Linux API速查手册

书栈(BookStack.CN)

目　录

- stat
- basename
- 目录操作
- getcwd

进程控制

- fork
- vfork
- exec~
- wait

进程通信

管道

- pipe
- mkfifo

信号处理

- 信号类型
- psignal
- kill
- raise
- signal
- sigaction
- 信号阻塞
- sigsuspend

IPC对象/XSI-IPC

- 消息队列
  - msgget
  - msgctl
  - msgrcv
- 信号量
  - semget
  - semctl
  - semop
- 共享内存
  - shmget
  - shmctl
  - shmdt

网络编程

- 套接字结构
- 套接字函数

致谢

当前文档 《Linux API速查手册》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-01-31。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/linuxapi

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Linux API速查手册

https://github.com/guodongxiaren/LinuxAPI/wiki

果冻虾仁的Linux系统编程学习记录

Linux环境编程API

内容概要

C语言API包含部分标准C的API、POSIX标准的系统编程API(一些Linux独有的系统API会单独注明)。

头文件源码

大部分头文件源码在**/usr/include**目录下。

安装man手册

因为涉及到大量的POSIX编程。所以最好下载POSIX函数的man手册。

Ubuntu

```
1. sudo apt-get install manpages-posix
2. sudo apt-get install manpages-posix-dev
```

默认安装了**manpages-dev**，所以不装POSIX的man手册是可以查看绝大部分API的。
但是不装的话，有些API是不能看到的，比如**posix_spawn**函数。

CentOS

```
1. yum install man-pages.noarch
```

> 关于目录
>
> > 右侧的目录并非以函数为索引依据，主要是以系统的man手册页面为索引依据。
> > 比如exec里面包含6个函数、pipe里面包含pipe()和pipe2()两个函数，但是它们都是属于一个man页面中。

涉及头文件

- time.h
- sys/time.h

结构体

struct timeval

```
/* 在头文件<sys/time.h>中定义 */

struct timeval {
    time_t      tv_sec;     /* 秒 */
    suseconds_t tv_usec;    /* 微秒 */
};
```

实际上结构体成员中的秒和微秒都是 `long` 类型。

struct timespec

```
struct timespec {
    long    tv_sec;         /* 秒 */
    long    tv_nsec;        /* 纳秒 */
};
```

struct tm ANSI C

```
/* 在头文件<time.h>中定义 */
struct tm
{
    int tm_sec;            /* 秒.    [0-60] (1 leap second) */
    int tm_min;            /* 分钟.    [0-59] */
    int tm_hour;        /* 小时.    [0-23] */
    int tm_mday;        /* 天.        [1-31] */
    int tm_mon;            /* 月.    [0-11] */
    int tm_year;        /* 年    - 1900.  */
    int tm_wday;        /* 每周第几天.    [0-6] */
    int tm_yday;        /* 每年第几天.[0-365]    */
    int tm_isdst;        /* DST.        [-1/0/1]*/
    ...
};
```

ANSI

- time
- difftime
- localtime

函数原型 ANSI C

```
1. #include <time.h>
2.
3. time_t time(time_t *t);
```

参数

如果t是非空指针，那么该函数的返回值也将保存在t中。

返回值

返回自epoch（1970年1月1日00:00:00）以来的经过了的总秒数。

epoch被称为UNIX纪元。

计算两个时间的差值

一般人们会用两次time()的结果直接相减来获取差值，但是time_t的类型ANSI C并未规定，有的系统是整型有的系统是浮点型或其他编码。为了兼容性，ANSI C的difftime()统一返回双精度浮点数。

函数原型

```
1. #include <time.h>
2. 
3. double difftime(time_t time1, time_t time0);
```

备注

实际上在POSIX系统中，常被定义成：

```
1. #define difftime(t1,t0) (double)(t1 - t0)
```

函数原型

```
#include <time.h>

struct tm *localtime(const time_t *timep);
```

描述

将参数中timep代表的秒数转换为结构体tm类型。出现错误时返回NULL。
注意，此时参数timep中的必须是经过time()初始化过的变量。
所以通常的用法是：

```
struct tm* t;
time_t currentTime;

time(&currentTime);
t = localtime(&currentTime);
```

注意返回的**tm**的年份是从**1900**开始计数的，月份是从**0**开始计数的

tm struct

```
struct tm {
    int tm_sec;    /* Seconds (0-60) */
    int tm_min;    /* Minutes (0-59) */
    int tm_hour;   /* Hours (0-23) */
    int tm_mday;   /* Day of the month (1-31) */
    int tm_mon;    /* Month (0-11) */
    int tm_year;   /* Year - 1900 */
    int tm_wday;   /* Day of the week (0-6, Sunday = 0) */
    int tm_yday;   /* Day in the year (0-365, 1 Jan = 0) */
    int tm_isdst;  /* Daylight saving time */
};
```

POSIX

- getitimer/setitimer

函数原型

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

参数

结构体timeval

```
struct timeval
{
    long tv_sec;/*秒数*/
    long tv_usec;/*微秒数*/
};
```

当前时间会返回给这个结构体指针tv。

结构体timezone

可以设为NULL。

返回值

- 0 ：成功
- -1: 失败

系统管理

函数	描述
uname	得到内核的名称和信息
getpwuid	通过**uid**获得相应的结构体passwd
getpwnam	通过用户名获得相应的结构体passwd
getspnam	通过用户名获得结构体spwd（内包含密码）
getgrgid	通过**gid**获得相应的结构体group
getgrnam	通过组名获得相应的结构体group
getenv	获取系统环境变量的值

函数原型

```
#include <sys/utsname.h>

int uname(struct utsname *buf);
```

参数

该函数的参数是用来返回的，即声明一个结构体utsname类型的变量，然后放入函数中。待uname()执行完毕后，会将系统内核信息返回到这个结构体utsname变量中。

返回值

成功返回0，失败返回-1，并设置errno。

---

utsname

```
struct utsname {
    char sysname[];    /* 操作系统名称 (如, "Linux") */
    char nodename[];   /* Name within "some implementation-defined
                      network" */
    char release[];    /* 操作系统发行版本 (如, "2.6.28") */
    char version[];    /* 操作系统版本 */
    char machine[];    /* 硬件标识符 */
    #ifdef _GNU_SOURCE
    char domainname[]; /* NIS 或 YP 域名 */
    #endif
};
```

函数原型

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

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);

int getpwnam_r(const char *name, struct passwd *pwd,
                   char *buf, size_t buflen, struct passwd **result);
int getpwuid_r(uid_t uid, struct passwd *pwd,
                   char *buf, size_t buflen, struct passwd **result);
```

> 缩写的含义
> **getpwnam**–->get password name
> **getpwuid**–->get password uid

结构体passwd

定义在头文件*pwd.h*中

```
struct passwd {
        char   *pw_name;       /* 用户名 */
        char   *pw_passwd;     /* 用户密码 */
        uid_t   pw_uid;        /* 用户ID */
        gid_t   pw_gid;        /* 用户组ID */
        char   *pw_gecos;      /* 用户信息 */
        char   *pw_dir;        /* home目录 */
        char   *pw_shell;      /* shell程序 */
};
```

函数原型

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

struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
int getgrnam_r(const char *name, struct group *grp,
          char *buf, size_t buflen, struct group **result);

int getgrgid_r(gid_t gid, struct group *grp,
          char *buf, size_t buflen, struct group **result);
```

> 缩写含义
> **getgrnam**–->get group name
> **getgruid**–->get group uid

结构体group

定义在头文件*grp.h*中

```
struct group {
        char   *gr_name;        /* 组名 */
        char   *gr_passwd;      /* 组密码 */
        gid_t   gr_gid;         /* 组ID */
        char  **gr_mem;         /* 组成员 */
};
```

命令行参数识别

函数原型

```
#include <unistd.h>
int getopt(int argc, char * const argv[],
            const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

#include <getopt.h>

int getopt_long(int argc, char * const argv[],const char *optstring,
const struct option *longopts, int *longindex);

int getopt_long_only(int argc, char * const argv[],const char *optstring,
   const struct option *longopts, int *longindex);
```

参数

getopt的前两个参数无须多言，关键是第三个optstring

- 单个字符，表示选项
- 单个字符后面接一个冒号:表示该选项后必须跟一个参数。参数紧跟在选项后或者用空格隔开，该参数的指针赋给optarg
- 单个字符后面接两个冒号::表示该选项后可以跟一个参数，且必须紧跟在选项后，不能以空格隔开，该参数的指针赋给optarg

返回值

getopt()成功执行后将返回第一个选项，并设置如下全局变量。

- optarg: 指向当前选项的参数（如果有）的指针
- optind: 再次调用getopt()时的下一个argv指针的索引
- optopt: 存储不可知或错误选项

文件IO

- ANSI
- POSIX

ANSI

- fread
- fgets
- ferror
- rewind
- ftell
- fopen
- fclose

fwrite

函数原型

```
#include <stdio.h>

size_t fwrite(const void *ptr,size_t size,size_t n,FILE *fp);
```

参数

ptr为基本类型或自定义的结构体类型的指针。
size代表要写入的数据字节数，通常为sizeof(数据类型)。
n代表写入的数据的个数。
fp即为文件指针。

返回值

返回写入的数据的个数。
如果成功写入则，文件内部指针会向右移动n*size。

fread

函数原型

```
#include <stdio.h>

size_t fread(void *ptr,size_t size,size_t n,FILE *fp);
```

参数

与fwrite含义相同。

返回值

fread不区分文件结束和错误。如有必要，请使用feof和ferror。

fputs

写入字符串到文件

函数原型

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

功能描述

将s指向的字符串，舍去结束标记'\0'后写入到fp指向的文件缓冲区。

返回值

返回写入的实际字符数；出现错误，返回EOF。

fgets

从文件读取字符串

函数原型

```
char *fgets(char *s,int size,FILE *fp);
```

功能描述

从fp所指向的文件缓冲区当前位置读取n-1个字符，
在其后补充一个'\0'，并写入到s指向的内存区。

返回值

返回s对应的地址，可以理解为返回字符串。出现错误返回NULL。

feof/ferror

函数原型

```
1. int feof(FILE *fp);
2. int ferror(FILE *fp);
```

功能描述

feof

测试fp指向的流是否设置了EOF标记（end-of-file indicator）。
即是否到达文件结尾。
该标记只能被clearerr函数清除。
若fp当前位置是文件结尾，则返回非0值，否则返回0。

ferror

测试fp指向的流是否设置了错误标记（error indicator）。
是否到达文件结尾。
该标记只能被clearerr函数清除。
若该标记被设置，则返回非0值。即表示出错。

示例

```
1. ...
2. while(!feof(fp))
3. {
4.     ...
5. }
```

若不是文件结尾则执行循环。

fseek

用于跳跃式地移动文件读写位置。

函数原型

```
#include <stdio.h>

int fseek(FILE *fp,long offset,int whence);
```

参数

offset为相较whence的偏移量。
whence为文件内部指针的基准:

- `SEEK_SET` 文件开始位置，其值为0
- `SEEK_CUR` 文件当前位置，其值为1
- `SEEK_END` 文件结束位置，其值为2

返回值

如果执行成功，将返回0。如果失败返回-1

rewind

rewind英文释义是 `倒带` 的意思。这里是重置文件指针到所指向的文件的开始位置。

函数原型

```
#include <stdio.h>

void rewind(FILE *fp);
```

ftell

tell就是说话的意思，该函数功能为报告当前位置距离文件开始处是第几个字符

函数原型

```
#include <stdio.h>

long ftell(FILE *fp);
```

返回值

返回fp所指向文件流的当前位置。如果出错返回-1

函数原型

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

`fopen`

参数

- path 要打开的文件路径名
- mode 文件的打开方式，6种取值

mode

字符	描述
r	只读，文件必须已存在
r+	允许读写，文件必须已存在
w	只写，文件不存在在创建，已存在则覆盖原内容写入
w+	允许读写，文件不存在在创建，已存在则覆盖原内容写入
a	只允许追加数据，文件不存在则创建
a+	允许读和追加数据，文件不存在则创建

返回值

如果调用成功，返回文件指针；否则返回**NULL**并设置适当的 `errno` 信息。

释放文件指针指向的资源，同时使文件描述符失效。

函数原型

```
#include <stdio.h>

int fclose(FILE *fp);
```

参数

参数为文件指针

返回值

返回值	状态
0	成功
EOF	失败

失败时会设置适当 `errno` 的值。

> EOF在stdio.h中定义，值为-1。

POSIX

- open
- read
- write
- lseek
- fcntl
- dup

基于文件描述符的文件打开方式

函数原型

```
#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);
```

> stat是status的缩写
>
> fcntl是file control的缩写

参数

flags

flags字段使用POSIX的几个宏，此时必须包含头文件 `<fcntl.h>` 才行。
可以是下面几个宏的逻辑或组合。这些宏共有三种类型：

访问方式	描述
O_RDONLY	只读
O_WRONLY	只写
O_RDWR	可读写

打开时标志	描述
O_CREAT	创建文件，需要指定第餐个参数mode
O_EXCL	与 `O_CREAT` 联用，如果文件已存在则返回错误
O_TRUNC	将清空文件的内容，仅对普通文件有用
O_NOCTTY	若打开的文件是终端设备，不让它作为该进程的控制终端
O_NOBLOCK	以非阻塞模式打开

IO操作方式	描述
O_APPEND	把数据写到文件末尾
O_NONBLOCK	对文件的read()/write()，当无立即可用输入(或输出不能立即写出)时能以EAGAIN错误状态标志立即返回
O_ASYNC	(异步)此标志被设置，文件描述符有输入数据时会生成SIGIO信号
O_SYNC	
O_DSYNC	

O_RSYNC

> `O_EXCL` 的EXCL是Exclusive（排他的，专有的）的缩写
>
> > 如果在打开已有文件的时候，没有使用 `O_TRUNC` 参数，而只是将fd的偏移置为文件头，
> > 这样会有问题，即之前的内容没有没删除，而是逐个覆盖。如果第二次输入的内容少于文件本身的内容，
> > 则会出现问题。

---

1. 一个进程当前有哪些打开的文件描述符可以通过/proc/进程ID/fd目录查看

函数原型

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

参数

参数	描述
fd	文件描述符
buf	读取的数据存放在buf指针指向的缓冲区
count	读取的字节数

关于**count**：如果**buf**是一个字符数组名，那么**count**就用它的**sizeof**值。若**buf**是字符指针（字符串）则**count**用它的**strlen**值。

返回值

若果函数执行成功，返回读取的字节数，如果遇到EOF，则返回**0**。出错返回**-1**，并设置相应errno值。

- 当我指定要读取100个字节的时候，在读完30个字节后，遇到了EOF，那么这时立即返回30，接下来继续执行read函数的时候返回0。
- 从终端设备读，通常以行为单位，读到换行符就返回。
- 当出错时（即返回-1），如果errno的值是**EINTR**，表示遇到调用信号而中断了读取，那么我们可以再次尝试read。

相关函数

|[[write|write]]|[[fread|fwrite-fread#fread]]|
|————|————|

函数原型

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

参数

参数同read函数

参数	描述
fd	文件描述符
buf	读取的数据存放在buf指针指向的缓冲区
count	读取的字节数

返回值

如果函数调用成功，返回值为写入的字节数；否则返回值为**-1**，并设置相应的errno值。

函数原型

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

off_t lseek(int fd, off_t offset, int whence);
```

参数

- fd是文件描述符
- offset是偏移量
- whence是偏移量的基准位置。它的取值有三个
    - *SEEK_SET*： 开始位置
    - *SEEK_CUR*： 当前位置
    - *SEEK_END*： 末尾位置

> 为什么开始位置的后缀是_SET
>
> > 实际上，在man手册中可以看出。这三个宏的描述是
> >
> > - SEEK_SET The offset is set to offset bytes.
> > - SEEK_CUR The offset is set to its current location plus offset bytes.
> > - SEEK_END The offset is set to the size of the file plus offset bytes.

fd的偏移量

在内核中对一个文件描述符（fd）的偏移量只维护一个值，也就是说你用读写方式打开一个文件，如果先用read读取了n个字符，紧接着用write写入了n个字符，那么后来写入的n个字符并不是从文件第一个字符位置开始的，而是从n+1个字符位置开始的。所以通常我们需要使用lseek来使fd的偏移量置于文件开始位置。

函数原型

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

int fcntl(int fd, int cmd, ... /* arg */ );
```

参数

参数可能有两个，也可能有三个，具体看第二个参数的取值。

- fd: 文件描述符
- cmd: 命令
- arg: 命令的参数

常用cmd	arg	描述
F_DUPFD		复制文件描述符
F_GETFD	无	获取文件描述符标签
F_SETFD		设置文件描述符标签
F_GETFL		获取文件状态标签
F_SETFL		设置文件状态标签
F_GETFLK		获取文件锁
F_SETFLK		设置文件锁
F_SETLKW		类似**F_SETLK**，但等待完成
F_GETOWN		获取收到SIGIO信号的进程或进程组ID
F_SETOWN		设置接收SIGIO信号的进程或进程组ID

文件描述符标签

- 文件描述符标签（flags）是一个整型，它的每一个二进制为，表明一种标志。
- 复制的文件描述符，标签不会被复制，每个文件描述符有各自的标签

当前，只有一个标志**FD_CLOEXEC**，表明当执行exec()函数时，将关闭该文件描述符。默认情况下，此位是清除的，所以在执行exec()之后，之前的文件描述符会保留。

```
fcntf(fd,F_SETFD,FD_CLOEXEC);//设置该标志，但是其他标志就消失了。
//良好的写法是：
int oldflags = fcntl(fd,F_GETFD);
oldflags |= FD_CLOEXEC;//设置该标志
fcntl(fd,F_SETFD,oldflags);

//如果要清除该标志
```

```
8.  oldflags &= ~FD_CLOEXEC;
```

文件状态标签

文件状态标签被所有复制的文件描述符共享。表明文件打开的属性，也就是open()的flags参数所指明的。
当判断一个文件的读写标志时，不能简单的通过&操作来判断，因为文件的读写标志有三种状态，系统中并非设置了三个独立位来表示，实际上是用了两个位来表示的。所以最好使用**O_ACCMODE**(0x3)来进行&操作。
其他的标志可以直接用&来判断，如 `flags&O_APPEND`

复制文件描述符号

函数原型

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

另外还有dup3()，不常用。

dup, dup2

- dup参数是一个文件描述符，返回一个文件描述符，值是当前未使用的最小数字，指向的位置和参数相同。
- dup2，可以自己指定要返回的文件描述的数值newfd，如果newfd是一个已经打开的文件描述符，则会将其关闭。

dup2, fcntl

```
dup2(fd,fd2);
//等价于
close(fd2);
fcntl(fd,F_DUPFD,fd2);
```

功能上可以等价于close()和fcntl()的组合，但是dup2是一个原子操作（关闭fd2，和复制fd不会被中断）。

Linux和各种Unix-like系统中有一重要概念——万物皆文件

分类

- [x] 根据处理方法的不同，分为：

- 缓冲区文件
- 非缓冲区文件

- [x] 根据其数据组织形式的不同，分为：

- 文本文件
- 二进制文件

- [x] 根据其存放数据的作用的不同，分为：

- **-** 普通文件（regular）
- **d** 目录文件
- **l** 符号链接文件
- 设备文件
    - **b** 块设备文件
    - **c** 字符设备文件
- **p** 知名管道文件（FIFO）
- **s** 套接字文件（socket）

索引节点

索引节点（inode）所包含的信息都封装在结构体stat中：

```
1.  struct stat {
2.      dev_t     st_dev;     /* ID of device containing file */
3.      ino_t     st_ino;     /* inode number */
4.      mode_t    st_mode;    /* protection */
5.      nlink_t   st_nlink;   /* number of hard links */
6.      uid_t     st_uid;     /* user ID of owner */
7.      gid_t     st_gid;     /* group ID of owner */
8.      dev_t     st_rdev;    /* device ID (if special file) */
9.      off_t     st_size;    /* total size, in bytes */
10.     blksize_t st_blksize; /* blocksize for filesystem I/O */
11.     blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
12.     time_t    st_atime;   /* time of last access */
13.     time_t    st_mtime;   /* time of last modification */
14.     time_t    st_ctime;   /* time of last status change */
15. };
```

函数原型

```
#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);
```

实际执行需要root权限。也就是使用root身份或者sudo来运行该程序的可执行文件。

> If the owner or group is specified as -1, then that ID is not changed.

区别联系

- lchown修改符号链接时，修改的是符号链接权限。
- chown修改符号链接时，修改符号链接指向的文件的权限。
- fchown修改符号连接时，行为和chown相同。只不过它的参数是文件的描述符。

给文件改名，移动文件位置。

函数原型

```
1. #include <stdio.h>
2. 
3. int rename(const char *oldpath, const char *newpath);
```

返回值

成功返回0，失败返回-1，并设置相应errno。

函数原型

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

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

三个函数的关系与chown，fchown，lchown的关系相同。

- stat检查符号链接时，实际检查的是符号链接所引用的文件
- lstat检查符号链接时，检查的是符号链接本身
- fstat功能与stat相同，不过它的参数是文件的描述符

返回值

成功返回**0**；失败返回**-1**，并设置相应errno的值。

结构体stat的内容

```
struct stat {
    dev_t     st_dev;     /* 设备号（主设备号，次设备号）*/
    ino_t     st_ino;     /* inode的数量 */
    mode_t    st_mode;    /* 文件的类型和存取的权限 */
    nlink_t   st_nlink;   /* 硬链接的数量 */
    uid_t     st_uid;     /* 所用者的uid */
    gid_t     st_gid;     /* 所有者的组id */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* 总大小,字节数 */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* 最后访问时间（access）*/
    time_t    st_mtime;   /* 最后修改时间（modification）文件内容的改动时间 */
    time_t    st_ctime;   /* 最后改动时间（change）文件属性的改动时间 */
};
```

basename()和**dirname()**是一对。

函数原型

```
#include <libgen.h>

char *dirname(char *path);

char *basename(char *path);
```

功能比较

path | dirname| basename
|—|—|—
/usr/lib| /usr| lib
/usr/ | / | usr
usr | . | usr
/ | / | /
. | . | .
.. | . | ..

也可以用于**url**。

> 目录其本质也是一种文件，它的r权限是 `ls` ，x权限是 `cd`

DIR结构体

Unix系统为用户提供了一种和文件结构**FILE**类似的目录结构**DIR**。它被称为目录流，目录中的目录项用dirent结构表示（但*DIR*的并非包含*dirent*成员）

dirent结构

结构体成员	类型	描述
d_into	ino_t	文件的inode号
d_name[ ]	char	以NULL结尾的文件名

常用函数（库调用）

函数名	描述	
[[opendir	opendir]]	根据目录的路径字符串，返回一个DIR指针
[[readdir	readdir]]	根据DIR指针，返回DIR指针所指向的目录项的指针（dirent *）
[[closedir	closedir]]	关闭DIR指针所指向的目录流
[[telldir	telldir]]	返回DIR指针所指向目录流的当前位置（long类型）
[[seekdir	seekdir]]	将DIR指针所指向的目录流偏移跳跃到某一位置

函数原型

```
#include <unistd.h>

char *getcwd(char *buf,size_t size);
```

返回值

返回当前目录名的绝对路径。

进程控制

fork

新建一个子进程。具有一次调用，两次返回的特点。

evecve

有6种表达：

从父进程派生出子进程，子进程完全拷贝父进程的stack，data，heap segment。

> 两者并不共享地址空间，所以的变量是独立的，一方修改，另一方不会变化。

函数原型

```
1. #include <unistd.h>
2.
3. pid_t fork(void);
```

特点

一次调用，两次返回

函数原型

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

pid_t vfork(void);
```

描述

同样是创建子进程，其效率比fork()要快。两者区别有：

- vfork()不会创建并复制父进程的地址空间，而是和父进程共享
- vfork()会阻塞父进程，只运行子进程运行
- 当子进程调用exec()或_exit()时，内核返回地址空间给父进程并唤醒它

exec函数族的作用是让fork出的子进程能够执行与父进程不同的代码段，实现不同的功能。

exec由6个函数组成

```
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 encp[]);

int execv(const char *path,const char *argv[]);
int execvp(const char *file,const char *argv[]);
int execve(const char *path,const char *argv[],char *const envp[]);
```

如何记忆

字符	原型	描述
l	list	选项列表，对应函数的省略号，可以写多个参数，以NULL结尾
v	vector	将命令的多个选项放到一个数组里，作为第二个参数
p	path	在系统PATH变量的路径里搜索，无p字符的函数则需要给出可执行文件的全路径名
e	enviroment	表示将一份新的环境变量传给他？

样例

```
execlp("ls", "ls", "-a", (char *)NULL);
char *v[] = {"ls", "-a", NULL};
execvp("ls", v);
```

函数原型

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

pid_t wait(int *status);
```

返回值

-1	错误
其他	被终止的子进程的id

错误类型

错误时，系统记录的错误代码**errno**，有两种：

ECHILD	没有子进程
EINTER	收到中断信号**signal**，立即返回

检测退出状态的宏

宏	缩写含义	描述
WIFEXITED	wait if exited	子进程正常退出时返回真值
WEXITSTATUS	wait exit status	当上面宏为真时，返回子进程正常退出时状态
WIFSIGNALED	wait if signaled	子进程由于信号导致终止，返回真值
WTERMSIG	wait terminate signal	当上面宏为真时，返回终止子进程的信号类型
WIFSTOPPED	wait if stopped	
WSTOPSIG	wait stop signal	
WIFCONTINUED	wait if continued	

上表记忆方式为3+1,三对加一单

> 另外有书中提及**WCOREDUMP**，即wait core dump（核心转储），不过man手册中未提及此宏

进程通信

进程间通信（Inter-Process Communication），简称**IPC**。

分类

- 管道通信
- 信号通信
- 共享内存通信*
- 信号量通信*
- 消息队列通信*
- 套接口（Socket）通信
- 全双工管道通信（部分Unix系统支持，Linux只支持半双工管道）

> 加星号*的三种通信方式，是源自于AT&T发行的System V(SYSV)版本的新IPC机制。

管道

- 管道**pipe**
- 命名管道**FIFO**

FIFO

命名管道FIFO，应该是静态实体，因此创建方式有两种：

- 函数*mkfifo*创建
- 终端使用命令*mknod*，*mkfifo* 创建

通道

- `pipe`
- `mkfifo`

创建一个管道，Linux系统中有pipe()和pipe2()两个函数。

> Linux仅支持半双工管道

pipe

函数原型

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

参数

```
pipefd[0]  读端
pipefd[1]  写端
```

返回值

```
0  成功
-1 失败，并自动设置 errno
```

---

pipe2

pipe2非POSIX标准，仅被Linux支持

函数原型

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

int pipe2(int pipefd[2],int flags);
```

3 创建一个命名管道

函数原型

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

int mkfifo(const char *pathname, mode_t mode);
```

参数

第二个参数的语义与[[open()|open]]的第3个参数相同，即权限。

信号

信号是一种进程间通信（IPC）机制，主要用于处理异步事件。

> 不同的Unix衍生版所支持的信号类型并不完全相同。除了支持**POSIX**规定的信号外，还支持其他信号。

术语解释

术语	解释
生成信号	发生了一个需要引起进程注意的事件而导致信号出现时。也叫发送信号
信号交付	被发送信号的那个进程识别到了信号并采取了适当动作。也叫接收信号
信号句柄	当信号出现时调用进行专门处理的函数。这个函数称为捕获函数或信号句柄
信号捕获	若信号交付时进程执行信号句柄，称进程捕获了信号
悬挂信号	当一个信号已经生成，但还未交付时，称该信号是悬挂的

信号类型

Linux支持62个信号。可以通过在终端输入 `kill -l` 或 `man 7 signal` 来查看。本页附录也有记录。

信号的宏定义和编号都定义在**signal.h**中。

信号产生

信号的产生方式多种多样，主要有3种:

- 程序错误
  - 程序异常，如零做除数
  - 进程超越CPU或文件大小的限制
- 外部事件
  - 通过键盘终端
- 显示请求
  - kill命令（或函数）

通过键盘终端

组合键	信号
Ctrl+C	SIGINT
Ctrl+\	SIGQUIT
Ctrl+Z	SIGSTOP

通过kill命令

```
1. kill -信号编号 进程号        #比如 kill -15 4264
2. kill -信号的宏定义 进城号     #比如 kill -SIGTERM 4264
```

调用系统函数

- [[kill|kill]]
- [[raise|raise]]
- [[alarm|alarm]]
- [[abort|abort]]

信号处理

主要有3种：

1. 忽略信号，对该信号不做处理，进程继续执行。但**SIGKILL**和**SIGSTOP**不能忽略
2. 捕捉信号，使进程执行指定的程序代
3. 默认处理方法，系统为每一个信号都设置了默认处理方法，通常为终止进程：
    - 流产
    - 终止
    - 忽略
    - 挂起
    - 继续

> 还有其他的一些处理方法，比如将信号挂起，不影响当前程序的执行，待需要时再处理该信号。

Linux常用**31**个信号（**1~31**）。signal.h中有个常量**NSIG**定义了信号的个数，其值通常为为64。

编号	信号	编号	信号	编号	信号
1	SIGHUP	2	SIGINT	3	SIGQUIT
4	SIGILL	5	SIGTRAP	6	SIGABRT
7	SIGBUS	8	SIGFPE	9	SIGKILL
10	SIGUSR1	11	SIGSEGV	12	SIGUSR2
13	SIGPIPE	14	SIGALRM	15	SIGTERM
16	SIGSTKFLT	17	SIGCHLD	18	SIGCONT
19	SIGSTOP	20	SIGTSTP	21	SIGTTIN
22	SIGTTOU	23	SIGURG	24	SIGXCPU
25	SIGXFSZ	26	SIGVTALRM	27	SIGPROF
28	SIGWINCH	29	SIGIO	30	SIGPWR
31	SIGSYS				

这31个信号传统UNIX支持的信号，后来又扩充了一些信号（实时UNIX系统支持的信号）见附录

分类

程序错误类信号

程序终止类信号

- SIGHUP
- SIGINT（^C）中断。不产生core文件
- SIGKILL
- SIGQUIT(^)结束。会产生核心转储的core文件
- SIGTERM

闹钟类信号

IO类信号

作业控制类信号

- [ ] **SIGCHLD**:进程终止时，会向其父进程发送该信号。此信号默认动作是忽略。如果父进程想要在子进程状态发生改变时得到通知，就必须捕获此信号

操作错误类信号

其他信号

SIGUSR1和SIGUSR2这两个信号是专门留给用户应用程序自己定义使用的，默认动作是终止进程。

附录

编号	信号	编号	信号	编号	信号
34	SIGRTMIN	35	SIGRTMIN+1	36	SIGRTMIN+2
37	SIGRTMIN+3	38	SIGRTMIN+4	39	SIGRTMIN+5
40	SIGRTMIN+6	41	SIGRTMIN+7	42	SIGRTMIN+8
43	SIGRTMIN+9	44	SIGRTMIN+10	45	SIGRTMIN+11
46	SIGRTMIN+12	47	SIGRTMIN+13	48	SIGRTMIN+14
49	SIGRTMIN+15	50	SIGRTMAX-14	51	SIGRTMAX-13
52	SIGRTMAX-12	53	SIGRTMAX-11	54	SIGRTMAX-10
55	SIGRTMAX-9	56	SIGRTMAX-8	57	SIGRTMAX-7
58	SIGRTMAX-6	59	SIGRTMAX-5	60	SIGRTMAX-4
61	SIGRTMAX-3	62	SIGRTMAX-2	63	SIGRTMAX-1
64	SIGRTMAX				

> 31和34之间是没有32和33的。前面31个信号1（1~31）是不可靠信号(非实时的)，扩充的31个信号（34~64）称做可靠信号(实时信号)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会

函数原型

```
#include <signal.h>

void psignal(int sig, const char *msg);
```

描述

打印sig对应信号的描述信息到标准错误流。

参数

sig为信号对应的数。
msg如果不为NULL，那么将msg作为输出消息的前缀。在msg和消息描述之间默认会有一个冒号和一个空格。

相关函数

[[psiginfo|psiginfo]]

函数原型

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

int kill(pid_t pid,int sig);
```

描述

pid	描述
>0	kill发送信号sig给进程pid
0	kill发送信号给和当前进程在同一进程组的所有进程
-1	信号发送给系统内的所有进程
<-1	kill发送信号sig给进程组-pid中的每个进程

返回值

- 如果成功完成返回值0
- 失败返回-1，并设置errno

函数原型

```
#include <signal.h>

int raise(int sig);
```

描述

发送一个sig信号给当前进程。raise()是线程安全的函数。与kill()的不同之处是，kill()发射信号给指定的进程（通过pid参数）

当raise()发射的信号，导致了一个信号句柄被调用的时候，raise()在信号句柄返回之后被返回。

返回值

成功0，失败返回非**0**值（不一定是-1）

相关函数

[[kill|kill]]

对一个信号指定新动作或回到其原先的动作

函数原型

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

参数

signum即信号值。后面的handler就是处理这个信号的动作。它的值为:

- SIG_DFL: 默认动作
- SIG_IGN: 忽略该信号（SIGKILL和SIGSTOP无法忽略，因为要保证root的绝对控制权）
- 信号句柄

信号句柄

信号句柄即捕获函数（这里是函数指针类型），该函数必须是只有一个整型参数，且返回值为void。

信号发送时，如果建立了信号句柄，系统在把控制转移到信号句柄之前有两种做法:

- 将阻塞后继新的信号直至信号句柄完成为止
- 或这首先改变该信号的动作为SIG_DEL(相当于调用signal(signum,SIG_DEL)，在执行完一次信号句柄之后，将其恢复为默认动作)

BSD系统使用前者，系统V使用后一种。Linux默认采用BSD的做法，但当设置了特征测试宏 `_XOPEN_SOURCE` 时，则采用系统V的做法，此时编译时要这样编译

```
gcc test.c -D_XOPEN_SOURCE
```

返回值

返回值是指向信号signum前一次有效动作的指针。它和函数第二个参数类型相同。返回值是:

- SIG_DFL
- SIG_IGN
- 信号句柄指针

可以保存这个值，并且在以后用它作为函数第二个参数再次调用**signal()**，从而恢复信号原来的动作

如果出错，返回**SIG_ERR**并设置**errno**。唯一地错误码（errno）是**EINVAL**

相关函数

[[sigaction|sigaction]]

本节包括sigaction函数和结构体sigaction两个部分。

函数sigaction

`System Call`用于测试和改变一个信号的行为。

函数原型

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                  struct sigaction *oldact);
```

参数	描述
signum	指定要改变的信号
act	一个函数指针，如果不为空，则指定收到该信号后的行为
oldact	如果oldact不为空，则将原行为保存在函数指针oldact中

结构体sigaction

结构体原型

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
};
```

sa_handler

是一个函数指针，其含义与 signal 函数中的信号处理函数类似。

sa_sigaction

是另一个信号处理函数，它有三个参数，可以获得关于信号的更详细的信息。当 sa_flags 成员的值包含了 SA_SIGINFO 标志时，系统将使用 sa_sigaction 函数作为信号处理函数，否则使用 sa_handler 作为信号处理函数。

> 在某些系统中，成员 sa_handler 与 sa_sigaction 被放在联合体中，因此使用时不要同时设置。

sa_mask

成员用来指定在信号处理函数执行期间需要被屏蔽的信号，特别是当某个信号被处理时，它自身会被自动放入进程的信号掩码，因此在信号处理函数执行期间这个信号不会再度发生。

sa_flags

成员用于指定信号处理的行为，它可以是以下值的“按位或”组合。

- SA_RESTART：使被信号打断的系统调用自动重新发起。
- SA_ONSTACK：系统将在调用[[sigalstack|sigalstack]]替代信号栈上运行信号句柄；否则使用用户栈来交付信号。
- SA_NOCLDSTOP：使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号。
- SA_NOCLDWAIT：使父进程在它的子进程退出时不会收到 SIGCHLD 信号，这时子进程如果退出也不会成为僵尸进程。
- SA_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号。
- SA_RESETHAND：信号处理之后重新设置为默认的处理方式。
- SA_SIGINFO：使用 sa_sigaction 成员而不是 sa_handler 作为信号处理函数。

> 该段文字整理自博客园

相关函数

- [[signal|signal]]
- [[sigemptyset|sigemptyset]]

阻塞信号是保持该信号并推迟发送，直到阻塞解除，但不会丢失。

结构体sigset_t（信号集合）

其中每一位对应系统支持的一种信号。结构体内部是数组。

函数

函数名	描述	
[[sigemptyset	sigempty]]	初始化信号集为空集
[[sigfillset	sigfillset]]	初始化信号集包含全部信号
[[sigaddset	sigaddset]]	向信号集中添加信号
[[sigdelset	sigdelset]]	从信号集中删除信号
[[sigismember	sigismember]]	测试某信号是否在信号集中
[[sigprocmask	sigprocmask]]	使信号集中的信号被屏蔽掉
[[sigpending	sigpending]]	返回当前进程因受阻而未交付的信号集

注意前面四个函数只对信号集类型（sigset_t）变量进行改动，但是并不实际阻塞信号。
程序在使用sigset_t类型的数据对象之前，必须要调用sigemptyset()或sigfillset()这两个函数之一。

函数原型

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

描述

用mask所指的信号集临时替代调用进程的信号屏蔽，然后挂起调用进程直到有不属于mask的信号到达为止。

返回值

一直返回**-1**，有错误会设置errno

相关函数

[[pause()|pause]]

因为最早出自System V系统中，故又称System V IPC。
分为：

- 消息队列
- 信号量
- 共享内存

这个三种通信方式共用了许多概念。都用到一个头文件**ipc.h**。

> ipc.h位于/usr/include/linux/ipc.h

IPC结构

```
struct ipc_perm {
    key_t          __key; /* Key supplied to semget(2) */
    uid_t          uid;   /* Effective UID of owner */
    gid_t          gid;   /* Effective GID of owner */
    uid_t          cuid;  /* Effective UID of creator */
    gid_t          cgid;  /* Effective GID of creator */
    unsigned short mode;  /* 权限 */
    unsigned short __seq; /* Sequence number */
};
```

权限mode字段，对于IPC结构而言，都没有执行的权限。
消息队列和共享内存使用术语读（read）和写（write），而信号量使用术语读（read）和更改（alter）。

> 结构限制
> 每种结构都有内置的限制。Linux可以用 `ipcs -l` 命令查看

函数

公共函数

- ftok

其他函数

分类	创建函数	控制函数	独立函数		
消息队列	[[msgget	msgget]]	[[msgctl	msgctl]]	msgsnd/msgrcv
信号量	[[semget	semget]]	[[semctl	semctl]]	semop
共享内存	[[shmget	shmget]]	[[shmctl	shmctl]]	shmat/shmdt

操作

三个函数中都会使用一个操作参数。有四个操作是公共操作，定义在ipc.h中。
下面以消息队列举例子讲解这四个公共操作：

IPC_RMID

删除消息队列。从系统中删除给消息队列以及仍在该队列上的所有数据，这种删除立即生效。
仍在使用这一消息队列的其他进程在它们下一次试图对此队列进行操作时，将出错，并返回EIDRM。 此命令只能由如下两种进程执行：
其有效用户ID等于msg_perm.cuid或msg_perm.guid的进程。
另一种是具有超级用户特权的进程。

IPC_SET

设置消息队列的属性。按照buf指向的结构中的值，来设置此队列的msqid_id结构。
该命令的执行特权与上一个相同。

IPC_STAT

读取消息队列的属性。取得此队列的msqid_ds结构，并存放在buf*中。

IPC_INFO

读取消息队列基本情况。

消息队列

消息的链式队列。
重要的数据结构

msqid_ds

```
1.  //位置/usr/include/linux/msg.h
2.  struct msqid_ds {
3.      struct ipc_perm msg_perm;
4.      struct msg *msg_first;        /* 指向消息头 */
5.      struct msg *msg_last;        /* 指向消息尾 */
6.      __kernel_time_t msg_stime;    /* 最近msgsnd（发送消息）时间 */
7.      __kernel_time_t msg_rtime;    /* 最近msgrcv（接收消息）时间 */
8.      __kernel_time_t msg_ctime;    /* 最近修改时间 */
9.      unsigned long  msg_lcbytes;    /* Reuse junk fields for 32 bit */
10.     unsigned long  msg_lqbytes;    /* ditto */
11.     unsigned short msg_cbytes;    /* 当前队列的字节数 */
12.     unsigned short msg_qnum;    /* 当前队列中消息个数 */
13.     unsigned short msg_qbytes;    /* 队列最大字节数 */
14.     __kernel_ipc_pid_t msg_lspid;    /* 最近msgsnd的pid */
15.     __kernel_ipc_pid_t msg_lrpid;    /* 最近receive的pid */
16. };
```

key值与ID值

每一个IPC机制都有一个ID，只有ID值相同才能传递数据。它的类型是key_t(*int*)，但是设置任意数字为key值有违软件设计的思想。所以Linux提供了函数ftok来创建key值，以文件为参数，提高了与文件的关联度。

创建消息队列。如果把消息队列看做一个文件的话，那么该函数就相当于**open**。

函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

参数

第一个参数是key值。
第二个参数的地位用来确定消息队列的访问权限。可以附加参数：

- IPC_CREAT //如果key不存在，则创建*(类似open函数的O_CREAT)*
- IPC_EXCL //如果key存在，则返回失败*(类似open函数的O_EXCL)*
- IPC_NOWAIT //如果需要等待，则直接返回错误

比如： `msgid=msgget(key,0666|IPC_CREAT)`

返回值

成功执行时，返回消息队列标识符。失败返回-1，errno被设为以下的某个值

- EACCES：指定的消息队列已存在，但调用进程没有权限访问它，而且不拥有CAP_IPC_OWNER权能
- EEXIST：key指定的消息队列已存在，而msgflg中同时指定IPC_CREAT和IPC_EXCL标志
- ENOENT：key指定的消息队列不存在同时msgflg中不指定IPC_CREAT标志
- ENOMEM：需要建立消息队列，但内存不足
- ENOSPC：需要建立消息队列，但已达到系统的限制

该函数用来对消息队列的基本属性进行控制、修改。

函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

参数

- msqid为消息队列标识符。
- cmd为执行的控制命令(在ipc.h中定义):
    - `#define IPC_RMID 0`
    - `#define IPC_SET 1`
    - `#define IPC_STAT 2`
    - `#define IPC_INFO 3`
- buf

cmd字段除了公共的四个操作以外，还有自己独立的操作。

IPC_RMID

删除消息队列。从系统中删除给消息队列以及仍在该队列上的所有数据，这种删除立即生效。
仍在使用这一消息队列的其他进程在它们下一次试图对此队列进行操作时，将出错，并返回**EIDRM**。
此命令只能由如下两种进程执行:

- 其有效用户ID等于msg_perm.cuid或msg_perm.guid的进程。
- 另一种是具有超级用户特权的进程。

IPC_SET

设置消息队列的属性。按照**buf**指向的结构中的值，来设置此队列的msqid_id结构。
该命令的执行特权与上一个相同。

IPC_STAT

读取消息队列的属性。取得此队列的**msqid_ds**结构，并存放在**buf***中。

IPC_INFO

读取消息队列基本情况。

这是一对函数，用于消息队列的发送和接收

描述

msgsnd函数用于将新的消息添加到消息队列的尾端。
msgrcv函数用于从消息队列中读取msqid指定的消息

函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
                      int msgflg);
```

参数

msgsnd

- msqid: 消息队列标识符(由msgget生成)
- msgp: 指向用户自定义的缓冲区（msgp）
- msgsz:接收信息的大小。范围在0～系统对消息队列的限制值
- msgflg:指定在达到系统为消息队列限定的界限时应采取的操作。
    - IPC_NOWAIT 如果需要等待，则不发送消息并且调用进程立即返回，errno为**EAGAIN**
    - 如果设置为0，则调用进程挂起执行，直到达到系统所规定的最大值为止，并发送消息

msgrcv

- msqid: 消息队列标识符
- msgp: 指向用户自定义的缓冲区（msgp）
- msgsz: 如果收到的消息大于msgsz，并且msgflg&MSG_NOERROR为真，则将该消息截至msgsz字节，并且不发送截断提示
- msgtyp: 用于指定请求的消息类型:
    - msgtyp=0: 收到的第一条消息，任意类型。
    - msgtyp>0: 收到的第一条msgtyp类型的消息。
    - msgtyp<0: 收到的第一条最低类型（小于或等于msgtyp的绝对值）的消息。
- msgflg: 用于指定所需类型的消息不再队列上时的将要采取的操作:
    - 如果设置了IPC_NOWAIT，若需要等待，则调用进程立即返回，同时返回-1，并设置errno为**ENOMSG**
    - 如果未设置IPC_NOWAIT，则调用进程挂起执行，直至出现以下任何一种情况发生:
        - 某一所需类型的消息被放置到队列中。
        - msqid从系统只能怪删除，当该情况发生时，返回-1，并将errno设为**EIDRM**。
        - 调用进程收到一个要捕获的信号，在这种情况下，未收到消息，并且调用进程按 `signal(SIGTRAP)` 中指定的方式恢复执行。

返回值

毋庸置疑，成功0，失败-1。
其他要注意的是，消息队列数据结构（msqid_ds类型）成员的变化：

msgsnd

成功时：

- msg_qnum 以1为增量递增
- msg_lspid 设置为调用进程的pid
- msg_stime 设置为当前时间。

msgrcv

成功时：

- msg_qnum 以1为增量递减
- msg_lrpid 设置为调用进程的pid
- msg_rtime 设置为当前时间。

---

msgp

该类型需要自己在编程时定义，用于存储消息的内容。
下面给出一个范例，注意，里面的名称随意。

```
struct msgbuf{
    long mtype;   //消息类型
    char mtext[1];//数组大小编程时自己指定
};
```

原理

信号量通信机制主要实现进程间同步，信号量值用来标识系统可用资源的个数。
实际应用中，两个进程间通信可能会使用多个信号量，因此Linux在管理时以信号量集合的概念来管理。
通常所说的创建一个信号量实际上是创建了一个信号量的集合。整个信号量集合由以下部分组成：

- 信号量集合数据结构：在此结构中定义了整个信号量集合的基本属性，如访问权限。
- 信号量：信号量集合使用指针指向一个由数组组成的信号量单元，在此信号量单元中存储了创建时申请的信号量。

数据结构和共用体

semid_ds

信号量集合数据结构。一般不直接调用，而是作为[[semctl()|semctl]]的第四个参数semun的成员出现。

```
/*位于/usr/include/linux*/
struct semid_ds {
    struct ipc_perm    sem_perm;        /* 权限 .. see ipc.h */
    __kernel_time_t    sem_otime;        /* 最近semop时间 */
    __kernel_time_t    sem_ctime;        /*最近 修改时间*/
    struct sem    *sem_base;        /* 队列第一个信号量 */
    struct sem_queue *sem_pending;        /* 阻塞信号量 */
    struct sem_queue **sem_pending_last;    /* 最后一个阻塞信号量*/
    struct sem_undo    *undo;    /* undo队列 */
    unsigned short    sem_nsems; /* no. of semaphores in array */
};
```

sem

每个信号量的数据结构

```
/*位置不明。。*/
struct sem{
        int semval; /*信号量的值*/
        int sempid; /*最近一个操作的进程号PID*/
};
```

sembuf

被[[semop ( ) |semop]]用到

```
struct sembuf{
         unsigned short sem_num;  /* 信号量标号 */
         short          sem_op;   /* 信号量操作（加减） */
```

```
4.         short          sem_flg;  /* 操作标识 */
5. };
```

semun

这个是个union类型。注意它包含了几个不同类型的成员，具体用到哪个依据[[semctl()|semctl]]第三个参数的不同而不同。

```
1. union semun {
2.         int              val;    /* SETVAL的值 */
3.         struct semid_ds *buf;    /* IPC_STAT, IPC_SET的缓冲 */
4.         unsigned short  *array;  /* GETALL, SETALL的数组 */
5.         struct seminfo  *__buf;  /* IPC_INFO的缓冲(Linux-specific) */
6. };
```

> System V提供的三种IPC进制，有异曲同工之妙。

semget

创建信号量结合

函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

参数

参数含义，与msgget类似，只是比它多了第二个参数。

- key为ftok函数创建。
- nsems为创建的信号量的个数，每个信号量以数组方式存储。
- semflg用来标识信号量结合的权限。如**0700**。此外还可以附加以下ipc参数：

宏名	描述
IPC_CREAT	如果key不存在，则创建*(类似open函数的O_CREAT)*
IPC_EXCL	如果key存在，则返回失败*(类似open函数的O_EXCL)*
IPC_NOWAIT	如果需要等待，则直接返回错误

控制信号量集合、信号量

函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

参数

可以有三个参数，也可以有四个参数（利用的可变参数个数的函数定义）。

- semid是信号量集合的标识符，一般由semget函数返回。
- semnum为集合中信号量的编号。
    - 如果标识某个信号量，此值为信号量下标（0~n-1）
    - 如果标识整个信号量集合，则设置为0【与上面冲突？】
- cmd为要执行的操作：
    - 共有的IPC四个操作
    - `GETVAL` //返回的是semnum的值
    - `GETALL` //设置semnum为0，那么获取信号集合的地址传递给第四个参数。返回值为0，-1
    - `GETNCNT` //设置semnum为0，返回值为等待信号量值的递增进程数，否则返回-1
    - `GETZCNT` //设置semnum为0，返回值是等待信号量值的递减进程数，否则返回-1
    - `SETVAL` //将第四个参数指定的值设置给编号为semnum的信号量。返回值为0，1
    - `SETALL` //设置semnum为0，将第四个参数传递个所有信号量。返回值0，1

> GET的四个操作中，只有GETVAL成功时返回0，其余的都是返相应取的值。SET两个操作都返回的是值

- [第四个参数]，根据cmd的不同而不同。其类型为semun的联合：
    - 如果cmd为SETVAL，那么该参数为val
    - 如果cmd为IPC_STAT&IPC_SET，那么该参数为struct semid_ds结构体变量
    - 如果cmd为GETVAL&SETALL，则该参数为数组地址array。
    - 如果cmd为IPC_INFO，则该参数为strcut seminfo结构体变量__buf

---

semun

```
union semun {
    int              val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO (Linux-specific) */
};
```

该函数的操作对象为信号量，而非信号量集合。这是一个原子操作。
为 `semaphore operate` 的缩写

函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);

int semtimedop(int semid, struct sembuf *sops, unsigned nsops,
               struct timespec *timeout);
```

参数

semid

要操作的信号量集合标识符

sops

此结构体为：

```
struct sembuf {
    unsigned short  sem_num;    /* 数组中的信号量下标 */
    short           sem_op;         /* 信号量操作 */
    short           sem_flg;    /* 操作标识 */
};
```

成员解读：

- sembuf为信号量的编号
- sem_op是要进行的操作（PV操作）：
    - 如果为正整数，表示增加信号量的值（若为3，则加上3）
    - 如果为负整数，表示减小信号量的值
    - 如果为0，表示对信号量当前值进行是否为0的测试
- sem_flg为操作标识：
    - IPC_NOWAIT：如果不能对信号量集合进行操作，则立即返回
    - SEM_UNDO：当进程退出后，该进程对sem进行的操作将撤销

nsops

PV

信号量是最早出现的用来解决进程同步与互斥问题的机制。
P原语操作的动作是：

1. sem减1；
2. 若sem减1后仍大于或等于零，则进程继续执行；
3. 若sem减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转进程调度。

V原语操作的动作是：

1. sem加1；
2. 若相加结果大于零，则进程继续执行；
3. 若相加结果小于或等于零，则从该信号的等待队列中唤醒一等待进程，然后再返回原进程继续执行或转进程调度。

共享内存用于实现进程间大量的数据传输。
共享内存空间是在内存中单独开辟的一段内存空间。这段空间有自己特有的数据结构，包括访问权限、大小和最近访问时间等。

重要的数据结构

shmid_ds

```
/*位于/usr/include/linux/shm.h*/
struct shmid_ds {
    struct ipc_perm        shm_perm;    /* 操作权限 */
    int            shm_segsz;    /* 段大小(bytes) */
    __kernel_time_t        shm_atime;    /* 最近attach时间 */
    __kernel_time_t        shm_dtime;    /* 最近detach时间 */
    __kernel_time_t        shm_ctime;    /* 最近change时间 */
    __kernel_ipc_pid_t    shm_cpid;    /* 创建者pid */
    __kernel_ipc_pid_t    shm_lpid;    /* 最近操作者pid */
    unsigned short        shm_nattch;    /* no. of current attaches */
    unsigned short         shm_unused;    /* compatibility */
    void             *shm_unused2;    /* ditto - used by DIPC */
    void            *shm_unused3;    /* unused */
};
```

配合信号量使用

一般，不允许几个进程同时对共享内存进行写操作。因此有必要使用二元信号量来同步两个进程以实现对共享内存的写操作。

与管道的对比

使用管道（pipe或FIFO）从一个文件传输信息到另一个文件需要复制4次：

- 服务端将信息从输入文件复制到server临时缓存区
- 从server临时缓冲区复制到管道
- 客户端从管道复制到client临时缓冲区
- 再从client临时缓存区复制到输出文件

使用共享内存只需要复制两次：

- 从服务端将信息从输入文件复制到共享内存
- 客户端从共享内存复制到输出文件

> pipe管道中，输入、输出文件对应的是标准输入、输出。

创建共享内存，通过key返回id。

函数原型

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

参数

key

不消多说

size

欲创建的共享内存段的大小

shmflg

共享内存段的创建标识：

- 公共的IPC选项（在**/usr/include/linux/ipc.h**中定义）
  - `IPC_CREAT //如果不存在就创建`
  - `IPC_EXCL //如果存在则返回失败`
  - `IPC_NOWAIT //如不等待直接返回`
- 共享内存自己的选项(在**/usr/include/linux/shm.h**中定义)
  - `SHM_R //可读`
  - `SHM_W //可写`

共享内存的控制

函数原型

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

参数

shmid

由shmget函数生成，不同的key值对应不同的id值。

cmd

操作字段，包括：

- 公共的IPC选项（ipc.h中）：
  - `IPC_RMID //删除`
  - `IPC_SET //设置ipc_perm参数`
  - `IPC_STAT //获取ipc_perm参数`
  - `IPC_INFO //如ipcs`
- 共享内存自己的选项（shm.h中）【需要root权限】
  - `SHM_LOCK //锁定共享内存段`
  - `SHM_UNLOCK //解锁共享内存段`

shmat是shared memory attach的缩写。而attach本意是贴的意思。
如果进程要使用一段共享内存，那么一定要将该共享内存与当前进程建立联系。即经该共享内存挂接（或称映射）到当前进程。
shmdt则是shmat的反操作，用于将共享内存和当前进程分离。在共享内存使用完毕后都要调用该函数。

函数原型

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

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

参数

shmid

共享内存空间的标识符，即ID。

shmaddr

指定共享内存的映射地址。
shmat中：如果为0（NULL），则由系统选择映射的地址，推荐设置为0。如果非0，并且没有指定**SHM_RND**,则该值即为映射共享内存的地址。

shmflg

指定共享内存的访问权限和映射条件：

- SHM_RDONLY //只读
- SHM_RND //取整，取向下一个SHMLBA边界
- SHM_REMAP //take-over region on attach
- SHM_EXEC //执行权限

如果设置为0的话，则是读写权限。

返回值

- shmat
    - 成功，则返回共享内存的地址
    - 失败，则返回-1，并设置errno
- shmdt
    - 成功，则返回0

- 失败，则返回-1，并设置errno

SHMLBA

低边界地址倍数，它总是2的乘方。

网络编程

Unix/Linux网络编程常用的头文件有:

- arpa/inet.h
- netinet/in.h
- sys/socket.h
- netdb.h
    - hostent(结构体)表示主机
    - servent（结构体）表示服务数据库的登记项信息

套接字地址结构

地址结构	说明
sockaddr_un	UNIX通信域套接字地址
sockaddr_in	IPv4套接字地址
sockaddr_in6	IPv6套接字地址

sockddr_un

```
struct sockaddr_un
{
    sa_familly_t sun_family;
    char sun_path[];
}
```

sockaddr_in

```
/* 在头文件<netinet/in.h>中定义 */
struct in_addr
{
    in_addr_t s_addr;
};
struct sockaddr_in
{
    uint8_t         sin_len; /* POSIX不要求这个字段，它是OSI协议中新增的 */
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8]; /* 未使用 */
};
```

套接字地址结构的每个成员都是以**sin**_开头的。表示的就是socket internet。

- sin_family地址族字段：IPv4为**AF_INET**
- sin_zero这个字段一般置为0。

数据类型

数据类型	头文件	说明
sa_family_t	<sys/socket.h>	
socklen_t	<sys/socket.h>	uint32_t
in_addr_t	<netinet/in.h>	IPV4地址uin32_t
in_port_t	<netinet/int.h>	端口uint16_t

由于历史原因，地址类型(in_addr)定义成了矢量（即结构体），实际上因为其只包含一个字段，完全可以用标量来表示。

> 节选自《UNP》的解释
>
> > 早期版本（4.2BSD）把in_addr定义为多种结构的联合，允许访问一个32为IPv4地址的所有4个字节，或者2个16位值。在IP地址划分为A、B、C类的时期，便于获取地址中的适当字节（比如单独获取网络号或主机号）。然后随着子网划分技术的来临和五分类编址（CIDR）的出现，各种地址类正在消失。那个联合已经不再需要了。
> > 不难理解，IPv4的地址是32位，端口是16位（端口号取值范围0~65535）

sockaddr_in6

```
struct in6_addr
{
    uint8_t s6_addr[16]; /* IPv6地址是128位（8×16） */
};
#define SIN6_LEN  /* 用于编译时测试 */
struct sockaddr_in6
{
    uint8_t          sin6_len;
    sa_family_t      sin6_family;
    in_port_t        sin6_port;

    uint32_t         sin6_flowinfo;
    struct in6_addr  sin6_addr;

    uint32_t         sin6_scope_id;
};
```

sin6_family为地址族字段：IPv6为AF_INET6

函数	描述	
[[socket	socket]]	创建一个套接字描述符
[[socketpair	socketpair]]	创建一个套接字偶对
[[shutdown	shutdown]]	断开套接字连接
[[close	close]]	销毁套接字

套接字选项

函数	描述
[[getsockopt	sockopt]]
[[setsockopt	sockopt]]

适用于流式套接字和数据报套接字

流套接字

通信双方称为对等套接字，请求连接的客户端套接字称为主动套接字，等待连接的服务端套接字为被动套接字。

> 一个套接字开始时是主动的，在调用了listen()之后才变成被动的。只有主动套接字才可以用于connect()，只有被动套接字才可以用accept()。

函数	描述		
[[bind	bind]]	允许服务进程给予套接字一个地址并建立连接的一方	
[[connect	connect]]	客户进程调用。完成三次握手	
[[accept	accept]]	客户进程完成connect()后，服务进程用该函数完成连接	
[[listen	listen]]	创建一个保存连接请求的侦听队列	
[[getsockname	getsockname]]	连接建立后，获取本地套接字地址	
[[getpeername	getpeername]]	连接建立后，获取与本地套接字连接的对等套接字地址	
[[send	send]]	向已连接的套接字发送数据	
[[recv	recv	]]	从已连接的套接字接收数据

数据报套接字

函数	描述	
[[recvfrom	recvfrom]]	接收数据报
[[sendto	sendto]]	发送数据报

recvfrom和sendto也可以用于流式套接字，但是很少这样用

字节序

- 大端：起始地址存储高序字节
- 小端：起始地址存储低序字节

网络协议都使用网络字节序（大端）。主机上的字节序称为主机字节序（有的系统采用大端，有的系统采用小端）

字节序转换函数函数原型

```
/* 有些系统中#include <netinet/in.h> */
#include <arpa/inet.h>

uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);

uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

函数名中

- h:host(主机序)
- n:net(网络序)
- s: short类型。16位
- l:long类型。32位。有的系统（如*Digital Alpha*）尽管*long*是*64*位，但*htonl*和*ntohl*返回的仍然是*32*位值

上面四个函数，进行主机序和网络序之间16和32位的转换。即IP地址和端口号的转换。

因特网程序使用**inet_aton**、**inet_addr**（已废弃）和**inet_ntoa**函数实现IP地址和点分十进制串之间的转换。

函数原型

```
#include <arpa/inet.h>

/*将一个点分十进制串转换位网络字节顺序的IP地址，字符串有效返回1，否则为0*/
int inet_aton(const char *cp,struct in_addr *inp);

/*若字符串有效则返回32位二进制网络序IPv4地址，否则为INADDR_NONE*/
in_addr_t inet_addr(const char *cp);
/*将一个网络字节顺序的IP地址转换位它所对应的点分十进制串*/
char *inet_ntoa(struct in_addr in);
```

> 缩写释义
> arpa貌似是ARPANEAT（阿帕网）的缩写。
> n表示network（网络）或numeric（数值）
> a表示application（应用）

inet_addr存在一些问题，所以已被废弃。新的代码都改用inet_aton函数来代替。

参数

注意，函数中的参数都是地址（**in_addr**）而不是套接字地址结构！
inet_aton和inet_nota两个函数的参数中都有32位二进制网络序IPv4地址。不同的是：

- inet_aton的参数的是指向in_addr结构的指针。（因为这是要返回的）
- 而inet_ntoa的参数是in_addr结构本身。（因为这个是要传入的）

随着IPv6的出现，产生了两个新的函数：inet_pton和inet_ntop。它们对于IPv4和IPv6都适用。
名称中的：

- n表示numeric（数值）
- p表示presentation（表达）

完成从数值类型到表达格式之间的转换。所谓表达格式就是IPv4中的点分十进制；IPv6中的冒号分十六进制。

函数原型

```
#include <arpa/inet.h>

int inet_pton(int af, const char *strptr, void *addrptr);
const char *inet_ntop(int af, const void *addrptr,
                      char *strptr, socklen_t len);
```

参数

`void *ddrptr`，即地址结构（in_addr或in6_addr）
和前面讨论的三个函数相比，这两个函数的参数中的地址均是指针（inet_aton参数中地址的是指针，但inet_ntoa参数中的地址是结构本身）

`char *strptr`，就是要传入或返回的表达格式。它不可以是一个空指针。

len实际是size_t类型，它的值可以使用以下两个宏：

```
#define INET_ADDRSTRLEN 16    /* IPv4 */
#define INET6_ADDRSTRLEN 46   /* IPv6 */
```

UNIX系统内部用一个主机网络地址数据库来记住主机名和IP地址直接的映射，这一数据库由文件/etc/hosts/或DNS提供。

关键头文件

netdb.h

关键结构体

hostent

用于报错一台主机的完整地址信息

```
1. struct hostent //host entry的缩写
2. {
3.     char  *h_name;        /* 主机的正式名字 */
4.     char **h_aliases;     /* 主机的可选别名 */
5.     int    h_addrtype;    /* 主机地址类型 */
6.     int    h_length;      /* 每一个地址的长度，以字节为单位 */
7.     char **h_addr_list;   /* 指向主机网络地址数组的指针 */
8. }
```

部分字段解释

字段名	描述
h_aliases	它是一个以空结尾的字符串向量
h_addrtype	它的值是AF_INET或AF_INET6，也可以是其他类型
h_length	对于IPv4，其值为4，对于IPv6，其值为16
h_addr_list	主机可能与多个网络连接，因而每一个网络有不同的地址，地址数组以空指针表示结束

主要函数

函数	描述	
[[gethostbyname	gethostbyname]]	通过域名或数-点或冒号形式的IP地址来返回主机信息
[[gethostbyaddr	gethostbyaddr]]	通过地址结构（如in_addr）返回主机信息
[[sethostent	sethostent]]	打开主机地址数据库
[[gethostent	gethostent]]	逐一扫描主机地址数据库登记项
[[endhostent	endhostent]]	关闭主机地址数据库

UNIX系统有一个记录标准服务的数据库，这个数据库由头文件**/etc/services**或域名服务器提供。

关键头文件

netdb.h

关键结构体

```
struct servent     /* server entry的缩写 */
{
    char *s_name;  /* 服务程序的正式名字 */
    char *s_alises;/* 服务程序的别名，为一字符串数组，空指针标志该数组结束 */
    int s_port;    /* 端口号，按网络字节序给出 */
    char *s_proto; /* 与该服务一起使用的协议名 */
}
```

主要函数

函数	描述	
[[getservbyname	getservbyname]]	通过服务名和协议名返回该服务的servent结构
[[getservbyport	getservbyport]]	通过端口号和协议名返回该服务的servent结构
[[setservent	setservent]]	打开服务数据库以准备 开始扫描
[[getservent	getservent]]	返回服务数据库的下一项，如果不再有下一项，则返回空指针
[[endservent	endservent]]	关闭服务数据库

带外数据即简称**OOB**（out-of-band）

带外数据是流式套接字独有的。当出现紧急情况时，无法立即通知接收进程。带外数据正用于解决这一问题。带外数据在正常的数据流之外发送，其效果相当于越过套接字上所有等待数据。当它到达接收进程时，接收进程会收到一个信号，从而进程可以立即处理这个数据。

带外数据的发送

比较简单，只需用MSG_OOB标志调用[[send()|send]]即可

带外数据的接收

有两种方式：

- 使用信号
- 使用[[select()|select]]

发送程序发送的每一个带外数据对接收程序都生成了一个**SIGURG**信号。接收者收到带外数据的时机是不确定的。

带外数据标志

```
    带外数据如果没有设置**SO_OOBINLINE**选项，即没有**嵌入**到普通数据之中，那么在收到带外数据通知后直接调用recv()就能读带外数据。
1.  但如果设置了这个选项，使带外数据嵌入到了普通数据之中我们要怎么读带外数据呢？
```

流套接字在接收带外数据时会自动放置一标志于正常数据流中。这个标志指出带外数据发送时原来所在的位置。带外数据标志用来区分哪些数据位于带外数据之前，哪些数据位于带外数据之后。
需要用[[sockatmark()|sockatmart]]来读带外数据

线程是进程中的一个独立控制流。一个进程包含一个或多个线程。
线程基本上不拥有系统资源（只有少量运行中必不可少的资源），但它可与同属于一个进程的其他线程共享该进程的全部资源，包括地址空间（数据段和堆段）、通用信号处理机制、数据与I/O。而线程有自己的栈（自动变量）。

> 进程是系统资源分配的最小单位，线程是CPU调度的最小单位

pthread

Linux中的线程相关函数都是以pthread开头的，它的含义是：POSIX thread，即POSIX标准的线程。
关于POSIX线程的更多概念，大家可以去 `man pthreads` 查看。

API对比

功能	线程	进程
创建	pthread_create	fork,vfork
退出	pthread_exit	exit
等待	pthread_join	wait,waitpid
取消/终止	pthread_cacel	abort
读取ID	pthread_self	getpid
通信机制	信号，信号量，互斥锁，读写锁，条件变量	管道（有名/无名），信号，信号量，消息队列，共享内存

调度

线程和进程支持同样的调度策略：

- SCHED_OTHER 分时调度策略，（默认的）
- SCHED_FIFO 实时调度策略，先到先服务
- SCHED_RR 实时调度策略，时间片轮转

编译

在编译（**gcc**）包含pthread函数的程序时，要显式地链接线程库，即**-lpthread**。

基本编程

- pthread_create
- pthread_exit
- pthread_join

在调用这个函数的进程中创建一个新的线程

函数原型

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

参数

thread

属于结果参数。函数结束时，返回线程ID并存储到thread中。
`pthread_t` 被定义成 `unsigned long int` 类型。

attr

设置线程的属性，主要是栈相关的属性。

start_routine

start_routine是一个回调函数（函数指针实现）。指明了线程要执行的函数。

arg

回调函数start_routine()执行时的参数（实参）。

返回值

成返回0，失败返回非0值。

结束一个线程

函数原型

```
#include <pthread.h>

void pthread_exit(void *retval);
```

参数

retval用来保存线程退出状态

返回值

为空。因为该函数永远成功。

为了回收资源，主线程会等待子线程结束。该函数就是用来等待线程终止的。类似与进程中的wait函数。
此函数将阻塞调用当前线程的进程，直到此线程退出。

函数原型

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

参数

thread

被等待线程的ID

retval

如果此值非NULL，pthread_join复制线程的退出状态

返回值

成功返回0，失败返回非0

互斥锁

条件变量

POSIX信号量

类型

pthread_mutex_t

互斥锁基本操作

函数	描述	
[[pthread_mutex_init	pthread_mutex_init]]	初始化互斥锁
[[pthread_mutex_lock	pthread_mutex_lock]]	阻塞申请互斥锁
[[pthread_mutex_unlock	pthread_mutex_unlock]]	释放互斥锁
[[pthread_mutex_trylock	pthread_mutex_trylock]]	非阻塞申请互斥锁
[[pthread_mutex_destroy	pthread_mutex_destroy]]	销毁互斥锁

互斥锁属性的基本操作

函数	描述	
[[pthread_mutexattr_init	pthread_mutexattr_init]]	初始化属性对象
[[pthread_mutexattr_destroy	pthread_mutexattr_destroy]]	销毁属性对象
[[pthread_mutexattr_settype	pthread_mutexattr_settype]]	设置属性对象的属性
[[pthread_mutexattr_gettype	pthread_mutexattr_gettype]]	获得属性对象的属性

头文件

- semaphore.h
- sys/stat.h
- fcntl.h

常用函数

函数	说明	
[[sem_open	sem_open]]	打开一个有名信号量
[[sem_close	sem_close]]	关闭一个信号量
[[sem_unlink	sem_unlink]]	删除一个信号量
[[sem_post	sem_post]]	【V操作】释放操作：信号量的值加1
[[sem_wait	sem_wait]]	【P操作】分配操作：信号量的值减1
[[sem_getvalue	sem_getvalue]]	获取信号量的值
[[sem_init	sem_init]]	初始化一个无名信号量
[[sem_destroy	sem_destroy]]	破坏一个无名信号量

IO复用

I/O多路复用常用于I/O操作可能会被阻塞的情况

- select()出自BSD系统，poll()出自 System V系统。两者原理等价，实现方式不同。POSIX.1定义了两者。
- epoll()出自Linux2.6，其他Unix系统无此函数，BSD系统（包括OSX系统）有类似的函数kequeue()

- select
- poll
- epoll模型

函数原型

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
                   fd_set *exceptfds, struct timeval *timeout);
```

参数

nfds

指定被监听的文件描述符的总数。它通常被设置为select()监听的最大文件描述符加1（文件描述符从0计数）

readfds、writefds、exceptfds

分别指向可读、可写和异常事件对应的文件描述符集合。这三个参数都是fd_set类型。
这三个参数是值-结果参数，在select()返回的的时候，将把就绪的fd更新到对应的集合中。

timeout

设置select()超时时间。

- NULL: select()将无限等待
- 非NULL: select()等待相应秒数，若无就绪则超时返回
    - 特例非NULL，但其值为0秒: select()检查完fd集合后立即返回

这是个值-结果参数，select()成功返回时，内核将修改它的值为剩余的时间。
但当该调用失败的时候，它返回的值是不确定的。该结构体（timeval）提供微秒级的分辨率。

返回值

- 成功时返回就绪（可读、可写和异常）文件描述符的总数。
- 如果在超时时间内没有任何文件描述符就绪，select()将返回0。
- 失败时返回-1，并设置errno。

如果在select()等待期间，程序接收到信号，则select()立即返回-1，并设置errno为**EINTR**。

---

fd_set

一个结构体名（被typedef命名，所以使用时不加struct）。实际上里面只包含一个整型（long int）数组。
该数组的每一位标记一个文件描述符。可以通过以下函数宏来访问fd_set结构体中的位。

返回值	函数宏	描述

void|**FD_ZERO**(fd_set *set)*|清除*set*所有的位
void|***FD_CLR***(*int fd, fd_set* set)|清除set的为fd
void|**FD_SET**(int fd, fd_set *set)*|设置*set*的位*fd*
int |***FD_ISSET***(*int fd, fd_set* set)|测试set的位fd是否被设置

在使用fd_set类型的变量的时候，一定要用 `FD_ZERO` 初始化

poll()和select()原理相同，但实现方式不同。select()用的更为频繁，但 据说 poll()效率更高，不过自 Linux 2.5.44 后被epoll取代。

函数原型

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

参数

fds

pollfd结构的数组。
检测每个结构中的fd对应的events事件是否就绪。在poll返回的时候将就绪情况写入到revents中。

nfds

nfds_t是 unsigned long 类型，据《UNP》所言，该类型过大了。

timeout

- -1: 永远等待（一说：负数）
- 0: 不等待，立即返回
- > 0: 等待指定数目的毫秒数

返回值

- 0: 表示超时返回
- -1: 表示出错
- > 0: 表示就绪的的文件描述符的数量

> poll()与select()返回正数时，其含义有差别：
>
> - select(): 如果同一个文件描述符在多个fd集（select有三个fd集合）中返回，则返回值也统计多次
> - poll(): 即使同一个fd出现在多个pollfd元素的 revents 成员中，也只被统计一次。

pollfd

非 typedef 的结构体类型

成员	类型	说明
fd	int	
events	short	
revents	short	

events和revents都是 `bit masks` 。events是输入，revents是输出（返回），events由用户设置，revents在函数返回时被设置。其取值可以是如下宏的按位或的结果：

输入事件bit位	events的输入？	revents的结果？	说明
POLLIN	√	√	有数据可读
POLLRDNORM	√	√	等价于**POLLIN**
POLLPRI	√	√	高优先级数据可读
POLLRDHUP	√	√	对等套接字关闭
输出事件**bit**位	**events**的输入？	**revents**的结果？	说明
POLLOUT	√	√	普通数据可写
POLLWRNORM	√	√	等价于**POLLOUT**
POLLWRBAND	√	√	优先级数据可写
错误事件**bit**位	**events**的输入？	**revents**的结果？	说明
POLLERR		√	发生错误
POLLHUP		√	发生挂起
POLLNVAL		√	（invalid）fd没有被打开

> POLLRDHUP 从 `Linux 2.6.17` 开始被支持
> 差异性：《UNP》中提到的POLLRDBAND其实并不被Linux支持；此外《UNP》中POLLIN与POLLRDNORM，POLLOUT与POLLWRNORM也并非等价。

主要函数

函数	描述	
[[epoll_create	epoll_create]]	创建一个epoll的文件描述符
[[epoll_ctl	epoll_ctl]]	epoll的事件注册函数
[[epoll_wait	epoll_wait]]	收集在epoll监控的事件中已经发送的事件

结构体

epoll_event

```
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

HINT

epoll_create生成的epfd，是内核中epoll结构的唯一标识。**epoll**结构不直接面向应用程序员。它维持着每个epoll处理要监视的fd及其感兴趣事件。要修改它只能通过epoll_ctl。

函数原型

```
#include <sys/epoll.h>

int epoll_create(int size);
```

参数

size参数自从Linux 2.6以后被忽略了。

> epoll_create1

```
#include <sys/epoll.h>
int epoll_create1(int flags);
```

函数原型

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

参数

epfd

是[[epoll_create|epoll_create]]的返回值。

op

表示动作，它由三个宏来表示

- EPOLL_CTL_ADD：注册新的fd到epfd中；
- EPOLL_CTL_MOD：修改已经注册的fd的监听事件；
- EPOLL_CTL_DEL：从epfd中删除一个fd；

fd

要监听的文件描述符

event

可以是以下几个宏 逻辑或 的组合

宏	描述

EPOLL**IN** |表示对应的文件描述符可以读（包括对端SOCKET正常关闭）
EPOLL**OUT**|表示对应的文件描述符可以写
EPOLL**PRI**|表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）
EPOLL**ERR**|表示对应的文件描述符发生错误
EPOLL**HUP**|表示对应的文件描述符被挂断
EPOLL**ET**| 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的
EPOLL**ONESHOT**|只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

检测event(struct epoll_event类型)是否包含某一标识

```
if(event & EPOLLHUP){
...
```

```
3. }
4. if(event & (EPOLLPRI|EPOLLERR|EPOLLHUP)){
5. ...
6. }
```

函数原型

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

参数

events

出参，记录准备好的fd。该参数为向量（数组），由调用方分配空间。

maxevents

最大监听**fd**。epoll_wait会检测从0到maxevents的所有fd是否就绪，如果就绪就保存到events中。

timeout

- 0，不阻塞立即返回
- -1，阻塞直到监听的一个fd上有一个感兴趣事件发生
- > 0，阻塞指定时间。直到监听的fd上有感兴趣事件发生，或者捕捉到信号

返回值

- > 0，返回准备好的fd数量
- 0，超时
- -1，出错

通常遍历events[0]~events[返回值-1]的元素。这些都是就绪的。并且保存了就绪的事件。

> events下标与fd并不相同

Linux 异步IO

> 编译时链接实时库，使用选项 `-lrt`

aiocb struct

aiocb是”asynchronous I/O control block”的缩写。

```
struct aiocb {
    /* The order of these fields is implementation-dependent */

    int             aio_fildes;     /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void  *aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
    int             aio_reqprio;    /* Request priority */
    struct sigevent aio_sigevent;   /* Notification method */
    int             aio_lio_opcode; /* Operation to be performed;
                                       lio_listio() only */

    /* Various implementation-internal fields not shown */
};

/* aio_lio_opcode: */
enum { LIO_READ, LIO_WRITE, LIO_NOP };
```