# Shell 命令

## root 命令

sudo passwd root root 设置密码

su root 切换到root

su 用户名 切换回普通模式

## tar 命令

压缩：tar -jcv -f filename.tar.bz2 要被压缩的文件或者文件夹

查看：tar -jtv -f filenma.tar.bz2

解压缩：tar -jxv -f filename.tar.bz2 -C 欲解压的目录

压缩

tar –cvf jpg.tar \*.jpg //将目录里所有jpg文件打包成tar.jpg

tar –czf jpg.tar.gz \*.jpg //将目录里所有jpg文件打包成jpg.tar后，并且将其用gzip压缩，生成一 个gzip压缩过的包，命名为jpg.tar.gz

tar –cjf jpg.tar.bz2 \*.jpg //将目录里所有jpg文件打包成jpg.tar后，并且将其用bzip2压缩，生成一个bzip2压缩过的包，命名为jpg.tar.bz2

tar –cZf jpg.tar.Z \*.jpg //将目录里所有jpg文件打包成jpg.tar后，并且将其用compress压缩，生成一个umcompress压缩过的包，命名为jpg.tar.Z

rar a jpg.rar \*.jpg //rar格式的压缩，需要先下载rar for linux

zip jpg.zip \*.jpg //zip格式的压缩，需要先下载zip for linux

解压

tar –xvf file.tar //解压 tar包

tar -xzvf file.tar.gz //解压tar.gz

tar -xjvf file.tar.bz2 //解压 tar.bz2

tar –xZvf file.tar.Z //解压tar.Z

unrar e file.rar //解压rar

unzip file.zip //解压zip

## man 命令

man [命令] 显示该命令的说明文档

## cd命令

cd .. 返回上一级目录

cd 返回根目录

## pwd命令

pwd 用于显示当前路径

## rm删除文件命令

rm –rf filepatch 删除filepatch路径下的文件夹及文件夹下的所有文件(强制删除)

rm –r filepatch/filename 仅删除filepatch路径下filename这一个文件

## vi 命令：

i 在当前字符下插入文本

:wq 在命令模式下存盘退出

:q 在命令模式下退出vi

:w 在命令模式下存盘

Esc从编辑模式切换到命令模式

## Git 命令：

git config - -list 查看已有的配置信息

git config --global user.name "heming"

git config --global user.email heming@dniserver.dnish.net

git init 创建git版本库

git add 添加文件到仓库暂存区，

git add -A: [<path>]表示把<path>中所有tracked文件中被修改过或已删除文件和所有untracted的文件信息添加到索引库。

git commit 提交暂存区文件到仓库

git commit -m "wrote a readme file"

git diff 查看最近一次edit后对文件做了哪些修改，相对于最近一次的add

git status 查看最近一次add后修改了哪些文件，相对于最近一次的commit

git clone 从远程仓库克隆

git remote 查看当前配置的远程库

git remote add 添加远程库

git push 推送数据到远程仓库

git push origin 本地分支名:远程分支名

git fetch origin相当于是从远程获取最新版本到本地，不会自动merge

所取回的更新，在本地主机上要用”远程主机名/分支名”的形式读取。比如origin主机的master，就要用origin/master读取

git checkout -b newBrach origin/master在origin/master的基础上，创建一个新分支

git rebase origin/[branchname] 用于把一个分支的修改合并到当前分支

git merge branchname 合并branchname分支到当前分支

git pull：相当于是从远程获取最新版本并merge到本地

Git创建分支

git checkout –b branchname 创建并切换到branchname分支

或者

git branch branchname 创建分支

git checkout branchname 切换分支

git checkout –t branchname 拿出远程分支并跟踪

git checkout –t origin/gs418tpp

在克隆仓库时，Git 通常会自动创建一个名为 master 的分支来跟踪 origin/master。这正是git push 和 git pull 一开始就能正常工作的原因。

git branch 查看当前分支

git branch -d brandname 删除brandname分支

git log 显示提交日志

git reset --hard HEAD^ 版本回退

git checkout -- <filename> 让filename文件回到最近一次git commit或git add时的状态

git reset HEAD file把暂存区的修改撤销掉（unstage），重新放回工作区

删除文件跟踪并且删除文件系统中的文件file git rm file

提交刚才的删除动作，之后git不再管理该文件git commit

删除文件跟踪但不删除文件系统中的文件file ，git rm --cached file

Note:--cached Use this option to unstage and remove paths only from the index.Working tree files, whether modified or not, will beleft alone.只是将文件从暂存区去除（不影响工作区对应的文件）。

[git-rm - Remove files from the working tree and from the index]

提交刚才的删除动作，之后git不再管理该文件。但是文件系统中还是有file。 git commit

git format-patch 生成文件修改的patch文件

格式: git format-patch -n [commit ID] 从选定的ID起依次生成之前n次提交的patch

git apply [patchfile] 应用生成的补丁

git am 应用补丁并提交

标签管理

git tag tagname (commit ID)创建当前分支(或指定ID)的标签

git tag 查询标签

git tag -d tagname 删除标签

git push origin tagname 推送标签到远程

删除远程标签

git tag -d tagname 先从本地删除

git push origin :refs/tags/tagname 再从远程删除

合并多个commit:

1.git rebase –I commit ID #合并commit ID 之前的所有提交

2. #在弹出的文件中初叫距离commit ID最近的commit 设置

为pick外，其他commit都设置为squash，按ctrl+X退出

3. #在接下来弹出的文件中设置新提交的commit的description

## scp 跨服务器文件拷贝命令

scp 选项 参数

选项：

-1 使用ssh协议版本1

-2 使用ssh协议版本2

-4 使用ipv4

-6 使用ipv6

-p 指定远程主机的端口号

参数：

源文件 目标文件（user@host:filename）

## ls命令-显示目标列表

-a 显示所有的文件及目录

-d 仅显示目录名

-s显示文件和目录的大小

## find –文件查找命令

find –name “filename” 查找文件

## diff和patch命令

diff

diff可以完成比较功能，生成补丁文件

格式：diff [option] oldfile newfile

常用的option选项有：

-r 对目录进行递归处理

-u 输出统一格式，diff有"传统"和"统一"两种格式，现在一般使用"统一"格式，比较而言，统一格式生成的文件大，但包含了更多的信息，有利于阅读与定位

-N 补丁中包含整个新文件

-a 补丁中包含二进制文件

patch

patch [ -b [ -B Prefix ] ] [ -f ] [ -l ] [ -N ] [ -R ] [ -s ] [ -v ] [ -c | -e | -n ] [ -d Directory ] [ -D Define ] [ -F Number ] [ -i PatchFile ] [ -o OutFile ] [ -p Number ] [ -r RejectFile ] [ -x Number ] [ File ]

patch 命令失败或拒绝接受补丁时，会产生一个和原文件同名，以".rej"为后缀的差异文件。 　　　当知道 -b 时，会产生一个和原文件同名，以".orig"为后缀的备份文件。

常使用的 patch 参数：

-p 指定目录级别（从路径全称中除去几层目录）

如，如果补丁文件包含路径名称 /curds/whey/src/blurfl/blurfl.c，那么：

-p 0 使用完整路径名 -p 1 除去前导斜杠，留下 curds/whey/src/blurfl/blurfl.c。 -p 4 除去前导斜杠和前三个目录，留下 blurfl/blurfl.c。

-d Directory 打补丁前，更改当前目录到指定目录

-i PatchFile 从指定文件，而不是从标准输入中读取补丁信息

-R 逆向补丁，这个选项在防止打错补丁很有用处

## touch命令

创建文件或修改文件时间

-a 只更新访问时间，不改变修改时间

-c 不创建不存在的文件

-m 只更新修改时间，不改变访问时间

-r file 使用文件file的时间更新文件的时间

-t 将时间修改为参数指定的日期,如：

07081556代表7月8号15点56分

# Makefile

## 综述

makefile描述规则格式：

TARGET... : PREREQUISITES...

COMMAND

target：规则的目标

prerequisites：规则的依赖

command：规则的命令行

包含其它makefile文件：

include FILENAMES...

FILENAMES 是 shell 所支持的文件名

## make的函数

函数语法：

$(FUNCTION ARGUMENTS)

一些常用的函数：

### 文本处理函数

$(subst FROM,TO,TEXT)

函数功能：把字串“ TEXT”中的“ FROM”字符替换为“ TO”

返回值：替换后的新字符串

$(patsubst PATTERN,REPLACEMENT,TEXT)

函数功能：搜索“ TEXT”中以空格分开的单词，将否符合模式

“ TATTERN”替换为“ REPLACEMENT”

返回值：替换后的新字符串

$(strip STRINT)

函数功能：去掉字串（若干单词，使用若干空字符分割）“ STRINT”

开头和结尾的空字符，并将其中多个连续空字符合并为一个空

字符

返回值：无前导和结尾空字符、使用单一空格分割的多单词字符

串

### 文件名处理函数

$(dir NAMES…)

函数功能：从文件名序列“ NAMES…”中取出各个文件名的目

录部分

返回值：空格分割的文件名序列“ NAMES…”中每一个文件的

目录部分

## Linux内核Makefile

顶层Makefile:

#line-433开始，进入下面变量指定的子目录，每个子目录生成一

#个built-in.o，最后head-y表示的文件和build-in.o一起链接成

#vmlinux

init-y := init/

drivers-y := drivers/ sound/

net-y := net/

libs-y := lib/

core-y := usr/

#直接包含了体系结构的Makefile

include $(srctree)/arch/$(ARCH)/Makefile

体系结构Makefile：

#line-94

head-y := arch/arm/kernel/head$(MMUEXT).o

arch/arm/kernel/init\_task.o

编译选项和连接选项：

全局：

CFLAGS：编译C文件的选项

AFLAGS：编译汇编文件的选项

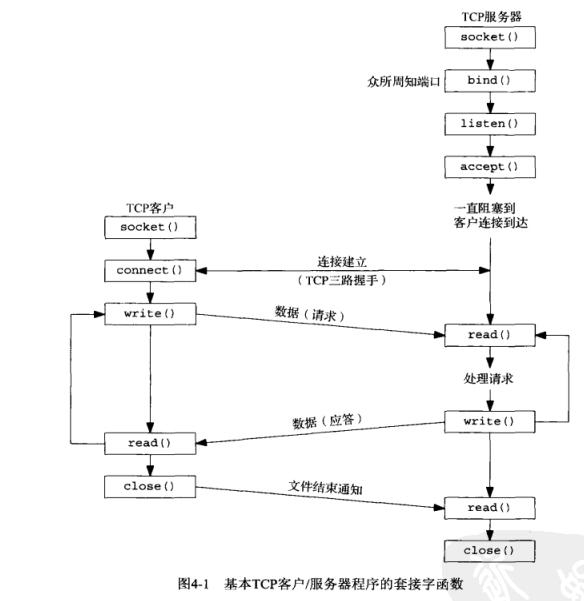
LDFLAGS：连接文件的选项

ARFLAGS：制作库文件的选项

# 进程管理

# socket

## TCP套接字



### socket函数

#include<sys/socket.h>

int socket(int family, int type, int protocot)

family: AF\_INET IPV4协议 AF\_INET6 IPV6协议

type: SOCK\_STREAM 字节流套接字

SOCK\_DGRAM 数据包套接字

SOCK\_RAW 原始套接字

protocol: IPPROTO\_CP TCP传输协议

IPPROTO\_UDP UDP传输协议

### connect函数

//建立与TCP服务器的连接

#include<sys/socket.h>

int connect(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen)

servaddr：指向服务器套接字地址结构的指针

### bind函数

//把一个本地协议地址赋予一个套接字

#include<sys/socket.h>

int bind(int sockfd, const struct sockaddr \*myaddr, socklen\_t addrlen)

### listen函数

//仅由TCP服务器调用，将一个未连接的套接字转换为一个被动套接

字

#include<sys/socket.h>

int listen(int sockfd, int backlog)

backlog：内核为相应套接字排队的最大连接个数

### accept函数

//由TCP服务器调用，用于从已完成连接队列队头返回下一个已完成

连接

#include<sys/socket.h>

int accept(int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen)

sockfd：监听套接字，由socket函数创建

close函数

include<unistd.h>

int close(int sockfd)

### recv和send函数

#include<sys/socket.h>

ssize\_t recv(int sockfd,void \*buff,size\_t nbytes,int flags)

ssize\_t send(int sockfd,const void \*buff,size\_t nbytes,int flags)

flags：MSG\_DONTROUTE 绕过路由表查找

MSG\_DONTWAIT 仅本操作非阻塞

MSG\_WAITALL 等待所有数据

### select函数

//等待多个事件中的任何一个发生

#include<sys/select.h>

#include<sys/time.h>

int select(int maxfdpl, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptset,

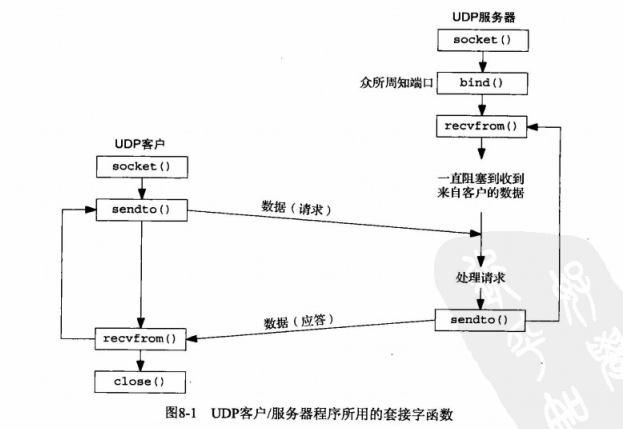
const struct timeval \*timeout)

maxfdpl：待测试的描述符的个数，值为最大描述符加1

fd\_set：读、写和异常测试的描述符集

timeout：等待所指定描述符中任何一个就绪可花多长时间

## UDP套接字



### recvfrom函数和sendto函数

#include<sys/socket.h>

size\_t recvfrom(int sockfd, void \*buffer, size\_t nbytes, int flags

struct sockaddr \*from, socklen\_t \*addrlen)

size\_t sendto(int sockfd, void \*buffer, size\_t nbytes, int flags

struct sockaddr \*to, socklen\_t \*addrlen)

# ALSA音频

## 内核驱动

snd\_card{ //声卡的结构

int number

struct list\_head devices //声卡下所有逻辑设备的链表

struct list\_head controls //声卡下所有控制单元的链表

void \*private\_data //声卡的私有数据

}

snd\_minor{ //保存了声卡下某个逻辑设备的上下文信息

int type

int card

int device

const struct file\_operations \*f\_ops

void \*private\_data

struct device \*dev

}

创建声卡：

snd\_card\_create(Linux2.6.30及以后api)，

snd\_card\_new(Linux2.6.30以前api)

struct snd\_card \*snd\_card\_new(int idx,const char \*xid,

struct module \*module, int extra\_size

//创建声卡的逻辑设备

int snd\_device\_new(struct snd\_card \*card, snd\_device\_type\_t type,

void \*device\_data, struct snd\_device\_ops \*ops)

通常，alsa-driver的已经提供了一些常用的部件的创建函数，而不必直接调用snd\_device\_new()，比如：

PCM ---- snd\_pcm\_new()

RAWMIDI -- snd\_rawmidi\_new()

CONTROL -- snd\_ctl\_create()

TIMER -- snd\_timer\_new()

INFO -- snd\_card\_proc\_new()

JACK -- snd\_jack\_new()

void snd\_pcm\_set\_ops(struct snd\_pcm \*pcm, int direction, struct

snd\_pcm\_ops \*ops)

//注册声卡

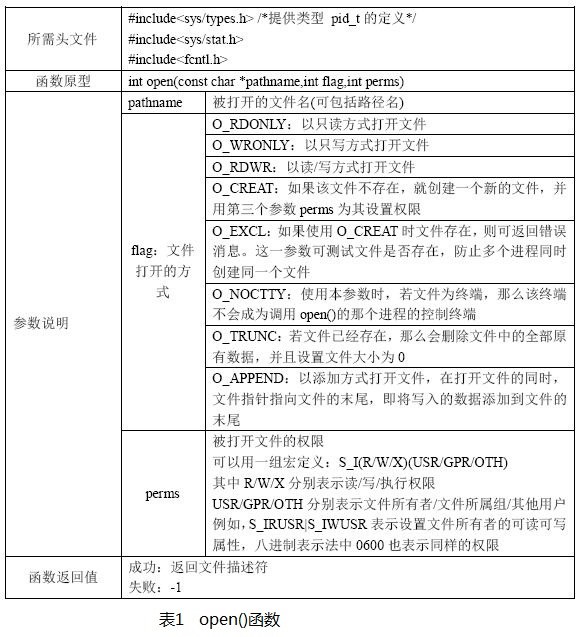
int snd\_card\_register(struct snd\_card \*card)

## 用户API

# file

## open

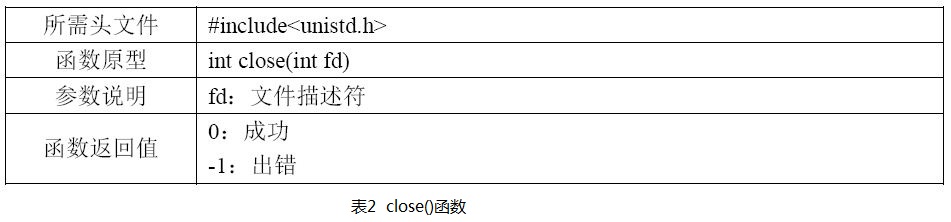
● open()函数用于打开或创建文件，在打开或者创建文件时可以指定文件的属性及用户的权限等各种参数。



在open()函数中，flag参数可通过 “|” 组合构成，但前3个标志常量（O\_RDONLY、O\_WRONLY及O\_RDWR）不能相互组合。perms是文件的存取权限，既可以用宏定义表示法，也可以用八进制表示法

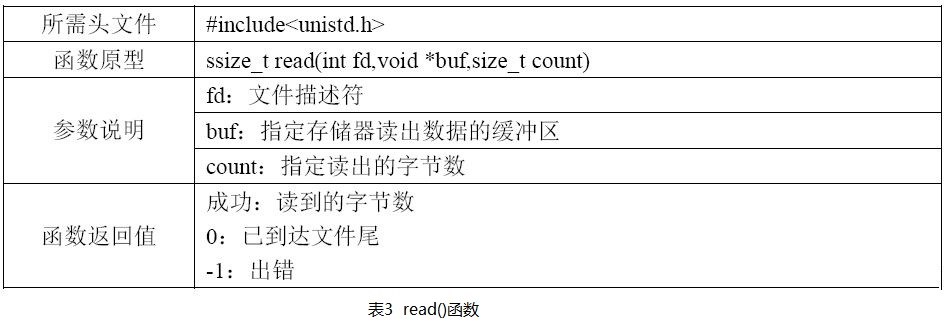
## close

● close()函数用于关闭一个被打开的文件。当一个进程终止时，所有被它打开的文件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。



## read

● read()函数用于将从指定的文件中读出的数据放到缓存区中，并返回实际读入的字节数。若返回0，则表示没有数据可读，即已到达文件尾。读操作从文件的当前指针位置开始。当从设备文件中读出数据时，通常一次最多读一行。

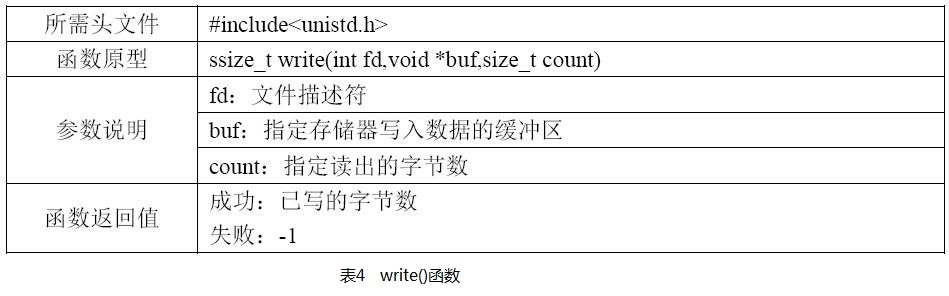


在读普通文件时，若读到要求的字节数前已到达文件的尾部，则返回的字节数会小于希望读书的字节数

也可以简单的记住：ssize\_t是有符号整形，size\_t是无符号整形

## write

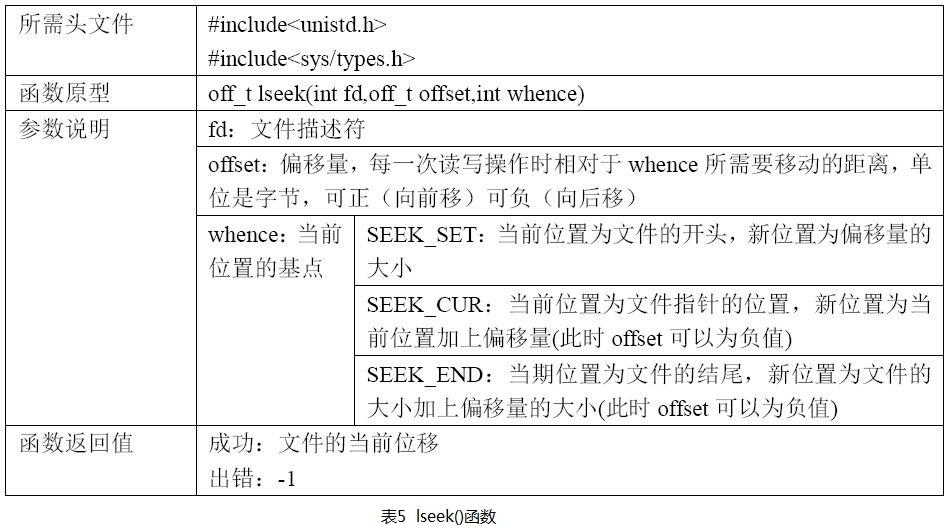
● write()函数用于向打开的文件写数据。写操作从文件的当前指针位置开始，对磁盘文件进行写操作，若磁盘已满或者超出该文件的长度，则write()函数返回失败。



在写普通文件时，写操作从文件的当前指针位置开始

## lseek

● lseek()函数用于在指定的文件描述符中将文件指针定位到相应的位置。每一个已打开的文件都有一个读写位置，当打开文件时，其读写位置通常指向文件开头；若是以附加的方式打开文件（如O\_APPEND），则读写位置会指向文件尾。当read()或者write()时，读写位置会随之增加，lseek()便是用来控制该文件的读写位置的。它只能用在可定位（可随机访问）文件操作中。管道、套接字和大部分字符设备文件是不可定位的，所以在这些文件的操作中无法使用lseek()调用



下面是lseek较特别的使用

☆ 欲将读写位置移到文件开头时：lseek(int fd,0,SEEK\_SET)

☆ 欲将读写位置移到文件尾时：lseek(int fd,0,SEEK\_END)

☆ 想要取得目前文件位置时：lseek(ind fd,0,SEEK\_CUR)

注意：Linux系统不允许lseek()对tty装置作用，此动作会令lseek()返回ESPIPE。

另外，其实还有一个文件创建函数creat()，它的函数原型是int creat(const char \*pathname,int perms),它相当于使用下列的调用方式调用open()：

open(const char \*pathname,(O\_CREAT|O\_WRONLY|O\_TRUNC))

文件锁包括建议性锁和强制性锁。建议性锁要求每个上锁文件的进程检查是否有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁。强制性锁是由内核执行的锁，当一个文件被上锁执行写入操作时，内核将阻止其他任何文件对其进行读写操作。采用强制性锁对性能影响很大，每次读写都必须检查是否有锁存在。

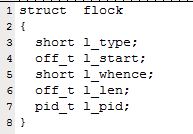
在Linux中，实行文件上锁的函数有lockf()和fcntl()，其中lockf()用于对文件施加建议性锁，而fcntl()不仅可以施加建议性锁，还可以施加强制性锁。同时，fcntl()还能对文件的某一记录上锁，也就是记录锁。

记录锁又可分为读取锁和写入锁，其中读取锁又称为共享锁，它能够使多个进程都能在文件的同一部分建立读取锁。而写入锁又称为排斥锁，在任何时刻只能有一个进程在某个部分建立写入锁。当然，在文件的同一部分不能同时建立读取锁和写入锁。

fcntl()函数具有很丰富的功能，它可以对已打开的文件描述符进行各种操作，不仅包括管理文件锁，还包括获得设置文件描述符和文件描述符标志、文件描述符的复制等很多功能！这一次我先学习一下fcntl()函数建立文件锁的方法，关于它的另外的用法...先学会了这个再说吧！

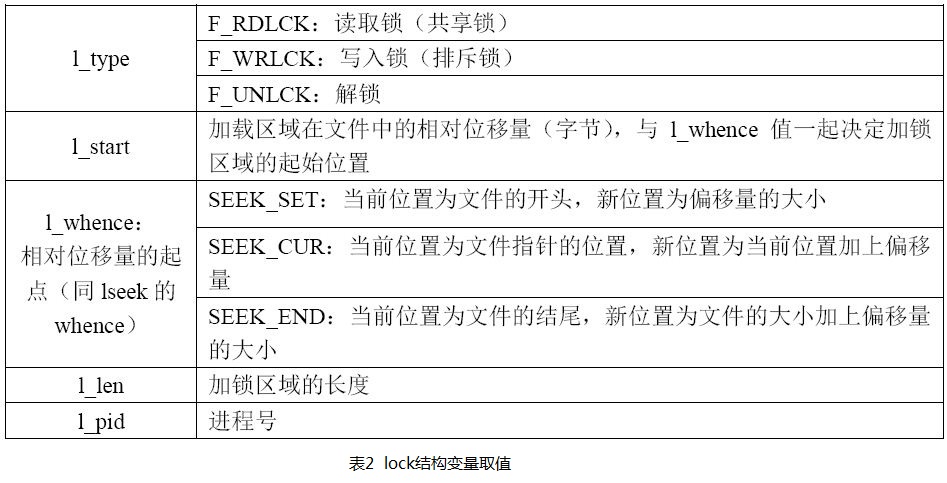


表1中的lock是一个flock结构体，结构如下：



上图中的 off\_t 就是数据类型 long int ；pid\_t 就是数据类型 int,不懂这里有解释：点此解释

那么这个结构体lock中每个变量的取值含义如下表2



mknod命令

mknod [options] name {bc} major minor

mknod [options] name p

GNU 选项（缩写）：

[-m mode] [--help] [--version] [--]

描述

mknod 用指定名称产生一个FIFO（命名管道），字符专用或块专用文件。

文件系统中的一个专用文件存贮着三种信息（布朗型、整型、整型）。布朗型在字符文件与块文件之间作出选择，两个整型是主、次设备号。

通常，一个专用文件并不在磁盘上占用空间，仅仅是为操作系统提供交流，而不是为数据存贮服务。一般地，专用文件会指向一个硬件设备（如：磁盘、磁带、打印机、虚拟控制台）或者操作系统提供的服务（如：/dev/null, /dev/random）。

块文件通常类似于磁盘设备（在数据可以被访问的地方赋予一个块号，意味着同时设定了一个块缓存）。所有其他设备都是字符文件。（以前，两种文件类型间是有差别的。比如：字符文件I/O没有缓存，而块文件则有。）

mknod命令就是用来产生这种类型文件的。

以下参数指定了所产生文件的类型：

p

FIFO型

b

块文件

c

字符文件

GNU版本还允许使用u（'unbufferd'非缓冲化），以保持与C语言的一致。

当创建一个块文件或字符文件时，主、次设备号必须在文件类型参数后给出。（十进制或八进制以0开头；GNU 版本还允许使用以0x开头的十六进制）缺省地，所产生的文件模式为0666（'a+rw'）。

选项

-m mode, --mode=mode

为新建立的文件设定模式，就象应用命令chmod一样，以后仍然使用缺省模式建立新目录。

GNU 标准选项

--help

在标准输出上显示使用信息并顺利退出。

--version

在标准输出上显示版本信息并顺利退出

--

终端选项列表

# 线程管理

创建一个线程

#include<pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t

\* attr,void \*(\*start\_routine)(void\*), void \* arg)

thread：返回一个不透明的，唯一的新线程标识符。

attr：不透明的线程属性对象。可以指定一个线程属性对象，或者

NULL为缺省值。

start\_routine：线程将会执行一次的C函数。

arg: 传递给start\_routine单个参数，传递时必须转换成指向void

的指针类型。没有参数传递时，可设置为NULL

终止当前线程

void pthread\_exit(void \*value\_ptr)

//获取线程标识符

pthread\_t pthread\_self(void)

//中断另外一个线程的运行

int pthread\_cancel(pthread\_t thread)

//等待另一个指定线程结束，

int pthread\_join(pthread\_t thread, void \*\*value\_ptr)

thread：等待退出线程的线程号。

value\_ptr：退出线程的返回值

int pthread\_detach(pthread\_t thread)

# 消息

## Posix消息队列

#include <fcntl.h>

#include <sys/stat.h>

#include <mqueue.h>

Link with -lrt.

### mq\_open

//打开或创建一个posix消息队列

mqd\_t mq\_open(const char \*name, int oflag)

mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode,

struct mq\_attr \*attr)

参数name为posix IPC名字，为了便于移植，需要指定为“/name”的格式。

参数oflag必须要有O\_RDONLY（只读）、标志O\_RDWR(读写)， O\_WRONLY（只写）之一，除此之外还可以指定O\_CREAT（没有该对象则创建）、O\_EXCL（如果O\_CREAT指定，但name不存在，就返回错误），O\_NONBLOCK（以非阻塞方式打开消息队列，在正常情况下mq\_receive和mq\_send函数会阻塞的地方，使用该标志打开的消息队列会返回EAGAIN错误）。

当操作一个新队列时，使用O\_CREAT标识，此时后面两个参数需要被指定，参数mode为指定权限位，attr指定新创建队列的属性

int mq\_close(mqd\_t mqdes)

关闭之后调用进程不在使用该描述符，但消息队列不会从系统中删除，进程终止时，会自动关闭已打开的消息队列，和调用mq\_close一样。参数为mq\_open()函数返回的值

### mq\_unlink

//从系统中删除某个消息队列

int mq\_unlink(const char \*name)

### mq\_setattr

//设置和和获取消息队列属性

int mq\_getattr(mqd\_t mqdes, struct mq\_attr \*attr)

int mq\_setattr(mqd\_t mqdes, struct mq\_attr \*newattr, struct mq\_attr \*oldattr)

### mq\_attr

struct mq\_attr {

long mq\_flags /\* Flags: 0 or O\_NONBLOCK \*/

long mq\_maxmsg /\* Max. # of messages on queue \*/

long mq\_msgsize /\* Max. message size (bytes) \*/

long mq\_curmsgs /\* # of messages currently in queue \*/

}

参数mq\_flags在mq\_open时被初始化（oflag参数），其值为0 或者 O\_NONBLOCK。

参数mq\_maxmsg和mq\_msgsize在mq\_open时在参数attr中初始化设置，mq\_maxmsg是指队列的消息个数最大值；mq\_msgsize为队列每个消息的最大值。

参数mq\_curmsgs为当前队列消息。

mq\_getattr()函数把队列当前属性填入attr所指向的结构体。

mq\_setattr()函数只能设置mq\_flags属性，另外的域会被自动忽略，mq\_maxmsg和mq\_msgsize的设置需要在mq\_open当中来完成， 参数oldattr会和函数mq\_getattr函数中参数attr相同的值

### mq\_send

//发送消息和接收消息

int mq\_send(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned msg\_prio)

ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned \*msg\_prio)

mqdes为消息队列描述符

msg\_ptr为指向消息的指针。

msg\_len为消息长度，该值不能大于属性值中mq\_msgsize的值。

msg\_prio为优先级，数值越大，优先级越高，消息在队列中将按照优先级大小顺序来排列消息。调用 mq\_receive 时总是返回队列中 最高优先级的最早消息。如果不需要可以置为0。

如果消息队列已满，mq\_send()函数将阻塞，直到队列有可用空间再次允许放置消息或该调用被信号打断；如果O\_NONBLOCK被指定，mq\_send()那么将不会阻塞，而是返回EAGAIN错误。如果队列空，mq\_receive()函数将阻塞，直到消息队列中有新的消息；如果O\_NONBLOCK被指定，mq\_receive()那么将不会阻塞，而是返回EAGAIN错误。

mq\_receive中的msg\_len要大于或等于发送的消息长度

### mq\_notify

# 同步

## 互斥锁

pthread\_mutex\_t{

}

//动态分配的互斥锁初始化

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr)

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex)

//线程用pthread\_mutex\_lock()函数去锁定指定的mutex变量，若该

mutex已经被另外一个线程锁定了，该调用将会阻塞线程直到mutex

被解锁

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

//pthread\_mutex\_trylock()尝试着去锁定一个互斥量，然而，若互斥量

已被锁定，程序会立刻返回并返回一个EBUSY错误值。该函数在优先级改变情况下阻止死锁是非常有用的

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)

//线程可以用pthread\_mutex\_unlock()解锁自己占用的互斥量。在一个

线程完成对保护数据的使用，而其它线程要获得互斥量在保护数据上

工作时，可以调用该函数

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

## 条件变量

pthread\_cond\_t{

}

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,

const pthread\_condattr\_t \*restrict attr)

int pthread\_cond\_destroy(pthread\_cond\_t \*cond)

//阻塞调用线程直到指定的条件受信（signaled）。该函数应该在互

斥量锁定时调用，当在等待时会自动解锁互斥量。在信号被发送，线程被激活后，互斥量会自动被锁定，当线程结束时，由程序员负责解锁互斥量

//将mutex互斥量解锁，将cond条件变量加锁，挂起等待

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex)

1，线程放在等待队列上，解锁

2，等待 pthread\_cond\_signal或者pthread\_cond\_broadcast信号之后去竞争锁

3，若竞争到互斥索则加锁

//用于向其他等待在条件变量上的线程发送信号（激活其它线程）。应

该在互斥量被锁定后调用。若不止一个线程阻塞在条件变量上，则应

用pthread\_cond\_broadcast()向其它所以线程发生信号

//唤醒等待cond条件变量的线程

int pthread\_cond\_signal(pthread\_cond\_t \*cond)

一般用法

thread1:

pthread\_mutex\_lock(&mut);

while (x <= y) {

pthread\_cond\_wait(&cond, &mut);

}

/\* operate on x and y \*/

pthread\_mutex\_unlock(&mut);

thread2:

pthread\_mutex\_lock(&mut);

/\* modify x and y \*/

if (x > y)

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mut);

## POSIX信号量

主要目的是提供进程间通信方式

互斥锁必须总是由给它上锁的线程解锁。而信号量没有这种限制，一个线程可以等到某个给定信号量，而另一个线程可以挂出该信号量

### 有名信号量

有名信号量既可用于线程间同步，也可用于进程间同步

#include <semaphore.h>

//如果oflag参数指定了O\_CREAT，后面两个参数需要

sem\_t \*sem\_open(const char \*name, int oflag,mode\_t mode,

unsigned int value)

//关闭信号量

int sem\_close(sem\_t \*sem)

//删除信号量

int sem\_unlink(const char \*name)

//如果被某个信号中断，返回EINTR错误

int sem\_wait(sem\_t \*sem)

//如果信号量已经为0，并不阻塞线程。而是返回EAGAIN错误

int sem\_trywait(sem\_t \*sem)

int sem\_post(sem\_t \*sem)

int sem\_getvalue(sem\_t \*sem,int \*valp)

### 无名信号量

//也叫基于内存的信号量

//shared为0，信号量是在同一进程间各个线程间共享的，否则是在进程间共享的，此时该信号量必须存放在某种类型的共享内存区中

int sem\_init(sem\_t \*sem, int shared, unsigned int value)

int sem\_destroy (sem\_t \*sem)

int sem\_wait(sem\_t \*sem)

int sem\_post(sem\_t \*sem)

# 共享内存区

1，,由shm\_open创建或打开一个共享内存区对象

2，由mmap把这个共享内存区映射到调用进程的地址空间

## 内存映射

//mmap把一个文件或一个POSIX共享内存区对象映射到调用进程的

地址空间

1.使用普通文件以提供内存映射I/O

2.使用特殊文件以提供匿名内存映射I/O

3.使用shm\_open以提供无亲缘关系进程间的POSIX共享内存区

#include <sys/mman.h>

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

addr指定描述符fd应被映射到的进程内空间的起始地址，通常为NULL空指针以让内核去选择起始地址

length为要映射的字节数，从被映射文件开头起第offset个字节处开始算，offset通常设置为0

port指明内存映射区的保护,PROT\_READ, PROT\_WRITE

flag为MAP\_SHARED，MAP\_PRIVATE等

函数返回值为描述符fd所映射到内存区的起始地址

int munmap(void \*addr, size\_t length)

保持内存映射文件和内存映射区的同步：

int msync(void \*addr, size\_t length, int flags)

addr和length指代内存中的内存映射区

flags可取MS\_ASYNC(异步写)，MS\_SYNC(同步写)等

fork对内存映射文件进行特殊处理，父进程在调用fork之前创建的内存映射关系由子进程共享

## POSIX共享内存

创建一个新的共享内存区对象或打开一个已经存在的共享内存区对象

int shm\_open(const char \*name, int oflag, mode\_t mode)

//删除一个共享内存区对象的名字

int shm\_unlink(const char \*name)

对于一个共享内存区对象，函数把该对象的大小设置为length个字节

int ftruncate(int fd, off\_t length)

获取共享内存区对象的信息

int fstat(int fd,struct stat \*buf)

# 信号

## 信号安装

### signal

第一个参数指定信号的值，第二个参数指定针对前面信号值的处理，可以忽略该信号（参数设为SIG\_IGN）；可以采用系统默认方式处理信号(参数设为SIG\_DFL)；也可以自己实现处理方式(参数指定一个函数地址)。   
如果signal()调用成功，返回最后一次为安装信号signum而调用signal()时的handler值；失败则返回SIG\_ERR。

#include <signal.h>

ypedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

### sigaction

sigaction函数用于改变进程接收到特定信号后的行为。该函数的第一个参数为信号的值，可以为除SIGKILL及SIGSTOP外的任何一个特定有效的信号（为这两个信号定义自己的处理函数，将导致信号安装错误）。第二个参数是指向结构sigaction的一个实例的指针，在结构sigaction的实例中，指定了对特定信号的处理，可以为空，进程会以缺省方式对信号处理；第三个参数oldact指向的对象用来保存原来对相应信号的处理，可指定oldact为NULL。如果把第二、第三个参数都设为NULL，那么该函数可用于检查信号的有效性

#include <signal.h>

int sigaction(int signum, const struct sigaction \*act,

struct sigaction \*oldact);

## 发送信号

（1） int raise(int sig); 对当前进程发送指定信号

（2） int pause(void); 将进程挂起等待信号

（3） int kill(pid\_t pid,int sig); 通过进程编号发送信号

（4） unsigned int alarm(unsigned int seconds); 指定时间（秒）发送SIGALRM信号。 seconds 为0 时取消所有已设置的alarm请求；

（5）int sigqueue(pid\_t pid,int sig,const union sigval val);类似于kill函数，多了附带共用体 union sigval形数，将共用体中的成员 int sival\_int 或 void \*sival\_ptr 的值传递给 信号处理函数中的定义类型 siginfo\_t 中的 int si\_int 或 void \*si\_ptr；

（6）int setitimer(int which,const struct itimerval \*value,struct itimerval \*oldvalue); 可定时发送信号，根据which可指定三种信号类型：SIGALRM、SIGVTALRM 和 SIGPROF；作用时间也因which值不同而不同;struct itimerval 的成员 it\_interval定义间隔时间，it\_value 为0时，使计时器失效；

（7） void abort(void) 将造成进程终止；除非捕获SIGABORT信号;

## 信号集

sigfillset(sigset\_t \*set); 设置所有的信号到set信号集中；

sigemptyset(sigset\_t \*set); 从set信号集中清空所有信号；

sigaddset(sigset\_t \*set,int sig);在set信号集中加入sig信号；

sigdelset(sigset\_t \*set,int sig);在set信号集中删除sig信号；

## 阻塞信号

int sigprocmask(int how,const sigset\_t \*set,sigset\_t \*set); 根据how值，设置阻塞信号集，或释放阻塞的信号集

int sigpending(sigset\_t \*set); 获取在阻塞中的所有信号；

int sigsuspend(const sigset\_t \*set); 类似于 pause()函数！

# 内核驱动

## USB

### 框架

usb\_device\_descriptor{ //usb设备描述符

}

// USB 端点被绑在接口中. USB 接口只处理一类 USB 逻辑连接

usb\_interface\_descriptor{ //usb接口描述符

}

// 端点是USB 通讯的最基本形式，端点可看作一个单向的管道

usb\_interface\_descriptor{ //端点描述符

}

usb\_config\_descriptor{ //配置描述符

}

### USB主机控制器

usb\_hcd{ //usb主机控制器驱动

const char \*product\_desc //厂商名称字符串

char irq\_descr[24] //驱动和总线类型

const struct hc\_driver \*driver //控制器驱动使用的回调函数

unsigned rh\_registered:1 //是否注册根hub

int irq //控制器的中断请求号

}

usb\_hcd可看作一个通用的控制器描述符，ohci\_hcd是usb\_hcd的具

体实现

ohci\_hcd{ // OHCI控制器驱动

struct ohci\_regs \_\_iomem \*regs //用于HC通信的主内存地址

}

### usb设备驱动

设备驱动结构：

usb\_driver{

const char \*name

int (\*probe) (struct usb\_interface \*intf,

const struct usb\_device\_id \*id)

void (\*disconnect) (struct usb\_interface \*intf)

}

USB请求块：

urb{

struct kref kref //主引用计数

}

struct urb \*usb\_alloc\_urb(int iso\_packets, gfp\_t mem\_flags)

void usb\_free\_urb(struct urb \*urb)

### usb驱动程序框架

基本数据结构：

usb\_skel{ //usb设备

struct usb\_device \*udev

struct usb\_interface \*interface

unsigned char \*bulk\_in\_buffer

}

usb\_skel\_init()

usb\_skel\_exit()

## USB storage

### 初始化

struct usb\_driver usb\_storage\_driver{

.probe = storage\_probe

struct usbdrv\_wrap drvwrap{

struct device\_driver driver

int for\_devices

}

}

struct bus\_type usb\_bus\_type{

.match = usb\_device\_match

}

usb\_stor\_init()

usb\_register(&usb\_storage\_driver)

usb\_register\_driver(struct usb\_driver \*new\_driver)

new\_driver->drvwrap.driver.bus = &usb\_bus\_type

new\_driver->drvwrap.driver.probe = usb\_probe\_interface

retval = driver\_register(&new\_driver->drvwrap.driver)

struct scsi\_host\_template usb\_stor\_host\_template{

.queuecommand = queuecommand

}

storage\_probe()

struct Scsi\_Host \*host

host = scsi\_host\_alloc(&usb\_stor\_host\_template, sizeof(\*us))

usb\_stor\_acquire\_resources(us)

//创建usb\_stor\_control\_thread内核线程

th = kthread\_create(usb\_stor\_control\_thread)

wake\_up\_process(th)

usb\_stor\_control\_thread()

scsi\_add\_host(host, &intf->dev)

//创建usb\_stor\_scan\_thread内核线程

th = kthread\_create(usb\_stor\_scan\_thread)

wake\_up\_process(th)

usb\_stor\_scan\_thread()

### 读写

1. scsi层:当用户需要访问usb设备，scsi-core会调用scsi\_dispatch\_cmd

函数

scsi\_dispatch\_cmd()

struct Scsi\_Host \*host = cmd->device->host

host->hostt->queuecommand(cmd, scsi\_done)

usb\_stor\_host\_template. queuecommand()

queuecommand()

//唤醒usb control线程

srb->scsi\_done = done

us->srb = srb

up(&(us->sema))

2. queuecommand唤醒了usb\_stor\_control\_thread线程