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）

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 [0-9][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

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都没有关联的文件，这些文件可以用 Linux 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

文件系统装载

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

挂载 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，回车后，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

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 为鹦 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 只显示字数 -l 只显示行数 -w 只显示单词数

正则表达式

支持工具：awk, ed, egrep, grep, sed, vi

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

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 块的。字符 c 用来确定块的大小，默认为 512 个字节。+N 表示大于 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

通过管道使用正则表达式搜索

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 中有对应的 UID 或者名称的用户的进程信息(UID 或用户名由逗号分开)。

-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 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 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 arraya[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，但在数组里使用好象是可以通用的。

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

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
  elif exp2
    elif1-commands
  else
    else-commands
fi
```

4. 选择结构：case 和 select

```
case variable in
  pattern1)
command1 ;;
  pattern2)
command2 ;;
  pattern3)
command3 ;;
esac
```

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 功能强大调试器

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 getppid 以获得其父进程的进程 ID
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 ID、实际组 ID、有效用户 ID、有效组 ID 等。父、子进程之间的区别在于 fork 的返回值，进程 ID，不同的父进程 ID，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(fildes)创建.
pipe 系统调用: int fildes [2]：int pipe (fildes)：

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上.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、进程管理、内存管理模块

#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）

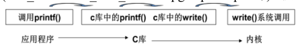
Linux 使用了其中的两个：特权级 0 和特权级 3 ，即内核模式(kernel mode)和用户模式(user mode）

地址空间（space）

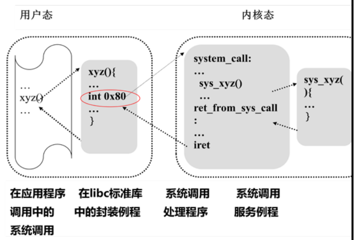
每个进程的虚拟地址空间：用户空间和内核空间。在用户态下只能访问用户空间；核心态下，均可访问
内核空间在每个进程的虚拟地址空间中都是固定的（虚拟地址为 3G~4G 的地址空间）。

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指定赋予存储区的保护级别。
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_t 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。
（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。 注意:父进程忘记了用 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

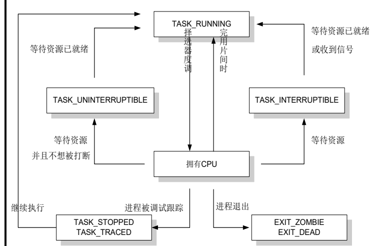
Linux 没有真正意义上的线程概念。但通过 clone()系统调用支持轻量级进程(线程);还支持内核线程，内核线程永远在核心态运行，没有用户空间。

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

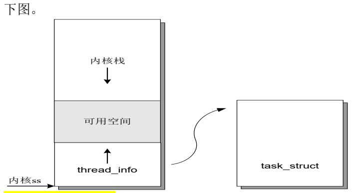
进程的 task_struct 是进程存在的唯一标志。task_struct 容纳了一个进程的所有信息，是系统对进程进行管理和控制的有效手段，是系统实现进程调度的主要依据。
进程创建时，系统就为其建立一个 task_struct 进程控制块。运行结束时，撤消该进程的 task_struct 。
Linux 在内存空间中开辟了一个专门的区域存放所有进程的 task_struct 。系统中的最多进程数目受 task 数组大小的限制，缺省值一般为 512。

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转换成可中断的等待态(INTERRUPTIBLE)，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复制给子进程，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，如把进程状态设置为 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,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( ),将该子进程插入到可运行队列。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
进程创建
系统调用流程 fork -> sys_fork -> do_fork
Do_Fork（）
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

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 schedule()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）

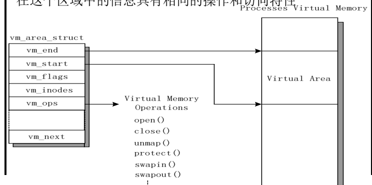
```
#define MODULE
#include <linux/module.h>
int init_module(void)   {//入口
  printk(“<1> Hello World!\n”);
  return 0;   }
void cleanup_module(void) {   //出口
  printk(“<1>Goodbye!\n”);        }
MODULE_LICENSE(“GPL”);
```

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

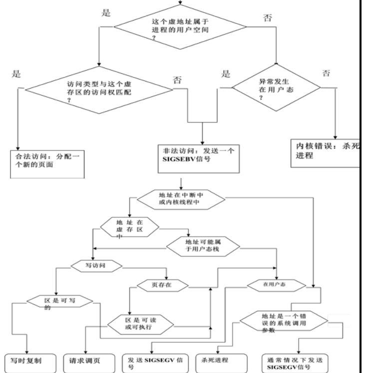
存储管理
虚拟存储空间
Linux 操作系统采用了请求式分页虚拟存储管理方法
虚拟内存共 4G 字节，分为内核空间（最高的 1G 字节）和用户空间（较低的 3G 字节）两部分.
内核空间由所有进程共享，其中存放的是内核代码和数据，即“内核映象”；进程的用户空间中存放的是用户程序的代码和数据。进程运行时须有独占的堆栈空间→

堆栈段
空洞
数据段
代码段

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



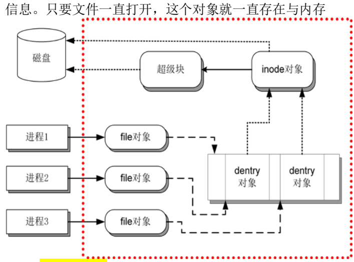
虚拟内存区域
fork()是通过拷贝或共享父进程的用户空间来实现的，即内核调用 copy_mm()函数，为新进程建立所有页表和 mm_struct 结构
分页存储
却也异常处理
4. How do you extract the kernel from the tarball *linux–2.6.14.tar.bz2*?
A.) tar x linux–2.6.14.tar.bz2
B.) untar linux–2.6.14.tar.bz2
C.) tar tzvf linux–2.6.14.tar.bz2
D.) tar xjf linux–2.6.14.tar.bz2



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 2^n 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 仅仅在于内存。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。
这些 inode 结构按照散列表的方式组织起来。

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
};
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。

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，然后初始化这个对象，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（）:

实验
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。

```
#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
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

操作系统原理
线程 相对优先级
THREAD_PRIORITY_TIME_CRITICAL THREAD_PRIORITY_HIGHEST +2
THREAD_PRIORITY_ABOVE_NORMAL +1
THREAD_PRIORITY_NORMAL 0
THREAD_PRIORITY_BELOW_NORMAL -1
THREAD_PRIORITY_LOWEST -2
THREAD_PRIORITY_IDLE
基本优先级
REALTIME_PRIORITY_CLASS，基本优先权为 24。
 HIGH_PRIORITY_CLASS，基本优先权为 13。 ABOVE_NORMAL_PRIORITY_CLASS，基本优先权为 10
//Windows NT and Windows Me/98/95: This value is not supported.
NORMAL_PRIORITY_CLASS，基本优先权为 8。
 BELOW_NORMAL_PRIORITY_CLASS，基本优先权为 6
//Windows NT and Windows Me/98/95: This value is not supported.
IDLE_PRIORITY_CLASS，基本优先权为 4。

Sytem Call 3 种传参方法，1. Register 2. Pass address to register 3. Patermeter pushed to stack, poped by system
ch1
- The one program running at all times on the computer" is the kernel(内核).
- Efficient Fair Convenient
What is the difference between kernel mode and user mode?
Why is the difference important to an operating system?
In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address .
In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory.
The difference rather protect the computer system resources, while preventing from errant users.
bootstrap program is loaded at power-up or reboot
- Typically stored in ROM or EEPROM, generally known as firmware
- Initializates all aspects of system
- Loads operating system kernel and starts execution
批处理系统（batch System）
注意同一时间间隔（并发）和同一时刻（并行）的区别。在多道程序环境下，一段时间内，宏观上有多道程序在同时执行，而在每一时刻，单处理机环境下实际仅能有一道程序执行，故微观上这些程序还是在分时地交替执行。操作系统的并发性是通过分时得以实现的。
- 多任务并发
lack of interaction
分时操作系统（Time-Sharing Systems ——Linux a multi-user time-sharing system
所谓分时技术就是把处理器的运行时间分成很短的时间片，按时间片轮流把处理机分配给各联机作业使用。若某个作业在分配给它的时间片内不能完成其计算，则该作业暂时停止运行，把处理器让给其他作业使用，等待下一轮再继续运行。由于计算机速度很快，作业运行轮转得很快，给每个用户的感觉好像是自己独占一台计算机。
实时操作系统
实时操作系统的主要特点是及时性和可靠性。
运行机制
- 时钟管理 中断机制
- 系统控制的数据结构及处理
    - 系统中用来登记状态信息的数据结构很多，比如作业控制块、进程控制块(PCB)、设备控制块、各类链表、消息队列、缓冲区、空闲区登记表、内存分配表等。为了实现有效的管理，系统需要一些基本的操作，常见的操作有以下三种：
    - 进程管理：进程状态管理、进程调度和分派、创建与撤销进程控制块等。
    - 存储器管理：存储器的空间分配和回收、内存信息保护程序、代码对换程序等。
    - 设备管理：缓冲区管理、设备分配和回收等。
中断
- 外中断（中断）：I/O 中断、时钟中断
- 内中断（异常）：系统调用（陷入）、缺页异常、断点指令、其他程序性异常（如算数溢出）
    - 自愿中断（指令中断）、强迫中断（硬件故障、软件中断）

中断处理：The operating system preserves the state of the CPU by storing registers and the program counter.
从用户态转换为核心态的唯一途径是中断或异常
由用户态进入核心态，不仅仅是状态需要切换。而且，所使用的堆栈也可能需要由用户堆栈切换为系统堆栈，但这个系统堆栈也是属于该进程的。
I/O 方式程序 I/O (Programmed I/O)中断 I/O (Interrupt I/O):同步 I/O 和异步 I/O
DMA 方式
通道方式
Synchronous I/O(同步 I/O):After I/O starts, control returns to user program only upon I/O completion.
* Wait for I/O completion,tow wayswait instruction idles the CPU until the next interrupt wait loop (loop: jmp loop).
* At most one I/O request is outstanding at a time, no simultaneous I/O processing.
Asynchronous I/O (异步 I/O) : After I/O starts, control returns to user program without waiting for I/O completion.
System call 系统调用 – request to the operating system to allow user to wait for I/O completion.
Device-status table contains entry for each I/O device indicating its type, address, and state.
Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.
DMA
Used for high-speed I/O devices able to transmit information at close to memory speeds.
* Device controller transfers blocks of data from buffer storage

directly to main memory without CPU intervention.
* Only on interrupt is generated per block, rather than the one interrupt per byte.

System Calls :Programming interface to the services provided by the OS
The function implementation is inside the OS, or we name it the OS kernel.
Those functions are usually packed inside an object called the library file.
Three general methods used to pass parameters to the OS
- Pass the parameters in registers
- Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - This approach taken by Linux and Solaris
- Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system

Types of System Calls
- Process control
- File management
- Device management
- Information maintenance
- Communication: Provide the mechanism for creating virtual connections among processes, users, and computer systems

Simple Structure 简单结构
Layered Approach 层次化结构
Layered OS is more efficient than monolithic OS(X)错误，巨型内核直接访问更加高效

Microkernel System Structure （Windows NT ... Windows 8、Windows 10、Mac OS）
优点
- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure
缺点
- Performance overhead of user space to kernel space communication 费时间

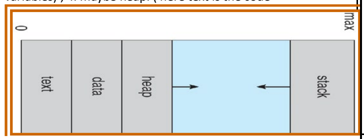
Monolithic Kernel Structure - packed into a single file （OS/360, VMS and Linux）
优点
- highly efficient because of direct communication between components
缺点
- Difficult to isolate source of bugs and other errors
- Hard to modify and maintain
- Kernel gets bigger as the OS develops.

Module（Linux , Solaris ）
Most modern operating systems implement kernel modules
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel

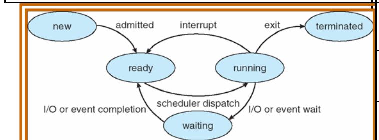
Virtual Machines 虚拟机
为了在通用操作系统管理下的计算机上运行一个程序，需要经历几个步骤。但是，__不一定需要向操作系统预定运行时间
需要，用控制台监控程序执行过程，将程序装入内存，确定起始地址，并从这个地址开始执行

进程

Why process rather than program?
1.different data with same program
2.different program with same data
A process include:
1.program counter; 2. stack (temporary data: parameters, return address, local variables); 3. data section (global variables) ; 4. maybe heap. (here text is the code



进程状态: A process has states while a program does not have states



PCB 的内容:
1.Process state
2.Program counter
3.CPU registers
4.CPU scheduling information
5.Memory-management information
6.Accounting information
7.I/O status information
8.page table or relocation register and limit register
9.file open table

调度队列的种类
Processes migrate among the various queues
1. job queue – set of all processes in the system
2. ready queue – set of all processes residing in main memory, ready and waiting to execute. Generally stored as a linked list
3. device queues – set of processes waiting for a particular I/O device, eg. a tape driver, a disk

调度器 scheduler 分类
long-term scheduler (or job scheduler) –selects which processes should be brought into the ready queue
short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
Attention: The long-term scheduler controls the **degree of multiprogramming**

Context-switch
Context-switch time is overhead; the system does no useful work while switching.
Process Termination
Process executes last statement and asks the operating system to delete it (exit)
1.Output data from child to parent (via wait)
2.Process' resources are de-allocated by operating system
Parent may terminate execution of children processes (abort)
1.Child has exceeded allocated resources
2.Task assigned to child is no longer required
3.If parent is exiting
Some operating system do not allow child to continue if its parent terminates--All children terminated - cascading termination
Advantages of process cooperation
Information sharing, Computation speed-up, Modularity and Convenience.
Interprocess Communication
1.shared memory and 2.message passing

进程同步种类: blocking (synchronous) vs. nonblocking (asynchronous)
1.**blocking send**: the sending process is blocked until the message is received by the receiving process or by the mailbox
2.**nonblocking send**: the sending process sends the message, and resumes operation
3.**blocking receive**: the receiver blocks until a message is available
4.**nonblocking receive**: the receiver retrieves either a valid message or a null, and resumes operation

生产者-消费者问题（指利用了 n-1 个单元）
```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
// producer
item nextProduced; // local variable
while (1) {
        /* produce an item in nextProduced */
while (((in + 1) % BUFFER_SIZE) == out)
        ;  // do nothing
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
// consumer
item nextConsumed;   // local variable
while (1) {
        while (in == out)    ; // buffer is empty
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
/* consume the item in nextConsumed */
}
```

线程

Benefits
1.Responsiveness (e.g. thread as GUI engine)
2.Resource Sharing (e.g. shared variable)
3.Economy (e.g. save memory)
4.Utilization of multiprocessor architectures
Thread can be a basic scheduling unit, but not one of resource allocation.
用户线程和内核线程
- 用户级线程：不依赖于 OS 核心（内核不了解用户线程的存在），应用进程利用 **线程库**提供创建、同步、调度和管理线程的函数来控制用户线程。
- 内核级线程：：依赖于 OS 核心，由内**核的内部需求**进行创建和撤销，用来执行一 个指定的函数。一个线程发起系统调用而阻塞
- 用户线程的维护由应用进程完成；**内核不了解用户线程的存在；**
- **用户线程切换不需要内核特权；**
- 用户线程调度算法可针对应用优化；
- **▲一个线程发起系统调用而阻塞，则整个进程在等待。**
- 内核维护进程和线程的上下文信息；**
- **线程切换由内核完成；**
- **时间片分配给线程，▲所以多线程的进程获得更多 CPU 时间；**
- **▲一个线程发起系统调用而阻塞，不会影响其他线程的运行

同步信号和异步信号
synchronous signals - 1. an illegal memory access, a division by zero 2. delivered to the same process that cause the signal
asynchronous signals
1. a user keystroke (Ctrl-C), a timer expiration
2. typically sent to another process

Multithreading models
Many to one:
1.many user-level threads mapped to single kernel thread
2.thread management is done in user space, used on systems that do not support kernel threads
3.drawbacks: whole process block if one thread blocks. unable to run parallel on multiprocessors
One to one:
1.each user-level thread maps to a kernel thread: more concurrency than many-to-one, support multiprocessors
2.drawback: overhead of creating kernel threads
3.examples: Windows 95/98/NT/2000, OS/2
Many to many:
1.multiplexes many user level threads to a smaller or equal number of kernel threads
2. allows the programmer to create a sufficient number of user threads
3. avoid bad blocking, support multiprocessors
Examples: Solaris 2, Windows NT/2000 with the ThreadFiber package
***Two-level Model --** M:M + 1:1
Similar to M:M, except that it allows a user thread to be **bound** to kernel thread.
Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier.

线程的终止: thread cancellation
1.target thread: the thread to be cancelled
2.asynchronous cancellation: one thread immediately terminates the target thread, may be cancelled in the middle of updating data shared with other threads, may not free a system-wide resource
3.deferred cancellation: the target thread can periodically check if it should terminate (at so called cancellation points in Pthread)

线程池
Why the thread pools?
1. avoid creation and termination overhead, so that it is faster to service a request
2. put bound on number of threads, thus limit CPU and memory usage
Considerations :
1.number of CPUs
2.amount of physical memory
3.expected number of concurrent requests
fork may cp all/current, exec replace whole.

CPU 调度

Maximum CPU utilization obtained with multiprogramming
CPU scheduling
Decisions may take place when a process, but not limited to:
1.Switches from running to waiting state
2.Switches from running to ready state
3.Switches from waiting to ready
4.Terminates
Scheduling under 1 and 4 is nonpreemptive.
All other scheduling is preemptive.
Dispatcher
Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
1.switching context
2.switching to user mode
3.jumping to the proper location in the user program to restart that program
Dispatch latency – time it takes for the dispatcher to stop one process and start another running.
CPU 调度的评判标准 Scheduling Criteria
1.CPU utilization – keep CPU as busy as possible
2.Throughput – number of processes that complete their execution per time unit
3.Turnaround time 周转时间– the interval from the time of submission of a process to the time of completion
4.Waiting time – amount of time a process has been waiting in the ready queue
5.Response time – amount of time it takes from when a request was submitted until the first response

CPU 调度算法
FCFS：等待时间不稳定，响应时间不稳定。
特点：easy to understand, easy to implement (an FIFO queue), large variance of waiting/response time, convoy effect, nonpreemptive, bad for time-sharing system

SJF：分为 preemptive(SRTF) & nonpreemptive.平均等待时间最短。
Nonpreemptive 会把当前正在执行的推掉。最优但不实际，是其他算法的判定标准。会造成 starvation(饥饿)
Approximation: predict the next burst length.
$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$
Each successive term has less weight than its predecessor.

Highest response Ratio Next，最高响应比优先
HRN = (W + T) / T ； W: Waiting Time   T: Burst Cycle Time

Priority: preemptive & nonpreemptive. 存在饥饿的问题，可以通过 aging 的方法来弥补。

Round Robin： q large ⇒ FIFO. q small ⇒ q must be large with respect to context switch, otherwise overhead is too high. No process waits more than (n-1)q time units. higher average turnaround than SJF, but better response. Typical time quanta: 10 to 100 ms. 80% of the CPU bursts should be shorter than the quantum.

other
Multilevel Queue: scheduling between the queues, commonly implemented as fixed-priority preemptive.
Each queue has its own scheduling algorithm.
foreground – RR background – FCFS
Multilevel Feedback-Queue: a process that waits too long is moved to a higher-priority queue. Prevents starvation.

进程同步

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
Atomic operation means an operation that completes in its entirety without interruption.
临界区问题的解决
1.Mutual Exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2.Progress. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.
3.Bounded Waiting. A bound must exist on the number of times that other processes are allowed(被允许而不一定要进入) to enter their critical sections after a process, as made a request to enter its critical section and before that request is granted.

算法 3: Peterson's Solution
```
do {
        flag [i]:= true;
        turn = 1-i;
        while (flag [1-i] and turn == 1-i) ;
                Critical Section
        flag [i] = false;
                Remainder Section
} while (1);
```
分析：满足了互斥、有空让进和有限等待。这个算法能够解决临界区问题。这里，turn 的含义是“当前的‘进入权’给了谁”。而 Flag[i]==ture 的含义为，Pi 已经准备好进入自己的临界区了。

Bakery Algorithm -- n processes
```
boolean choosing[n] = {false};
int number[n] = {0};
// 某一个线程
do
{
    choosing[i] = true; //表示进程 i 正在获取它的排队登记号，保证计算 number[i]的原子操作
    number[i] = max(number[0],number[1],...,number[n - 1]) + 1; //是进程 i 的当前排队登记号。如果值为 0，表示进程 i 未参加排队，不想获得该资源。
    choosing[i] = false; // 获取排队登记号完成
    for(j = 0; j < n; j++)
    {
        while(choosing[j]); // 确保比较的原子性
        while((number[j] != 0) && (number[j],j) < (number[i],i)); // a < c or (a == c and b < d) 理解 (a,b) = (ticket#, process_id#) order
```

```
    }
    // critical section
    number[i] = 0;  // 结束业务。退出排队
    // reminder section
}while(1);
```
Special atomic hardware instructions
基于 TestAndSet
```
boolean TestAndSet(boolean *target)    {
        boolean rv = *target;
        *target = true;
        return rv;
}
boolean lock = false  对于第 i 个进程:
do {
        while (TestAndSet(&lock)) ;
                Critical Section
        lock = false;
                Remainder Section
}while(1);
```
分析：满足了互斥，但不满足有限等待条件。这里某个号数的进程可以一直霸占进入临界区的机会，使得其他号数的进程无限等待。

基于 Swap 的算法:
```
boolean lock = false 对于第 i 个进程
do {
        key = true;
        while (key == true) Swap(lock,key);
                Critical Section
        lock = false;
                Remainder Section
} while(1);
```
分析：满足了互斥，但不满足有限等待条件。这里某个号数的进程可以一直霸占进入临界区的机会，使得其他号数的进程无限等待。

Bounded-waiting mutual exclusion with TestAndSet()
```
boolean waiting[n] = false ;
boolean lock = false ;
do {
        waiting[i]=true;
        key= true;
        while (waiting[i] && key)
                key=TestAndSet(lock);
        waiting[i]=false;
        //critical section;
        j= (i+1) % n ;
    while ((j != i) && !waiting[ j ])
            j= (j+1) % n ;
    if (j == i)    lock = false ;
    else waiting[ j ] = false ;
        //remainder section ;
}while(1)
```
Semaphores 基于信号量的算法:
```
把一个进程 P 屏蔽，从就绪队列转移到 L 上去：wait(S){
        S.value--;
        if (S.value < 0){
                //add this process to S.List;
                block;
        }
}
把一个进程 P 唤醒，从 L 转移到就绪队列中：
signal(S){
        S.value++;
        if (S.value <= 0){
                //remove a process P from S.List;
                wakeup(P);
        }
}
```
S.value>0 表示有 S 个资源可用；
S.value=0 表示无资源可用或表示不允许进程再进入临界区；
S.value<0 则|S.value|表示在等待队列中进程的个数或表示等待进入临界区的进程个数。
wait(S)表示申请一个资源；signal(S)表示释放一个资源。
如果**两个 wait 操作相邻**，那么它们的**顺序至关重要**，而**两个相邻的 signal 操作**的**顺序无关紧要**。一个同步 wait 操作与一个互斥 wait 操作在一起时，同步 wait 操作在互斥 wait 操作前。

Bounded-Buffer Problem
有限缓冲区问题：进程共享一块有界的缓冲区。如何实现安全同步？
采用信号量 full， empty， mutex，
初始值 full = 0, empty = n, mutex = 1。
```
    wait(full/EMPTY); CON/PRO
    wait(mutex);
    ……
    signal(mutex);
    signal(empty/FULL); PRO/CON
    ……
    while (1);
```

Readers-Writers Problem
```
mutex = 1, wrt = 1
//readers
    readcount = 0
    wait(mutex);
    readcount++;
    if (readcount == 1)
            wait(rt);
    signal(mutex);
    //reading is performed
    wait(mutex);
    readcount--;
    if (readcount == 0)
            signal(wrt);
    signal(mutex);
//writers
    wait(wrt);
    //writing is performed
    signal(wrt);
```
Dining-Philosophers Problem
```
semaphore chopstick[5] = 1。
对于第 i 哲学家:
    do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
                eat;
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
                think;
    } while (1);
```
其他解决方法:
1.allow at most four philosophers to be sitting simultaneously at the table--a spare resource.
2.allow a philosopher to pick up chopsticks only if both are available, and pick up them simultaneously.
3.use a asymmetric solution: an odd philosopher picks up first the left chopstick and then the right chopstick, whereas an even one picks up first the right and then the left.
Deadlock and Starvation
Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

死锁问题

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
1.Mutual exclusion: only one process at a time can use a resource.
2.Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. Circular wait: there exists a set {P1, P2, …, Pn} of waiting processes such that P1 is waiting for a resource that is held by P2, P2 is waiting for a resource that is held by P3, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P1.
Handling Deadlocks
1.Prevention 低设备使用率和系统吞吐率
条件 2：必须同时申请所有资源(之后不 wait)；或只有没有资源才可申请(之前不 hold)。资源利用率低 饥饿
条件 3：申请其它资源不能立即得到，那么现有资源可被抢占
适用 cpu 内存 不适用打印机 磁带驱动器
条件 4：资源排序，递增申请
2. Avoidance
3. Detection
4.Recovery
5.Ignore the problem and pretend that deadlocks never occur in the system (Unix)

Resource-Allocation Graph 该算法只适用单个实例资源
If no cycle exists, then the allocation of the resource will leave the system in a safe state.

Safe State
System is in safe state if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of all the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
safe state ⇒ no deadlocks
unsafe state ⇒ possibility of deadlock

Banker's Algorithm(Deadlock Avoidance)
Resource-Request Algorithm
1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2.If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.
3. Pretend to allocate requested resources to Pi by modifying the state as follows:
Available = Available - Requesti
Allocationi = Allocationi + Requesti
Needi = Needi – Requesti
If safe ⇒ the resources are allocated to Pi
If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Safety Algorithm
1.Let Work and Finish be vectors of length m and n, respectively. Initialize:
(a) Work = Available
(b) Finish [i] = false for i = 0, 1, …, n- 1.
2.Find an index i such that both:
(a)Finish[i] == false
(b)Requesti ≤ Work
If no such i exists, go to step 4.
3.Work = Work + Allocationi
Finish[i] = true
go to step 2.
4.If Finish [i] == true for all i, then the system is in a safe state.

Detection Algorithm(死锁检测算法)
Almost the same with safety algorithm
1.(b)For i = 1,2, …, n, if Allocationi ≠ 0, then
Finish[i] = false; otherwise, Finish[i] = true.
4.If Finish[i] == false, for some i, 1 ≤ i ≤ n, then the system is in deadlock state.
Moreover, if Finish[i] == false, then Pi is deadlocked.
The algorithm requires an order of O(m x n2) operations to detect whether the system is in deadlocked state
Wait for 图算法：简化了资源分配图，只适用单个实例资源

内存管理

Binding 地址绑定
Addresses in the program are generally symbolic.
A compiler binds these symbolic addresses to relocatable addresses.
A linkage editor (loader) binds these relocatable addresses to absolute addresses.
1. Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
2. Load time: Must generate relocatable code if memory location is not known at compile time.
3. Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

Logical vs. Physical Address Space
Central to proper memory management
1.logical address – generated by the CPU; also referred to as virtual address
2.physical address – address seen by the memory unit
Differ in execution-time address-binding scheme and same in compile-time and load-time.

Memory-Management Unit(内存管理单元)
MMU is a <u>hardware device</u> that maps virtual to physical address.
In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
The user program deals with logical addresses; it <u>never</u> sees the real physical addresses.

Dynamic Linking
linking postponed until execution time. Particularly useful for (shared) libraries; better memory usage; easier library updates; operating system need to check if routine is in processes' memory address.

Swapping
Major part of swap time is transfer time
Backing store – fast disk large enough

Overlay
An overlay is when a process replaces itself with the code of another program.
法 1：不能换出有待处理 IO 的进程
法 2：IO 只能操作系统缓冲区

1.Contiguous Allocation
Base register contains value of smallest <u>physical address</u>.
Limit register contains <u>range of logical addresses.</u>
Logical address must be less than limit register
MMU maps logical address dynamically

Dynamic Storage-Allocation Problem
First-fit: Allocate the *first* hole that is big enough
Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size; Produces the smallest leftover hole
Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
first fit and best fit are better than worst fit. first fir a little better than best fit.

Fragmentation
External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
Compaction is possible only if relocation is dynamic, and is done at execution time.

2.Paging
Divide physical memory into <u>fixed-sized</u> blocks called <u>**frames**</u> (size is power of 2, 512B~ 8,192B).
Divide logical memory into blocks of same size called <u>**pages**</u>.
Page table- translate logical to physical addresses
Page number (*p*) – used as an index into a *page table* which contains <u>base address</u> of each page in physical memory
Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	*d*
m − *n*	*n*

Implementation of Page Table
Page table is kept in main memory.
Base register (PTBR) points to the page table
Length register (PTLR) indicates size of the table.
Every data/instruction access requires <u>two memory accesses</u>.
One for the page table and one for the data/instruction.
Can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs). EAT = (t+□) □ + (t + t + □) (1 – □)

Memory Protection - Valid-invalid bit
Shared Pages
Shared code must appear in same location in the logical address space of all processes.

Structure of the Page Table
1.Hierarchical Paging (Two-Level Paging)
Page-number(2-level) + Page-offset
2.Hashed Page Tables
The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
3.Inverted Page Tables

3.Segmentation
Segment table – maps 2d physical addresses. 动态链接
Logical address consists <segment-number, offset>. Each table entry has:
base – contains the starting physical address where the segments reside in memory
limit – specifies the length of the segment
STBR and STLR
Allocation: segments vary in length→dynamic storage-allocation problem ; first fit/best fit ; <u>external fragmentation</u>
Sharing : same segment number

Segmentation and paging
Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single segment table entry with a page-table. Paged segmentation handles the case of having very long segments that require a lot of time for allocation. By paging the segments, we reduce wasted memory due to external fragmentation and simplify the allocation.

虚拟存储

Basic Concepts
Virtual memory – separation of user logical memory from physical memory. 当物理内存较小时提供一个较大的外部虚拟存储
1.only part of the program needs to be in memory for execution. 2.logical address space can therefore be much larger than physical address space. 3.allows address spaces to be shared by several processes. 4.allows for more efficient process creation. 5.virtual memory allows other benefits during process creation: copy-on-write ; memory-mapped file.

Demand Paging
Bring a page into memory only when it is needed
1.Less I/O needed 2.Less memory needed
3.Faster response 4.More users

Page Fault
1.Operating system looks at another table to decide:(1)Invalid reference ⇒ abort (2)Just not in memory 2.Get empty frame 3.Swap page into frame 4.Reset tables 5.Set validation bit = **v** 6. Restart the instruction that caused the page fault
Page Fault Rate $0 \le p \le 1.0$. EAT = (1 – *p*) x memory access + *p* (page fault overhead + swap page out + swap page in + restart overhead)

Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory until either process modifies. More efficient process creation.

Page Replacement Algorithm 页置换的算法
Goal - lowest page-fault rate.
Reference string: page numbers only, with adjacent duplications eliminated.
1.**First-In-First-Out** Algorithm - use a FIFO queue to hold all pages in memory. When a page is brought into memory, insert it at the tail. When a free frame is needed, we replace the page at the queue. **Belady's Anomaly**
2.**Optimal** Page Replacement OPT/MIN- replace the page that <u>won't be used for the longest period of time</u>
3.**Least Recently Used** - associates with each page the time of that page's last use. When a page must be replaced, choose the page that <u>has not been used for the longest period of time</u>.
a. <u>Counter implementation</u> - A time-of-use field. Copy the clock into it when referenced. Look for smallest time-of-use field. Difficulties: requires a search of the page table; a write to memory for each memory access; page-table maintain in context switch; overflow of the clock.
b. <u>Stack implementation</u> - Keep a stack of page numbers in a double linked list. Move a page to the top when referenced. Requires 6 pointers to be changed. Always replace the page at the bottom. No search for replacement.
PS: both need special hardware support(TLB).

4.**LRU-Approximation** Page Replacement
Reference Bit - with each page associate a bit, initially = 0, when page is referenced, bit set to 1.
a. <u>Additional Reference Bits</u>
Use right-shift history byte for each page, current reference bit shift into the left-most bit. choose the page with lowest number.
b. <u>Second-Chance Algorithm</u>
FIFO + reference bit. Circular Queue. Inspect the current frame, if the reference bit is set, reset it, and skip to next frame; otherwise, replace it.
c. <u>Enhanced Second-Chance Algorithm</u>
Modify bit (dirty, bit). Replace the first page encountered in the lowest nonempty class. Drawback: may have to scan several times. (Mac)
5.**Counting-based** Algorithms - keep a counter of number of references to each page.
a. Least-frequently-used (LFU) - page with the smallest count is replaced. counter shift right at regular interval , exponentially decaying average.
b. Most-frequently-used (MFU) - based on the argument that page with the smallest count was probably just brought in and has yet to be used.

Frame Allocation
1.Fixed Allocation – Equal or Proportional(比例).
2.Priority Allocation
<u>Global vs Local Replacement</u>.
Global(more common used) - process selects from the set of all frames. High priority process can take a frame from a lower priority process. Results in greater system throughput. Problem: a process cannot control its own page-fault rate.
Local – process selects from only its own set of allocated frames. the number of frames allocated to a process does not change.

Thrashing & Working-Set Model
If a process doesn't have "enough" pages, the page-fault rate is very high ⇒ 1.low CPU utilization 2.os thinks it needs to increase the degree of multiprogramming 3.another process added to the system 4.worse the condition.
<u>Thrashing</u> :≡ A process is busy swapping pages in and out. Σ size of locality > total memory size, thrashing occur. Δ ≡ working-set window ≡ a fixed number of page references. <u>Working-Set Size WSS</u> = total number of pages referenced in the most recent Δ (varies in time). Δ too small will not encompass entire locality. Δ too large will encompass several localities. Δ = ∞ will encompass entire program.
D = Σ WSSi ≡ total demand frames. if D > m ⇒ thrashing. Policy: if D > m, then suspend one of the processes (and swap it out completely).
Working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible, it optimizes CPU utilization.
Implementation: approximate with <u>interval timer</u> + <u>a reference bit.</u>
Example: Δ = 10,000, timer interrupts every 5000 references, keep in memory 2 bits for each page. Whenever a timer interrupts, copy and sets the values of all reference bits to 0. If one of the bits in memory = 1 ⇒ page in working set

Memory-Mapped Files
Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.
1.Simplifies file access by tring file I/O through memory rather than read() write() system calls. 2.Allows several processes to map the same file - pages in memory to be shared.

Allocating Kernel Memory
1.Buddy System - Allocates memory from fixed-size segment consisting of physically contiguous pages. (用高位地址判断是否 Buddy)
2.Slab Allocator - <u>Slab</u> is one or more physically contiguous pages. <u>Cache</u> consists of one or more slabs. Single cache for each unique kernel data structure. When cache created, filled with objects marked as <u>free.</u> When structures stored, objects marked as <u>used</u>. If slab is full of used objects, next object allocated from empty slab. If no empty slabs, new slab allocated. Benefits include no fragmentation, fast memory request satisfaction.

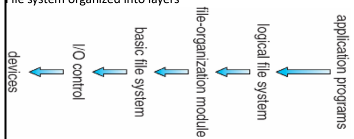
文件系统接口

File – A sequence of logical records.
File Attributes
1.Name 2.Identifier – unique tag identifies file within file system 3.Type 4.Location 5.Size 6.Protection 7.Time, date, and user identification
8.Information about files kept in the directory structure, which is maintained on the disk.
File Operations - 1.Create 2.Write 3.Read 4.Reposition within file 5.Delete 6.Truncate 截短
Open-file table
A file can be opened simultaneously by more than one processes in name of several users
A <u>per-process</u> table tracks all files that a process has open. May have a pointer to an entry in the system-wide table.
A <u>system-wide</u> table contains process -independent information. Have an <u>open count</u> tracks the number of processes.
Info associated with an open file: 1.File pointer 2.File-open count 3.Disk location of the file 4.Access rights
File Types – Name, Extension
Access Methods - Sequential & Direct Access
File Directory
A directory is a collection of nodes containing information about all files. Both the directory structure and the files reside on disk.
Information in dir: name, type, address , current length, maximum length, date last accessed (for archival), date last updated (for dump), owner ID (who pays), protection information.
Operations: 1.Search 2.Create 3.Delete 4.List a directory 5.Rename 6.Traverse the file system
Directory Structure
Goal: 1.Efficiency 2.Naming 3.Grouping
1.<u>Single-Level</u> (2)(3)X limited name length
2.<u>Two-Level</u> (3)X
3.<u>Tree-Structured</u> <u>Absolute or relative</u> path name
4.<u>Acyclic-Graph</u> have shared subdirectories and files. A file may have multiple absolute path names. Distinct file names may refer to the same
Delete file ⇒ dangling pointer. To solve, using (1)Backpointers using a daisy chain (2) Every-entry-count. 保证无圈方法:a.只允许指向文件的链接; b.每次增加 link 时检查有无圈.
5.General Graph – Need garbage collection
File System Mounting
A file system must be mounted before it can be accessed. An unmounted file system is mounted at a mount point.A directory structure can be built out of <u>multiple partitions.</u>
File Sharing
Desirable on multi-user systems. May be done through a <u>protection</u> scheme.
(1)User IDs and Group IDs
(2)Remote File Systems: NFS, CIFS
File Protection
Mode of access: read, write, execute
three classes of users

a) owner access	7	⇒	1 1 1
b) group access	6	⇒	1 1 0
c) public access	1	⇒	0 0 1

文件系统实现

File-System Structure
File structure goal:
1.Logical storage unit - data blocks
2.Collection of related information - FCB
File system organized into layers



File system resides on secondary storage
Device driver controls the physical device
File-System Implementation
On disk:
1.<u>Boot control block</u> contains info needed by system to boot OS from that volume.
boot block(UFS) & partition boot sector(NTFS)
2.<u>Volume control block</u> contains volume details
(# of blocks, free block count and pointers, free FCB count and pointers)
superblock(UFS) & master file table(NTFS)
3.Directory structure organizes the files.
inode numbers(UFS) & master file table(NTFS)
4.Per-file File Control Block (FCB) contains many details about the file.
In memory
1.in-memory mount table
2. in-memory directory structure
3. system-wide open-file table - contains a copy of the FCB of each open files, open count, etc.
4.per-process open-file table - contains a pointer to the entry in the system-wide open-file table, read/write position, etc.
File Control Block
1.file permissions 2.file dates(create, access, write) 3.file owner, group, ACL 4. file size 5. file data blocks or pointers to file data blocks.
File Open
1.the directory structure is searched for the given file name 2.if found, the FCB is copied into a system-wide open-file table in memory 3.an entry is made in the per-process open-file table 4.returns a pointer to the entry in the per-process open-file table -- file descriptor in UNIX, file handle in Windows
File Close
1.the per-process table entry is removed. 2.the system-wide table entry's open count is decremented. if the count hits zero, all updated info is copied back to the disk, and the system-wide table entry is removed.
Virtual File Systems
Provides an object-oriented way of implementing file systems.
Allows the same system call interface(API) to be used for different types of file systems. The API is to the VFS interface, rather than any specific type of file system.
Three Layers: 1. file-system interface – based on open, close, read, write and on file descriptors.
2. virtual file system.
Directory Implementation
1.<u>Linear list</u> of file names with pointer to the data blocks 2.<u>Hash Table</u> – linear list with hash data structure
Allocation Methods 磁盘分配方法
Contiguous allocation
Benefits: simple, only starting location and length (number of blocks) are required; good support of random access.
Problem: dynamic storage-allocation - first fit, best fit; external fragmentation – compaction.
Difficulty: files grow, difficult to determine how large a file.
Extent-Based Systems - a modified contiguous allocation scheme. It allocate disk blocks in extents; an extent is a contiguous block on the disk; extents are allocated for file allocation; a file consists of one or more extents.
LA/512 Block X + start address Displacement Y
Linked allocation
Benefits: simple, need only starting address in directory entry; no external fragmentation.
Drawbacks: no random access, can be used effectively only for sequential-access files; space required for the pointers in every block - use clusters(簇) 按簇(几个块一起)分配,减少指针消耗,但会增加内碎片
Reliability: pointers scatter all over the disk, and may lost or damage - double FAT!
LA/511 Block X Displacement Y+1
Indexed allocation
brings all pointers into the index block; directory entry contains the address of the index block.
Benefits: support random access; no external fragmentation
Drawbacks: waste space(whole block may contains one or two pointers)
Index table of variant size files
1.Linked scheme 2.Multilevel index 3.combined scheme(12+256+256^2+256^3)
LA/512 X =displacement in index table
Y=displacement in block
File Allocate Table(FAT)
Disk-space allocation used by MS-DOS and OS/2
A section at the beginning of each partition is set aside to contain the table. A FAT is used as a linked list. 0 means free block; a special value (-1) means end-of-file; double FAT to increase the reliability. FAT cause a lot of disk <u>head seeks</u>.
FAT is usually cached in memory; random access time is improved, we can find the location of any block by reading the FAT (provided it is cached in memory). 类似于游标的一种实现.
Free-Space Management 空闲空间管理
1.<u>bit map, or bit vector (n blocks)</u>
Each block is represented by 1 bit(1 free 0 occupied). Bit map requires extra space.
First free block number calculation=(number of bits per word) *(number of 0-value words) +offset of first 1 bit
Ex: block size = 512B disk size = 1GB *n* = 1GB / 512B = 2M = 256KB clustering blocks in group (say, of four blocks) reduces the number to 64KB.
Easy to get contiguous files
2.<u>Linked Free-space List</u>
Cannot get contiguous space easily; no waste of space; an FAT implementation may help to improve performance; FIFO or stack?

磁盘系统

Transfer rate is rate at which data flow between drive and computer.
<u>Positioning time</u> (<u>**random-access time**</u>) is time to move disk arm to desired cylinder (<u>**seek time**</u>) and time for desired sector to rotate under the disk head (<u>**rotational latency**</u>)
Drive attached to computer via <u>I/O bus</u>
1.Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI.
2.Host controller in computer uses bus to talk to disk controller built into drive or storage array.
Moving-head Disk Mechanism
A read-write head "flies" just above each surface of every platter. The heads are attached to a <u>disk arm</u> that moves all the heads as a unit. The surface of a platter is logically divided into circular <u>tracks</u>, which are subdivided into <u>sectors</u>. The set of tracks that are at one arm position makes up a <u>cylinder</u>.
Disk Structure
Disk drives are addressed as large 1-dimensional arrays of <u>logical blocks</u>, where the logical block is the smallest unit of transfer - typical called a sector, 512B or 1KB
The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
1.sector 0 is the first sector of the first track on the outermost cylinder.
2.mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.
Disk Scheduling
The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
Access time has two major components
1.<u>Seek time</u> is the time for the disk are to move the heads to the cylinder containing the desired sector. 2.<u>Rotational latency</u> is the additional time waiting for the disk to rotate the desired sector to the disk head
<u>Minimize seek time</u>
<u>Seek time ≈ seek distance</u>
Disk <u>bandwidth</u> is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
Disk Scheduling Algorithm 磁盘调度算法
1.**FCFS – First Come First Service**
2.**SSTF** – 选寻道时间最短的. May cause starvation
3.**SCAN**(elevator algorithm) The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues. 扫到头
4.**C-SCAN** – 基于 SCAN，磁盘是圆形的，从一头倒另一头很快并且不执行 JOB. 扫到头，快速扫回
5.**LOOK** 扫到有 JOB 的 c 最远的位置，不用到头
6.**C-LOOK** 快速扫回，注意不是环形，只是快速扫回，中途不干活，比如设备只能单向存取.
Disk Management
1.Low-level formatting, or physical formatting — Dividing a disk into sectors that the disk controller can read and write.
2.To use a disk to hold files, the operating system still needs to record its own data structures on the disk. a. Partition the disk into one or more groups of cylinders. b. Logical formatting or "making a file system" c. To increase efficiency most file systems group blocks into clusters. (1)Disk I/O done in blocks. (2)File I/O done in clusters
Master Boot Record
RAID Structure
RAID – multiple disk drives provides reliability via <u>redundancy</u>. Increases the mean time to failure
Frequently combined with NVRAM(Non-volatile RAM) to improve write performance. RAID is arranged into six different levels.
RAID 0: non-redundant striping.

RAID 1: mirrored disks.
RAID 2: memory-style error correcting codes.
RAID 3: bit-interleaved parity.
RAID 4:block-interleaved parity.
RAID 5: block-interleaved distributed parity.
RAID 6: P + Q redundancy.

I/O 系统

An I/O channel is a special processor.
Virtual devices are implemented with **spooling**!

Basic I/O Concepts

device drivers– present a uniform device access interface to the I/O subsystem

I/O subsystem – separates the rest of the kernel from the complexity of managing I/O devices

Port – a device communicates with the machine via a connection point, e.g. a serial port

Bus – a set of wires and a protocol

Controller– a collection of electronics that operate a port, a bus, or a device: the processor communicates with the controller by reading and writing bit patterns in data and control registers; **Special I/O instruction or Memory-mapped I/O**.

Polling 轮询

Determines state of device. Command-ready; busy; error. Inefficient when speed differ much.

Interrupt 中断

1.CPU interrupt request line triggered by I/O device.
2.**Interrupt handler** receives interrupts.
3. Maskable to ignore or delay some interrupts
Two lines: Nonmaskable and maskable interrupt.
4.Interrupt vector to dispatch interrupt to correct handler: based on priority; Some nonmaskable.
5.Interrupt mechanism also used for exceptions.

DMA(Direct Memory Access)

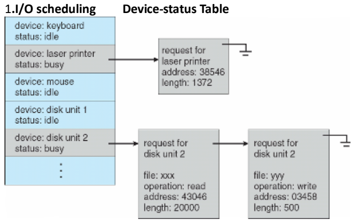
1.Used to avoid programmed I/O for large data movement
2.requires DMA controller- a special-purpose processor.
3.Bypasses CPU to transfer data directly between I/O device and memory. Handshake between the main processor and the DMA controller: two wires - DMA-request wire and DMA-acknowledge wire

Application I/O Interface I/O 应用接口

1.I/O system calls encapsulate device behaviors in generic classes. 2.Device-driver layer hides differences among I/O controllers from kernel.
Block and Character Devices   Network Devices
Clocks and Timers Blocking and Nonblocking I/O

Kernel I/O Subsystem

1.I/O scheduling   **Device-status Table**



2.Buffering

1.To cope with a speed mismatch between the producer and consumer of a data stream.
2.To adapt between devices that have different data-transfer sizes.
3.To support copy semantics for application I/O.

3. Caching

Fast memory holding copy of data. Always just a copy. More efficient than access to the original.
Difference between caching and buffering: a buffer may hold the only copy of data whereas a cache always holds a copy on faster storage.

4.Spooling and Device Reservation

A buffer that holds output for a device, such as a printer, that cannot accept interleaved(交叉) data streams. Spooling is one way OS can coordinate concurrent output.
Device Reservation provides exclusive access to a device. System calls for allocation and deallocation. Watch out for deadlock.

5.error handling

6.I/O Protection

User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions. a. All I/O instructions defined to be privileged(特权). b. I/O must be performed via system calls - Memory-mapped and I/O port memory locations must be protected too.

I/O port registers

1.status register 2.control register
3.data-in register(s) 4.data-out register(s)

Kernel I/O subsystem summary

1. management of name of files and devices
2. access control to files and devices
3. file system space allocation
4. device allocation (including reservation)
5. buffering, caching, spooling
6. I/O scheduling
7. device status monitoring, error handling
8. device driver configuration