## 文件IO

### 说明

linux文件IO，没有缓冲区

### 打开或者新建文件

#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);

参数：

pathname:需要打开或者创建的文件

flags：打开方式

必选项（三个选项互斥）：

O\_RDONLY:只读

O\_WRONLY: 只写

O\_RDWR: 读写

可选项:

O\_CREAT: 文件不存在，新建文件，文件存在相当于没写这个选项

O\_APPEND: 在文件末尾追加

mode：

1、若是新建文件需要通过这个参数指定权限，

2、填八进制 如 0777，

3、创建出来的权限需要减去 umask

4、可以通过umask命令查看默认权限 ，open函数生成文件的权限: mode & ~umask

返回值：

成功返回：文件描述符

失败返回：-1，设置errno

注意：

C语言默认打开的三个文件描述符

STDIN\_FILENO：输入  
STDOUT\_FILENO：输出  
STDERR\_FILENO：错误输出

### 关闭文件文件描述符

#include <unistd.h>

int close(int fd);

参数:

fd: open需要关闭的文件描述符

返回值:

成功: 返回0

失败: 返回-1

### 读文件

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

参数：

fd：需要读取的文件描述符

buf：保存读取数据的地方

count：需要读取的字节数

返回值：

成功返回：读取到的字节数字

返回0表示：读到了文件末尾,或者写端关闭

失败返回：-1，设置errno

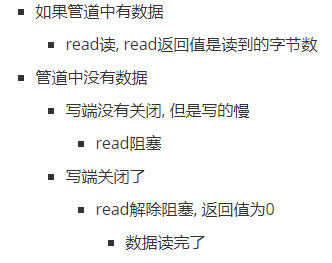
errno = 11，非阻塞情况下读取到缓冲区中没有数据的文件描述符

errno =104，发送数据太快会出现

注意：

1、函数阻塞与否由文件描述符决定

2、读取pipe和mkfifo创建的文件描述符，函数会阻塞



### 写文件

创建一个

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

参数：

fd：需要写的文件描述符

buf：需要写的数据

count：需要写的字节数，如果写入数据不足，末尾自动补 \0

返回值：

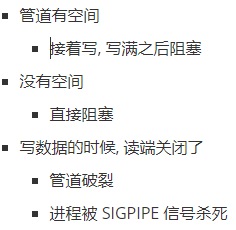
成功返回：写入的字节数

失败返回：-1，设置errno

注意：

1、函数阻塞与否由文件描述符决定

2、对pipe和mkfifo创建的文件描述符进行写操作时，函数会阻塞



### 文件指针操作

#include <sys/types.h>

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

参数：

fd：文件描述符

offset：偏移量

whence：指针的位置

SEEK\_SET--文件头

SEEK\_CUR--当前位置

SEEK\_END--末尾

返回值：返回文件指针的位置

示例：

// 1. 移动文件指针到文件头

lseek(fd, 0, SEEK\_SET);

// 2. 获取当前文件指针的位置

lseek(fd, 0, SEEK\_CUR);

// 3. 获取文件长度

lseek(fd, 0, SEEK\_END);

// 4. 拓展文件长度, 拓展100字节

lseek(fd, 100, SEEK\_END);

// 要进行一次写操作

write(fd, " ", 1);

### 操作文件描述符

#include <unistd.h>

int dup(int oldfd);`

功能：复制一个文件描述符

参数：

oldfd：需要被复制的文件描述符

返回值：

成功返回：一个新的文件描述符

失败返回：-1，设置errno

注意：

1、此时有两个文件描述符记得都要关闭

#include <unistd.h>

int dup2(int oldfd, int newfd);

功能：重定向一个文件描述符

参数：

oldfd：不需要不改变指向文件描述符

newfd：需要改变指向的文件描述符

返回值：

成功返回： newfd

失败返回：-1，设置errno

注意：

1、两个文件描述符都需要关闭

2、此时两个文件描述符指向同一个文件

#include <unistd.h>

#include <fcntl.h>

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

功能：该函数可以对文件描述符作一些设置，如修改读写操作，设置文件描述符带阻塞等等

代码示例：

[src\linux系统编程\文件操作\ddq.c](src/linux系统编程/文件操作/ddq.c)

## linux常用系统函数

### 目录操作函数

#### 文件重命名

#include <stdio.h>

int rename(const char \*oldpath, const char \*newpath);

参数：

oldpath：旧的文件名

newpath：新的文件名

返回值：

成功返回：0

失败返回：-1，设置errno

#### 修改进程的工作目录

#include <unistd.h>

int chdir(const char \*path);

参数：

path：切换到那个工作目录

返回值：

成功返回：0

失败返回：-1，设置errno

#### 获取进程的当前工作目录

#include <unistd.h>

char \*getcwd(char \*buf, size\_t size);

参数：

buf：传入参数保存进程当前工作目录

size：buf大小

返回值：

成功返回：参数buf的首地址

失败返回：NULL，设置errno

#### 创建目录

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char \*pathname, mode\_t mode);

参数：

pathname：目录名称

mode：目录的权限，8进制，mod & ~umask

返回值：

成功返回：0

失败返回：-1

#### 删除目录

#include <unistd.h>

int rmdir(const char \*pathname);

参数：

pathname：目录名

返回值：

成功返回：0

失败返回：-1

注意：

1、相当于 rm -f 只能删除空目录

#### 打开一个目录

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(const char \*name);

参数:

- name: 要打开的目录名

返回值:

成功返回：DIR

失败返回：NULL

注意：

1、//关闭目录

int closedir(DIR \*dirp);

#### 读取目录中的数据

#include <dirent.h>

struct dirent\* readdir(DIR \*dirp);

参数：

dirp：打开的目录

返回值：

成功返回：dirent结构体,里面保存了一个文件的属性

失败返回：NULL，不设置errno

读取到目录尾返回：NULL，设置errno

注意:

1、目前想不到这个函数可以能会出现失败的情况，所以把NULL当做读到文件尾即可，传入错误参数只会段错误，不会函数调用失败

struct dirent

{

ino\_t d\_ino; // 此目录进入点的inode

ff\_t d\_off; // 目录文件开头至此目录进入点的位移

signed short int d\_reclen; // d\_name 的长度, 不包含NULL 字符

unsigned char d\_type; // d\_name 所指的文件类型

har d\_name[256]; // 文件名

};

d\_type

DT\_BLK - 块设备

DT\_CHR - 字符设备

DT\_DIR - 目录

DT\_LNK - 软连接

DT\_FIFO - 管道

DT\_REG - 普通文件

DT\_SOCK - 套接字

DT\_UNKNOWN - 未知

struct dirent \*readdir(DIR \*dirp);

- 返回一个结构体, 这个对应一个文件

代码案例：

1、小案例

[src\linux系统编程\目录操作\main.c](src/linux系统编程/目录操作/main.c)：

2、演示

[src\linux系统编程\目录操作\demo.c](src/linux系统编程/目录操作/demo.c)

### 文件属性函数

#### 判断文件权限、存在

#include <unistd.h>

函数功能：判断是否有操作某个文件的权限，或者判断文件是否存在

int access(const char \*pathname, int mode);

参数：

- pathname：文件名

- mode：

R\_OK：判断是否有读权限

W\_OK：判断是否有写权限

X\_OK：判断是否有执行权限

F\_OK：判断文件是否存在

返回值：

有操作权限、文件存在返回：0

没有有操作权限、文件存在返回：-1，设置erron

#### 修改文件权限

#include <sys/stat.h>

int chmod(const char \*filename, int mode);

参数：

- filename：文件名

- mode：八进制数设置文件权限 如：0777

返回值：

成功返回：0

失败返回：-1，设置errno

#### 修改文件所有者

#include <unistd.h>

int chown(const char \*path, uid\_t owner, gid\_t group);

参数：

path：要修改的文件名

owner：用户ID

group：组ID

返回值：

成功返回：0

失败返回：-1，设置errno

#### 修改文件大小

#include <unistd.h>

#include <sys/types.h>

int truncate(const char \*path, off\_t length);

参数：

path：要操作的文件

length：最终的文件大小

返回值：

成功返回：0

失败返回：-1，谁知errno

### 获取文件属性stat()

功能：获取一个文件的详细属性

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int stat(const char \*pathname, struct stat \*statbuf);

参数：

pathname：文件名

buf: 传出参数, 保存文件的属性信息

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

注意：

stat函数: 获取的是连接文件指向的文件的属性

lstat函数: 获取的是连接文件本身的属性

struct stat {

dev\_t st\_dev; //文件的设备编号

ino\_t st\_ino; //节点

mode\_t st\_mode; //文件的类型和存取的权限

nlink\_t st\_nlink; //连到该文件的硬连接数目，刚建立的文件值为1

uid\_t st\_uid; //用户ID

gid\_t st\_gid; //组ID

dev\_t st\_rdev; //(设备类型)若此文件为设备文件，则为其设备编号

off\_t st\_size; //文件字节数(文件大小)

blksize\_t st\_blksize; //块大小(文件系统的I/O 缓冲区大小)

blkcnt\_t st\_blocks; //块数

time\_t st\_atime; //最后一次访问时间

time\_t st\_mtime; //最后一次修改时间

time\_t st\_ctime; //最后一次改变时间(指属性)

};

关于变量 st\_mode:

- st\_mode -- 16位整数

0-2 bit -- 其他人权限

- S\_IROTH 00004 读权限

- S\_IWOTH 00002 写权限

- S\_IXOTH 00001 执行权限

- S\_IRWXO 00007 掩码, 过滤 st\_mode中除其他人权限以外的信息

3-5 bit -- 所属组权限

- S\_IRGRP 00040 读权限

- S\_IWGRP 00020 写权限

- S\_IXGRP 00010 执行权限

- S\_IRWXG 00070 掩码, 过滤 st\_mode中除所属组权限以外的信息

6-8 bit -- 文件所有者权限

- S\_IRUSR 00400 读权限

- S\_IWUSR 00200 写权限

- S\_IXUSR 00100 执行权限

- S\_IRWXU 00700 掩码, 过滤 st\_mode中除文件所有者权限以外的信息

12-15 bit -- 文件类型

- S\_IFSOCK 0140000 套接字

- S\_IFLNK 0120000 符号链接（软链接）

- S\_IFREG 0100000 普通文件

- S\_IFBLK 0060000 块设备

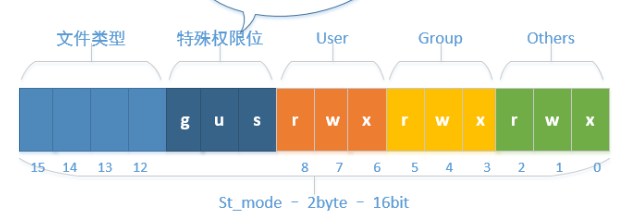
- S\_IFDIR 0040000 目录

- S\_IFCHR 0020000 字符设备

- S\_IFIFO 0010000 管道

- S\_IFMT 0170000 掩码,过滤 st\_mode中除文件类型以外的信息

(st\_mode & S\_IFMT) == S\_IFREG



### exec

功能：使用一个进程拉起另外一个进程，被拉起进程的用户区 替换 拉起进程的用户区，内核区不变，所以不会创建出新的进程

#include<unistd.h>

int execlp(const char \*file,const char \*arg)

参数：

file:命令、程序名称，会在path里面找，一般用来执行命令

arg:参数可用多个，用NULL结束

#include<unistd.h>

int execl (const char \*file,const char \*arg)

参数：

file:命令、程序名称，不会在path里面找，所以一般用于执行程序

arg:参数可用多个，用NULL结束

返回值：

成功：无返回

失败，返回-1，设置errno

## 进程

### 进程说明：

1、每一个进程都拥有独立的虚拟地址空间，进程是linux系统资源的最小分配单位。

2、fork后，只有少量内核中的少量数据不会被拷贝，目前已知PID。

3、孤儿进程，父进程在子进程之前推出了程序，子进程会认PID为1的进程做父进程，由PID为1的进程负责释放

4、僵尸进程，子进程结束后父进程不负责回收，同时父进程还在运行中，子进程就会变成僵尸进程，僵尸进程需要杀死父进程才会死亡

### 创建进程：

#include<unistd.h>

pid\_t fork() (void)

成功返回：

父进程返回：父进程pid

子进程返回：0

失败返回：-1，设置errno

### 获取当前进程的PID和获取父进程的PID

#include <sys/types.h>

#include <unistd.h>

获取当前进程的PID:

pid\_t getpid(void);

获取父进程的PID:

pid\_t getppid(void);

### 进程回收函数

功能：回收一个结束的子进程

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*wstatus)

参数：wstatus: 传一个int数据进去就行

WIFEXITED(wstatus)值为非0：进程正常结束

WEXITSTATUS(wstatus):

1.子进程的返回值，如果没有返回值，结果可能是随机数

2.值是无符号1字节数据，如果返回值为负数，接收可能不正确

a. -1 == 255，-2==254 ，以此类推

WIFSIGNALED(wstatus): 值为为非0：进程异常终止

WTERMSIG(wstatus):

终止进程的信号编号(整型数据)

返回值:

成功返回：子进程PID

失败返回：-1

注意：

1、函数带阻塞

2、只回收一个进程

3、通过4个宏函数，获取子进程退出状态

可以设置非阻塞和回收指定进程的进程回收函数

pid\_t waitpid(pid\_t pid, int \*wstatus, int options)

参数：

pid:

pid= -1: 所有子进程都可以回收

pid> 0: 需要回收的子进程PID

pid= 0: 没写，不知道

wstatus：用法和上一个函数相同

options：传WNOHANG不阻塞，传0阻塞

返回值:

>0: 回收掉的子进程PID

-1: 无子进程

=0: 参数options为非阻塞，且子进程未结束

### 进程通信pipe (管道)

功能：匿名管道返回两个文件描述符，父子进程之间通信

#include <unistd.h>

int pipe(int pipefd[2]);

参数：

pipefd：传出参数

pipefd[0]:管道的读端

pipefd[1]:管道的写端

返回值：

成功返回：0

失败返回：-1，设置errno

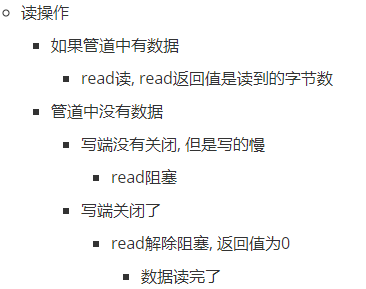
注意：

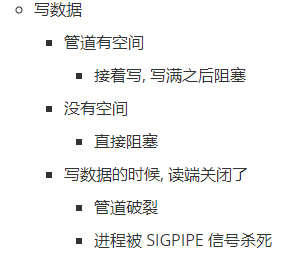
1、管道是单向的，建议读端关闭写端，写端关闭读端

2、管道的默认大小4k 查看ulimit -a

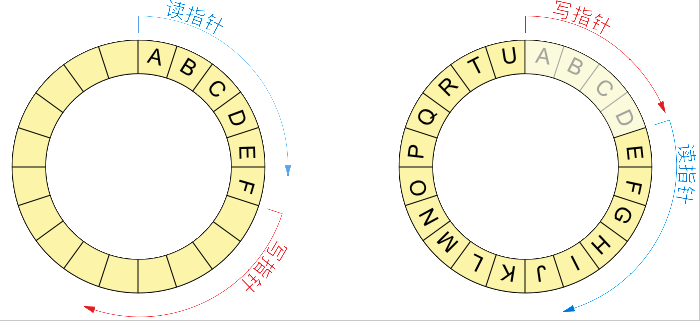
3、管道可以的会动态扩展一部分空间

4、管道文件描述符带阻塞





5、管道的结构，环形队列



### 进程通信mkfifo(管道)

功能：有名管道，创建一个管道文件，进行进程通信

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char \*pathname, mode\_t mode);

参数：

pathname：创建的文件名

mode：文件的权限

返回值：

成功返回：0

失败返回：-1，设置ernno

注意：

1、除了管道的大小其他的应该都后上面一样

### 进程通信mmap（内存映射区）

功能：把一个硬盘中的普通文件映射到 程序的内存中 相当于两个进程拥有同一块内存

#include <sys/mman.h>

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

参数：

addr：传NULL即可，映射到具体的内存位置

length：映射区的大小，一般用文件的大小即可

prot：文件的读写权限

PROT\_READ读

PROT\_WRIFE 写

PROT\_READ | PROT\_WRIFE 读写

flags：映射区的内容是否同步到文件

MAP\_SHARED：同步到文件，进程通信必须设置同步

MAP\_PRIVATE：不同步到文件

fd：被映射文件的文件描述符

offset：映射文件的开始偏移量，必须是4k的整数倍，传0表示从文件其实位置开始映射

返回值：

成功返回：映射区的首地址

失败返回：MAP\_FAILED #define MAP\_FAILED ((void \*) -1)

内存映射区释放：

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

参数：

addr：映射区的首地址

length：映射区的大小

返回值：

成功返回：0

失败返回：-1，设置errno

代码示例：

1、[src\linux系统编程\进程通信\mmap.c](src/linux系统编程/进程通信/mmap.c)

### 守护进程

守护进程就是后台服务进程

#### 守护进程的特点

1、守护进程一般用d作为结尾

2、守护进程是脱离终端的

3、周期性的执行某些任务

#### 示例：

//父进程fork子进程, 然后退出

pid\_t pid = fork();

if(pid>0 || pid<0)

{

exit(0);

}

//子进程调用setsid函数创建会话

setsid();

//修改当前的工作目录

chdir("/home/itcast/log");

//重设文件掩码

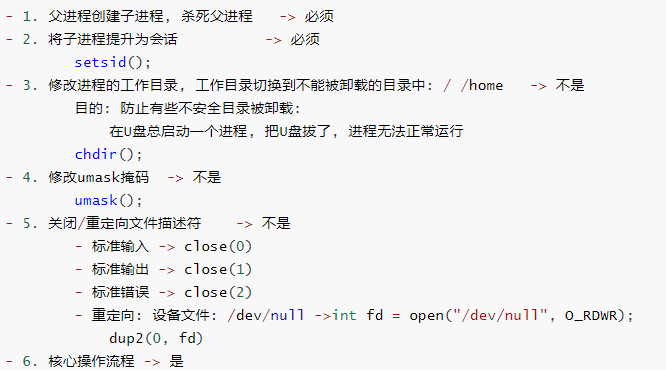
umask(0000);

//关闭文件描述符

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);



## 信号

### 信号说明:

1、一个进程执行过程中如果收到某个信号，会放下当前执行的代码，去处理这个信号，信号处理完成后在接着执行当前代码

2、信号可以被阻塞、捕获 - SIGKILL, SIGSTOP 这两个信号 不允许被捕捉、阻塞

#### 信号的状态

1、产生

2、未决

- 信号没有被处理

- 阻塞是未决的一种状态

3、递达

- 信号已经被递达，开始处理

- 递达不保证这个信号被正确处理结束，因为在处理一个信号的过程中可能收到另外一个信号，导致当前信号，处理中断。

#### 信号列表

可以通过命令：kill -l 查询当前系统有哪些信号

#### 信号的产生和默认处理动作

通过命令：man 7 signal 查看每个信号详细信息

### 信号处理函数：

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

示例：

void函数名(int 处理的信号编号)

void lr\_sig(int signo)

注意：

1、信号处理函数执行中，默认会阻塞当前信号

2、信号处理函数执行中，如果收到其他未被阻塞信号，会执行会去执行其他信号

### 注信号处理函数1：（捕获信号）

功能：捕获某个信号，交给自定义的信号处理函数处理

#include <signal.h>

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

函数参数：

signum：信号的编号，需要捕捉的信号

handler：自定义的函数名, 可以是SIG\_IGN--表示忽略, SIG\_DFL--表示行默认动作

函数返回值: 无

注意：

1、在不同的平台下，这个函数可能不同，所以建议使用sigaction函数

2、 SIGKILL, SIGSTOP 这两个信号 不允许被捕捉, 阻塞和忽略

3、处理当前信号过程中会自动阻塞当前信号

### 注册信号处理函数2（捕获信号）

#include <signal.h>

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

参数：

signum:需要捕捉的信号

act:传入参数,处理方式

oldact:传出参数,旧的处理方式

返回值:

成功返回：0

失败返回：-1，设置errno

所需要结构体：

struct sigaction

{

- void (\*sa\_handler)(int); // 信号处理函数

- void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); //如果需要通过信号传输某些数据，就需要使用这个信号处理函数，可以不给值

- sigset\_t sa\_mask; //当前信号处理函数执行期间需要阻塞的信号，不清楚绑定的信号还是还是信号处理函数，没事过，应该是绑定了信号，一般来说初始化一下这个集合即可

- int sa\_flags; //传0即可，表示使用默认标识

- void (\*sa\_restorer)(void); //已经废弃不在使用，不用给值

};

注意：

1、 SIGKILL, SIGSTOP 这两个信号 不允许被捕捉, 阻塞和忽略

2、处理当前信号过程中会自动阻塞当前信号

### 给指定进程发送信号：

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

函数参数：

pid：

pid>0:发送信号给指定程序

pid=0:发送信号给统一进程组的所有进程

pid<-1:不会

pid=-1:发送信号给，进程有权限发送的，系统中所有进程

sig:信号编号，建议使用宏，不同操作系统可以编号不同

返回值:

成功返回：0

失败放回：-1，设置errno

### 指定时间后发送SIGALRM信号

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

参数:

seconds:几秒后发送信号

返回值：

若先前没有调用过alarm函数, 则返回0;

若先前有alarm函数调用, 返回上一次剩余的秒数

注意：

1、使用自然定时法

2、一个进程只有一个时钟, 若多次调用, 最后一次是有效的

3、alarm(0);就是取消时钟的意思

### 指定时间后发送信号

对比上个函数，这个可以重复发送信号

int setitimer(int which,

const struct itimerval \*new\_value,

struct itimerval \*old\_value);

参数：

which:

ITIMER\_REAL：使用自然定时法，真实时间，建议使用，发送的是SIGALRM信号

ITIMER\_VIRTUAL：只计算进程占用cpu的时间，发送的是SIGVTALRM信号

ITIMER\_PROF：计算占用cpu及执行系统调用的时间，发送的是SIGPROF信号

new\_value: 传入参数, 设置时钟触发的时间

old\_value: 传出参数, 先前的设置, 通常传NULL.

返回值：

成功返回：0

失败返回：-1，设置errno

注意：

过程中所需结构体：

struct itimerval

{

struct timeval it\_interval; //以后每次触发的间隔

struct timeval it\_value; //第一次触发的时间

};

struct timeval

{

time\_t tv\_sec; //秒

suseconds\_t tv\_usec; //微秒

};

### 未决信号集和阻塞信号集

说明：

1、信号被发送到程序内核的未决信号集，此时未决信号集会与阻塞信号集进行比较，如果信号未被阻塞，会被送达给程序

2、因为每个信号只占用一个位的空间，只能表示 存在 或者 不存在 两种可能，所以程序收到多个相同未决信号，只有一个会被保存在未决信号集中

3、子进程会拷贝父进程里面的阻塞信号集，未决信号集没测过

4、子进程会拷贝父进程里面的信号捕捉和处理

#include <signal.h>

所需结构体：

typedef struct

{

nsigned long int \_\_val[\_SIGSET\_NWORDS];

}\_\_sigset\_t;

信号集创建

sigset\_t set;

信号集置0,表示不阻塞(初始化操作)

int sigemptyset(sigset\_t \*set);

信号集置1

int sigfillset(sigset\_t \*set);

将某个信号加入信号集

int sigaddset(sigset\_t \*set, int signum);

将某个信号移出信号集

int sigdelset(sigset\_t \*set, int signum);

读取当前进程的未决信号集

int sigpending(sigset\_t \*set);

以上函数返回值：

成功返回：0

失败返回：-1，设置errno

判断某个信号是否在信号集中

int sigismember(const sigset\_t \*set, int signum);

返回值：

在返回：1

不在返回：0

错误返回：-1，设置errno

设置阻塞信号集（生效）

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

参数：

how:处理方式

SIG\_BLOCK: 将用户设置的阻塞信息添加到内核中, 内核中原来的数据不变 mask |= set

SIG\_UNBLOCK: 清除用户在内核中设置的数据, mask &= ~set

SIG\_SETMASK: 用户传入信号集，覆盖内核阻塞信号集

set: 传入参数，是一个自定义信号集合。由参数how来指示如何修改当前信号屏蔽字。

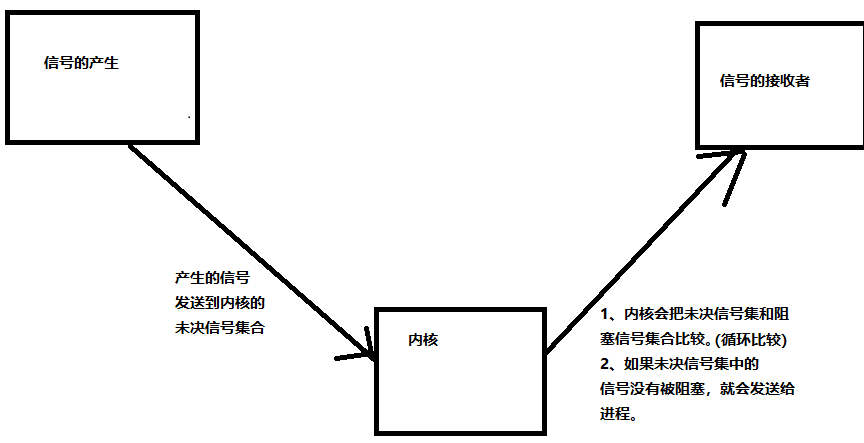
oldset：传出参数，保存旧的 内核阻塞信号集

返回值：

成功返回:0

失败返回:-1,设置erro

图片说明：



### 子进程结束信号SIGCHLD

产生SIGCHLD信号的条件

1 子进程收到SIGSTOP信号

2 子进程收到SIGCONT信号

3 子进程终止

产生SIGCHLD信号的条件

1. 子进程死了, 自杀, 他杀

2. 子进程被暂停了

3. 子进程由暂停状态重新恢复运行

以上三种情况子进程都会给父进程发送信号: SIGCHLD, 父进程默认处理动作是忽略这个信号

通过SIGCHLD信号回收子进程小案例

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include <sys/types.h>

#include <signal.h>

#include <sys/wait.h>

int ii = 0;

//处理信号的函数

void my\_waitpid(int signo)

{

int wpid = -1;

//回收子进程

while (( wpid = waitpid(-1,NULL, WNOHANG)) !=0 && wpid != -1 )

printf("回收了[%d]个进程，进程pid=[%d]\n", ++ii, wpid);

}

int main()

{

int i = 0;

pid\_t pid = 0;

//捕捉子进程死亡信号

signal(SIGCHLD, my\_waitpid);

dzz:

//主进程创建1000个子进程

pid = fork();

if (pid > 0)

{

if (++i < 1000)

goto dzz;

while (ii<1000){}

printf("程序结束所有子进程已回收\n");

}

//子进程做一下打印

if (pid == 0)

printf("第[%d]个子进程产生，PID=[%d]\n",i+1,getpid());

return 0;

}

## 线程

### 线程说明

1、线程没有独立的虚拟地址空间，多个子线程和主线程共享同一个地址空间

2、线程是系统调度执行的最小单位，会抢CPU资源

3、线程不是linux系统的库函数，是第三方库函数

4、线程拥有的一些独立资源

- 线程id

- 处理器现场和栈指针（内核栈）

- 独立的栈空间（用户空间栈）

- errno 变量

- 阻塞信号集

5、多个线程同一时间操作同一个数据时，需要加锁，防止数据异常

6、多个线程间通信 “条件变量”、“信号量”

7、子线程执行中发生异常会导致整个进程死亡

### 创建线程

#include <pthread.h>

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

const pthread\_attr\_t \*attr,

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

void \*arg);

参数：

thread：传出参数保存线程ID

attr:线程属性，设置线程是否分离状态，传NULL默认就是不分离

//定义线程属性变量

pthread\_attr\_t attr;

//初始化线程属性变量

pthread\_attr\_init(&attr);

//设置线程属性为分离状态

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

//释放线程属性

ret = pthread\_attr\_destroy(&attr);

void \*(\*start\_routine) (void \*)：线程执行的函数

arg：需要传入线程的参数，给第三个参数传参

返回值:

成功返回：0

失败返回：错误号, 可以使用strerror函数打印错误原因

### 使当前线程结束

#include <pthread.h>

功能：线程退出时使用这个函数，不要使用return，主线程使用return 会使其他线程也结束

void pthread\_exit(void \*retval);

参数：

retval：退出状态（需要使用全局或者申请的堆空间），不关心退出状态传NULL

注意：

1、是在需要退出的线程处调用

2、主线程退出必须使用这个函数否则其他线程也会死亡

### 获取当前的线程ID

#include <pthread.h>

pthread\_t pthread\_self(void);

返回值：返回线程的ID

### 回收子线程资源

#include <pthread.h>

功能：子线程运行结束后，主线程负责回收子进程的资源，设置线程分离后，不需要主线程回收子资源

int pthread\_join(pthread\_t thread, void \*\*retval);

参数：

thread：需要回收的子线程ID

retval：退出状态 pthread\_exit函数的参数，在这接收，不关心退出状态可以传NULL

返回值：

成功返回：0

失败: 返回错误号, 可以使用strerror函数打印错误原因

注意：

1、分离后的线程无法阻塞等待退出

2、子线程未结束函数会阻塞等待子线程退出

### 设置线程分离

#include <pthread.h>

功能：设置线程分离，子线程退出后不需要主线程回收资源

int pthread\_detach(pthread\_t thread);

参数：

thread：线程ID

返回值:

成功: 返回0

失败: 返回错误号, 可以使用strerror函数打印错误原因

### 取消线程

#include <pthread.h>

功能：在主线程中，取消一个执行中的子线程

int pthread\_cancel(pthread\_t thread);

参数：

thread：需要取消的线程ID

返回值:

成功: 返回0

失败: 返回错误号, 可以使用strerror函数打印错误原因

注意：

1、线程取消不是立即执行的，需要等待线程到达某个取消点(检查点)，可粗略认为一个系统调用(进入内核)即为一个取消点，调用pthread\_testcancel函数设置一个取消点。

### 比较两个线程是否相同

#include <pthread.h>

int pthread\_equal(pthread\_t t1, pthread\_t t2)；

参数：

t1：需要比较的线程ID

t2：需要比较的线程ID

返回值：

两个线程相同返回：非0 值

两个线程不同返回：0

注意：

1、pthread\_t线程ID，在不同的平台下可能封装不同，有些平台下pthread\_t线程ID，是一个结构体，所以比较线程需要使用这个函数

### 线程同步

多个线程同时访问一个数据，会出现数据异常所以需要线程同步

#### 互斥锁

说明：多个线程同时对一个数据操作时，会导致异常，所以需要访问共享数据时，建议加锁，某个线程在访问时，其他线程阻塞等待

1 定义一把锁

pthread\_mutex\_t mutex;

2 初始化锁（第二个参数可以传NULL）

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

const pthread\_mutexattr\_t \*restrict attr);

参数：

mutex：传入一把锁

attr：设置锁的属性，不设置可以传NULL

返回值：

成功返回：0

失败返回：错误号

3 加锁

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

参数：

mutex：传入一把锁

返回值：

成功返回：0

失败返回：错误号

4 解锁

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

参数：

mutex：传入一把锁

返回值：

成功返回：0

失败返回：错误号

5 释放互斥锁

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

参数：

mutex：传入一把锁

返回值：

成功返回：0

失败返回：错误号

注意：

1、线程对某个数据加锁前，最好释放当前的锁，防止死锁情况出现

2、线程对某个数据加锁前，请留意加锁代码阻塞情况，防止死锁情况出现

代码示例：

[src\linux系统编程\多线程\线程同步\互斥锁\main.c](src/linux系统编程/多线程/线程同步/互斥锁/main.c)

#### 读写锁

说明：读写锁时一种特别的锁，读时可以并行，写时独占

读写锁的使用步骤:

1 定义一把读写锁:

pthread\_rwlock\_t wrlock;

2 初始化读写锁:

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock,

const pthread\_rwlockattr\_t \*restrict attr);

参数:

- rwlock: 读写锁

- attr: 读写锁属性, 使用默认属性, 设置为: NULL

返回值：

成功返回：0

失败返回：错误号

3添加读锁:

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock); //

函数功能：加读锁

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

添加写锁:

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

函数功能：加写锁

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

4 解锁

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

：

4 释放读写锁:

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

注意：

1. 死锁：访问其他锁时建议解开自己的锁
2. 读写锁：读并行，写独占，当读写同时等待锁的时候写的优先级高

3. 适合于对数据结构读的次数远大于写的情况

#### 条件变量

说明：生产者和消费者模型，使用条件变量，生产者生产出商品后通知消费者消费，没有商品时消费者阻塞等待，防止供不应求

条件变量相关函数和使用步骤:

1 定义条件变量:

pthread\_cond\_t cond;

2 初始化条件变量:

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

const pthread\_condattr\_t \*restrict attr);

参数:

- cond: 条件变量的地址

- attr: 使用默认属性, 这个值设置为NULL

返回值：

成功返回：0

失败返回：错误号

3阻塞线程等待唤醒

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

pthread\_mutex\_t \*restrict mutex);

函数功能：调用该函数阻塞，同时自动解锁，等待signal 和 broadcast函数唤醒后，自动加锁 并解除阻塞 运行

参数:

- cond: 条件变量

- mutex: 传入锁

返回值：

成功返回：0

失败返回：错误号

注意：

1、唤醒后，执行前在使用while 循环判断一下，防止同时唤醒了多个线程

4 唤醒被wait 函数阻塞的一个或多个线程

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

参数：

- cond: 条件变量

返回值：

成功返回：0

失败返回：错误号

注意：

1、可以使用 pthread\_cond\_broadcast 唤醒所有的阻塞在 pthread\_cond\_wait / pthread\_cond\_timedwait 函数上的线程。

4 释放条件变量

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

参数：

cond：条件变量

返回值：

成功返回：0

失败返回：错误号

代码示例：

[src\linux系统编程\多线程\线程同步\条件变量\main.c](src/linux系统编程/多线程/线程同步/条件变量/main.c)

#### 信号量

说明：生产者和消费者模型，使用信号量，生产出商品后 ++ ，消费后 -- ，防止供不应求

头文件

#include <semaphore.h>

信号量的使用

1、定义信号量

sem\_t sem;

2、初始化信号量

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

参数:

sem: 信号量

pshared: 传 0 表示用于多线程, 非 0 值表示用于多进程

value: 信号量的初始值，小于等于 0 ，-- 函数阻塞

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

3、++ 函数

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

功能: 调用该函数一次, 相当于sem++

参数:

sem: 信号量

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

4、-- 函数

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

功能: 调用该函数一次, 相当于sem--, 当sem为 0 的时候, 引起阻塞

参数:

sem: 信号量

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

5、释放信号量

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

功能: 释放信号量资源

参数:

sem: 信号量

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

代码示例：

[src\linux系统编程\多线程\线程同步\信号量\main.c](src/linux系统编程/多线程/线程同步/信号量/main.c)