5.把父进程的有关信息拷贝给子进程，建立共享关系;
6.把子进程的 counter 设为父进程 counter 值的一半;
7.把子进程加入到可运行队列中;
8.结束 do_fork()函数返回 PID 值.

do_fork()的执行过程(源代码在 kernel/fork.c 文件中):

1) 调用 alloc_task_struct()分配子进程 task_struct 空间。严格地讲，此时子进程还未生成。
2) 把父进程 task_struct 的值全部赋给子进程 task_struct。
3) 检查是否超过了资源限制，如果是，则结束并返回出错信息。更改一些统计量的信息。
4) 修改子进程 task_struct 的某些成员的值使其正确反映子进程的状况，如进程状态被置成 TASK_UNINTERRUPTIBLE。
5) 调用 get_pid( )函数为子进程得到一个 pid 号。
6)共享或复制父进程文件处理、信号处理及进程虚拟地址空间等资源。
7) 调用 copy_thread( )初始化子进程的内核栈,内核栈保存了进程返回用户空间的上文。此处与平台相关，以 i386 为例，其中很重要的一点是存储寄存器 eax 值的位置被置 0,这个值就执行系统调用后子进程的返回值。
8) 将父进程的当前的时间配额 counter 分一半给子进程。
9) 利用宏 SET_LINKS 将子进程插入所有进程都在其中的双向链表。调用 hash_pid（）,将子进程加入相应的 hash 队列。
10) 调用 wake_up_process( ),将该子进程插入可运行队列。至此，子进程创建完毕,并在可运行队列中等待被调度运行。
11) 如果 clone_flags 包含有 CLONE_VFORK 标志，则将父进程挂起直到子进程释放进程空间。进程控制块中有一个信号量 vfork_sem 可以起到将进程挂起的作用。
12) 返回子进程的 pid 值，该值就是系统调用后父进程的返回值。

进程等待的机制

Linux 提供了多种使进程进入等待态的机制，如：wait、sleep 等
内核函数 sys_wait4()的功能就是使进程进入等待态。
当前进程在运行过程中需要等待它的子进程终止时，调用该函数使其状态从运行态转换成可中断的等待态(INTERUPTIBLE)，并加入到等待队列中。
当被等待的子进程运行终止后，发出信号通知处于等待态的父进程，把父进程唤醒，使父进程继续运行。

```
int sys_wait4(pid_t pid,unsigned int * stat_addr,
int options,struct rusage * ru)
```

pid：
pid > 0 等待进程标识为 pid 的子进程。
pid = 0 等待与当前进程属于同一个用户组的子进程。
pid < -1 等待最早创建的子进程终止。
pid = -1 等待所有子进程的终止。

state_addr：状态变量的地址：
函数执行时的状态代码存放在该变量中，待函数返回时，可以通过对这个变量的值的检测，了解到子进程终止的原因是正常终止、还是因信号被终止或因信号而暂停。
option：控制选项
用于指定调用该函数的进程中是等待子进程的结束，还是不需等待子进程的结束而继续运行。
ru：传递使用资源结构体。
在执行 sys_wait4( )过程中，把进程及其子进程使用资源的有关信息写入该结构体中。在函数返回后，可以从该结构体中得到这些信息。

进程的睡眠和唤醒

进程的睡眠
当前进程在运行过程中还可以通过系统调用函数 sleep_on() 和 interruptible_sleep_on()使其从运行态转换为等待状态。
把使用这些函数进入的等待态又称为睡眠态。

进程的唤醒
在等待队列中处于睡眠状态的进程，在等待的事件发生后需要使用内核函数把它们唤醒。
wake_up()：唤醒等待队列中的可中断态和不可中断态的进程
wake_up_interruptible()：唤醒可中断态进程。

进程的终止和撤消

进程完成本身的任务，自动终止
进程被内核强制终止
进程终止：do_exit()
(1)设定当前进程的标志
(2)释放系统中该进程在各种管理队列中的任务结构体
(3)释放进程使用的各种资源
(4)把进程的状态转为僵死态
(5)把退出码置入任务结构体
(6)变更进程族亲关系
(7)执行进程，选择下一个使用 CPU 的进程。

2. 进程调度信息

调度程序利用这部分信息决定系统中哪个进程最应该运行，并结合进程的状态信息保证系统运转的公平和高效。这一部分信息通常包括进程的类别（普通进程还是实时进程）、进程的优先级等等。如表 4.2 所示：

need_resched	调度标志
Nice	静态优先级
Counter	动态优先级
Policy	调度策略
rt_priority	实时优先级

当 need_resched 被设置时，在“下一次的调度机会”就调用调度程序 schedule()。 counter 代表进程剩余的时间片，是进程调度的主要依据，也可以说是进程的动态优先级，因为这个值在不断地减少；nice 是进程的静态优先级，同时也代表进程的时间片，用于对 counter 赋值，可以用 nice()系统调用改变这个值；policy 是适用于该进程的调度策略，实时进程和普通进程的调度策略是不同的；rt_priority 只对实时进程有意义，它是实时进程调度的依据。
进程的调度策略有三种，

SCHED_OTHER	其他调度	普通进程
SCHED_FIFO	先来先服务调度	实时进程
SCHED_RR	时间片轮转调度	

普通进程的调度

实时进程调度的中心思想是，让处于可执行状态的最高优先级的实时进程尽可能地占有 CPU，因为它有实时需求；而普通进程则被认为是没有实时需求的进程，于是调度程序力图让各个处于可执行状态的普通进程和平共处地分享 CPU，从而让用户觉得这些进程是同时运行的。
与实时进程相比，普通进程的调度要复杂得多。内核需要考虑两件麻烦事：

一、动态调整进程的优先级

按进程的行为特征，可以将进程分为“交互式进程”和“批处理进程”：
交互式进程（如桌面程序、服务器、等）主要的任务是与外界交互。这样的进程应该具有较高的优先级，它们总是睡眠等待外界的输入。而在输入到来，内核将其唤醒时，它们又应该很快被调度执行，以做出响应。比如一个桌面程序，如果鼠标点击后半秒种还没反应，用户就会感觉系统“卡”了；
批处理进程（如编译程序）主要的任务是做持续的运算，因而它们会持续处于可执行状态。这样的进程一般不需要高优先级，比如编译程序多运行了几秒种，用户多半不会太在意；
如果用户能够明确知道进程应该有怎样的优先级，可以通过 nice、setpriority 系统调用来对优先级进行设置。（如果要提高进程的优先级，要求用户进程具有 CAP_SYS_NICE 能力。）
然而应用程序未必就像桌面程序、编译程序这样典型。程序的行为可能五花八门，可能一会儿像交互式进程，一会儿又像批处理进程。以致于用户难以给它设置一个合适的优先级。
再者，即使用户明确知道一个进程是交互式还是批处理，也多半碍于权限或因为偷懒而不去设置进程的优先级。（你又是否为某个程序设置过优先级呢？）
于是，最终，区分交互式进程和批处理进程的重任就落到了内核的调度程序上。
调度程序关注进程近一段时间内的表现（主要是检查其睡眠时间和运行时间），根据一些经验性的公式，判断它现在是交互式的还是批处理的？程度如何？最后决定给它的优先级做一定的调整。
进程的优先级被动态调整后，就出现了两个优先级：
1、用户程序设置的优先级（如果未设置，则使用默认值），称为静态优先级。这是进程优先级的基准，在进程执行的过程中往往是不改变的；
2、优先级动态调整后，实际生效的优先级。这个值是可能时时刻刻都在变化的；

二、调度的公平性

在支持多进程的系统中，理想情况下，各个进程应该是根据其优先级公平地占有 CPU。而不会出现“谁运气好谁占得多”这样的不可控的情况。
linux 实现公平调度基本上是两种思路：
1、给处于可执行状态的进程分配时间片（按照优先级），用完时间片的进程被放到“过期队列”中。等可执行状态的进程都过期了，再重新分配时间片；
2、动态调整进程的优先级。随着进程在 CPU 上运行，其优先级被不断调低，以便其他优先级较低的进程得到运行机会；
后一种方式有更小的调度粒度，并且将“公平性”与“动态调整优先级”两件事情合而为一，大大简化了内核调度程序的代码。因此，这种方式也成为内核调度程序的新宠。
强调一下，以上两点都是仅针对普通进程的。而对于实时进程，内核既不能自作多情地去动态调整优先级，也没有什么公平性可言。

调度程序的效率

“优先级”明确了哪个进程应该被调度执行，而调度程序还必须要关心效率问题。调度程序跟内核中的很多过程一样会频繁被执行，如果效率不济就会浪费很多 CPU 时间，导致系统性能下降。
在 linux 2.4 时，可执行状态的进程被挂在一个链表中。每次调度，调度程序需要扫描整个链表，以找出最优的那个进程来运行。复杂度为 O(n)；
在 linux 2.6 早期，可执行状态的进程被挂在 N(N=140)个链表中，每一个链表代表一个优先级，系统中支持多少个优先级就有多少个链表。每次调度，调度程序只需要从第一个不为空的链表中取出位于链表头的进程即可。这样就大大提高了调度程序的效率，复杂度为 O(1)；
在 linux 2.6 近期的版本中，可执行状态的进程按照优先级顺序被挂在一个红黑树（可以想象成平衡二叉树）中。每次调度，调度程序需要从树中找出优先级最高的进程。复杂度为 O(logN)。

那么，为什么从 linux 2.6 早期到近期 linux 2.6 版本，调度程序选择进程时的复杂度反而增加了呢？

这是因为，与此同时，调度程序对公平性的实现从上面提到的第一种思路改变为第二种思路（通过动态调整优先级实现）。而 O(1)的算法是基于一组数目不大的链表来实现的，按我的理解，这使得优先级的取值范围很小（区分度很低），不能满足公平性的需求。而使用红黑树则对优先级的取值没有限制（可以用 32 位、64 位、或更多位来表示优先级的值），并且 O(logN)的复杂度也还是很高效的。

调度触发的时机

调度的触发主要有如下几种情况：
1、当前进程（正在 CPU 上运行的进程）状态变为非可执行状态。
进程执行系统调用主动变为非可执行状态。比如执行 nanosleep 进入睡眠、执行 exit 退出、等等；
进程请求的资源得不到满足而被迫进入睡眠状态。比如执行 read 系统调用时，磁盘高速缓存里没有所需要的数据，从而睡眠等待磁盘 IO；
进程响应信号而变为非可执行状态。比如响应 SIGSTOP 进入暂停状态、响应 SIGKILL 退出、等等；
2、抢占。进程运行时，非预期地被剥夺 CPU 的使用权。这又分两种情况：
进程用完了时间片、或出现了优先级更高的进程。
优先级更高的进程受正在 CPU 上运行的进程的影响而被唤醒。如发送信号主动唤醒，或因为释放互斥对象（如释放锁）而被唤醒；
内核在响应时钟中断的过程中，发现当前进程的时间片用完；
内核在响应中断的过程中，发现优先级更高的进程所等待的外部资源的变为可用，从而将其唤醒。比如 CPU 收到网卡中断，内核处理该中断，发现某个 socket 可读，于是唤醒正在等待读这个 socket 的进程；再比如内核在处理时钟中断的过程中，触发了定时器，从而唤醒对应的正在 nanosleep 系统调用中睡眠的进程；

Linux 线程

Linux 把线程和进程一视同仁，每个线程拥有唯一属于自己的 task_struct 结构。不过线程本身拥有的资源少，共享进程的资源，如：共享地址空间、文件系统资源、文件描述符和信号处理程序。
Linux 内核线程一在内核态下创建、独立执行的一个内核函数：

```
int kernel_thread(int (*fn)(void *), void * arg,
unsigned long flags)
{ pid_t p;
p =clone( 0, flags | CLONE_VM );
if ( p ) /* 父*/
return p;
else { /* 子*/
fn(arg); exit( );
} }
```

内核线程是通过系统调用 clone()来实现的,使用 CLONE_VM 标志说明内核线程与调用它的进程(current)具有相同的进程地址空间.
由于调用进程是在内核中调用 kernel_thread(),因此当系统调用返回时,子进程也处于内核态中,而子进程随后调用 fn,当 fn 退出时,子进程调用 exit()退出,所以子进程是在内核态运行的.
由于内核线程是在内核态运行的,因此内核线程可以访问内核中数据,调用内核函数..
内核线程也可以叫内核任务，它们一般用于周期性地执行某项工作，例如，磁盘高速缓存的刷新，网络连接的维护，页面的换入换出等等。

几个特殊身份的内核线程

swapper（或 idel）进程(init_task)，0 号进程（pid=0）一从无到有诞生的第一个线程，执行 cpu_idle()函数。它只有在 CPU 不能执行其他进程时才执行
init(pid=1)进程，既是内核线程也是 1 号用户进程的。init 进程创建和监控操作系统外层所有进程的活动。
还有另外几个线程，他们是守护进程（运行于后台的系统进程）：
kflushd (即 bdflush)线程：刷新“脏”缓冲区中的内容到磁盘以归还内存。
kupdate 线程：刷新旧的“脏”缓冲区中的内容到磁盘以减少文件系统不一致的风险。
kpiod 线程：把属于共享内存映射的页面交换出去。
kswapd 线程：执行内存回收功能。
用命令 ps aux 参看所有进程，包括守护进程。

Linux 的进程调度

Linux 进程调度方式：
Linux 系统采用抢占调度方式。Linux2.6 内核是抢占式的，这意味着进程无论是处于内核态还是用户态，都可能被抢占，
Linux 的调度基于分时技术（time-sharing）。对于优先级相同进程进程采用时间片轮转法。
根据进程的优先级对它们进行分类。进程的优先级是动态的。

Linux 的进程分为普通进程和实时进程，实时进程的优先级高于普通进程

Linux 进程的策略 policy 有:
普通进程按照 SCHED_NORMAL 调度策略进行进程调度。
实时进程按照 SCHED_FIFO 或 SCHED_RR 策略进行调度。
SCHED_BATCH 是 2.6 新加入的调度策略，这种类型的进程一般都是后台处理进程，总是倾向于跑完自己的时间片，没有交互性。所以对于这种调度策略的进程，调度器一般给的优先级比较低，这样系统就能在没什么事情做的时候运行这些进程，而一旦有交互性的进程需要运行，则立刻切换到交互性的进程，从用户的角度来看，系统的响应性/交互性就很好。
Linux 2.4 的调度算法：
遍历进程可运行队列，算法时间 O(n)
时间片，动态优先级
Linux 2.6.1-2.6.22 调度算法（O(1)算法）：
优先级 0-139，使用 active 和 expire 两个队列，按照优先级调度，算法时间 O(1)
Linux 2.6.23-调度算法（CFS 算法）：
非实时：CFS(完全公平调度器）进程调度器，使用红黑树选取下一个被调度进程，O(lg N)
实时：优先级队列

Linux 2.6 进程调度

2.6 调度系统从设计之初就把开发重点放在更好满足实时性和多处理机并行性上,并且基本实现了它的设计目标。主要设计者,传奇式人物 Ingo Molnar 将新调度系统的特性概括为如下几点：
继承和发扬 2.4 版调度器的特点：

交互式作业优先	轻载条件下调度/唤醒的高性能
公平共享	基于优先级调度
高 CPU 使用率	SMP 高效亲和

实时调度和 cpu 绑定等调度手段
Linux 2.6 的进程设置 140 个优先级。实时进程优先级为 0-99，普通进程优先级 100-139 的数。0 为最高优先权，139 为最低优先权。
优先级分为静态优先级和动态优先级
调度程序根据动态优先级来选择新进程运行。
基本时间片： 静态优先权本质上决定了进程的基本时间片，即进程用完了以前的时间片时，系统分配给进程的时间片长度。静态优先权和基本时间片的关系用下列公式确定：
base time quantum（ms）：
（140-static_priority）*20 if static_priority <120
（140-static_priority）*5 if static_priority >=120

task_struct 相关域

sleep_avg
进程的平均等待时间（nanosecond）
反映交互式进程优先与分时系统的公平共享
值越大，计算出来的进程优先级也越高
run_list
串连在优先级队列中
优先级数组 prio_array 中按顺序排列了各个优先级下的所有进程
调度器在 prio_arrary 中找到相应的 run_list，从而找到其宿主结构 task_struct
time_slice
进程的运行时间片剩余大小
进程的默认时间片与进程的静态优先级相关
进程创建时，与父进程平分时间片
运行过程中递减，一旦归零，则重置时间片，并请求调度
递减和重置在时钟中断中进行（scheduler_tick()）
进程退出时，如果自身并未被重新分配时间片，则将自己剩余的时间片返还给父进程
static_prio 静态优先级
与 2.4 版本中的 nice 值意义相同，但取值区间不同，是用户可影响的优先级
通过 set_user_nice()来改变
static_prio= MAX_RT_PRIO + nice + 20
MAX_RT_PRIO 定义为 100
进程初始时间片的大小仅决定于进程的静态优先级
核心将 100~139 的优先级映射到 200ms~10ms 的时间片上
优先级数值越大，优先级越低，分配的时间片越少
实时进程的 static_prio 不参与优先级 prio 的计算
prio 动态优先级
相当于 2.4 中 goodness() 的计算结果，在 0~MAX_PRIO-1 之间取值（MAX_PRIO 定义为 140），其中：
0~MAX_RT_PRIO-1 （MAX_RT_PRIO 定义为 100）属于实时进程范围；
MAX_RT_PRIO~MAX_PRIO-1 属于非实时进程。数值越大，表示进程优先级越小。
2.6 中，动态优先级不再统一在调度器中计算和比较，而是独立计算，并存储在进程的 task_struct 中，再通过描述的 priority_array 结构自动排序。
普通进程 prio = max (100, min (static priority -bonus + 5, 139))
Bonus 是范围从 0 到 10 的值，bonus 的值小于 5 表示降低动态优先权以示惩罚，bonus 的值大于 5 表示增加动态优先权以示额外奖赏。Bonus 的值依赖于进程过去的情况，说得更准确一些是与进程的平均睡眠时间相关。
unsigned longrt_priority
实时进程的优先级
sys_sched_setschedule()
一经设定在运行时不变，作为其动态优先级
prio_array_t *array
记录当前 CPU 活动的就绪队列
以优先级为序组成数组

runqueue 结构（kernel/sched.c）

runqueue 结构是 Linux2.6 调度程序最重要的数据结构。系统中的每个 CPU 都有它自己的运行队列，所有的 runqueue 结构存放在 runqueues 每 CPU（per-CPU）变量中。
系统中的每个可运行进程属于且只属于一个运行队列。只要可运行进程保持在同一个运行队列中，它就只可能在拥有该运行队列的 CPU 上执行。但是，可运行进程会从一个运行队列迁移到另一个运行队列。
runqueue 结构的字段：
prio_array_t *active, *expired, arrays[2]
每个 CPU 均有两个具有优先级的队列，按时间片是否用完分为“活动队列”（active 指针所指）和“过期队列”（expired 指针所指）。active 指向时间片没用完、当前可被调度的就绪进程，expired 指向时间片已用完的就绪进程。
每一类队列用一个 struct prio_array 表示（优先级排序数组）
一个任务的时间片用完后，它会被转移到“过期”的队列中
在该队列中，任务仍然是按照优先级排好序的
当活动队列中的任务均被执行完时，就交换两个指针