

Linux 程序设计总结

2017-yyf

目录

Chapter1 Linux 基础	3
文件系统（File System）	3
Boot Loader.....	3
虚拟终端（Vitual Terminal）	3
命令提示.....	3
UNIX 结构	5
重定向.....	5
管道.....	6
环境变量.....	6
常见环境变量.....	6
Chapter2.....	6
执行脚本文件.....	6
用户变量.....	7
引号的使用.....	7
参数变量和内部变量.....	7
条件测试.....	7
基本算术运算（需要计算 <code>var1=var2+*/var3</code> 这种）	8
<code>\$()</code> 、 <code>`</code> （反引号）捕获命令输出	8
条件语句.....	9
循环语句.....	9
命令组合.....	10
函数.....	10
杂项命令.....	10
Chapter 3.....	11
gcc	11
make 命令.....	11
编译连接过程.....	12
虚拟文件系统（VFS）	12
软连接（符号链接）、硬链接.....	13
系统调用与库函数.....	13
基础 I/O 系统调用	13
open/create.....	13
close	14
read/write	15
lseek	15
dup/dup2	16
fcntl	16
标准 I/O 库	17
setbuf/setvbuf.....	17

fopen/fclose	17
getc/fgetc/getchar	18
putc/fputc/putchar	18
fgets/gets.....	18
fputs/puts.....	19
fread/fwrite	19
scanf/fsacnf/sscanf.....	19
printf/fprintf/sprintf.....	20
fseek/ftell/rewind fgetpos/fsetpos	20
fflush	20
fileno/fdopen	20
tmpnam/tmpfile.....	21
stat/fstat/lstat	21
access	22
chmod/fchmod.....	22
chown/fchown/lchown	22
umask	23
link/unlink	23
symlink/readlink.....	23
mkdir/rmdir.....	23
chdir/fchdir.....	24
getcwd.....	24
opendir/closedir/readdir/telldir/seekdir	24
lockf.....	25
Chapter 4.....	26
内核简介.....	26
内核结构.....	26
Linux 内核模块与应用程序区别	26
开发驱动的注意事项.....	27

Chapter1

文件系统（File System）

- 操作系统中负责存取和管理文件的部分
- 一个文件及其某些属性的集合。它为这些文件的序列号（file serial numbers）提供了一个名称空间
- 类型
 - ◆ VFS，虚拟文件系统，与以下的磁盘文件系统（即文件的分区格式不同），为底层的文件系统提供了统一的抽象。（更多内容参见 Chapter3 VFS）
 - ◆ EXT2、EXT3、FAT32

Boot Loader

- Boot Loader 加载和启动 Linux 系统内核。Linux 下的 Boot Loader 为 GRUB。

虚拟终端（Vitual Terminal）

Linux 下有 6 个虚拟终端 VT1-6

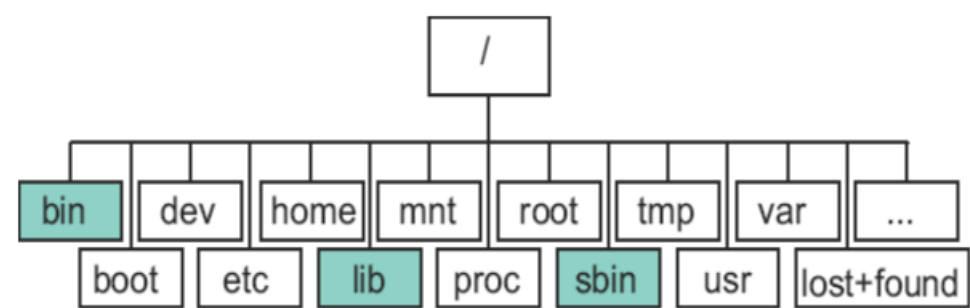
命令提示

- 可以自行配置
- \$: 普通用户
- #: Root 用户

文件和目录

- 文件：
 - ◆ 文件是一数据的集合。
 - ◆ 文件结构：字节流、记录序列（Record Sequence）、记录树（Record Tree）。Linux 下为字节流
 - ◆ 文件类型：
 - （-）普通文件：纯文本文档、二进制文件、数据格式文件等
 - （c）字符设备文件：与设备进行交互的文件。Linux 中所有的设备都被抽象为了对应的文件。字符设备是按字符读取。
 - （b）块设备文件：同字符设备文件，但是是按块读取。
 - （p）数据输送文件（FIFO, pipe）。他主要的目的在解决多个程序同时存取一个文件所造成的错误问题。
 - （s）socket 文件。我们可以启动一个程序来监听客户端的要求，而客户端就可以通过这个 socket 来进行数据的沟通。如启动 Mysql 服务会创建一个对应的 socket 文件。一般在/var/run 目录中
 - （l）符号链接
 - （d）目录
- 目录结构
 - ◆ Linux 中所有的目录均包含在一个统一的、虚拟的统一文件系统（Unified File System）中。

- ◆ 物理设备被挂载到对应挂载点，抽象为一个文件。没有类似于 C:/ 的驱动盘符
- ◆ 根目录下文件夹的作用：



目录	描述
/bin	必要的命令二进制文件。 包含了会被系统管理者和用户使用的命令，但是需要在没有任何文件系统挂载时使用。也可能会在脚本文件中间接使用。
/boot	Boot Loader 相关的静态文件。 包含了所有需要在加载阶段使用的文件（不含一些不需要在启动阶段用到的配置文件）。保存了所有内核在执行用户态程序之前用到的数据。
/dev	设备文件。是一些特殊或设备的文件。
/etc	主机相关的系统配置。主要包含了配置文件。
/lib	必要的共享二进制文件或内核模块。比如在/bin 或/sbin 里需要用到的库
/media	可删除 media 的挂载点。如软盘等。
/mnt	临时文件系统的挂载点。
/opt	额外的应用软件包安装目录
/sbin	必要的系统二进制文件
/srv	系统提供的有关服务的数据
/tem	临时文件
/usr	第二级层次树。/usr 是共享的、只读的数据。大型的软件包不应该使用/usr 下的层次
/var	可变数据
可选的目录	
/home	用户 home 目录
/lib<qual>	必要共享二进制文件的可选格式
/root	ROOT 用户的 HOME 目录

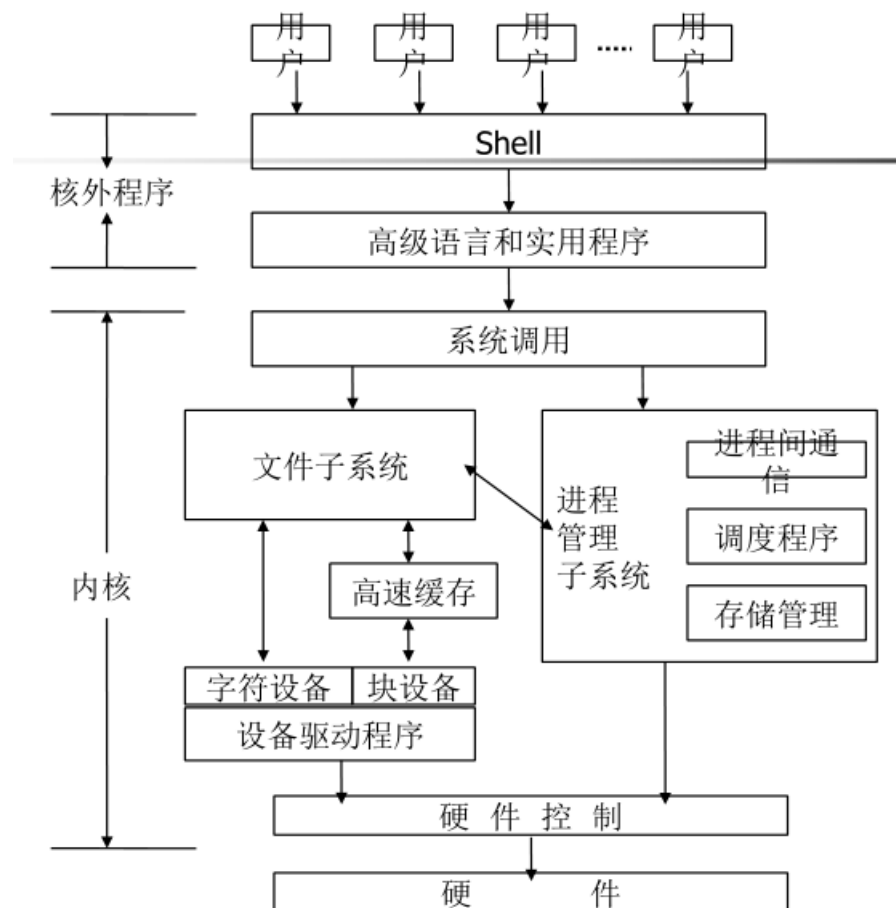
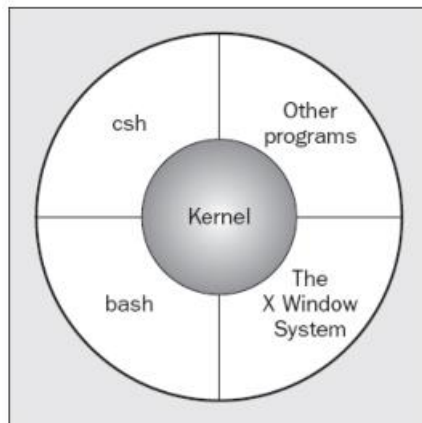
文件权限

- 文件权限分为三个层次：用户、组、其他用户
- 三个类型：读、写、执行
- 更改权限：参见 chmod

进程

- 进程是一个正在执行的程序实例。由执行程序、它的当前值、状态信息以及通过操作系统管理此进程执行情况的资源组成。

UNIX 结构



重定向

- 标准输入、标准输出、标准错误
 - ◆ 对应的文件描述符: 0, 1, 2
 - ◆ C 语言变量: stdin, stdout, stderr
- < > >> >!
 - ◆ > 输出重定向到一个文件或设备 覆盖原来的文件

- ◆ >! 输出重定向到一个文件或设备 强制覆盖原来的文件
- ◆ >> 输出重定向到一个文件或设备 追加原来的文件
- ◆ < 输入重定向到一个程序

ls -al > list.txt	将显示的结果输出到 list.txt 文件中，若该文件以存在则覆盖
ls -al >> list.txt	将显示的结果输出到 list.txt 文件中，若该文件以存在则追加
ls -al 1> list.txt 2> list.err	将显示的数据，正确的输出到 list.txt 错误的数据输出到 list.err

管道

- 一个进程的输出作为另一个进程的输入
- 使用 |
- 例子
 - ◆ ls | wc -l
 - ◆ ls -lF | grep ^d
 - ◆ ar t /usr/lib/libc.a | grep printf | pr -4 -t

环境变量

- 操作环境的参数
- 查看和设置环境变量：
 - ◆ echo 显示环境变量 echo \$PATH
 - ◆ env 显示系统所有的环境变量
 - ◆ set 显示本地定义的 shell 变量
 - ◆ export 设置新的环境变量 export HELLO="hello"

常见环境变量

- HOME 用于保存注册目录的完全路径名。
- PATH 用于保存用冒号分隔的目录路径名，Shell 将按 PATH 变量中给出的顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行。
- TERM 终端的类型。
- UID 当前用户的识别字，取值是由数位构成的字串。
- PWD 当前工作目录的绝对路径名，该变量的取值随 cd 命令的使用而变化。
- PS1 主提示符，在特权用户下，默认的主提示符是#，在普通用户下，默认的主提示符是\$。

Chapter2

执行脚本文件

- \$sh script_file
- chmod +x script_file（增加可执行权限）

- `./script_file`
- `source script_file` 或 `.script_file`

用户变量

- 用户在 shell 脚本里定义的变量
- 赋值：
`var=value`（注意不能有任何空格）
- 使用：
`$var` 或 `${var}`
- 删除
`unset var`（不能加\$）
- `read` 命令
使用 `read` 从标准输入中读取一行数据
`read name`
可以使用提示
`read -p "Enter your name:" name`
更多内容参照命令 `read`

引号的使用

- 单引号内所有的字符都是其字面意思。
- 双引号内，除了\$、`（弯弯上面的）和\外，所有字符保持本身含义

参数变量和内部变量

- 参数变量和内部变量：
 - 调用脚本程序时如果带有参数，对应的参数和额外产生的一些变量。

环境变量	说明
<code>\$#</code>	传递到脚本程序的参数个数
<code>\$0</code>	脚本程序的名字
<code>\$1, \$2, ...</code>	脚本程序的参数
<code>\$*</code>	一个全体参数组成的清单，它是一个独立的变量，各个参数之间用环境变量IFS中的第一个字符分隔开
<code>\$@</code>	" <code>\$*</code> "的一种变体，它不使用IFS环境变量。

条件测试

`test expression` 或 `[expression]`（注意 `expression` 前后需要有空格）或 `((expression))`，但是 `(())` 里只能使用 C 风格的比较（`<` `<=` `>` `>=` `!=`）（无法使用 `-eq` 等比较命令）

字符串比较	结果
str1 = str2	两个字符串相同则结果为真
str1!=str2	两个字符串不相同则结果为真
-z str	字符串为空则结果为真
-n str	字符串不为空则结果为真

算术比较	结果
expr1 -eq expr2	两个表达式相等则结果为真
expr1 -ne expr2	两个表达式不等则结果为真
expr1 -gt expr2	expr1 大于 expr2 则结果为真
expr1 -ge expr2	expr1 大于或等于 expr2 则结果为真
expr1 -lt expr2	expr1 小于 expr2 则结果为真
expr1 -le expr2	expr1 小于或等于 expr2 则结果为真

文件条件测试	结果
-e file	文件存在则结果为真
-d file	文件是一个子目录则结果为真
-f file	文件是一个普通文件则结果为真
-s file	文件的长度不为零则结果为真
-r file	文件可读则结果为真
-w file	文件可写则结果为真
-x file	文件可执行则结果为真

逻辑操作	结果
! expr	逻辑表达式求反
expr1 -a expr2	两个逻辑表达式 “And”（“与”）
expr1 -o expr2	两个逻辑表达式 “Or”（“或”）

基本算术运算（需要计算 var1=var2+*/var3 这种）

使用 let 命令

```
let result=num1+num2（变量前不需要添加$）
```

使用\$[]操作符

```
result=$((num1+num2))
```

使用\$()

```
result=$((num1+num2))
```

\$(), ``（反引号）捕获命令输出

命令组，在里面命令的将会优先执行，并返回执行结果

```
$(ls *.sh)
```

\${}参数扩展

Parameter Expansion	Description
<code>\${param:-default}</code>	If param is null, then set it to the value of default.
<code>\${#param}</code>	Gives the length of param
<code>\${param%word}</code>	From the end, removes the smallest part of param that matches word and returns the rest
<code>\${param%%word}</code>	From the end, removes the longest part of param that matches word and returns the rest
<code>\${param#word}</code>	From the beginning, removes the smallest part of param that matches word and returns the rest
<code>\${param##word}</code>	From the beginning, removes the longest part of param that matches word and returns the rest

条件语句

- if 语句:

形式

```
if [ expression ]
then
    statements
elif [ expression ]
then
    statements
elif ...
else
    statements
fi
```

注意: shell 中表达式为 0 表示真, **expression** 为 0 时经过 then (与 C 相反), 在使用函数确定真假是注意返回值为 0 的时候表示真

- case 语句:

```
case str in
    str1 | str2) statements;;
    str3 | str4) statements;;
    *) statements;;
esac
```

循环语句

- for 语句

```
for var in list
do
    statements
done
```

这种类型适合对一系列字符串进行处理, 通常会结合一些基本命令

ps: 也可以写为 C 风格

```
for ((i=1;i<5;i++))
do
```

```

        statements
    done
while 语句
    while condition
    do
        statements
    done
until 语句
    until condition
    do
        statements
    done          （不建议使用）
select 语句
    select item in itemlist
    do
        statements
    done          （生成菜单列表）

```

命令组合

分号串联。

command1;command2 依次执行

条件组合。（短路特性）

AND 命令。command1&&command2 执行 command1 成功之后才执行 command2

OR 命令。command1||command2 执行 command1 失败之后才执行 command2

函数

声明

```

func_name(){
    statements
    [return]
}

```

调用

func_name [paras]

函数内部获取参数

使用\$1 \$2 获取参数，同 shell 脚本类似的变量

获取返回值

执行函数后使用\$?可以获得返回值

杂项命令

break: 跳出当前循环

continue: 调到下一次循环

exit n: 以退出码 n 退出脚本执行

return: 函数返回

export: 将变量导出到 shell，使之成为 shell 的环境变量

set: 为 shell 设置参数变量

unset: 从环境中删除变量或函数

trap: 指定在收到操作系统信号后执行的动作

“.”(冒号命令): 空命令

“.”(句点命令)或 source: 在当前 shell 中执行命令

Chapter 3

gcc

可用于编译（gcc -c）、链接（gcc -o）

Usage:

- gcc [options] [filename]

Basic options:

- -E: 只对源程序进行预处理(调用cpp预处理器)
- -S: 只对源程序进行预处理、编译
- -c: 执行预处理、编译、汇编而不链接
- -o output_file: 指定输出文件名
- -g: 产生调试工具必需的符号信息
- -O/On: 在程序编译、链接过程中进行优化处理
- -Wall: 显示所有的警告信息

- -Idir: 指定额外的头文件搜索路径
- -Ldir: 指定额外的库文件搜索路径
- -lname: 链接时搜索指定的库文件
- -DMACRO[=DEFN]: 定义MACRO宏

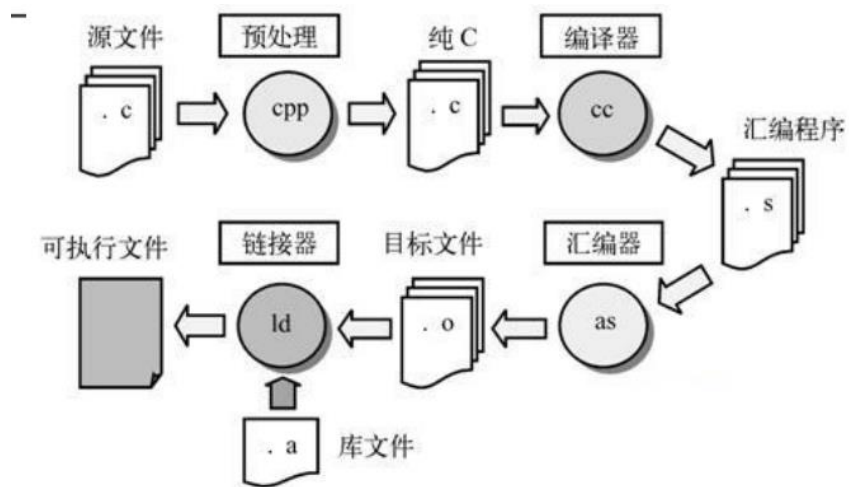
make 命令

makefile 带来的好处就是——“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。

（不会重点考，稍微了解了解，可以参考

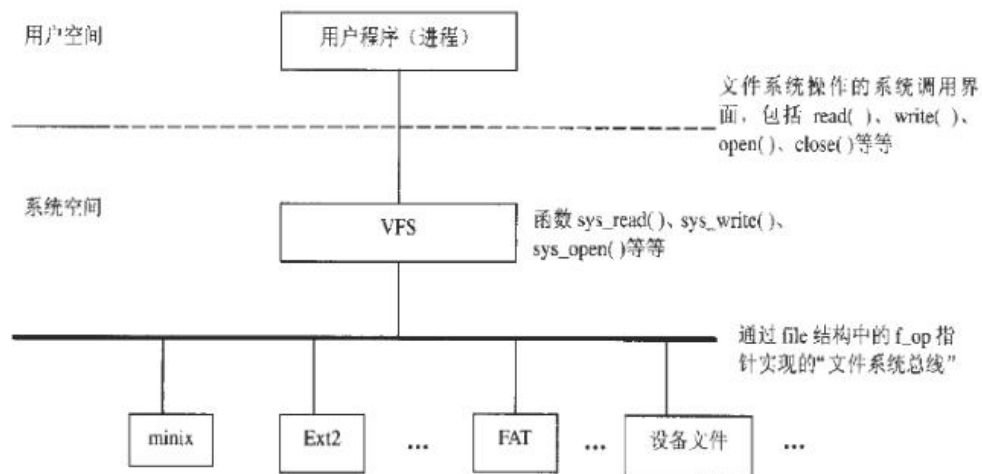
<http://blog.csdn.net/ruglcc/article/details/7814546/>）

编译连接过程



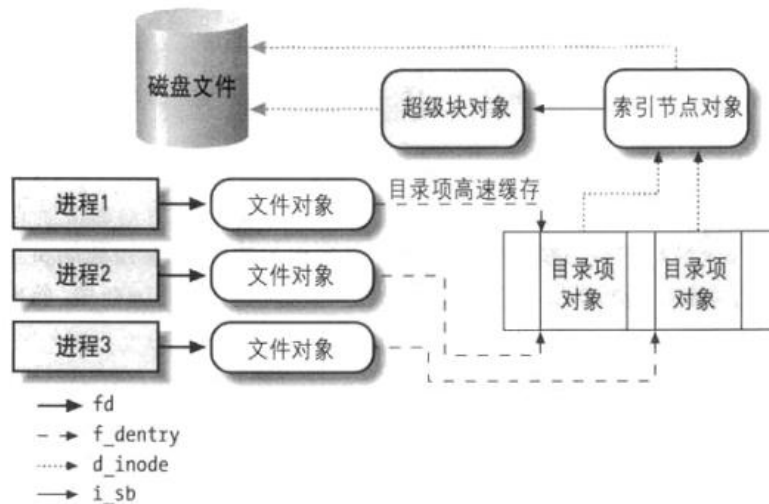
虚拟文件系统（VFS）

- 结构



- VFS Model

- ◆ 只存在与内存之中
- ◆ 分为 4 个部分：超级块（Super block）、索引节点对象（i-node Object）、文件对象（File Object）、目录项对象（Dentry Object）



软连接（符号链接）、硬链接

■ Hard link

- 不同的文件名对应同一个inode
- 不能跨越文件系统
- 对应系统调用link

■ Symbolic link

- 存储被链接文件的文件名(而不是inode)实现链接
- 可跨越文件系统
- 对应系统调用symlink
- 软连接类似于 windows 下的快捷方式，删除原文件后（原文件没有任何硬链接文件）软连接不可用。
- 硬链接创建 `ln [原文件名] [连接文件名]`
- 符号链接 `ln -s [原文件名] [连接文件名]`

系统调用与库函数

- 都以 C 函数的形式出现
- 系统调用。Linux 内核的对外接口；用户程序和内核之间唯一的接口；提供最小接口。需要切换到内核进行相关操作。编译运行速度快。
- 库函数。依赖于系统调用，提供复杂的功能。可移植性好。如：标准 I/O 库。

基础 I/O 系统调用

- 文件描述符。int fd。一个非负整数。标准输入（**STDIN_FILENO**）、标准输出（**STDOUT_FILENO**）、标准错误（**STDERR_FILENO**）对应的文件描述符依次为 0, 1, 2。

open/create

- ◆ 作用：打开或创建一个文件，并获得对应文件的文件描述符
- ◆ 原型

Open and possibly create a file or device

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
(Return: a new file descriptor if success; -1 if failure)
```

◆ 参数

- **pathname**: 文件名
- **flags**: 文件打开模式, 表示以什么方式打开, 可以是 **O_RDONLY**、**O_WRONLY**、**O_RDWR**, 依次表示只读、只写、读写中的一个。以上 3 个模式可以和一些附加模式做按位或 (|) 运算, 添加其他的模式。如:
 - **O_APPEND**: 以追加模式打开
 - **O_TRUNC**: 若文件存在, 则长度被截为 0, 属性不变。(覆盖模式)
 - **O_CREAT**: 如果文件不存在, 则创建它
 - **O_EXCL**: 同 **O_CREAT** 一起使用, 使得当文件存在的时候会出现错误。
 - **O_NONBLOCK**: 对于设备文件, 以 **O_NONBLOCK** 方式打开可以做非阻塞 I/O (Nonblock I/O)。

创建一个文件等价于用 **O_CREAT|O_WRONLY|O_TRUNC** 模式打开文件

- **mode**: 当创建一个文件时指定文件的权限。值为一个无符号整数, 低 9 位确定权限 (同 **chmod** 里使用的值)。

取值	含义
S_IRUSR(00400)	Read by owner
S_IWUSR(00200)	Write by owner
S_IXUSR(00100)	Execute by owner
S_IRWXU(00700)	Read, write and execute by owner
S_IRGRP 00040	Read by group
S_IWGRP 00020	Write by group
S_IXGRP 00010	Execute by group
S_IRWXG 00070	Read, write and execute by group
S_IROTH 00004	Read by others
S_IWOTH 00002	Write by others
S_IXOTH 00001	Execute by others
S_IRWXO 00007	Read, write and execute by others

close

- ◆ 作用: 关闭文件描述符, 释放文件资源。

◆ 原型

```
#include <unistd.h>
int close(int fd);
(Return: 0 if success; -1 if failure)
```

◆ 参数

- **fd**: 需要关闭的文件描述符

read/write

- ◆ 作用：根据指定文件描述符的文件读取/写入文件内容
- ◆ 原型
 - Read from a file descriptor

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

(返回值: 读到的字节数, 若已到文件尾为0, 若出错为-1)
 - Write to a file descriptor

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

(返回值: 若成功为已写的字节数, 若出错为-1)
- ◆ 参数
 - fd: 对应的文件描述符
 - buf: 待读取/写入文件的缓冲区
 - count: 读取/写入文件的字节数
- ◆ 使用
 - ```
while ((n = read(STDIN_FILENO, buf, BUFSIZE)) > 0)
 if (write(STDOUT_FILENO, buf, n) != n)
 err_sys("write error");
if (n < 0)
 err_sys("read error");
```

## lseek

- ◆ 作用：设置读写文件的偏移
- ◆ 原型

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

(Return: the resulting offset location if success; -1 if failure)
- ◆ 参数
  - fildes: 对应的文件描述符
  - offset: 偏移量
  - whence: 偏移的方式, 有 3 个取值
    - SEEK\_SET: 偏移到 offset 位置处 (相对文件头)
    - SEEK\_CUR: 偏移到当前位置+offset 位置处;
    - SEEK\_END: 偏移到文件尾+offset 位置处;
- ◆ 使用:
  - 返回当前的偏移量

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```
  - 返回文件大小

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_END);
```
  - 拓展文件, 主要使用场景是先扩展一个文件空间, 然后再填充内部的内容

(如一些下载工具)

```
off_t file_size = 1024*8;
lseek(fd, file_size-1, SEEK_END)
write(fd, "0", 1);
```

## dup/dup2

- ◆ 作用: 复制文件描述符。**dup** 产生一个相同的文件描述符指向同一个文件; **dup2** 复制一个旧的文件描述符到新的文件描述符, 使得新的文件描述符与旧的文件描述符完全一样, 过程主要是先关闭新的文件描述符对应的文件, 然后进行复制。
- ◆ 原型:

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
(Return: the new file descriptor if success; -1 if failure)
```
- ◆ 参数:
  - **oldfd**: 旧的文件描述符
  - **newfd**: 新的文件描述符 (目标文件描述符)
- ◆ 使用:
  - 通常用于重定向, 与 **STDERR\_FILENO**、**STDIN\_FILENO**、**STDOUT\_FILENO** 结合使用。

```
fd2=dup(STDOUT_FILENO); //保存标准输出
fd = open(filename, O_WRONLY|O_CREAT, fd_mode); //打开文件
dup2(fd, STDOUT_FILENO); //把输出重定向到 fd 标识的文件
close(fd);
```

## fcntl

- ◆ 作用: 操作一个文件描述符, 改变一个已经打开的文件的属性
- ◆ 原型:

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
(返回值: 若成功则依赖于cmd, 若出错为-1)
```
- ◆ 参数:
  - **fd**: 对应的文件描述符
  - **cmd**: 具体的操作, **fcntl** 进行的操作依赖于 **cmd**。有以下取值:
    - **F\_DUPFD**: 复制文件描述符, 返回新的文件描述符。
    - **F\_GETFD**/**F\_SETFD**: 获取/设置文件描述符标志 (目前就只有 **close-on-exec** 标记)。这个标志的目的是解决 **fork** 子进程执行其他任务 (使用 **exec**、**excel** 等命令) 导致了父进程的文件描述符被复制到子进程中 (实际子进程不需要), 使得对应文件描述符无法被之后需要的进程获取。设置了这个标记后可以使得子进程在执行 **exce** 等命令时释放对应的文件描述符资源。



- **F\_GETFL/F\_SETFL**: 获得/设置文件状态标志 (`open/creat` 中的 `flags` 参数), 目前只能更改 `O_APPEND`, `O_ASYNC`, `O_DIRECT`, `O_NOATIME`, `O_NONBLOCK`
- **F\_GETOWN/F\_SETOWN**: 管理 I/O 可用相关的信号。获得或设置当前文件描述符会接受 `SIGIO` 和 `SIGURG` 信号的进程或进程组编号
- **F\_GETLK/F\_SETLK/F\_SETLKW**: 获得/设置文件锁, 设置为 **F\_GETLK** 时需要传入 `flock*` 指针用于存放最后的锁信息。**S\_SETLK** 需要传入 `flock*` 指针表示需要改变的锁的内容, 如果不能被设置, 则立即返回 `EAGAIN`。  
**S\_SETLKW** 同 **S\_SETLK**, 但是在锁无法设置时会阻塞等待任务完成。
- ◆ 说明: 文件锁分为记录锁 (按记录加锁)、劝告锁 (检查, 加锁由应用程序自身控制, 不会强制应用程序不准访问)、强制锁 (检查, 加锁由内核控制; 影响 `open`、`read`、`write`)、共享锁 (可读)、排他锁 (读写均不可)
- ◆ 相关结构体:
 

```
struct flock{
 ...
 short l_type; /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
 short l_whence; /* How to interpret l_start: SEEK_SET, SEEK_CUR,
 SEEK_END */
 off_t l_start; /* Starting offset for lock */
 off_t l_len; /* Number of bytes to lock */
 pid_t l_pid; /* PID of process blocking our lock (F_GETLK only) */
 ...
}
```

## 标准 I/O 库

- 使用 `FILE *` 指向文件流 (类似于 `fd` 的作用)。预定义 3 个指针: 标准输入 (`stdin`), 标准输出 (`stdout`), 标准错误 (`stderr`)
- 3 种缓冲
  - 块缓冲 (全缓冲, `full buffered`, `block buffered`)
  - 行缓冲 (`line buffered`)
  - 无缓冲 (`unbuffered`)

### setbuf/setvbuf

- 作用: 设置文件缓冲区
- 原型:
 

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```
- 参数:
  - ◆ `stream`: 文件指针
  - ◆ `buf`: 对应的缓冲区
  - ◆ `mode`: 缓冲区类型, `_IOFBF` (满缓冲) `_IOLBF` (行缓冲) `_IONBF` (无缓冲)
  - ◆ `size`: 缓冲区内的字节数
- 使用:

### fopen/fclose

- 作用: 打开/关闭一个文件流

■ 原型:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *stream);
```

■ 参数:

- ◆ filename: 文件名
- ◆ mode: 打开的模式
  - "r": 只读方式打开
  - "w": 以覆盖方式写
  - "a": 以追加方式写
  - "r+": 以读写方式打开
  - "w+": 以读写方式打开, 文件不存在则创建; 覆盖方式写
  - "a+": 以读写方式打开, 文件不存在则创建; 追加方式写
- ◆ stream: 需要关闭的文件流

### getc/fgetc/getchar

- ◆ 作用: 从文件中读取下一个字符, 若到文件尾或出错, 则返回 EOF
- ◆ 原型:

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

(Result: Reads the next character from a stream and returns it as an **unsigned char cast to an int**, or **EOF** on end of file or error.)

### putc/fputc/putchar

- ◆ 作用: 写入一个字符
- ◆ 原型:

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

(Return: the character if success; -1 if failure)

### fgets/gets

- ◆ 作用: 读取一行字符串。fgets 最多读取 size-1 个字符, 并将其保存在 s 指向的缓冲区中, 遇到文件尾或换行符停止。在缓冲区的最后添加了'\0'字符。
- ◆ 原型:

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
char *gets(char *s); //not recommended.
```
- ◆ 参数:
  - s: 缓冲区指针
  - size: 大小
  - stream: 文件流

## fputs/puts

- ◆ 作用：批量写入字符，写入字符串。
- ◆ 原型：

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

- I/O 效率问题

使用缓冲可以减少读取文件的时间。在缓冲区较小的时候，增加缓冲区大小可以显著减小读取时间，因为这时候上下文切换（切换内核/用户态）次数过多，是主要影响读取时间的因素。当缓冲区大小到一定大小时，读取时间几乎不变，这是因为上下文切换已经不是主要影响因素，而主要耗时的操作在数据的读取的固定时间上。

## fread/fwrite

- ◆ 作用：读取、写入文件
- ◆ 原型：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
(Return: the number of a items successfully read or written.)
```

- ◆ 参数：
  - ptr: 缓冲区指针
  - size: 一次读取/写入的字节数
  - nmemb: 读取/写入的字节数
  - stream: 文件流

- ◆ 使用：

```
■ Read/write a binary array:
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
 err_sys("fwrite error");
■ Read/write a structure
struct {
 short count; long total; char name[NAMESIZE];
}item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
 err_sys("fwrite error");
```

## scanf/fsacnf/sscanf

- ◆ 作用：格式化输入
- ◆ 原型：

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

## printf/fprintf/sprintf

- ◆ 作用：格式化输出
- ◆ 原型：


```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

## fseek/ftell/rewind fgetpos/fsetpos

- ◆ 作用：重新设置流位置（fseek 类似 lseek）
- ◆ 原型：

### fseek, ftell, rewind functions

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
```



### fgetpos, fsetpos functions ( Introduced in ANSI C)

```
#include <stdio.h>
int fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

- ◆ 相关结构体：

```
typedef struct
{
 __off_t __pos; // 双下划线
 __mbstate_t __state;
}fpos_t;
```

## fflush

- ◆ 作用：刷新文件流，把流里的数据立刻写入文件
- ◆ 原型：

```
#include <stdio.h>
int fflush(FILE *stream);
```

## fileno/fdopen

- ◆ 作用：进行底层文件描述符与文件流之间的操作
- ◆ 原型：

确定流使用的底层文件描述符

```
#include <stdio.h>
int fileno(FILE *fp);
```

根据已打开的文件描述符创建一个流

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

- ◆ 参数：
  - mode 参见 fopen

## tmpnam/tmpfile

- ◆ 作用：进行临时文件的操作，生成一个临时文件名或者临时文件
- ◆ 原型：

Create a name for a temporary file

```
#include <stdio.h>
char *tmpnam(char *s);
(返回值: 指向唯一路径名的指针)
```

Create a temporary file

```
#include <stdio.h>
FILE *tmpfile(void);
(返回值: 若成功为文件指针, 若出错为NULL)
```

## stat/fstat/lstat

- ◆ 作用：获得文件的属性信息
- ◆ 原型：

### ■ Get file status

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *filename, struct stat *buf);
int fstat(int fildes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
(Return: 0 if success; -1 if failure)
```

把对应文件名的相关信息存到对应地址里  
同上, 使用文件描述符  
增加了符号链接  
文件名  
文件描述符  
存储信息的指针  
遇到文件软连接的时候得到的属性得到的是软链接文件的属性  
得到软链接本身文件的属性

- ◆ 相关结构体

```
struct stat {
 mode_t st_mode; /*file type & mode*/
 ino_t st_ino; /*inode number (serial number)*/
 dev_t st_rdev; /*device number (file system)*/
 nlink_t st_nlink; /*link count*/
 uid_t st_uid; /*user ID of owner*/
 gid_t st_gid; /*group ID of owner*/
 off_t st_size; /*size of file, in bytes*/
 time_t st_atime; /*time of last access*/
 time_t st_mtime; /*time of last modification*/
 time_t st_ctime; /*time of last file status change*/
 long st_blksize; /*Optimal block size for I/O*/
 long st_blocks; /*number 512-byte blocks allocated*/
};
```

硬链接计数  
当前文件所有者id  
当前文件所有者组id  
文件类型  
当前文件的索引节点号  
设备号, 特指当前文件的块设备的时候才有  
最近访问时间  
最近修改时间  
最近文件属性修改时间

st\_mode 里保存了文件的类型、权限等信息。低 9 位保存权限信息，每一位依次为用户的读、写、执行，用户组的读、写、执行，其他用户的读、写、执行。  
[9:11]位依次为 sticky、SGID、SUID, [12:17]保存了文件的类型。

|                           |            |                                               |
|---------------------------|------------|-----------------------------------------------|
|                           | SUID       | Program runs with effective user ID of owner  |
|                           | SGID       | Program runs with effective group ID of owner |
| 在这个文件夹里创建的文件是否具有用户的互斥性(<) | Sticky bit |                                               |

/tmp 任何用户都有写权限  
是否允许user1写入的文件被user2删除  
Sticky Bit控制这类权限  
user1创建的只能自身删除

预定义了判断文件类型的宏（参数为 st\_mode）

| Macro      | File type              |
|------------|------------------------|
| S_ISREG()  | regular file           |
| S_ISDIR()  | directory              |
| S_ISCHAR() | character special file |
| S_ISBLK()  | block special file     |
| S_ISFIFO() | fifo                   |
| S_ISLNK()  | symbolic link          |
| S_ISSOCK() | socket                 |

## access

- ◆ 作用：按实际用户 ID 和实际组 ID 测试文件的存取权限
- ◆ 原型：
 

```
#include <unistd.h>
int access(const char *pathname, int mode);
(Return: 0 if success; -1 if failure)
```
- ◆ 参数：
  - pathname: 文件路径
  - mode: 取值为 R\_OK, W\_OK, X\_OK, F\_OK 依次为测试读取、写入、执行权限和文件是否存在

## chmod/fchmod

- ◆ 作用：更改一个文件的权限
- ◆ 原型：
 

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
(Return: 0 if success; -1 if failure)
```
- ◆ 参数：
  - mode: 与 st\_mode 中的低九位相同（普遍的权限格式）

## chown/fchown/lchown

- ◆ 作用：改变一个文件的所有者
- ◆ 原型：

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
(Return: 0 if success; -1 if failure)
```

## umask

- ◆ 作用：为进程设置文件存取权限屏蔽字，并返回以前的值
- ◆ 原型：
 

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```
- ◆ 参数：
  - mask，对应的 mask 值（进程默认为 022，八进制）
- ◆ 说明：
  - mask 值是一个进程的文件保护机制，可以用来控制文件的权限，系统默认为八进制 022。文件的最终权限是设定的 mode 值与 mask 值的取反后做按位与的结果
  - $actual\_mode = mode \& \sim mask$   
 如初始权限为 rw-rw-rw-0666，mask 值为---w--w-022，计算后 actual\_mode 为 rw-r--r--0644，可以认为 mask 的值对应的权限为不能提供为用户的权限

## link/unlink

- ◆ 作用：创建/删除一个文件的（硬）链接
- ◆ 原型：
 

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
(Return: 0 if success; -1 if failure)

int unlink(const char *pathname);
(Return: 0 if success; -1 if failure)
```

## symlink/readlink

- ◆ 作用：创建/读取符号链接的值
- ◆ 原型：
 

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath);
(Return: 0 if success; -1 if failure)

int readlink(const char *path, char *buf, size_t bufsiz);
(Return: the count of characters placed in the buffer if success;
-1 if failure)
```

## mkdir/rmdir

- ◆ 作用：创建/删除空目录
- ◆ 原型：

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
(Return: 0 if success; -1 if failure)
int rmdir(const char *pathname);
(Return: 0 if success; -1 if failure)
```

### chdir/fchdir

- ◆ 作用：改变当前的工作目录
- ◆ 原型：

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
(Return: 0 if success; -1 if failure)
```
- ◆ 说明：当前工作目录是进程的属性，所以该函数只影响调用 chdir 的进程本身

### getcwd

- ◆ 作用：获得当前工作目录的绝对路径
- ◆ 原型：

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
(返回值: 若成功则为buf, 若出错则为NULL)
```
- ◆ 参数：
- ◆ 使用：

### opendir/closedir/readdir/telldir/seekdir

- ◆ 作用：目录的打开、关闭、读、定位
- ◆ 原型：

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir(const char *name);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
```

- ◆ 相关结构体：

#### ■ DIR

```
struct __dirstream
{
 void *__fd;
 char *__data;
 int __entry_data;
 char *__ptr;
 int __entry_ptr;
 size_t __allocation;
 size_t __size;
```



```

 __libc_lock_define(, __lock)
 };
 typedef struct __dirstream DIR;
 ■ dirent
 struct dirent {
 long d_ino; /* inode number 索引节点号 */
 off_t d_off; /* offset to this dirent 在目录文件中的偏移 */
 unsigned short d_reclen; /* length of this d_name 文件名长 */
 unsigned char d_type; /* the type of d_name 文件类型 */
 char d_name [NAME_MAX+1]; /* file name (null-terminated) 文件名，
 最长 255 字符 */
 }

```

◆ 使用：

```

DIR *dp;
struct dirent *entry;

if ((dp = opendir(dir)) == NULL)
 err_sys(...);
while ((entry = readdir(dp)) != NULL) {
 lstat(entry->d_name, &statbuf);
 if (S_ISDIR(statbuf.st_mode))
 ...
 else
 ...
}
closedir(dp);

```

## lockf

◆ 作用：对文件进行加/释放锁

◆ 原型：

```

#include <sys/file.h>
int lockf(int fd, int cmd, off_t len);

```

◆ 参数：

- fd: 对应的文件描述符
- cmd: 指定的操作类型，取值为以下
  - F\_LOCK: 给文件加互斥锁，若文件已被加锁，则会一直阻塞到锁被释放。
  - F\_TLOCK: 同 F\_LOCK，但若文件已被加锁，不会阻塞，而回返回错误。
  - F\_ULOCK: 解锁。
  - F\_TEST: 测试文件是否被上锁，若文件没被上锁则返回 0，否则返回 -1。
- len: 为从文件当前位置的起始要锁住的长度。

◆ 说明：lockf 是 fcntl 的一个库层次上的封装，但是不支持共享锁（只支持排他锁，参考 fcntl 文件锁说明）

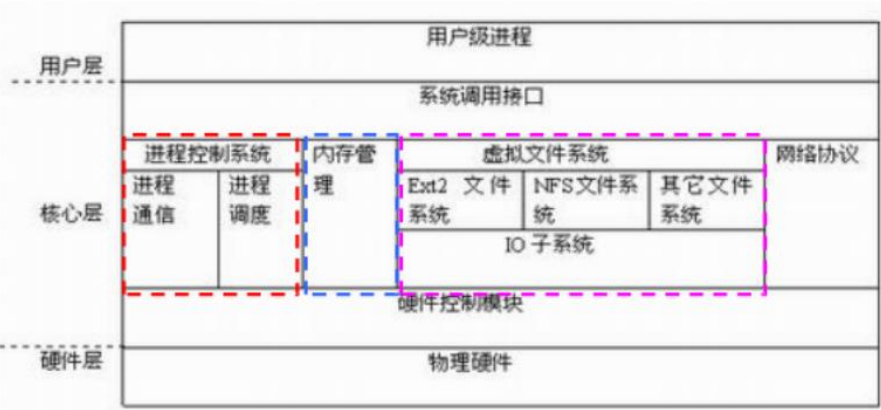
# Chapter 4

## 内核简介

### 什么是内核

- 操作系统是一系列程序的集合，其中最重要的部分构成了内核
- 单内核/微内核
  - 单内核是一个很大的进程，内部可以分为若干模块，运行时是一个独立的二进制文件，模块间通讯通过直接调用函数实现
  - 微内核中大部分内核作为独立的进程在特权下运行，通过消息传递进行通讯
- Linux内核的能力
  - 内存管理，文件系统，进程管理，多线程支持，抢占式，多处理支持
- Linux内核区别于其他UNIX商业内核的优点
  - 单内核，模块支持
  - 免费/开源
  - 支持多种CPU，硬件支持能力非常强大
  - Linux开发者都是非常出色的程序员
  - 通过学习Linux内核的源码可以了解现代操作系统的实现原理

## 内核结构



## Linux 内核模块与应用程序区别

|    | C 语言程序 | Linux 内核模块          |
|----|--------|---------------------|
| 运行 | 用户空间   | 内核空间                |
| 入口 | main() | module_init()指定     |
| 出口 | 无      | module_exit()指定     |
| 运行 | 直接运行   | insmod              |
| 调试 | gdb    | kdebug, kdb, kgdb 等 |

## 开发驱动的注意事项

- 不能使用 c 库来开发驱动程序
- 没有内存保护机制
- 小内核栈
- 并发上的考虑