**Shell命令**

**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)|shell-systemcall-kernel-hardware

Bash(BourneAgainShell)|tcsh(TCshell),csh,zsh|ksh(KornShell),pdksh

Ls(commands)🡺BASH(linuxshell)🡺01(convertedbinarycommands)🡺linuxKernel

LINUX系统是多进程、多用户和交互式的计算环境。

Shell命令可以被分为**内部命令和外部命令**。

1.内部命令:shell本身包含的一些命令,是整个shell代码的组成部分；2shell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**

Shell搜索的目录名字都保存在shell变量PATH（在TC shell中是path）中。

变量PATH中的目录名用符号分开。在bash中“:”

变量PATH保存在~/.profile或者~/.login中（~:主目录）

**chsh** 命令来改变默认**登录shell**, -l选项显示系统可用的shell

$echo $SHELL /bin/bash $tcsh %ps

<Ctrl-D> 或exit命令返回到原来的shell或默认的shell

**shell的启动文件**

/ect/passwd记录每个用户的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

**/bin/bash的启动**

继承getty设置的TERM、PATH等环境变量. 首先寻找/etc/profile脚本文件，并执行它。然后在~下寻找启动脚本文件.bash\_profile，执行它；如果不存在，则执行.bash\_login；如果.bash\_login也不存在，则执行.profile文件。然后bash将作为一个交互式shell执行.bashrc文件(注：“~”为登陆目录)

~/.bash\_out：退出时执行

TERM：终端类型

/etc/bashrc ：整个系统的别名和功能；

/etc/profile 整个系统的环境参数和启动程序；

~/.bashrc：用户的的别名和功能；

~/.bash\_profile：用户的环境参数和启动程序

~/.profile：用户登录环境配置文件(Bourne shell的)

**shell的环境变量**

显示shell的路径：echo $SHELL

查看环境变量值： set

设置环境变量值： 变量名=值

HISTFILE：用于贮存历史命令的文件

HISTSIZE：历史命令列表的大小

HOME：当前用户的主目录

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、cwd：当前工作目录。

SECONDS：当前shell开始后所流逝的秒数

**Shell元字符**

" 引用多个字符,允许替换 "$file".bak

‘ 引用多个字符 ‘$100,000’

$ 一行的结束/显示变量的值 $PATH

& 让一个命令在后台执行 command &

( ) 在子shell中执行命令 (cmd1;cmd2)

{ } 在当前shell中执行命令 {cmd1;cmd2}

\* 匹配0个或者多个字符 chap\*.ps

? 匹配单个字符 lab.?

[ ] 插入通配符 [a-s],[1,5-9]

^ 一行的开始/否定符号 [^3-8]

` 替换命令 PS1=`cmd`

| 创建命令间的管道 cmd1|cmd2

; 分割顺序执行的命令 cmd1;cmd2

< 重定向命令的输入 cmd<file

> 重定向命令的输出 cmd>file

\ 转义字符/允许在下一行中继续shell命令

! 启动历史记录列表中的命令和当前命令 !!,!4

% TC shell的提示符，或者指定一个任务号时作为起始字符 %或者%3

如：$ls [0-90[a-zA-Z].html

**常用命令**

passwd :修改密码命令

man, info

如：man password; info password; ls –help|more; man –S2 open; man 2 open

whatis：得到任何LINUX命令的更短的描述

whoami：显示用户名leafior

which：当某个工具或程序有多个副本时，用which来识别哪个副本在运行

who：显示现在正在使用系统的用户的信息

w：比who更加详细地列出系统上用户的信息

hostname：显示登录上的主机的名字Ubuntu

uname：显示操作系统的信息Linux

PATH=~/bin:$PATH:. 搜索路径中增加~/bin和.目录

cal [[month] year]如cal 4 2011

alias [name[=string]…] 为name命令建立别名string。如alias more=’pg’

unalias 删除别名 alias ll=’ls –C’

uptime显示系统运行时间命令

su [-][-c <command>] [username]

-c< cmd >执行完指定的指令后，即恢复原身份。

-改变身份时，也同时变更工作目录，及HOME、SHELL、PATH变量等

Username指定要变更的用户名，默认为root

pwd： 显示当前的工作目录 /home/leafier

<Ctrl-C>: 终止当前的命令or程序

<Ctrl-D>: 结束输入，or退出linux系统，or从上层shell返回

<Ctrl-Z>: 暂停当前命令执行

**文本编辑器**

vi filename(vim filename) 命令模式+插入模式

进入插入模式：I | 退出：ESC或者Ctrl+i

:wq保存并退出 | :q!不保存退出

命令模式下在命令前加：！可以执行shell，如：！pwd

编辑器：vi(m), *emacs, gedit, kedit*, pico/nano

**文件管理操作**

**文件类型**

Linux把所有东西看做文件处理（打印机、键盘…）

【普通文件-】

扩展名无意义。除保留字符外均可命名，<=255char

【目录文件d】只能被系统修改，用户进程只可读取

Linux继承了UINX，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inode。

Linux的目录项主要由文件名和inode号(唯一)组成

【设备文件】字符设备文件c & 块设备文件b

fd0(floppy drive 0); had(harddisk a); lp0(line printer 0);tty(teletype terminal)

【管道(FIFO)文件】进程间传输数据

【链接文件=符号链接文件】提供共享文件方法之一

【Socket文件】

**文件系统结构**

主目录~（登陆目录）echo $HOME /home/leafior

当工作目录 “.”目录本身；“..”父目录

绝对路径（/usr/src）相对路径（~/course…）

**标准目录**

/根目录：包含了所有的目录和文件。

/bin：也称二进制目录，包含了那些供系统管理员和普通用户使用的重要的Linux命令的可执行文件。目录/usr/bin下存放了大部分的用户命令。

/boot ：包括Linux内核的二进制映像。内核文件名是vmlinux加上版本和发布信息。

/dev：包含所有linux 系统中使用的外部设备。但是这里并不是放的外部设备的驱动程序。

/etc：存放了系统管理时要用到的各种配置文件和子目录。网络配置文件，文件系统，x系统配置文件，设备配置信息，设置用户信息等都在这个目录下。

/sbin：系统管理员的系统管理程序。

/home：/home/leafior

/lib：几乎所有的应用程序都会用到这个目录下的系统动态连接共享库。

/mnt ：这个目录主要用来临时装载文件系统mount

/opt：该目录用来安附加软件包

/proc ：进程和系统得信息，可以在这个目录下获取系统信息。这些信息是在内存中，由系统自己产生的。

/root ：根（root）用户的主目录

/sbin，/usr/sbin，/usr/root/sbin：存放了系统管理的工具、应用软件和通用的root用户权限的命令

/tmp：用来存放不同程序执行时产生的临时文件

/usr：存放了可以在不同主机间共享的只读数据。

/lost+found : 存放所有和其他目录没有关联的文件，这些文件可以用Linux工具fsck查找得到。

/var : 存放易变数据。/var/spool/mail 存放收到的电子邮件，/var/log 存放系统的日志，/var/ftp

**文件系统挂载**

**mount [-t fstype] [-o options] /dev/xxyN dirname**

如：mount –t vfat /dev/hda1 /mnt/c

类型 设备文件 挂在目录

挂接U盘：mount –t vfat /dev/sda1 /mnt/usb

fstype: iso9660/vfat/ntfs/msdos

/dev 保存所有设备文件的目录

xx：IDE硬盘为hd、SCSI硬盘和usb盘为sd，软盘fd

y：同种设备的顺序号，第一个硬盘为a

N：同一个设备编号，硬盘1-4为前面四个主分区，5开始为逻辑分区。

/dev/hda1（第一个硬盘的第一个分区）

/dev/hda5、/dev/hda6（逻辑分区）

/dev/fd0（软盘） /dev/hdc（光盘）

/dev/sda1（通常为移动硬盘或U盘的第一个分区）

mount –a：挂载在/etc/fstab中列出的所有设备

umount /dev/hda5 或umount /dosd卸载

**文件结构导航**

**echo**

-E不解释转义字符/-e解释(“\c”回车不换行)

-n不输出行尾换行符

**ls**

-r 将文件以相反次序显示(原定依英文字母次序)

-t 将文件依建立时间之先后次序列出

-i 显示inode号

-A 同-a ，但不列出“.”及“..”隐藏文件也来

-F 可执行文件则加“\*”, 目录则加“/”

-R 若目录下有文件，则以下的文件亦皆依序列出

ls –l [文件列表]

ls –ld [目录列表]

-rw-r--r-- 1 root root 44870 Dec 27 16:33 Makefile

(1) -普通文件，b块特殊文件，c字符特殊文件，d目录，l连接，p命名管道（FIFO）

所有者、组和其它用户的访问权限

(2)连接数 (3)所有者的登录名 (4)所有者的组名

(5)文件大小byte (6)(7)(8)修改时间 (9)文件名

ls ~/ [!0-9]\*.[c,C]ls [0-9][a-zA-Z].html #两个字符构成

**mkdir [options] dirnames**

-p 若上层目录目前尚未建立，则创建父目录

-m设置目录的权限。设置法与chmod 指令相同

$ mkdir –m 777 dir\_name

**rmdir [options] dirnames**

-p 删除指定目录之后，若该目录的上层目录已变成空目录，则将其一并删除

**touch** sample.h文件不存在，新建空；存在，修改时间

**file [options] file-list**：显示文件内容类型命令

**cat/** tac nl == cat -n

-E 每行尾显示符号$ -n 显示每一行的行号

$ cat 执行指令，不加任何参数

键入任何文字后，回车，系统回应一模一样的文字

$ cat file1 file2 > file3

file1和file2合成file3，若文件file3已经存在，覆盖；若用“>>”，新的内容就会附加在原有内容之后。

空格下一页，回车下一行，Q退出

cat sample

**pr**把文件分割成多页并在每页插入页眉，包含日期、当前时间、文件名和页号

df 命令可显示所有文件系统对i节点和磁盘块的使用情况。

该命令各个选项的含义如下：

-a 显示所有文件系统的磁盘使用情况，包括0块（block）的文件系统，如/proc文件系统。

-k 以k字节为单位显示。

-i显示i节点信息，而不是磁盘块。

-t 显示各指定类型的文件系统的磁盘空间使用情况。

-x 列出不是某一指定类型文件系统的磁盘空间使用情况（与t选项相反）。

-T 显示文件系统类型。

**more** 、pg 一次显示一个屏幕的内容

+/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 –c –10 file1 #显示文件file1的内容，每10行显示一次，而且在显示之前先清屏。

$ more sample phones

一次显示一屏幕的sample文件的内容，然后以同样的方式显示phones文件的内容

more –c -10 file1每10行显示一次，显示前先清屏

**less** 可上下翻页

-N 显示行号

-o file 当命令输入是管道的时候，除了显示屏，还把输出写到文件file里；若已经存在，会提示是否覆盖。

-p pattern 在文件中查找匹配“pattern”的第一处位置。

**head** 默认显示10行，-N控制显示开始的行

**tail** 默认显示10行，-f如果文件正在被追加，会继续显示追加的行，直到键入<Ctrl-C>。

-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

-r 逆序显示（最后一行先显示）

**cp**

-a 在备份中保持尽可能多的源文件结构和属性

-d 拷贝时保留链接

-f 删除已存在的目标文件不提示

-i 提示是否覆盖已存在的目标文件

-p 保持原先文件的所有者，组权限和时间标志

-r 递归复制目录，把所有非目录文件当普通文件复制

-R 递归复制目录

**mv**

mv [option] file1 file2 重命名file1为覅leor移动文件

mv [opion] file-list dirctory把文件列表中所有文件移到目录下

-i 在覆盖目标文件前提示用户 | -f 强制转移

**rm**

rm [option] file-list

-f 强制删除 | -i 删除前提示 | -r 递归删除

**wc** [options] file-list

显示文件大小，有行数、单词数和字符数（字节数）

-c 只显示字符数 -l 只显示行数 -w 只显示单词数

**diff** [options] [file1] [file2]

如果用-代替file1或file2，将从标准输入读取输入

-b 忽略行尾的空白

-e 生成并且显示一个脚本，编辑器ed可以用它来把文件file1转化成file2

-h 快速比较（此时不能使用-e选项）

若使用diff的时候不带任何选项，它会生成一系列指令，如果两个文件不同的话就可以使用这些指令把文件file1转化成file2：

L1aL2,L3：在F1第L1行后添加F2的L2行到L3行

L1,L2cL3,L4：把file1中的L1行到L2行替换成file2中的L3行到L4行

L1,L2dL3：删除file1中的L1行到L2行

**zdiff和zcmp**可以用来比较两个压缩文件

**uniq** [options] [+N][input-file] [output-file]

删除已经排序好的文件input-file中的所有重复行， 并把处理后不重复的行输出到output-file中

如果未指定output-file，输出到标准输出设备上

如果未指定input-file，从标准输入设备中输入

-c 在每行之前显示他们出现的次数

-d 只显示重复行 -u 只显示未重复行

**lpr** [options] file-list 打印

-# N 把file-list中的文件打印N份；默认只打印一份

-P ptr 把打印请求提交给打印机ptr

-T title 在标题页上打印标题title

-m 打印结束后发送电子邮件

-p 用命令pr来格式化输出

lpr提交一个需要打印的文件

lpq显示打印任务的状态

lprm从打印队列里清楚一个或多个打印任务

lpc激活打印控制程序

lptest生成测试打印机的波纹图案

**正则表达式**

支持工具：awk, ed, egrep, grep, sed, vi

x|y|z x或y或z

/L..e/ Love, Live, Lose, …

^x 以x开始的string

x$ 以x结束的string

\\* \*

(xy)+或\(xy\)+ xy,xyxy,xyxyxy, …

xy? x, xy

xy\* x, xy, xyy, xyyy

xy+ xy, xyy, xyyy, ..

[], [^] /[Hh]ello/ /[^A-KM-Z]ove/ Love

{n} 匹配n次

{n,} 匹配n或n+次

{n,m} >=n, <=m

**tar [options][filename-list]**

-c 建立新的备份文件。

-r 将文件附加在备份文件后面。

-f archname 用archname作为存档或恢复文件的备份文件名；默认是/dev/mto。如果archname是-，从标准输入读（对解压文件），或写到标准输出（对建立档案文件），这是当tar用作管道时的一个特性。

-t 以类似ls –l格式列出磁带上的内容（备份在磁带上的文件名）。

-u 将把比备份文件中更新的文件加入到备份文件中。

-x 从备份文件中取出文件。

-z 在tar创建备份文件时，使用gzip命令对它进行压缩；而从备份文件提取文件时，用gzip命令来解压备份文件。

-v 详细显示文件处理过程，用x选项解压文件的过程或存档文件的过程。

$ tar -cvf bash.help.tar \*.help

$ tar -zxvf linux-2.5.15.tar.gz

**rpm [options] [rpm-filename]**

**-v** 显示安装过程的详细信息。

**-h** 显示安装进度。

-a 查询所有安装的软件包。

-f filename 查询指定文件名的软件包。

-p rpm-filename 查询指定的软件包。

-d 只有列出文件。

**-i** 显示软件包信息，包含名称、版本和描述。

-l 列出软件包的文件。

-R 列出相关的软件包。

-s 显示软件包内文件的状态。

**-U** rpm-filename 升级指定的软件包。

-q 使用交互模式。

**-e** rpm-filename 删除指定的软件包。

-F rpm-filename 更新指定的软件包。

-i rpm-filename 安装指定的套件。

**sort [选项] 文件列表**

-b 忽略前导的空白

-d 按字典顺序排序，比较时仅字母、数字、空格和制表符有意义

-f 将小写字母与大写字母同等对待

-k pos1,pos2 指定一个或几个**字段**作为排序关键字，字段位置从pos1开始，到pos2为止。如不指定pos2，则关键字为从pos1到行尾。字段,字符的位置从1开始。

-r 按逆序输出排序结果

-c 检查给定文件是否已排好序，如果它们没有都排好序，则打印一个出错信息，并以状态值1退出。

-o 输出文件将排序输出写到输出文件中而不是标准输出，如果输出文件是输入文件之一，sort先将该文件的内容写入一个临时文件，然后再排序和写输出结果。

Sort –k 2 students 姓氏第2个字段开始作为比较键值

Sort –k 4 –r –b students用电户作为比较键值，忽略行首的空格和tab，逆序排列结果

**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

**find [目录列表][表达式]**

-exec CMD 如果命令(CMD)返回0（返回值为真），则该文件符合要求；命令必须以\; 结束

-ok CMD 和-exec相同，执行命令需要确认

-inum N 搜索inode为N的文件

-links N 搜索有N个链接的文件

-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

-perm oct 权限等于oct(八进制数字，如777)的文件

-print 显示符合要求的文件路径和文件名

-size ±N[c] 搜索文件大小为N块。字符c用来确定块的大小，默认为512个字节。+N表示大小超过N块的，-N就是小于N 块的

-user name搜索所有权为name的文件

\( expr \) 当表达式为真结果为真；表达式可用OR和AND组合

! expr 取反，当表达式为假时结果为真

得到一个文件的所有硬链接：

$ find /usr . –inum 258072 –print

括号用来标明需要匹配的表达式，在\(和-o前后必须要有空格。这个命令没有提示直接删除匹配的文件；如果想要提示，用参数-ok替换-exec。

$ find . \( -name core –o –name ‘\*.ps’ –o –name ‘\*.p’ \) –print –exec rm {} \;

**whereis [options] [file-list]**查明系统上是否存在特定一个目录，若存在，给出路径whereis –b cat

-b 只搜索可执行文件

-m 只搜索帮助文件

-s 只搜索源代码

**grep [选项]模式[文件列表] 中间**

**egrep[选项] [字符串] [文件列表]** 最慢，最灵活

**fgrep[选项] [表达式] [文件列表]** 最快,限制多

如果没有文件列表，则从标准输入读入数据

-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，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 '[a-z]\{8\}' students

包含至少连续8个小写字母单词的行

$ egrep “^J|^K” students

**权限**

对于目录：

r :列出目录的内容 w：建立，删除

x :允许用户搜索这个目录，如果你没有对目录的执行特权，那么就不能使用ls –l命令来列出目录下的内容或者是使用cd命令来把该目录变成当前目录。

usr(u) / group(g) / others(o)

1 bit表示每一种权限，用户有8种可能的操作权限

3 bits表示某一用户的权限；3种用户，9 bits

**chmod**参数只有1或2位8进制数，按从右到左匹配

chmod [option] octal-mode file-list

chmod [option] symbolic-mode file-list

-R 递归修改or设置文件、目录及其子目录的访问权限

-f 强制改变文件访问特权，若不是文件的拥有者，得不到任何错误信息

Chmod u=rwx courses

Chmod 740 courses

**umask [mask]** :新创建的文件或目录的访问特权都将设置为1，除了在参数mask中设置为1的对应位。

文件访问权限 = 默认的访问权限–mask

Umask 013 对于一个新建的可执行文件764

**默认访问权限:** 执行文件为777 文本文件为666

**通过文件连接共享**

**ln [options] existing-file new-file**

**ln [options] existing-file-list directory**

-f 强迫建立链接

-n 如果“new-file”已存在，不创建链接。

-s 建立一个符号链接而不是硬链接

-d 建立目录的硬链接

**硬链接**：一个指向文件索引节点的指针，inode相同；不可跨越文件系统；只有超级用户才可以建立目录硬链接；不占用空间(极少)

**软链接**：可跨越文件系统，甚至跨越网络(NFS)；如果链接指向的文件从一个目录移动到另一个目录，就无法通过符号链接访问它；占有少量空间，存inode

**进程**

进程创建和终止是LINUX系统处理外部命令所采用的唯一机制。

内部命令：代码本身就是shell进程的一部分。.、alias、bg、cd、continue、echo、exec、exit、fg、jobs、pwd 、set、shift、test、time、umask 、unset和wait

外部命令：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

Shell执行二进制文件：

1.Shell使用fork创建子进程；2.子进程执行exec，用命令对应的可执行文件覆盖自身；3.命令执行，bash等待命令结束。

Shell执行脚本文件：

创建一个子shell并让子shell依次执行脚本中命令，执行与从键盘输入的命令采用相同的方式

子shell为每一个要执行的命令创建一个子进程。

子shell执行脚本文件中的命令时，父shell等待子shell 结束。子shell遇到脚本文件的EOF终止。

子shell终止，父shell结束等待状态，开始重新执行。

**ps**

-a 显示所有终端上执行的进程信息，包括其他用户

-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。

-p 根据进程ID显示对应的信息。

-u ulist 显示在ulist列表中有对应的UID或者名称的用户的进程信息(UID或者用户名由逗号分开)。

-t tlist 选取列在tlist中的终端上的进程；如果没有tlist，显示不带参数ps命令执行的结果。

命令ps aux参看所有进程，包括守护进程

**top**：实时监视CPU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

后台/作业

$cmd &

[作业号]进程PID——作业是一个不运行于前台的进程，并且只能在关联的终端上访问。这样的进程通常在后台执 行或者成为被挂起的进程。

**fg[%jobid]** 后台🡪前台

%|%+ 当前的作业 %- 以前的作业 %N作业号为N

%Name开头名字为Name %?Name 命令中含Name

**<Ctrl-Z>** 挂起前台进程

**bg** 挂起进程🡪后台，参数同fg（后台🡺前台）

**jobs** 显示所有挂起/停止的和后台进程的作业号

**suspend** 可以挂起当前shell进程

**守护进程(Daemons)**：运行于后台的系统进程。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)

**;顺序执行 &并发执行 |重定向**

最后一个&和之前的&间所有命令用一个进程完成。

$ date & who; whoami ; uname; echo Hello, World! &

date用一个进程执行，其他用另外一个进程执行

**命令组：** 命令组中的所有命令都在一个进程中执行（在当前shell的子shell中）

$ (date;echo Hello,World! )

Ctrl+C 终止前台进程

**命令的有条件执行**

cmd1**&&**cmd2：1成功则2 cmd1 **| |** cmd2：1失败则2

**kill [-signal\_number] proc-list kill –l**

发送signal\_number信号到PID或者jobID在proc-list中的进程；jobID必须以“%”号开始。

kill –l返回所有信号的号码以及名字的列表

1 挂断:退出系统,用调制解调器使用系统时挂断电话

2 中断<Ctrl-C> 3 退出<Ctrl-\>

9 强制终止 15 终止进程（默认的信号号码）

为了终止一个忽略15号信号或者其它信号的进程，需要使用9号信号，即强制终止信号

进程号0可以指代所有在当前登录期间创建的进程。kill –9 0可以终止所有登录时产生的进程（即当前会话中的所有进程），这样，你就不得不退出系统。

**ps –e f或pstree**：用图的形式显示当前系统中执行进程的进程树——$ pstree –a | more

**ulimit –u：**Bash下显示用户可同时执行的最大进程数

**重定向**

command < input-file > output-file

command > output-file < input-file

>换成>>则追加文件，否则替换

**sdin— 0 stout — 1 sderr — 2**

cat lab1 lab2 1> output 2>&1

2> &1：使文件描述符2为文件描述符1 的拷贝，导致错误信息送往和该命令输出相同的地方

在命令行的解析中，文件的重定向顺序是从左到右

$ cat lab1 lab2 lab3 2>&1 1>output

标准出错先设置成显示器，标准输出才改为output

**tee [options] file-list**

从标准输入中得到输入，送到标准输出和file-list中

cmd1| tee file1…fileN|cmd2

cmd1标准输出作为tee的标准输入，tee输出送到文件file1到fileN中，同时作为cmd2的标准输入

**< &-** 关闭标准输入 **> &-** 关闭标准输出

**m<&或m>&-** 将文件描述符m关闭

<> file将标准输入和输出都分配给文件file

n <file将文件file设为文件描述符n

n> file将文件描述符n指向文件file

> &file标准输出和出错输出重定向到文件file

**Bash 编程**

**一． Bash特殊字符**  
**1． 通配符：**  
\*：匹配任何字符串  | ?：匹配任何单个字符 | 集合运算符：用一些单个字、一个连续范围或断续的字符集合作为通配符  [set]：用字符集合作通配符匹配单个字符，如：[aeiou]，[a-o]，[a-h, w-z]  [!set]：除了集合外的所有字符组成的集合作通配符   
**2． 花括号展开式（可以嵌套）：**  
格式：[前导字符串]{字符串1[{嵌套字符串1…}] [, 字符传2…]}[后继字符串]

如：c{a{r, t, n}, b{r, t, n}}s就等于  cars cats cans cbrs cbts cbns

**3． 其它特殊字符：**  
< ：输入重定向   
>; ：输出重定向(没有文件则创建，有则覆盖)   
>;>; ：输出重定向(没有则创建，有则追加到文件尾部)   
| ：管道  \ ：引用后面的单个字符   
‘ ：强引用字符串，不解释特殊字符   
“ ：弱引用字符串，解释所有特殊字符   
~ ：根目录 | ` ：命令替换 | # ：行注释   
; ：命令分隔符（命令终止符），运行在一行里执行多条命令   
$ ：变量表达式  & ：在后台执行命令   
\* ：字符串通配符  ? ：单个字符通配符   
**二． Bash变量   
1． 自定义变量**  
用户自定义的变量由字母、数字和下划线组成，并且变量名的第一个字符不能为数字，且变量名大小写敏感。   
varname=value **注意bash不能在等号两侧留空格**  
shell语言是非类型的解释型语言，给一个变量赋值实际上就是定义了变量，而且可以赋不同类型的值。引用变量有两种方式，$varname和${varname}，为防止变量在字符串中产生歧义建议使用第二种方式，引用未定义的变量其值为空。   
declare [option][name[=value]] declare –i age=20

typeset [option][name[=value]]

-a 数组 -f 函数 -i 整数 -r 变量 –x 全局变量  
**2． 环境变量**  
可以用set命令给变量赋值或查看环境变量值，使用unset命令清除变量值，使用export导出变量将可以使其它进程访问到该环境变量。   
**3． 位置变量**  
位置变量对应于命令行参数，其中$0为脚本名称，$1为第一个参数，依次类推，参数超过9个必须使用${}引用变量。shell保留这些变量，不允许用户以另外的方式定义它们，传给脚本或函数的位置变量是局部和只读的，而其余变量为全局的（可以用local关键字声明为局部）。   
**4． 其它变量**

$?：保存前一个命令的返回码 $-：在Shell启动或使用set命令时提供选项 $$：当前shell的进程号 $!：上一个子进程的进程号

$#：传给脚本或函数的参数个数，即位置变量数减1，不含脚本名称。  
$\* ：传给脚本或函数的参数组成的单个字符串，即除脚本名称后从第一个参数开始的字符串，每个参数以$IFS分隔（一般内部域分隔符$IFS为1空格）。形同”…”   
$@ ：传给脚本或函数的参数列表，这些参数被表示为多个字符串。形同”” “” “”…。$\*和$@之间的不同方便使用两种方法处理命令行参数，但是在打印时参数外观没有区别。

**5. 数组** 声明一个数组：declare -a array

(1) array=(var1 var2 var3 ... varN)

(2) array=([0]=var1 [1]=var2 [2]=var3 ... [n]=varN)

(3) array[0]=var1 arrya[1]=var2 array[n]=varN

计算数组元素个数：

${#array[@]} 或者 ${#array[\*]}

BASH的特殊参数 @ 和 \* 都表示“扩展位置参数，从1开始”，但形式稍有差异，但在数组里使用好像是可以通用的。

引用数组：echo ${array[n]}

遍历数组：filename=(`ls`)

for var in ${filename[@]};do

echo $var

done

**三． Bash操作符**  
**1． 字符串操作符（替换操作符）**

$var ${var}如果var存在且不为空，返回它的值，否则返回null   
${var:-word} 如果var存在且不为空，返回它的值，否则返回word   
${var:=word} 如果var存在且不为空，返回它的值，否则将word赋给var， 返回它的值   
${var:+word} 如果var存在且不为空，返回word，否则返回空   
${var:?message} 如果var存在且不为空，返回它的值，   
否则显示“bash2:$var:$message”，然后退出当前命令或脚本   
${var:offset[]} 从offset位置开始返回var的一个长为length的子串， 若没有length，则默认到var串末尾

$(var) 替换这个命令为输出的结果  
**2． 模式匹配操作符**  
${var#pattern} 从var头部开始，删除和pattern匹配的最短模式串，然后返回 剩余串   
${var##pattern} 从var头部开始，删除和pattern匹配的最长模式串，然后返回 剩余串，basename path＝${path##\*/}   
${var%pattern} 从var尾部开始，删除和pattern匹配的最短模式串，然后返回 剩余串，dirname path＝${path%/\*}   
${var%%pattern} 从var尾部开始，删除和pattern匹配的最长模式串，然后返回 剩余串   
${var/pattern/string} 用string替换var中和pattern匹配的最长模式串   
**3． 文件测试操作符**  
-d file file存在并且是一个目录  -e file file存在   
-f file file存在并且是一个普通文件   
-g file file存在并且是SGID(设置组ID)文件 -r file 对file有读权限   
-s file file存在并且不为空 -w file 对file有写权限   
-u file file存在并且是SUID(设置用户ID)文件   
-x file 对file有执行权限，如果是目录则有查找权限   
-O file 拥有file -L file file为符号链接   
-G file 测试是否是file所属组的一个成员   
file1 –nt file2 file1比file2新 file1 –ot file2 file1比file2旧   
**4． 字符串操作符**  
-n str str的长度大于0（不为空） -z str str的长度为0（空串）

另一种逻辑操作符 逻辑与expr1 –a expr2 逻辑或expr1 –o expr2

**5.数值处理**

let expression 或（（expression））计算表达式

expression是一个包含项和操作符的表达式,项可以是一个变量或是一个整数常数

2． 确定性循环：for   
for value in list  
do  
 command-list  
done

expr args

计算参数args的值

**五． Shell流控制**  
**1． 条件语句**：if   
if exp1

then then-commands

**4． 选择结构：case和select**  
case variable in

pattern1) command1 ;;

pattern2) command2 ;;

pattern3) command3 ;;

esac

elif exp2

elif1-commands

else

else-commands

fi  
  
**3． 不确定性循环：**

while 条件/ until 条件   
do  
 语句  
done

**5． 命令shift**  
将存放在位置变量中的命令行参数依次向左传递   
shift n 命令行参数向左传递n个串   
**六． Shell函数**  
定义： function fname ( )  {  commands commands }    
调用：fname [ parm1 parm2 parm3 ... ]   
说明： 函数在使用前定义，两种定义功能相同   
函数名和调用函数参数成为函数的位置变量   
函数中的变量应该使用local声明为局部变量   
**七． 输入输出**  
**1．I/O重定向**  
< ：输入重定向   
>; ：输出重定向(没有文件则创建，有则覆盖)   
>;>; ：输出重定向(没有则创建，有则追加到文件尾部)   
<< ：输入重定向(here文档)   
格式： command << label   
**2．字符串I/O操作**  
字符串输出：**echo**   
命令选项： -e：启动转义序列 -n：取消输出后换行   
转义序列： \a：Alt/Ctrl+G(bell) \b：退格Backspace/Ctrl+H   
\c：取消输出后换行 \f：Formfeed/Ctrl+J   
\r：Return/Ctrl+M \v：Vertical tab   
\n：八进制ASCII字符 \\：单个\字符 \t：Tab制表符   
字符串输入：**read**  
可以用于用户交互输入，也可以用来一次处理文本文件中的一行   
命令选项： -a：将值读入数组，数组下标从0开始   
-e：使用GNU的readline库进行读入，允许bash的编辑功能   
-p：在执行读入前打印提示

**概要**

最为重要的决策之一是采用 GPL（GNU General Public License）。设计Linux三原则（实用|有限目标|简单设计）

内核版本的序号：major(主版本号).minor(次版本号).patchlevel（对当前版本的修订次数）

Linux系统结构(计算机硬件|linux内核|shell|应用层|用户)

**GNU C的扩充**

吸收了C++中的inline和const. 为了支持64位CPU，增加了新的基本数据类型long long int.

分支声明: 使用likely()和unlikely()宏对条件选择进行优化

if(foo){…}

当foo大多数时间都会为1时：if(likely(foo)){…}

当foo大多数时间都会为0时if(unlikely(foo)){…}

许多C语言支持属性描述符，如“aligned”，“packed”等。由于这些在ANSI C中不是保留字，所以可能引起冲突。GNU C支持在前后加上“\_\_”来区分。“\_\_inline\_\_”等于保留字“inline”。

.a .so .sa库文件a:传统的静态函数库；so代表共享函数库

gcc –I/usr/openwin/include power.c包含保存在子目录或非标准位置中的include文件

GCC编译器缺省的头文件目录是/usr/include目录及其子目录下。

数学库libm.a连接到power.o

gcc –o power power.o -lm文件名中“lib”以后，扩展名以前的部分 gcc –o power power.o /usr/lib/libm.a

**Linux中的汇编语言**

寄存器名要加上“%”作为前缀;小写字母;立即数要加上“$”作为前缀; gas -o hello.o hello.s

make工具，与之有关的makefile or Makefile文件

gdb 功能强大调式器

**fork（）函数**

fork:创建一个新子进程

#include <sys/types.h>#include <unistd.h>pid\_t fork(void);

返回值：调用一次，返回两次。子进程的返回值是0，父进程的返回值则是子进程的进程ID,出错为-1

if ( (pid=fork()) < 0) { /\* error handling \*/}

else if (pid == 0) { /\* child \*/}

else { /\* parent \*/};

用fork函数创建子进程后，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，该进程完全由新程序代换，而新程序则从其main函数开始执行。调用exec并不创建新进程，所以前后的进程PID并未改变。exec只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

子进程调用g e t p p i d以获得其父进程的进程I D

子进程和父进程共享很多资源，除了打开文件之外，很多父进程的其他性质也由子进程继承：如实际用户I D、实际组I D、有效用户I D、有效组I D等。父、子进程之间的区别包括fork的返回值，进程I D，不同的父进程I D，父进程设置的锁，子进程不继承等。

Linux实现**进程间通信**(IPC Inter Process Communication)方法有：System V IPC机制（ 信号量， 消息队列，共享内存）；管道（pipe）、命名管道；套接字（socket）；信号( signal )

**管道**是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称为pipe文件。向管道（共享文件）提供输入的发送进程（即写进程），而接收管道输出的接收进程（即读进程）可从管道中接收数据。管道通信是基于文件系统形式的一种通信方式.

管道分为无名管道和有名管道两种类型。无名管道是一个只存在于打开文件机构中的一个临时文件，从结构上没有文件路径名，不占用文件目录项。无名管道利用系统调用pipe(filedes)创建.

pipe系统调用: int filedes [2]；int pipe (filedes)；

**Linux线程**

Linux 2.6内核支持clone()系统调用创建线程

pthread\_create()：创建线程函数；

pthread\_exit()：主动退出线程；

pthread\_join()：用于将当前线程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。

pthread\_cancel()：终止另一个线程的执行。

Linux是一个单内核，Linux内核运行在单独的内核地址空间.Linux吸取了微内核的精华：其引以为豪的是模块化设计、抢占式内核、支持内核线程以及动态装载和卸载内核模块。

**GNU C Library （glibc）**提供了连接内核的系统调用接口，还提供了在用户空间应用程序和内核之间进行转换的机制。内核和用户空间的应用程序使用的是不同的保护地址空间。每个用户空间的进程都使用自己的虚拟地址空间，而内核则占用单独的地址空间

**Linux内核源程序安装在/usr/src/linux**

arch：该子目录包括了所有和体系结构相关的内核代码。

Include: 该子包括编译内核所需要的大部分头文件。

init：该子目录包含内核的初始化代码，包含两个文件main.c和version.c。

mm：该子目录包括所有独立于cpu体系结构的内存管理代码，如页式存储管理内存的分配和释放等；而和体系结构相关的内存管理代码则位于arch/\*/mm/，例如arch/i386/mm/fault.c

kernel：主要的内核代码，实现了大多数内核函数， sched.c；

drivers：放置系统所有的设备驱动程序; lib：放置内核的库代码。

net：内核与网络相关的代码。

ipc：这个目录包含内核的进程间通讯的代码。

fs：所有的文件系统代码和各种类型的文件操作代码，它的每一个子目录支持一个文件系统，例如fat和ext2;

scripts：此目录包含用于编译内核的脚本文件等。

一个Kconfig文件和一个Makefile文件，这两个文件都是编译时使用的辅助文件.

Linux系统引导过程使用内核镜像，/boot目录下文件名称如：vmlinuz-2.6.15.5：普通内核镜像：zImage（Image compressed with gzip），大小不能超过512k. 大内核镜像：bzImage（big Image compressed with gzip），包含了大部分系统核心组件：系统初始化、进程调度、内核管理模块

#sudo apt-get install libncurses5-dev 若无ncurses库安装之

#su //输入密码，用户权限改为root权限。或用sudo命令

#mv linux-2.6.28.tar.gz /usr/src //把内核代码文件移到相应的目录。也可以在自己的主目录下对内核进行修改。

# cd /usr/src

# tar zxvf linux-2.6.28.tar.gz //解压内核包，生成的内核源代码放在linux.2.6.28目录中

# cd linux-2.6.28

//使用系统的原配置文件，在/boot目录下的以config文件名开头的文件：

# cp /boot/config-`uname -r` .config

# make menuconfig

编译内核# make –j4

编译和安装内核模块# make modules\_install //模块安装

安装内核# make install //生成linux启动

Sudo mkinitramfs -o /boot/initrd.img-2.6.36

Sudo update-initramfs -c -k 2.6.36

Sudo update-grub2 //自动修改系统引导配置，产生grub.cfg启动文件

**系统调用**

#include <linux/unistd.h>

系统调用在用户空间进程和硬件设备之间添加了一个中间层。应用程序调用操作系统提供的功能模块（函数）。

用户程序通过系统调用从用户态（user mode）切换到核心态（kernel mode），从而可以访问相应的资源。

**运行模式（mode）**

Linux使用了其中的两个：特权级0和特权级3 ，即内核模式(kernel mode)和用户模式(user mode）

**地址空间（space）**

每个进程的虚拟地址空间：用户空间和内核空间。在用户态下只能访问用户空间；核心态下，均可访问

内核空间在每个进程的虚拟地址空间中都是固定的（虚拟地址为3G～4G的地址空间）。

**上下文（context）**。三个部分：

用户级上下文：正文、数据、用户栈以及共享存储区；

寄存器上下文：通用寄存器、程序寄存器（IP）、处理机状态寄存器（EFLAGS）、栈指针（ESP）；

系统级上下文：进程控制块task\_struct、内存管理信息(mm\_struct、vm\_area\_struct、 pgd、pmd、pte等)、核心栈等。

strace ls 查看操作系统命令所调用的系统调用

**内核函数**在形式上与普通函数一样，但它是在内核实现的，系统调用是用户进程进入内核的接口层，它本身并非内核函数，但它是由内核函数实现的。进入内核后，不同的系统调用会找到各自对应的内核函数，这些内核函数被称为系统调用的“服务例程”。当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数。

**内核初始化期间**调用trap\_init()函数建立IDT表中128(0x80)号向量对应的表项：set\_system\_gate(0x80， &system\_call);

system\_call()函数实现了系统调用中断处理程序 ：

它首先把系统调用号和该异常处理程序用到的所有CPU寄存器保存到相应的栈中， SAVE\_ALL

把当前进程task\_struct （thread\_info）结构的地址存放在ebx中

对用户态进程传递来的系统调用号进行有效性检查。若调用号大于或等于NR\_syscalls，系统调用处理程序终止。 （sys\_call\_table）

若系统调用号无效，函数就把-ENOSYS值存放在栈中eax寄存器所在的单元，再跳到ret\_from\_sys\_call()

根据eax中所包含的系统调用号调用对应的特定服务例程，服务例程的返回值必须写到eax寄存器中（每个系统调用至少有一个参数，即通过eax寄存器传递来的系统调用号）

程序执行系统调用步骤：

1、程序调用libc库的封装函数。

2、调用软中断 int 0x80 进入内核。

3、在内核中首先执行system\_call函数，接着根据系统调用号在系统调用表中查找到对应的系统调用服务例程 。

4、执行该服务例程。

5、执行完毕后，转入ret\_from\_sys\_call例程，从系统调用返回

**System Calls:**

(1). **int read(int fd, char \*buf, int n);** 读文件

(2). **int write(int fd, char \*buf, int n);**  写文件

(3). **int open(char \*name, int rwmode);** 打开文件

rwmode: 0 read 1 write 2 w/r

(4). **int creat(char \*name, int perms);**  创建文件

perms: rwx 三位八进制数

(5).**int lseek(int fd, long offset, int origin)** 在文件中定位

origin: 0 文件开始 1 当前位置 2 文件结束

(从当前读写位置移动到offset处, offset是相对origin计算得出的)

定位至文件结尾处: lseek(fd, 0L, 2);

定位至文件开始处: lseek(fd, 0L, 0);

取得当前位置: pos = lseek(fd, 0L, 1);

(6).创建低级进程: 新进程会覆盖老进程（几个函数的不同之处主要在于命令传递方式）

**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 agrv[]);**

(7).**int dup(int fd)**

在最低序号的未分配的文件描述符上复制文件描述符fd, 返回指向相同打开文件的一个新的文件描述符.

**int dup2(int oldfd, int newfd);** 让newfd和oldfd指向同一个文件描述符oldfd, 如果可能的话关闭newfd

(8**)pid\_t fork(void);** 创建一个子进程（Copy-On-Write）

**int clone(int (\*fn)(void \*), void \* child\_stack, int flags, void \* args);** 子进程可以使用父进程一些execution context

**pid\_t vfork(void);** 创建一个子进程并把父进程挂起

(9)**void exit(int status)** 结束进程

**pid\_t wait(int \*status)** 父进程挂起，直到其中一个子进程

(10).**int fcntl(int fd, int cmd);**

**int fcntl(int fd, int cmd, long arg);**

**int fcntl(int fd, int cmd, struct flock\* lock);**

在以文件描述符为fd的文件上执行cmd指定的命令

Cmd: F\_DUPFD, F\_GETFD, F\_SETFD

**mode\_t umask(mode\_t mode);** 把calling process的file mode设置为 mode & 0777

(11)**int chdir(const char\* path)** 把当前工作目录改为path指定的目录

**int fchdir(int fd);** directory是由fd指定的，其它跟chdir 类似

**int chmod（const char\* path, mode\_t mode）**由path指定的文件的模式改为mode

(12).**int mkdir(const char\* path, mode\_t mode)** 以mode为模式创建由path指定的目录

**int rmdir(const char\* path)** 移除由path指定的目录，该目录必须为空

**int rename(const char\* oldpath, const char\* newpath)** 改变文件的位置或名称

(13).**int link(const char\* oldpath, const char\* newpath)** 为已存在的文件创建一个链接，创建后两个文件名指向同一个文件，完全等同

**int symlink(const char\* oldpath, const char\* newpath)** 为oldpath指定的文件创建一个名为newpath的符号链接（可能存在，也可能不存在）

**int unlink(const char\* path)**删除path指定的文件，如果path指定的文件为该文件的最后一个链接，那么调用unlink后，文件被删除，而且空间被释放。如果还有进程在使用该文件，那么直到进程结束，文件才会被删除。

(14).文件系统I节点相关信息获得:

struct stat stbuf;

**stat(char \*name, &stbuf);** 文件由name指定

**fstat(fd, &stbuf);** 文件由fd指定

(15)**int gettimeofday(struct timeval\* tv, struct timezone \*tz);** 获取时间

**int settimeofday(const struct timeval\* tv, const struct timezone \*tz);** 设置时间

struct timeval

{

time\_t tv\_sec; //seconds

suseconds tv\_usec; //microseconds

};

Stcuct timezone

{

int tz\_minuteswest; //minutes west of Greenwich

int tz\_dsttime; //type of DST correction

};

**int stime(time\_t \* t);** 把系统时间设为t，t为从00：00：00 GMT 1970.1.1开始算的秒

**time\_t time(time\_t \* t)** 得到系统时间，返回值为从00:00:00 UTC 1970.1.1开始算的秒，如果t不为NULL，那么结果也将保存到t里

**clock times(struct tms\* buf);** 获取进程时间,把当前进程时间存在buf里

struct buf

{

clock\_t tms\_utime; //user time

clock\_t tms\_stime; //system time

clock\_t tms\_cutime; // user time of dead children

clock\_t tms\_cstime; //system time of dead children

};

(16)**int uname(struct utsname\* buf)**;获取当前内核的名称和相关状态

struct utsname

{

char sysname[];

char nodename[];

char release[];

char version[];

char machine[];

#ifdef \_GNU\_SOURCE

char domainname[];

#endif

};

**int brk(void \*end\_data\_segment);**把进程的data segment 的end的值设为end\_data\_segment， 以达到改变data segment 大小的目的。

**void \*mmap(void \* start, size\_t length, int prot, int flags, int fd, off\_t offset);**

将由fd指定的文件（或设备）的offset处开始的length 字节映射到内存，start指定希望映射后的内存的起始地址，但不一定是，所以一般可以设为0, 而映射后的真正起始地址将由mmap()函数返回。Prot指定贴身后内存的保护级别。

**int munmap(void\* start, size\_t length);** 取消文件（设备）的内存映射

**uid\_t getuid()** 获取当前进程real user的ID

**uid\_t geteuid()** 获取当前进程effective user的ID

**int setuid(uid\_t uid)** 设置当前进程effective user 的ID

(17). **typedef void (\*sighandler\_t)(int);**

**sighandler signal(int signum, sighandler\_t handler);** 为由signum指定的信号添加新的处理该信号的函数

**int kill(pid\_t pid, int sig)** 向由pid指定的进程发由sig指定的信号

**int pipe(int filedes[2])** 为管道创建一对文件描述符存在filedes[0]和filedes[1]中，其中filesdes[0]是用来读的，filedes[1]是用来写的

**exec（）函数**

在Linux系统中，使程序执行的唯一方法是使用系统调用exec()。exec函数族把当前进程映像替换成新的程序文件，而且该程序通常main函数开始执行。

exec指的是一组函数，一共有6个，分别是：

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[]);

其中只有execve是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

exec函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，就是在调用进程内部执行一个可执行文件。

参数argc指出了运行该程序时命令行参数的个数，

数组argv存放了所有的命令行参数，

数组envp存放了所有的环境变量。如PATH，HOME

execv开头的函数是以"char \*argv[]"这样的形式传递命令行参数，而execl开头的函数采用了把参数一个一个列出来，然后以一个NULL（作用和argv数组里的一样）表示结束。

在全部6个函数中，只有execle和execve使用了char \*envp[]传递环境变量，其它的4个函数都没有这个参数，这4个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而execle和execve会用指定的环境变量去替代默认的那些。

除execlp和 execvp之外的4个函数都要求，它们的第1个参数path必须是一个完整的路径，如"/bin/ls"；而execlp和execvp的第1个参数 file可以简单到仅仅是一个文件名，如 "ls"，这两个函数可以自动到环境变量PATH制定的目录里去寻找。

Exec

sys\_execve

ebx中拿到filename地址

ecx中拿到参数地址

edx中拿到环境变量地址

do\_execve()

分配可执行文件信息结构bprm=kmalloc()

检查对传入的filename的访问权限

填充bprm结构

拷贝父进程的ldt

计算参数个数并比较传入参数与所需参数

计算环境变量个数并比较传入环境变量与所需环境变量

进一步检查文件是否可以被执行

分配临时的bprm内存页

拷贝文件名等信息到内存页中

调用search\_binary\_handler执行新的bprm

load\_linary()进行加载

清理资源

结束do\_execve

结束sys\_execve

Kill

sys\_exit

do\_exit

标记退出

释放各种空间与资源

通知相应进程将要退出

主动放弃cpu

（正常结束的话程序到此为止没有返回）

**wait（）函数**原型为:pid\_t wait(int \*status)

　　当进程退出时,它向父进程发送一个SIGCHLD信号,默认情况下总是忽略SIGCHLD信号,此时进程状态一直保留在内存中,直到父进程使用wait函数收集状态信息,才会清空这些信息。用wait来等待一个子进程终止运行称为回收进程。　　wait()要与fork()配套出现,如果在使用fork()之前调用wait(),wait()的返回值则为-1,正常情况下wait()的返回值为子进程的PID。如果先终止父进程,子进程将继续正常进行，只是它将由init进程(PID 1)继承,当子进程终止时,init进程捕获这个状态。当父进程忘了用wait()函数等待已终止的子进程时,子进程就会进入一种无父进程清理自己尸体的状态,此时的子进程就是僵尸进程.如子进程虽然执行完毕,但父进程没有调用wait(),出现子进程虽然死亡,而不能在内核中清理尸体的情况.父进程用wait()会回收掉尸体.

　　如果参数的值不是NULL,wait就会把子进程退出时的状态取出并存入其中,这是一个整数值(int),指出了子进程是正常退出还是被非正常结束的,

　　由于这些信息被存放在一个整数的不同二进制位中,所以就设计了一套专门的宏来完成这项工作，其中最常用的两个：

　　1,WIFEXITED(status)这个宏用来指出子进程是否为正常退出的,如果是,它会返回一个非零值.

　　2,WEXITSTATUS(status)当WIFEXITED返回非零值时,我们可以用这个宏来提取子进程的返回值,如果子进程调用exit(5)退出,WEXITSTATUS(status)就会返回5;

　　如果子进程调用exit(7),WEXITSTATUS(status)就会返回7.请注意,如果进程不是正常退出的,也就是说,WIFEXITED返回0,这个值就毫无意义.

**exit（）命令**

用于退出当前shell，在shell脚本中可以终止当前脚本执行。

格式：exit n--Cause the shell to exit with a status of n.

格式：exit--即为最后一个命令的退出码。

格式：$?--上一个命令的退出码。

格式：trap "commands" EXIT--退出时执行commands指定的命令。（ A trap on EXIT is executed before the shell terminates.）

退出码（exit status，或exit code）的约定：

0表示成功（Zero - Success）

非0表示失败（Non-Zero - Failure）

2表示用法不当（Incorrect Usage）

127表示命令没有找到（Command Not Found）

126表示不是可执行的（Not an executable）

>=128 信号产生

**进程管理**

Linux没有真正意义上的线程概念。但通过clone()系统调用支持轻量级进程(线程);还支持内核线程，内核线程永远在核心态运行，没有用户空间。

**Linux进程的组成**

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)。

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。

Text:存放着进程要执行的指令代码，只读。User segment: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(包括进程控制块PCB)

task\_struct结构中定义：

volatile long state ; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

long exit\_state;表征进程退出时候的状态。

**进程状态**

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，父进程已经调用了wait4()或者waitpid()。

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的奖励或者惩罚。



**Linux进程控制块**PCB--task\_struct

进程的task\_struct是进程存在的唯一标志。task\_struct容纳了一个进程的所有信息，是系统对进程进行管理和控制的有效手段，是系统实现进程调度的主要依据。

进程创建时，系统就为其建立一个task\_struct进程控制块。运行结束时，撤消该进程的task\_struct 。

Linux在内存空间中开辟了一个专门的区域存放所有进程的task\_struct 。系统中的最多进程数目受task 数组大小的限制，缺省值一般为512。

**task\_struct结构**（include/linux/sched.h文件）：

include/linux/sched.h, line 701

701 struct task\_struct {

702 volatile long state;

/\* -1 unrunnable, 0 runnable, >0 stopped \*/

703 struct thread\_info \*thread\_info;

704 atomic\_t usage;

705 unsigned long flags;

/\* per process flags, defined below \*/

...

713 int prio, static\_prio;

714 struct list\_head run\_list;

715 prio\_array\_t \*array;

...

719 unsigned long sleep\_avg;

720 unsigned long long timestamp, last\_ran;

721 unsigned long long sched\_time;

/\* sched\_clock time spent running \*/

722 int activated;

723

724 unsigned long policy;

……

}

1. 进程的状态和标志

volatile long state //进程的状态

unsigned long flags //进程的标志

这些标志的含义分别为：

PF\_ALIGNWARN 标志打印“对齐”警告信息。

PF\_STARTING 进程正被创建。

PF\_EXITING 标志进程开始关闭。

PF\_DEAD 标志进程已经完成退出。

PF\_FORKNOEXEC 进程刚创建，但还没执行。

PF\_SUPERPRIV 超级用户特权标志。

PF\_DUMPCORE 标志进程是否清空core文件。

PF\_SIGNALED 标志进程被信号杀出。

PF\_MEMALLOC 进程分配内存标志。

PF\_FLUSHER 负责磁盘写回。

PF\_USED\_MATH 若没有置位，则使用fpu之前必须初始化。

PF\_FREEZE 由于系统要进入休眠，进程正在被停止。

PF\_NOFREEZE 系统睡眠的时候，这个进程不能被停止。

PF\_FROZEN 系统要进入睡眠，进程被停止。

PF\_FSTRANS 在一个文件系统事务之中。

PF\_KSWAPD kswapd内核线程。

PF\_SWAPOFF 在换出页的过程中。

PF\_LESS\_THROTTLE 尽可能少把我换出。

PF\_SYNCWRITE 负责把脏页写回。

PF\_BORROWED\_MM 内核线程借用进程的mm。

PF\_RANDOMIZE 随机虚拟地址空间。

PF\_SWAPWRITE 允许被写到swap中去。

2.进程的标识

int pid //进程标识号

unsigned short uid, gid //用户标识号，组标识号

unsigned short euid, egid //用户有效标识号，组有效标识号

unsigned short suid, sgid //用户备份标识号，组备份标识号

unsigned short fsuid, fsgid //用户文件标识号，组文件标识号

3.进程的族亲关系

struct task\_struct \*p\_opptr //指向祖先进程PCB的指针

struct task\_struct \*p\_pptr //指向父进程PCB的指针

struct task\_struct \*p\_cptr //指向子进程PCB的指针

struct task\_struct \*p\_ysptr //指向弟进程PCB的指针

struct task\_struct \*p\_osptr //指向兄进程PCB的指针

4. 进程间的链接信息

struct task\_struct \*next\_task //指向下一个PCB的指针

struct task\_struct \*prev\_task //指向上一个PCB的指针

struct task\_struct \*next\_run //指向可运行队列的下一个PCB的指针

struct task\_struct \*prev\_run //指向可运行队列的上一个PCB的指针

5.进程的调度信息

long counter //时间片计数器

long nice //进程优先级

unsigned long rt\_priority //实时进程的优先级

unsigned long policy //进程调度策略

6.进程的时间信息

long start\_time //进程创建的时间

long utime //进程在用户态下耗费的时间

long stime //进程在核心态下耗费的时间

long cutime //所有子进程在用户态下耗费的时间

long cstime //所有子进程在核心态下耗费的时间

unsigned longtimeout //进程申请延时

7.进程的虚存信息

struct mm\_struct \*mm //进程的虚存信息

struct desc\_struct \*ldt //进程的局部描述符表指针

unsigned long saved\_kernel\_stack //核心态下堆栈的指针

unsigned long kernel\_stack\_page //核心态下堆栈的页表指针

8. 进程的文件信息

struct fs\_struct \*fs //进程的可执行映象所在的文件系统

struct files\_struct \*files //进程打开的文件

9.与进程间通信有关的信息

unsigned longsignal //进程接收到的信号

unsigned longblocked //阻塞信号的掩码

struct signal\_struct\*sig //信号处理函数表的指针

int exit\_signal //进程终止的信号

struct sem\_undo \*semundo //进程要释放的信号量

struct sem\_queue \*semsleeping //与信号量操作相关的等待队列

10．其它信息

int errno //系统调用的出错代码

long debugreg[8] //进程的8个调试寄存器

char comm[16] //进程接收到的信号

PID是32位的无符号整数，它被顺序编号，最大值为32768。

每个进程都属于某个用户、组task\_struct结构中定义有用户标识符UID（User Identifier）和组标识符GID（Group Identifier）

为了表示当前正在运行的进程，定义了一个current宏，可以把它看作全局变量来用，例如current->pid返回正在执行的进程的标识符

**内核堆栈&task\_struct**

2.6 thread\_info代替了原先task\_struct的位置，跟内核堆栈放在一块，thread\_info中放置一个指向task\_struct的指针，如下图。



**进程组织方式-链表**

把可运行状态的进程组成一个双向循环链表，也叫可运行队列（runqueue）

task\_struct中定义两个指针：struct task\_struct \*next\_run, \*prev\_run; init\_task起链表头的作用

**等待队列**:一组睡眠的进程wait\_queue

sleep\_on()函数:让正在运行的进程等待某一特定事件

wake\_up（）:让待唤醒的进程进入TASK\_RUNNING状态

等待机制：wait、sleep。

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

**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

**进程创建的原理：**

系统创建的第一个进程是init进程。系统中所有的进程都是由当前进程使用系统调用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。

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。

#include <sys/type.h>/\* 提供类型pid\_t的定义,在PC上int同\*/ #include <unistd.h> /\* 提供系统调用的定义 \*/

**Fork() 函数进程创建的过程**:

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.

三个系统调用sys\_clone(),sys\_vfork(),sys\_fork() 可以实现创建子进程，这三个系统调用最终都会调用do\_fork()函数完成主要工作。Fork()🡺sys\_fork()🡺do\_fork()

三个系统调用的源程序在arch/i386/kernel/process.c中

do\_fork()函数的第一个参数**clone\_flags**可由多个标志位组成，常见的标志位有：

CLONE\_VM 子进程父进程共享进程空间;

CLONE\_FS 子进程父进程共享文件系统信息;

CLONE\_FILES 子进程父进程共享打开的文件;

CLONE\_VFORK 如果父进程想使子进程释放空间时唤醒它，则置该位。

CLONE\_SIGHAND 父进程和子进程共享信号处理函数表

CLONE\_PTRACE 如果父进程被跟踪的话，那么子进程也被跟踪。

sys\_clone()对应的clone\_flags可能是多个标志位的组合，取决于具体情况。

sys\_fork()对应的clone\_flags值是SIGCHILD。SIGCHILD的作用是子进程终结或暂停时给父进程发信号。

sys\_vfork()对应的clone\_flags值是CLONE\_VFORK|CLONE\_VM|SIGCHILD。

**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( )初始化子进程的核心模式栈时,核心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。

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。内核线程是通过系统调用clone()来实现的

**进程调度**

Linux系统采用抢占调度方式。无论内核态还是用户态。

分时技术，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。

**unsigned long policy: 进程调度策略**

162 #define SCHED\_NORMAL 0 普通进程时间片轮转

163 #define SCHED\_FIFO 1 实时进程先进先出

164 #define SCHED\_RR 2 实时进程时间片轮转

165 #define SCHED\_BATCH 3 后台处理进程（无交互性）

**调度对象**是可运行队列，每个处理器有一个可运行队列

普通进程的**权值**就是它的counter的值（处置21），而实时进程的权值是它的rt\_priority的值加1000

**调度算法**

**kernel/sched.c。**核心函数是**schedule( )**,该函数的任务是选出一个可运行的进程。过程：

1) 检查是否有软中断服务请求，如果有，则先执行这些请求。

2) 若当前进程调度策略是SCHED\_RR， 且 counter为0，则将该进程移到可执行进程队列的尾部并对counter重新赋值。

3) 检查当前进程的状态，如为 TASK\_INTERRUPTIBLE， 且该进程有信号接收，则将进程状态置为 TASK\_RUNNING。

4) 当前进程的状态不是TASK\_RUNNING，则将其从可执行进程队列中移出，然后将当前进程控制块的need\_resched恢复成0。

5) 进入函数的核心部分。可运行进程队列的每个进程都将被计算出一个权值，主要是利用goodness()函数（**计算进程的当前权值）**。最终最大的权值保存在变量c中，与之对应的进程控制块保存在变量next中。

6) 检查c是否为0。若为0则表明所有可执行进程的时间配额都已用完，因而对所有进程的counter重新计算（赋值）。

7) 如果next进程就是当前进程,则结束shedule()的运行。否则进行进程切换，CPU改由next进程占据。

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

调度程序依靠几个最重要的函数有：

scheduler\_tick( ) ，维持当前最新的 time\_slice 计数器

try\_to\_wake\_up( )，唤醒睡眠进程

recalc\_task\_prio( )，更新进程的动态优先权

schedule( )，选择要被执行的新进程

load\_balance()，维持多处理器系统中运行队列的平衡。

**内核模块**

动态可加载内核模块（Loadable Kernel Module LKM）

#define MODULE

#include <linux/module.h>

int init\_module(void) {//入口

printk(“<1> Hello World!\n”);

return 0; }

void cleanup\_module(void) { //出口

printk(“<1>Goodbye!\n”); }

MODULE\_LICENSE(“GPL”);

insmod [path]modulename.ko调用insmod程序将把需要插入的模块以目标代码的形式插入到内核中，insmod会自动运行在init\_module()函数中定义的过程

lsmod显示当前系统中正在使用的模块信息，与cat /proc/modules等价

rmmod 命令卸载模块，会自动运行在cleanup\_module()函数中定义的过程

**存储管理**

**虚拟存储空间**

Linux操作系统采用了请求式分页虚拟存储管理方法

虚拟内存共4G字节，分为内核空间（最高的1G字节）和用户空间（较低的3G字节）两部分.

内核空间由所有进程共享，其中存放的是内核代码和数据，即“内核映象” ；进程的用户空间中存放的是用户程序的代码和数据。进程运行时须有独占的堆栈空间🡺

一个进程的用户地址空间主要由mm\_struct结构和vm\_area\_structs结构来描述. mm\_struct结构它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；vm\_area\_struct结构是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

虚拟内存区域

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

**分页存储**

却也异常处理

**do\_page\_fault()**

页面异常的处理程序两个参数：

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,调度时不会优先考虑。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:

1. 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；

2. 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

**最近最少使用（LRU）**

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

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

**物理内存**

**基于区的伙伴系统及slab分配器**

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。

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。

Linux设置了一个mem\_map[]数组管理内存页帧。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。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。

**Buddy算法**是把内存中的所有页帧按照2n划分，其中n=0～10。划分后形成了大小不等的存储块，称为页帧块，简称页块。数组free\_area[]来管理各个空闲页块组，申请空间的函数为alloc\_pages( )；释放函数为free\_pages( )**；**

**slab分配器：**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。基于伙伴系统的slab分配器

**VFS**

VFS并不是一种实际的文件系统。ext2等物理文件系统是存在于外存空间的，而VFS仅存在于内存。文件系统的源代码可以在 linux/fs 中找到.

**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

struct **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。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

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

**super\_operations**

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

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

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;

｝

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。

**Inode operations**

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。

**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

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 \*/

……

}

对目录项进行操作的一组函数，由d\_op指向 dentry\_operation结构：

struct dentry\_operations {

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

**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。

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;

......

};

**file\_operations**

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

};

**文件的open()操作**

open()系统调用内核函数是sys\_open( )。

第一个参数是打开文件的路径名。第二个参数是文件的访问标志。

**sys\_open( )**：

1、Invokes getname( ) to read the file pathname from the process address space.

2、用get\_unused\_fd( )在current->files->fd所指向的文件对象指针数组中查找一个未使用的文件号，存储在局部变量fd中。

3、调用filp\_open（）函数，工作主要分成两步：

第一步：调用**open\_namei（）**函数，找到目标节点（可以是文件、目录）所对应的dentry对象，与dentry对象相对应的inode对象此时也应该在物理内存中。

第二步 ：调用dentry\_open( )函数，该函数申请一个file对象空间，然后初始化该对象，其中的一步使file对象的f\_dentry指向已获得的dentry对象。

4、调用fd\_install（）函数，将文件对象装入当前进程的打开文件表：current->files->fd[fd] = file;然后返回文件号fd。

最重要的步骤是open\_namei（）完成的。函数的执行过程：

a.确定从哪一个dentry对象出发进行路径解析。根据指定文件路径名是相对路径还是绝对路径从current中得到相应的dentry对象。

b.调用path\_walk（）函数进行路径解析，该函数执行一个循环，每一循环都是得到一个dentry对象，该对象对应文件路径的一个子路径。

**read()的实现**

read( )在内核中的对应函数为sys\_read（）。

(1)根据文件描述符调用函数fget（），找到相应的文件对象file。

(2)检查file->f\_mode是否允许读。

(3)调用file->f\_op->read（）函数执行读的操作。对于大部分文件系统实际是generic\_file\_read( )函数。该函数根据文件位置和要读出的长度确定相应的页面，然后再检查该页面是否在pagecache中存在，如果不存在，就要调用inode节点的i\_mapping->a\_ops->readpage( )方法,将其从磁盘读入。不同磁盘文件系统的readpage方法不 同,ext2文件系统相应的函数为ext2\_readpage。

为了提高性能，读操作采用了预读机制。

**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;

…………………………

}

**进程创建**

struct **task\_struct** {

/\* offsets of these are hardcoded elsewhere - touch with care \*/

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/ //进程当前的状态

unsigned long flags; /\* per process flags, defined below \*/ //反应进程状态的信息，但不是运行状态，定义见下

int sigpending; //进程收到了信号，但尚未处理

mm\_segment\_t addr\_limit; /\* thread address space: //虚存地址上限0-0xBFFFFFFF for user-thead, 0-0xFFFFFFFF for kernel-thread \*/

struct exec\_domain \*exec\_domain;

volatile long need\_resched; //与进程调度有关表示用户从系统空间按返回用户空间要执行的一次调度

unsigned long ptrace;

int lock\_depth; /\* Lock depth \*/

/\*

\* offset 32 begins here on 32-bit platforms. We keep

\* all fields in a single cacheline that are needed for

\* the goodness() loop in schedule().

\*/

long counter; //与进程调度相关

long nice;

unsigned long policy; //实用于本进程的调度政策

struct mm\_struct \*mm;

int processor;

/\*

\* cpus\_runnable is ~0 if the process is not running on any

\* CPU. It's (1 << cpu) if it's running on a CPU. This mask

\* is updated under the runqueue lock.

\*

\* To determine whether a process might run on a CPU, this

\* mask is AND-ed with cpus\_allowed.

\*/

unsigned long cpus\_runnable, cpus\_allowed;

/\*

\* (only the 'next' pointer fits into the cacheline, but

\* that's just fine.)

\*/

struct list\_head run\_list;

unsigned long sleep\_time;

struct task\_struct \*next\_task, \*prev\_task; //内核会对每一个进程做点什么事情的时候，常常需要将其连成一个队列，这2个指针用于这个目的

struct mm\_struct \*active\_mm;

struct list\_head local\_pages;

unsigned int allocation\_order, nr\_local\_pages;

/\* task state \*/

struct linux\_binfmt \*binfmt;//应用文件格式

int exit\_code, exit\_signal;

int pdeath\_signal; /\* The signal sent when the parent dies \*/

/\* ??? \*/

unsigned long personality; //进程的个性化信息，详细见下

int did\_exec:1;

unsigned task\_dumpable:1;

pid\_t pid; //进程号

pid\_t pgrp;

pid\_t tty\_old\_pgrp;

pid\_t session;

pid\_t tgid;

/\* boolean value for session group leader \*/

int leader;

/\*

\* pointers to (original) parent process, youngest child, younger sibling,

\* older sibling, respectively. (p->father can be replaced with

\* p->p\_pptr->pid)

\*/

struct task\_struct \*p\_opptr, \*p\_pptr, \*p\_cptr, \*p\_ysptr, \*p\_osptr; //用于族谱信息的，例如p\_opptr指向父进程

struct list\_head thread\_group;

/\* PID hash table linkage. \*/

struct task\_struct \*pidhash\_next;

struct task\_struct \*\*pidhash\_pprev; //pid是随机分配的，我们常常使用kill pid想进程发送信号（大部分人认为是杀死进程，其实这是个发送信号的指令，默认的参数为杀死。如果想暂停某进程，只需kill STOP 进程的PID），这里可以看到根据pid寻找进程的操作是经常被使用的，而pid又是随机分配，于是这里边用这2个指针指向一个杂凑数组，数组是按照杂凑的算法，以pid为关键字建立，方便根据pid来寻找task\_struct

wait\_queue\_head\_t wait\_chldexit; /\* for wait4() \*/

struct completion \*vfork\_done; /\* for vfork() \*/

unsigned long rt\_priority; //优先级

unsigned long it\_real\_value, it\_prof\_value, it\_virt\_value;

unsigned long it\_real\_incr, it\_prof\_incr, it\_virt\_incr;

struct timer\_list real\_timer;

struct tms times; //运行时间的总汇

unsigned long start\_time;

long per\_cpu\_utime[NR\_CPUS], per\_cpu\_stime[NR\_CPUS]; //在多个处理器上运行于系统空间和用户空间的时间

/\* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific \*/

unsigned long min\_flt, maj\_flt, nswap, cmin\_flt, cmaj\_flt, cnswap;//发生页面异常的次数和换入换出的次数

int swappable:1;

/\* process credentials \*/

uid\_t uid,euid,suid,fsuid;

gid\_t gid,egid,sgid,fsgid; //与文件权限有关的

int ngroups;

gid\_t groups[NGROUPS];

kernel\_cap\_t cap\_effective, cap\_inheritable, cap\_permitted; //权限，比如该进程是否有权限从新引导系统，这里是大概介绍

int keep\_capabilities:1;

struct user\_struct \*user; //指向该进程拥有的用户

/\* limits \*/

struct rlimit rlim[RLIM\_NLIMITS]; //进程对各种资源使用数量的限制，详细见下

unsigned short used\_math;

char comm[16];

/\* file system info \*/

int link\_count, total\_link\_count;

struct tty\_struct \*tty; /\* NULL if no tty \*/

unsigned int locks; /\* How many file locks are being held \*/

/\* ipc stuff \*/

struct sem\_undo \*semundo;

struct sem\_queue \*semsleeping;

/\* CPU-specific state of this task \*/

struct thread\_struct thread;

/\* filesystem information \*/

struct fs\_struct \*fs;

/\* open file information \*/

struct files\_struct \*files;

/\* namespace \*/

struct namespace \*namespace;

/\* signal handlers \*/

spinlock\_t sigmask\_lock; /\* Protects signal and blocked \*/

struct signal\_struct \*sig;

sigset\_t blocked;

struct sigpending pending;

unsigned long sas\_ss\_sp;

size\_t sas\_ss\_size;

int (\*notifier)(void \*priv);

void \*notifier\_data;

sigset\_t \*notifier\_mask;

/\* Thread group tracking \*/

u32 parent\_exec\_id;

u32 self\_exec\_id;

/\* Protection of (de-)allocation: mm, files, fs, tty \*/

spinlock\_t alloc\_lock;

/\* journalling filesystem info \*/

void \*journal\_info;

};

long **do\_fork**(unsigned long clone\_flags, /\*the flags parameter of clone()\*/

unsigned long stack\_start, /\*Same as the child\_stack parameter of clone( )\*/

struct pt\_regs \*regs,/\*Pointer to the values of the general purpose registers\*/

/\*saved into the Kernel Mode stack when switching from User Mode to Kernel Mode\*/

unsigned long stack\_size,

int \_\_user \*parent\_tidptr,

int \_\_user \*child\_tidptr) /\*Same as the corresponding ptid and ctid parameters of clone()\*/

{ struct task\_struct \*p;

int trace = 0;

struct pid \*pid = alloc\_pid(); /\*Allocates a new PID for the child by looking in the pidmap\_array bitmap\*/

long nr;

if (!pid)

return -EAGAIN; /\*申请失败，返回错误码\*/

nr = pid->nr;

if (unlikely(current->ptrace)) {

trace = fork\_traceflag (clone\_flags); /\*Checks the ptrace field of the parent (current->ptrace): if it is not zero, the parent process is being traced by another process, thus do\_fork( ) checks whether the debugger wants to trace the child on its own (independently of the value of the CLONE\_PTRACE flag specified by the parent); in this case, if the child is not a kernel thread (CLONE\_UNTRACED flag cleared), the function sets the CLONE\_PTRACE flag.\*/

if (trace)

clone\_flags |= CLONE\_PTRACE;

}

p = copy\_process(clone\_flags, stack\_start, regs, stack\_size, parent\_tidptr, child\_tidptr, nr);

/\*Invokes copy\_process() to make a copy of the process descriptor. If all needed resources are available, this function returns the address of the task\_struct descriptor just created.\*/

if (!IS\_ERR(p)) { /\*成功创建一个新的PCB\*/

struct completion vfork;

if (clone\_flags & CLONE\_VFORK) {

p->vfork\_done = &vfork;

init\_completion(&vfork);

}

if ((p->ptrace & PT\_PTRACED) || (clone\_flags & CLONE\_STOPPED)) {

sigaddset(&p->pending.signal, SIGSTOP);

set\_tsk\_thread\_flag(p, TIF\_SIGPENDING);

}

/\*a.调整父进程和子进程的调度参数b.如果子进程和父进程在同一个cpu上运行，并且父子进程不共享同一组页表(CLONE\_VMflag被清空)就把子进程插入父进程运行队列，并让子进程位于父进程前面，迫使子进程先于父进程运行 c. 否则将子进程插入父进程运行队列的队尾\*/ if (!(clone\_flags & CLONE\_STOPPED))

wake\_up\_new\_task(p, clone\_flags);

else

p->state = TASK\_STOPPED; /\*如果CLONE\_STOPPED标志被设置，则把子进程置为TASK\_STOPPED状态\*/

if (unlikely (trace)) {

current->ptrace\_message = nr;

ptrace\_notify ((trace << 8) | SIGTRAP);

}/\*如果父进程被跟踪，则把子进程的PID存入 current的ptrace\_message字段并调用ptrace\_notify()，使当前进程停止运行，并向当前进程的父进程发送SIGCHLD信号，通知debugger进程\*/

if (clone\_flags & CLONE\_VFORK) {

wait\_for\_completion(&vfork);

if (unlikely (current->ptrace & PT\_TRACE\_VFORK\_DONE))

ptrace\_notify ((PTRACE\_EVENT\_VFORK\_DONE << 8) | SIGTRAP);

/\*If the CLONE\_VFORK flag is specified, it inserts the parent process in a wait queue and suspends it until the child releases its memory address space (that is, until the child either terminates or executes a new program).\*/

}

} else {

free\_pid(pid);

nr = PTR\_ERR(p);

}

return nr; /\*Terminates by returning the PID of the child.\*/

}

**Do\_Fork（）函数实现分析**

sys\_fork

默认参数调用do\_fork()

分配新的pid=alloc\_pidmap()

拷贝资源并生成新的PCB p=copy\_process(…)

拷贝父进程task\_struct dup\_task\_struct(…)

分配PCB空间 alloc\_task\_struct()

分配thread\_info alloc\_thread\_info()

进行拷贝

拷贝父进程资源

copy\_semundo files fs sighand signal mm keys

copy\_thread: 在这里p->thread.eip设置了子进程执行的起始位置

后面的语句是立刻退出fork函数并返回自己的pid

其中 copy mm

dup\_mm()

调用allocate\_mm 从salb中分配一个mm\_struct结构

使用memcpy拷贝父进程的mm\_struct

mm\_init(mm)初始化mm中的某些项

init\_new\_context()初始化ldt

dup\_mmap 拷贝父进程的虚存段

kmem\_cache\_alloc 分配vm\_struct

拷贝父进程的vm\_struct

拷贝父进程的vma权限 pol=mpol\_copy(vam\_polic(mpnt))

vma\_set\_colicy(tmp,pol)

将vm\_area\_struct加入自己的vma链表

将新的vma中的页表拷贝到自身页表中copy\_page\_range()

打开新的vma

设置新进程的调度 sched\_fork()

设置状态

与父进程均分当前时间片

结束copy\_process(…)

唤醒新生成的进程wake\_up\_task

清理资源

结束do\_fork()

结束sys\_fork

实验

共享内存

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

key\_t key 是这块共享内存的标识符。如果是父子关系的进程间通信的话，这个标识符用IPC\_PRIVATE来代替。如果两个进程没有任何关系，所以就用ftok()算出来一个标识符（或者自己定义一个）使用了。

int size 是这块内存的大小．

int flag 是这块内存的模式(mode)以及权限标识。

模式可取如下值：

IPC\_CREAT 新建（如果已创建则返回目前共享内存的id）

IPC\_EXCL   与IPC\_CREAT结合使用，如果已创建则则返回错误

然后将“模式” 和“权限标识”进行“或”运算，做为第三个参数。

如：    IPC\_CREAT | IPC\_EXCL | 0640

例子中的0666为权限标识,4/2/1 分别表示读/写/执行3种权限，第一个0是UID,第一个6（4+2）表示拥有者的权限，第二个4表示同组权限，第3个0表示他人的权限。

这个函数成功时返回共享内存的ID，失败时返回-1。

创建共享内存时，shmflg参数至少需要 IPC\_CREAT | 权限标识，如果只有IPC\_CREAT 则申请的地址都是k=0xffffffff，不能使用；

获取已创建的共享内存时，shmflg不要用IPC\_CREAT（只能用创建共享内存时的权限标识，如0640），否则在某些情况下，比如用ipcrm删除共享内存后，用该函数并用IPC\_CREAT参数获取一次共享内存（当然，获取失败），则即使再次创建共享内存也不能成功，此时必须更改key来重建共享内存

void    \*shmat( int shmid , char \*shmaddr , int shmflag );

shmat()是用来允许本进程访问一块共享内存的函数。

int shmid是那块共享内存的ID。

char \*shmaddr是共享内存的起始地址，如果shmaddr为0，内核会把共享内存映像到调用进程的地址空间中选定位置；如果shmaddr不为0，内核会把共享内存映像到shmaddr指定的位置。所以一般把shmaddr设为0。

int shmflag是本进程对该内存的操作模式。如果是SHM\_RDONLY的话，就是只读模式。其它的是读写模式  
成功时，这个函数返回共享内存的起始地址。失败时返回-1。

int shmdt( char \*shmaddr );

参数char \*shmaddr是那块共享内存的起始地址。

成功时返回0。失败时返回-1。

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

int shmid是共享内存的ID。

int cmd是控制命令，可取值如下：

        IPC\_STAT        得到共享内存的状态

        IPC\_SET         改变共享内存的状态

        IPC\_RMID        删除共享内存

struct shmid\_ds \*buf是一个结构体指针。IPC\_STAT的时候，取得的状态放在这个结构体中。如果要改变共享内存的状态，用这个结构体指定。

返回值：   成功：0

                失败：-1

管道：

int pipe(int filedes[2]);

函数说明

    pipe()会建立管道,并将文件描述词由参数 filedes 数组返回。

    filedes[0]为管道里的读取端，所以pipe用read调用的

    filedes[1]则为管道的写入端。

返回值: 若成功则返回零,否则返回-1,错误原因存于 errno 中。

错误代码:

    EMFILE 进程已用完文件描述词最大量

    ENFILE 系统已无文件描述词可用。

    EFAULT 参数 filedes 数组地址不合法。

例：

int filedes[2];

    char buf[80];

    pid\_t pid;

    pipe( filedes );

    if ( (pid=fork()) > 0 )

    {

        printf( "This is in the father process,here write a string to the pipe.\n" );

        char s[] = "Hello world , this is write by pipe.\n";

        write( filedes[1], s, sizeof(s) );

        close( filedes[0] );

        close( filedes[1] );

    }

    else

    {

        printf( "This is in the child process,here read a string from the pipe.\n" );

        read( filedes[0], buf, sizeof(buf) );

        printf( "%s\n", buf );

        close( filedes[0] );

        close( filedes[1] );

    }

消息队列：

int msgget(key\_t key, int msgflg); //创建消息队列

参数：

key：消息队列关联的键。

msgflg：消息队列的建立标志和存取权限。

返回说明：

成功执行时，返回消息队列标识值。失败返回-1

int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

//将消息送入消息队列

参数：

msqid：消息队列的识别码。

msgp：指向消息缓冲区的指针，此位置用来暂时存储发送和接收的消息，是一个用户可定义的通用结构，形态如下

struct msgbuf {

long mtype; /\* 消息类型，必须 > 0 \*/

char mtext[1]; /\* 消息文本 \*/

};

msgsz：消息的大小。

msgtyp：从消息队列内读取的消息形态。如果值为零，则表示消息队列中的所有消息都会被读取。

msgflg：用来指明核心程序在队列没有数据的情况下所应采取的行动。如果msgflg和常数IPC\_NOWAIT合用，则在msgsnd()执行时若是消息队列已满，则msgsnd()将不会阻塞，而会立即返回-1，如果执行的是msgrcv()，则在消息队列呈空时，不做等待马上返回-1，并设定错误码为ENOMSG。当msgflg为0时，msgsnd()及msgrcv()在队列呈满或呈空的情形时，采取阻塞等待的处理模式。

返回说明：

成功执行时，msgsnd()返回0, 失败返回-1

ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);

//从消息队列读取信息.

成功执行时,返回0, 失败返回-1

msgctl(qid, IPC\_RMID, NULL) //删除消息队列

信号量：

sem\_init：初始化信号量sem\_t，初始化的时候可以指定信号量的初始值，以及是否可以在多进程间共享。

sem\_wait：一直阻塞等待直到信号量>0。

sem\_timedwait：阻塞等待若干时间直到信号量>0。

sem\_post：使信号量加1。

sem\_destroy：释放信号量。和sem\_init对应。

编译内核：

make clean      删除大多数的编译生成文件， 但是会保留内核的配置文件.config， 还有足够的编译支持来建立扩展模块  
make mrproper   删除所有的编译生成文件， 还有内核配置文件， 再加上各种备份文件  
make distclean  mrproper删除的文件， 加上编辑备份文件和一些补丁文件。

apt-get install kernel-package libncurses5-dev fakeroot wget bzip2

//安装工具包

make config是有问必答的方式，每个内核选项它都会问你要、不要、模块，选错了一个就必须从头再来一遍；

make menuconfig提供一个基于文本的图形界面，它依赖于ncurses5这个包，键盘操作，可以修改选项，一般推荐用这个；

make xconfig需要你有x window system支持，就是说你要在KDE、GNOME之类的X桌面环境下才可用，好处是支持鼠标，坏处是X本身占用系统周期，而且X环境容易引起编译器的不稳定

make -j4 启动4个线程（双核）来编译内核文件生成.o等中间文件

内核文件bzImage的位置在/usr/src/linux/arch/i386/boot目录下。

make modules\_install 安装模块

make install使用命令make install将bzImage和System.map拷贝到/boot目录下。这样，Linux在系统引导后从/boot目录下读取内核映像到内存中

添加系统调用：

system\_call()函数实现了系统调用中断处理程序：

1.它首先把系统调用号和该异常处理程序用到的所有CPU寄存 器保存到相应的栈中， SAVE\_ALL

2.把当前进程task\_struct （thread\_info）结构的地址存放在ebx中

3.对用户态进程传递来的系统调用号进行有效性检查。若调 用号大于或等于NR\_syscalls，系统调用处理程序终止。 （sys\_call\_table）

4.若系统调用号无效，函数就把-ENOSYS值存放在栈中eax寄 存器所在的单元，再跳到ret\_from\_sys\_call()

5.根据eax中所包含的系统调用号调用对应的特定服务例程

实验修改的主要有3处地方：

for (p = &init\_task; (p = next\_task(p)) != &init\_task;)

//遍历进程

p->comm //comm类型为char[16],代表进程名

p->pid //当亲进程号

p->state //当前进程的状态

-1 unrunnable, 0 runnable, >0 stopped

p-> parent //指向父进程task\_struct的地址

添加文件系统：

/\* Structure of a directory entry \*/

#define EXT2\_NAME\_LEN 255

/\* The new version of the directory entry. Since EXT2 structures are stored in intel byte order, and the name\_len field could never be bigger than 255 chars, it's safe to reclaim the extra byte for the file\_type field. \*/

struct ext2\_dir\_entry\_2 {

\_\_u32 inode; /\* Inode number \*/

\_\_u16 rec\_len; /\* Directory entry length \*/

\_\_u8 name\_len; /\* Name length \*/

\_\_u8 file\_type;

char name[EXT2\_NAME\_LEN]; /\* File name \*/

};

/\* Ext2 directory file types. Only the low 3 bits are used. The other bits are reserved for now.\*/

enum {

EXT2\_FT\_UNKNOWN,

EXT2\_FT\_REG\_FILE,

EXT2\_FT\_DIR,

EXT2\_FT\_CHRDEV,

EXT2\_FT\_BLKDEV,

EXT2\_FT\_FIFO,

EXT2\_FT\_SOCK,

EXT2\_FT\_SYMLINK,

EXT2\_FT\_MAX

};

文件类型：

普通文件（文件名不超过255）

目录文件

字符设备文件和块设备文件：

fd0 (for floppy drive 0)

hda (for harddisk a)

lp0 ( for line printer 0)

tty(for teletype terminal)

管道(FIFO)文件

链接文件

socket文件

文件系统分三大类：

基于磁盘的文件系统，如ext2/ext3/ext4、VFAT、NTFS等。

网络文件系统，如NFS等。

特殊文件系统，如proc文件系统、devfs、sysfs（/sys）等。

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。

inode结构定义在文件<include/linux/fs.h>中：

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;

struct file\_system\_type {

const char \*name;

int fs\_flags;

struct super\_block \*(\*get\_sb) (struct file\_system\_type \*, int,

const char \*, void \*);

void (\*kill\_sb) (struct super\_block \*);

struct module \*owner;

struct file\_system\_type \* next;

struct list\_head fs\_supers;

};

name： 文件系统类型名字，比如“ext2”，“vfat”等等。

fs-\_flags：mount的文件系统的参数。

get\_sb：当这种类型的文件系统要被mount的时候，这个函数会被调用，用以得到相应文件系统的超级块。

kill\_sb：当这种类型的文件系统被umount的时候，这个函数被调用。

owner：VFS内部使用，大多数情况下，你只需要初始化为THIS\_MODULE。

next： 文件系统类型链表的后续指针。VFS内部使用，初始化为NULL。

list\_head fs\_supers：文件系统的超级块的双向链表。

#define NR\_OPEN 256

struct files\_struct {

int count; /\* 共享该结构的计数值\*/

fd\_set close\_on\_exec;

fd\_set open\_fds;

struct file \* fd[NR\_OPEN];

};

fd[]每个元素是一个指向file结构体的指针，该数组称为进程打开文件表。

进程打开一个文件时，建立一个file结构体，并加入到系统打开文件表中，然后把该file结构体的首地址写入fd[] 数组的第一个空闲元素中，一个进程所有打开的文件都记载在fd[]数组中。

dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换

命令语法：dd [选项]

常用参数：

if = 输入文件（或设备名称）

of = 输出文件（或设备名称）

bs = bytes 同时设置读/写缓冲区的字节数（等于设置ibs和obs）

count=blocks 只拷贝输入的blocks块

conv = ucase 把字母由小写转换为大写

conv = lcase 把字母由大写转换为小写。

例：dd if=/dev/zero of=myfs bs=1M count=1

/dev/zero：零设备“0”

/dev/loop：loopback device（回环设备、或虚拟设备）指是用文件来模拟块设备

ext2的操作都在结构ext2\_dir\_inode\_operations中定义

fs/ext2/namei.c下定义了具体实现方法

int open\_namei(int dfd, const char \*pathname, int flag,

int mode, struct nameidata \*nd)

{

int acc\_mode, error;

struct path path;

struct dentry \*dir;

int count = 0;

// #define ACC\_MODE(x) ("\000\004\002\006"[(x)&O\_ACCMODE])

// 审计模式

acc\_mode = ACC\_MODE(flag);

/\* O\_TRUNC implies we need access checks for write permissions \*/

// 截断标志, 基本上需要写权限, 除非要截断的长度实际大于文件本身长度

if (flag & O\_TRUNC)

acc\_mode |= MAY\_WRITE;

/\* Allow the LSM permission hook to distinguish append

access from general write access. \*/

// 添加标志, 也是需要写权限

if (flag & O\_APPEND)

acc\_mode |= MAY\_APPEND;

/\*

\* The simplest case - just a plain lookup.

\*/

// 不需要创建文件

if (!(flag & O\_CREAT)) {

// 直接找pathname的dentry和挂接点, 结果填在nd中

error = path\_lookup\_open(dfd, pathname, lookup\_flags(flag),

nd, flag);

if (error)

return error;

goto ok;

}

/\*

\* Create - we need to know the parent.

\*/

// 创建文件的dentry和挂接点, 数据填到nd中

error = path\_lookup\_create(dfd,pathname,LOOKUP\_PARENT,nd,flag,mode);

if (error)

return error;

/\*

\* We have the parent and last component. First of all, check

\* that we are not asked to creat(2) an obvious directory - that

\* will not do.

\*/

error = -EISDIR;

// 检查nameidata结构中的last参数是否合法

if (nd->last\_type != LAST\_NORM || nd->last.name[nd->last.len])

goto exit;

// 文件项dentry

dir = nd->dentry;

// 去掉查询父目录标志

nd->flags &= ~LOOKUP\_PARENT;

mutex\_lock(&dir->d\_inode->i\_mutex);

// 填充path参数, 又根据nd的信息搜索一次当前的缓存的dentry

// 不过dir与path.dentry难道不相同么?

path.dentry = lookup\_hash(nd);

path.mnt = nd->mnt;

do\_last:

// 检查path.entry是否合法

error = PTR\_ERR(path.dentry);

if (IS\_ERR(path.dentry)) {

mutex\_unlock(&dir->d\_inode->i\_mutex);

goto exit;

}

// 检查nd->intent.open.file是否合法, 这是最终要返回的文件指针

if (IS\_ERR(nd->intent.open.file)) {

mutex\_unlock(&dir->d\_inode->i\_mutex);

error = PTR\_ERR(nd->intent.open.file);

goto exit\_dput;

}

/\* Negative dentry, just create the file \*/

if (!path.dentry->d\_inode) {

// 创建新文件的inode, 然后返回

error = open\_namei\_create(nd, &path, flag, mode);

if (error)

goto exit;

return 0;

}

// 现在是打开已经存在的文件

/\*

\* It already exists.

\*/

mutex\_unlock(&dir->d\_inode->i\_mutex);

audit\_inode\_update(path.dentry->d\_inode);

error = -EEXIST;

// O\_EXCL标志是只必须打开的是不存在的文件, 文件已存在时错误

if (flag & O\_EXCL)

goto exit\_dput;

if (\_\_follow\_mount(&path)) {

error = -ELOOP;

if (flag & O\_NOFOLLOW)

goto exit\_dput;

}

error = -ENOENT;

if (!path.dentry->d\_inode)

goto exit\_dput;

// 如果dentry的具体FS的实现中定义了follow\_link操作, 转

// 不过大多数FS的实现中都没有定义该函数

if (path.dentry->d\_inode->i\_op && path.dentry->d\_inode->i\_op->follow\_link)

goto do\_link;

// 从路径中的dentry和mnt信息赋值到nameidata

path\_to\_nameidata(&path, nd);

error = -EISDIR;

// 如果是一个目录, 返回错误

if (path.dentry->d\_inode && S\_ISDIR(path.dentry->d\_inode->i\_mode))

goto exit;

ok:

// 对nd中的dentry及其inode进行打开前的错误检查

error = may\_open(nd, acc\_mode, flag);

if (error)

goto exit;

return 0;

// 下面是错误处理, 释放掉已分配的资源, 返回错误

exit\_dput:

dput\_path(&path, nd);

exit:

if (!IS\_ERR(nd->intent.open.file))

release\_open\_intent(nd);

path\_release(nd);

return error;

// 处理符号连接, 找到实际文件的inode,然后重新循环, 要注意回环情况的错误处理

do\_link:

error = -ELOOP;

if (flag & O\_NOFOLLOW)

goto exit\_dput;

/\*

\* This is subtle. Instead of calling do\_follow\_link() we do the

\* thing by hands. The reason is that this way we have zero link\_count

\* and path\_walk() (called from ->follow\_link) honoring LOOKUP\_PARENT.

\* After that we have the parent and last component, i.e.

\* we are in the same situation as after the first path\_walk().

\* Well, almost - if the last component is normal we get its copy

\* stored in nd->last.name and we will have to putname() it when we

\* are done. Procfs-like symlinks just set LAST\_BIND.

\*/

// 设置查找LOOKUP\_PARENT标志

nd->flags |= LOOKUP\_PARENT;

error = security\_inode\_follow\_link(path.dentry, nd);

if (error)

goto exit\_dput;

// 处理符号链接

error = \_\_do\_follow\_link(&path, nd);

if (error) {

/\* Does someone understand code flow here? Or it is only

\* me so stupid? Anathema to whoever designed this non-sense

\* with "intent.open".

\*/

release\_open\_intent(nd);

return error;

}

nd->flags &= ~LOOKUP\_PARENT;

// 检查最后一段文件或目录名的属性情况

if (nd->last\_type == LAST\_BIND)

goto ok;

error = -EISDIR;

if (nd->last\_type != LAST\_NORM)

goto exit;

if (nd->last.name[nd->last.len]) {

\_\_putname(nd->last.name);

goto exit;

}

error = -ELOOP;

// 出现回环标志: 循环超过32次

if (count++==32) {

\_\_putname(nd->last.name);

goto exit;

}

dir = nd->dentry;

mutex\_lock(&dir->d\_inode->i\_mutex);

// 更新路径的挂接点和dentry

path.dentry = lookup\_hash(nd);

path.mnt = nd->mnt;

\_\_putname(nd->last.name);

goto do\_last;

}

**open\_namei函数：**

实现了从文件名到inode的转换, 也是namei的处理入口.一下为其中的核心函数：

\_\_link\_path\_walk名称解析函数

去掉起始多余的"/"

空路径；

路径对应的inode；

For: 循环处理,每个循环提取文件路径的一个目录名, '/'分隔；

检查文件权限, 包括读写执行权限, 用户/组/其他权限, 返回0为合法；

EAGAIN表示该inode正在被操作, 检查其执行权限

而对于普通文件检查结果将是错误

出错中断循环

填充quickstring结构

name的第一个字符的数值  
 计算文件名的hash, 不包括'/'

目录(如果有的话)的名称长度

c为0表示是最后的具体文件名了

跳过中间的'/'

到名称尾, 说明文件名最后一个字符是'/'

如果第一个字符是'.'：

Default：是一个一'.'开头的文件或目录名称

Case 2：第2 个字符不是".", 是普通文件或路径名

以".."开头, 是父目录, 更新nd为父目录nameidata数据, inode相应更新重新循环；

Case1： 以'.'开头的当前目录, 忽略, 重新循环

底层FS实现中有自己的HASH算法

根据文件/目录名进行具体的查找（do\_lookup）

inode更新为本级文件目录的inode

找不到inode, 转错误处理

处理符号链接, 在其中考虑了递归互相链接的异常处理(do\_follow\_link)

更新inode为实际的inode

nd中得到下一级路径信息

继续循环找下一目录文件名称

最后的文件名了, 处理和前面类似

重复上面的工作…

……

lookup\_parent:

复制当前quickstring结构this信息到nd的last中  
类型为LAST\_NORM

return\_reval:

返回

return\_err：

出错

Open\_namei() 结构如下：

path\_lookup\_open    path\_lookup\_create  
     |                     |  
     V                     V  
   \_\_path\_lookup\_intent\_open(打开文件)  
               |  
               V  
        do\_path\_lookup（遍历整个目录，找到所需的dentry）  
               |  
               V  
        link\_path\_walk  
               |  
               V  
      \_\_link\_path\_walk  
               |  
               V  
           do\_lookup  
               |  
               V  
          real\_lookup

**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 |

**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 |