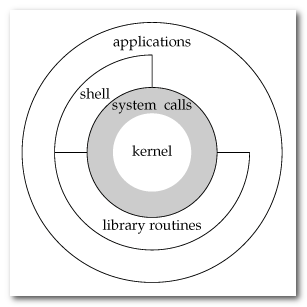
# 1. 系统调用

## 1.1 什么是系统调用

系统调用，顾名思义，说的是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务，比如用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件，可以通过时钟相关的系统调用获得系统时间或设置定时器等。

从逻辑上来说，系统调用可被看成是一个内核与用户空间程序交互的接口——它好比一个中间人，把用户进程的请求传达给内核，待内核把请求处理完毕后再将处理结果送回给用户空间。



系统服务之所以需要通过系统调用来提供给用户空间的根本原因是为了对系统进行“保护”，因为我们知道 Linux 的运行空间分为内核空间与用户空间，它们各自运行在不同的级别中，逻辑上相互隔离。

所以用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间函数。

比如我们熟悉的“hello world”程序（执行时）就是标准的用户空间进程，它使用的打印函数 printf 就属于用户空间函数，打印的字符“hello word”字符串也属于用户空间数据。

但是很多情况下，用户进程需要获得系统服务（调用系统程序），这时就必须利用系统提供给用户的“特殊接口”——系统调用了，它的特殊性主要在于规定了用户进程进入内核的具体位置。

换句话说，用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不准许肆意跳入内核。有了这样的陷入内核的统一访问路径限制才能保证内核安全无误。我们可以形象地描述这种机制：作为一个游客，你可以买票要求进入野生动物园，但你必须老老实实地坐在观光车上，按照规定的路线观光游览。当然，不准下车，因为那样太危险，不是让你丢掉小命，就是让你吓坏了野生动物。

## 1.2 系统调用的实现

系统调用是属于操作系统内核的一部分的，必须以某种方式提供给进程让它们去调用。CPU 可以在不同的特权级别下运行，而相应的操作系统也有不同的运行级别，用户态和内核态。运行在内核态的进程可以毫无限制的访问各种资源，而在用户态下的用户进程的各种操作都有着限制，比如不能随意的访问内存、不能开闭中断以及切换运行的特权级别。显然，属于内核的系统调用一定是运行在内核态下，但是如何切换到内核态呢？

答案是软件中断。软件中断和我们常说的中断（硬件中断）不同之处在于，它是通过软件指令触发而并非外设引发的中断，也就是说，又是编程人员开发出的一种异常（该异常为正常的异常）。操作系统一般是通过软件中断从用户态切换到内核态。

## 1.3 系统调用和库函数的区别

### 1.3.1 系统调用和库函数的区别

Linux 下对文件操作有两种方式：系统调用（system call）和库函数调用（Library functions）。

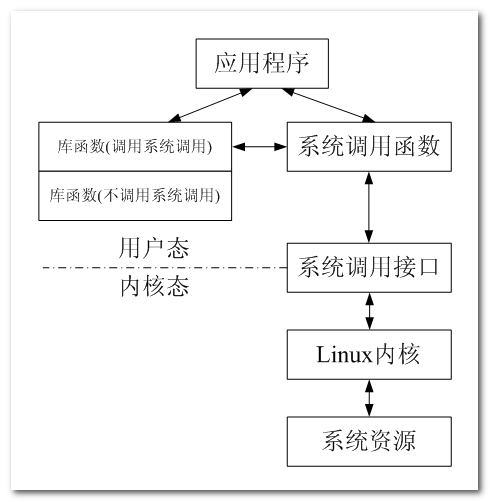
库函数由两类函数组成：

1）不需要调用系统调用

不需要切换到内核空间即可完成函数全部功能，并且将结果反馈给应用程序，如strcpy、bzero 等字符串操作函数。

2）需要调用系统调用

需要切换到内核空间，这类函数通过封装系统调用去实现相应功能，如 printf、fread等。



系统调用是需要时间的，程序中频繁的使用系统调用会降低程序的运行效率。当运行内核代码时，CPU工作在内核态，在系统调用发生前需要保存用户态的栈和内存环境，然后转入内核态工作。系统调用结束后，又要切换回用户态。这种环境的切换会消耗掉许多时间。

### 1.3.2 C库IO函数工作流程



库函数访问文件的时候根据需要，设置不同类型的缓冲区，从而减少了直接调用 IO 系统调用的次数，提高了访问效率。

这个过程类似于快递员给某个区域（内核空间）送快递一样，快递员有两种方式送：

1）来一件快递就马上送到目的地，来一件送一件，这样导致来回走比较频繁（系统调用）

2）等快递攒着差不多后（缓冲区），才一次性送到目的地（库函数调用）

# 2. 文件和目录

## 2.1 文件IO

### 2.1.1 概述

2.1.1.1 虚拟地址空间

每个进程都会分配虚拟地址空间，在32位机器上，该地址空间为4G：



在进程里平时所说的指针变量，保存的就是虚拟地址。当应用程序使用虚拟地址访问内存时，处理器（CPU）会将其转化成物理地址。

这样做的好处在于：

* 进程隔离，更好的保护系统安全运行
* 屏蔽物理差异带来的麻烦，方便操作系统和编译器安排进程地址

2.1.1.2 文件描述符

1) 文件描述

在 Linux 的世界里，一切设备皆文件。我们可以系统调用中 I/O 的函数（I：input，输入；O：output，输出），对文件进行相应的操作（ open()、close()、write() 、read() 等）。

打开现存文件或新建文件时，系统（内核）会返回一个文件描述符，文件描述符用来指定已打开的文件。这个文件描述符相当于这个已打开文件的标号，文件描述符是非负整数，是文件的标识，操作这个文件描述符相当于操作这个描述符所指定的文件。

程序运行起来后（每个进程）都有一张文件描述符的表，标准输入、标准输出、标准错误输出设备文件被打开，对应的文件描述符 0、1、2 记录在表中。程序运行起来后这三个文件描述符是默认打开的。

#define STDIN\_FILENO  0 //标准输入的文件描述符

#define STDOUT\_FILENO 1 //标准输出的文件描述符

#define STDERR\_FILENO 2 //标准错误的文件描述符

在程序运行起来后打开其他文件时，系统会返回文件描述符表中最小可用的文件描述符，并将此文件描述符记录在表中。



2) 最大打开的文件个数

Linux 中一个进程最多只能打开 NR\_OPEN\_DEFAULT （即1024）个文件，故当文件不再使用时应及时调用 close() 函数关闭文件。

* 查看当前系统允许打开最大文件个数：

cat /proc/sys/fs/file-max

* 当前默认设置最大打开文件个数1024

ulimit -a

* 修改默认设置最大打开文件个数为4096

ulimit -n 4096

### 2.2.2 错误处理函数

errno 是记录系统的最后一次错误代码。代码是一个int型的值，在errno.h中定义。查看错误代码errno是调试程序的一个重要方法。

当Linux C api函数发生异常时，一般会将errno全局变量赋一个整数值，不同的值表示不同的含义，可以通过查看该值推测出错的原因。

#include <stdio.h> //fopen

#include <errno.h> //errno

#include <string.h> //strerror(errno)

int main()

{

FILE \*fp = fopen("xxxx", "r");

if (NULL == fp)

{

printf("%d\n", errno); //打印错误码

printf("%s\n", strerror(errno)); //把errno的数字转换成相应的文字

perror("fopen err"); //打印错误原因的字符串

}

return 0;

}

查看错误号：

/usr/include/asm-generic/errno-base.h

/usr/include/asm-generic/errno.h

### 2.2.3 常用IO函数

2.2.3.1 open

#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);

**功能：**

打开文件，如果文件不存在则可以选择创建。

**参数：**

pathname：文件的路径及文件名

flags：打开文件的行为标志，必选项 O\_RDONLY, O\_WRONLY, O\_RDWR

mode：这个参数，只有在文件不存在时有效，指新建文件时指定文件的权限

**返回值：**

成功：成功返回打开的文件描述符

失败：-1

* flags详细说明

必选项：

|  |  |
| --- | --- |
| **取值** | **含义** |
| O\_RDONLY | 以只读的方式打开 |
| O\_WRONLY | 以只写的方式打开 |
| O\_RDWR | 以可读、可写的方式打开 |

可选项，和必选项按位或起来：

|  |  |
| --- | --- |
| **取值** | **含义** |
| O\_CREAT | 文件不存在则创建文件，使用此选项时需使用mode说明文件的权限 |
| O\_EXCL | 如果同时指定了O\_CREAT，且文件已经存在，则出错 |
| O\_TRUNC | 如果文件存在，则清空文件内容 |
| O\_APPEND | 写文件时，数据添加到文件末尾 |
| O\_NONBLOCK | 对于设备文件, 以O\_NONBLOCK方式打开可以做非阻塞I/O |

* mode补充说明

1. 文件最终权限：mode & ~umask
2. shell进程的umask掩码可以用umask命令查看

* umask：查看掩码（补码）
* umask mode：设置掩码，mode为八进制数
* umask -S：查看各组用户的默认操作权限

|  |  |  |
| --- | --- | --- |
| **取值** | **八进制** | 含义 |
| S\_IRWXU | 00700 | 文件所有者的读、写、可执行权限 |
| S\_IRUSR | 00400 | 文件所有者的读权限 |
| S\_IWUSR | 00200 | 文件所有者的写权限 |
| S\_IXUSR | 00100 | 文件所有者的可执行权限 |
| S\_IRWXG | 00070 | 文件所有者同组用户的读、写、可执行权限 |
| S\_IRGRP | 00040 | 文件所有者同组用户的读权限 |
| S\_IWGRP | 00020 | 文件所有者同组用户的写权限 |
| S\_IXGRP | 00010 | 文件所有者同组用户的可执行权限 |
| S\_IRWXO | 00007 | 其他组用户的读、写、可执行权限 |
| S\_IROTH | 00004 | 其他组用户的读权限 |
| S\_IWOTH | 00002 | 其他组用户的写权限 |
| S\_IXOTH | 00001 | 其他组用户的可执行权限 |

2.2.3.2 close

#include <unistd.h>

**int close(int fd);**

**功能：**

关闭已打开的文件

**参数：**

fd : 文件描述符，open()的返回值

**返回值：**

成功：0

失败： -1, 并设置errno

需要说明的是，当一个进程终止时，内核对该进程所有尚未关闭的文件描述符调用close关闭，所以即使用户程序不调用close，在终止时内核也会自动关闭它打开的所有文件。

但是对于一个长年累月运行的程序(比如网络服务器)，打开的文件描述符一定要记得关闭,否则随着打开的文件越来越多，会占用大量文件描述符和系统资源。

2.2.3.3 write/read

1) write

#include <unistd.h>

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

**功能：**

把指定数目的数据写到文件（fd）

**参数：**

fd : 文件描述符

buf : 数据首地址

count : 写入数据的长度（字节）

**返回值：**

成功：实际写入数据的字节个数

失败： - 1

2) read

#include <unistd.h>

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

**功能：**

把指定数目的数据读到内存（缓冲区）

**参数：**

fd : 文件描述符

buf : 内存首地址

count : 读取的字节个数

**返回值：**

成功：实际读取到的字节个数

失败： - 1

3) 阻塞、非阻塞

读常规文件是不会阻塞的，不管读多少字节，read一定会在有限的时间内返回。

从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用read读终端设备就会阻塞，如果网络上没有接收到数据包，调用read从网络读就会阻塞，至于会阻塞多长时间也是不确定的，如果一直没有数据到达就一直阻塞在那里。

同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

注意，阻塞与非阻塞是对于文件而言的，而不是指read、write等的属性。

以非阻塞方式打开文件：

#include <unistd.h> //read

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <errno.h> //EAGAIN

int main()

{

// /dev/tty --> 当前终端设备

// 以不阻塞方式(O\_NONBLOCK)打开终端设备

int fd = open("/dev/tty", O\_RDONLY | O\_NONBLOCK);

char buf[10];

int n;

n = read(fd, buf, sizeof(buf));

if (n < 0)

{

// 如果为非阻塞，但是没有数据可读，此时全局变量 errno 被设置为 EAGAIN

if (errno != EAGAIN)

{

perror("read /dev/tty");

return -1;

}

printf("没有数据\n");

}

return 0;

}

2.2.3.4 lseek

所有打开的文件都有一个当前文件偏移量(current file offset)，以下简称为 cfo。cfo 通常是一个非负整数，用于表明文件开始处到文件当前位置的字节数。

读写操作通常开始于 cfo，并且使 cfo 增大，增量为读写的字节数。文件被打开时，cfo 会被初始化为 0，除非使用了 O\_APPEND 。

#include <sys/types.h>

#include <unistd.h>

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

**功能：**

改变文件的偏移量

**参数：**

fd：文件描述符

offset：根据whence来移动的位移数（偏移量），可以是正数，也可以负数，如果正数，则相对于whence往右移动，如果是负数，则相对于whence往左移动。如果向前移动的字节数超过了文件开头则出错返回，如果向后移动的字节数超过了文件末尾，再次写入时将增大文件尺寸。

whence：其取值如下：

SEEK\_SET：从文件开头移动offset个字节

SEEK\_CUR：从当前位置移动offset个字节

SEEK\_END：从文件末尾移动offset个字节

**返回值：**

若lseek成功执行, 则返回新的偏移量

### 2.2.4 文件操作相关函数

2.2.4.1 stat/lstat

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

**int stat(const char \*path, struct stat \*buf);**

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

**功能：**

获取文件状态信息

**stat和lstat的区别：**

当文件是一个符号链接时，lstat返回的是该符号链接本身的信息；

而stat返回的是该链接指向的文件的信息。

**参数：**

path：文件名

buf：保存文件信息的结构体

**返回值：**

成功：0

失败 - 1

struct stat结构体说明：

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

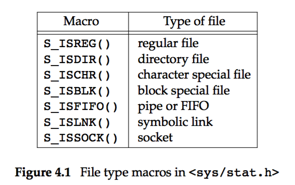
time\_t st\_ctime; //最后一次改变时间(指属性)

};

st\_mode(16位整数)参数说明：



文件类型判断应使用宏函数：



int main()

{

struct stat buf;

stat("a.txt", &buf);

if (S\_ISREG(buf.st\_mode))

{

printf("%s\n", "这是普通文件");

}

return 0;

}

2.2.4.2 access

#include <unistd.h>

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

**功能：**测试指定文件是否具有某种属性

**参数：**

pathname：文件名

mode：文件权限，4种权限

R\_OK： 是否有读权限

W\_OK： 是否有写权限

X\_OK： 是否有执行权限

F\_OK： 测试文件是否存在

**返回值：**

0： 有某种权限，或者文件存在

-1：没有，或文件不存在

2.2.4.3 chmod

#include <sys/stat.h>

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

**功能：**修改文件权限

**参数：**

filename：文件名

mode：权限(8进制数)

**返回值：**

成功：0

失败：-1

2.2.4.4 chown

#include <unistd.h>

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

**功能**：修改文件所有者和所属组

**参数**：

pathname：文件或目录名

owner：文件所有者id，通过查看 / etc / passwd 得到所有者id

group：文件所属组id，通过查看 / etc / group 得到用户组id

**返回值**：

成功：0

失败：-1

2.2.4.5 truncate

#include <unistd.h>

#include <sys/types.h>

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

**功能**：修改文件大小

**参数**：

path：文件文件名字

length：指定的文件大小

a)比原来小, 删掉后边的部分

b)比原来大, 向后拓展

**返回值**：

成功：0

失败：-1

2.2.4.6 link

#include <unistd.h>

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

**功能**：创建一个硬链接

**参数**：

oldpath：源文件名字

newpath：硬链接名字

**返回值**：

成功：0

失败：-1

2.2.4.7 symlink

#include <unistd.h>

**int symlink(const char \*target, const char \*linkpath);**

**功能**：创建一个软链接

**参数**：

target：源文件名字

linkpath：软链接名字

**返回值**：

成功：0

失败：-1

2.2.4.8 readlink

#include <unistd.h>

**ssize\_t readlink(const char \*pathname, char \*buf, size\_t bufsiz);**

**功能：**读软连接对应的文件名，不是读内容(该函数只能读软链接文件)

**参数：**

pathname：软连接名

buf：存放软件对应的文件名

bufsiz ：缓冲区大小(第二个参数存放的最大字节数)

**返回值：**

成功：>0，读到buf中的字符个数

失败：-1

2.2.4.9 unlink

#include <unistd.h>

**int unlink(const char \*pathname);**

**功能：**删除一个文件(软硬链接文件)

**参数：**

pathname：删除的文件名字

**返回值：**

成功：0

失败：-1

2.2.4.10 rename

#include <stdio.h>

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

**功能**：把oldpath的文件名改为newpath

**参数**：

oldpath：旧文件名

newpath：新文件名

**返回值**：

成功：0

失败：-1

### 2.2.5 文件描述符的复制

2.2.5.1 概述

dup() 和 dup2() 是两个非常有用的系统调用，都是用来复制一个文件的描述符，使新的文件描述符也标识旧的文件描述符所标识的文件。

这个过程类似于现实生活中的配钥匙，钥匙相当于文件描述符，锁相当于文件，本来一个钥匙开一把锁，相当于，一个文件描述符对应一个文件，现在，我们去配钥匙，通过旧的钥匙复制了一把新的钥匙，这样的话，旧的钥匙和新的钥匙都能开启这把锁。

对比于 dup(), dup2() 也一样，通过原来的文件描述符复制出一个新的文件描述符，这样的话，原来的文件描述符和新的文件描述符都指向同一个文件，我们操作这两个文件描述符的任何一个，都能操作它所对应的文件。

2.2.5.2 dup

#include <unistd.h>

**int dup(int oldfd);**

**功能：**

通过 oldfd 复制出一个新的文件描述符，新的文件描述符是调用进程文件描述符表中最小可用的文件描述符，最终 oldfd 和新的文件描述符都指向同一个文件。

**参数：**

oldfd : 需要复制的文件描述符 oldfd

**返回值：**

成功：新文件描述符

失败： -1

2.2.5.3 dup2

#include <unistd.h>

**int dup2(int oldfd, int newfd);**

**功能：**

通过 oldfd 复制出一个新的文件描述符 newfd，如果成功，newfd 和函数返回值是同一个返回值，最终 oldfd 和新的文件描述符 newfd 都指向同一个文件。

**参数：**

oldfd : 需要复制的文件描述符

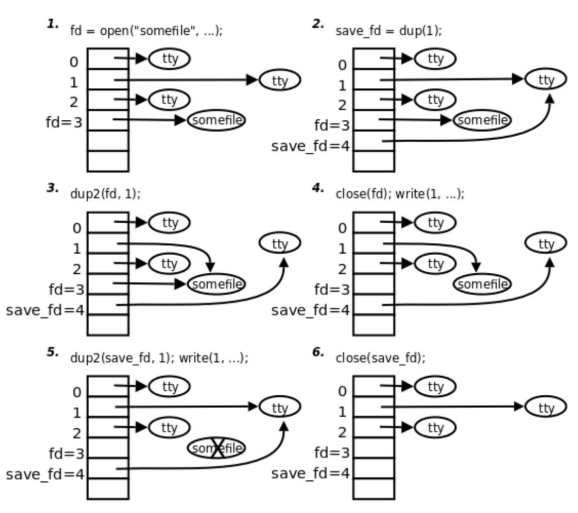
newfd : 新的文件描述符，这个描述符可以人为指定一个合法数字（0 - 1023），如果指定的数子已经被占用（和某个文件有关联），此函数会自动关闭 close() 断开这个数字和某个文件的关联，再来使用这个合法数字。

**返回值：**

成功：返回 newfd

失败：返回 -1

2.2.5.4 示例分析



### 2.2.6 操作文件的特性

#include <unistd.h>

#include <fcntl.h>

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

**功能：**改变已打开的文件性质，fcntl针对描述符提供控制。

**参数**：

fd：操作的文件描述符

cmd：操作方式

arg：针对cmd的值，fcntl能够接受第三个参数int arg。

**返回值**：

成功：返回某个其他值

失败：-1

fcntl函数有5种功能：

1. 复制一个现有的描述符（cmd=F\_DUPFD）
2. 获得／设置文件描述符标记(cmd=F\_GETFD或F\_SETFD)
3. 获得／设置文件状态标记(cmd=F\_GETFL或F\_SETFL)
4. 获得／设置异步I/O所有权(cmd=F\_GETOWN或F\_SETOWN)
5. 获得／设置记录锁(cmd=F\_GETLK, F\_SETLK或F\_SETLKW)

// 等价于dup()

int new\_fd = fcntl(fd, F\_DUPFD, 0);

// 获取文件状态标志

int flag = fcntl(fd, F\_GETFL, 0);

switch (flag & O\_ACCMODE)

{

case O\_RDONLY:

printf("read only\n");

break;

case O\_WRONLY:

printf("write only\n");

break;

case O\_RDWR:

printf("read and write\n");

break;

default:

break;

}

if (flag & O\_APPEND)

{

printf("append\n");

}

flag |= O\_APPEND; // 追加flag

fcntl(fd, F\_SETFL, flag); //设置文件状态标记

## 2.2 目录

### 2.2.1 getcwd

#include <unistd.h>

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

**功能：**获取当前进程的工作目录

**参数：**

buf : 缓冲区，存储当前的工作目录

size : 缓冲区大小

**返回值：**

成功：buf中保存当前进程工作目录位置

失败：NULL

### 2.2.2 chdir

#include <unistd.h>

**int chdir(const char \*path);**

**功能：**修改当前进程(应用程序)的路径

**参数：**

path：切换的路径

**返回值：**

成功：0

失败：-1

### 2.2.3 opendir

#include <sys/types.h>

#include <dirent.h>

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

**功能：**打开一个目录

**参数：**

name：目录名

**返回值：**

成功：返回指向该目录结构体指针

失败：NULL

### 2.2.4 closedir

#include <sys/types.h>

#include <dirent.h>

**int closedir(DIR \*dirp);**

**功能：**关闭目录

**参数：**

dirp：opendir返回的指针

**返回值：**

成功：0

失败：-1

### 2.2.5 readdir

#include <dirent.h>

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

**功能：**读取目录

**参数：**

dirp：opendir的返回值

**返回值：**

成功：目录项结构体指针

失败：NULL

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\_type 所指的文件类型

char d\_name[256]; // 文件名

};

d\_type文件类型说明：

|  |  |
| --- | --- |
| **取值** | **含义** |
| DT\_BLK | 块设备 |
| DT\_CHR | 字符设备 |
| DT\_DIR | 目录 |
| DT\_LNK | 软链接 |
| DT\_FIFO | 管道 |
| DT\_REG | 普通文件 |
| DT\_SOCK | 套接字 |
| DT\_UNKNOWN | 未知 |

# 3. 进程

## 2.1 概述

### 2.1.1 程序和进程

我们平时写的 C 语言代码，通过编译器编译，最终它会成为一个可执行程序，当这个可执行程序运行起来后（没有结束之前），它就成为了一个进程。

程序是存放在存储介质上的一个可执行文件，而进程是程序执行的过程。进程的状态是变化的，其包括进程的创建、调度和消亡。程序是静态的，进程是动态的。



程序就类似于剧本(纸)，进程类似于戏(舞台、演员、灯光、道具...)，同一个剧本可以在多个舞台同时上演。同样，同一个程序也可以加载为不同的进程(彼此之间互不影响)。

在 Linux 系统中，操作系统是通过进程去完成一个一个的任务，进程是管理事务的基本单元。

进程拥有自己独立的处理环境（如：当前需要用到哪些环境变量，程序运行的目录在哪，当前是哪个用户在运行此程序等）和系统资源（如：处理器 CPU 占用率、存储器、I/O设备、数据、程序）。

我们可以这么理解，公司相当于操作系统，部门相当于进程，公司通过部门来管理（系统通过进程管理），对于各个部门，每个部门有各自的资源，如人员、电脑设备、打印机等。

### 2.1.2 单道、多道程序设计

2.1.2.1 单道程序设计

所有进程一个一个排队执行。若A阻塞，B只能等待，即使CPU处于空闲状态。而在人机交互时阻塞的出现时必然的。所有这种模型在系统资源利用上及其不合理，在计算机发展历史上存在不久，大部分便被淘汰了。

2.1.2.2 多道程序设计

在计算机内存中同时存放几道相互独立的程序，它们在管理程序控制之下，相互穿插的运行。多道程序设计必须有硬件基础作为保证。

**时钟中断**即为多道程序设计模型的理论基础。并发时，任意进程在执行期间都不希望放弃cpu。因此系统需要一种强制让进程让出cpu资源的手段。时钟中断有硬件基础作为保障，对进程而言不可抗拒。 操作系统中的中断处理函数，来负责调度程序执行。

在多道程序设计模型中，多个进程轮流使用CPU (分时复用CPU资源)。而当下常见CPU为纳秒级，1秒可以执行大约10亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在运行。

1s = 1000ms

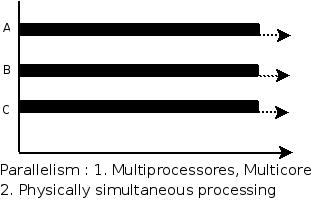
1ms = 1000us

1us = 1000ns

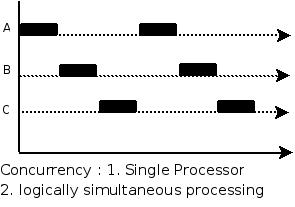
1s = 1000000000

### 2.1.3 并行和并发

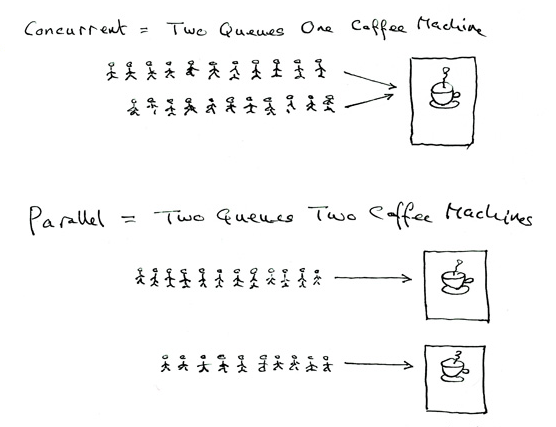
**并行(parallel)：**指在同一时刻，有多条指令在多个处理器上同时执行。



**并发(concurrency)：**指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



* 并行是两个队列**同时**使用两台咖啡机
* 并发是两个队列**交替**使用一台咖啡机



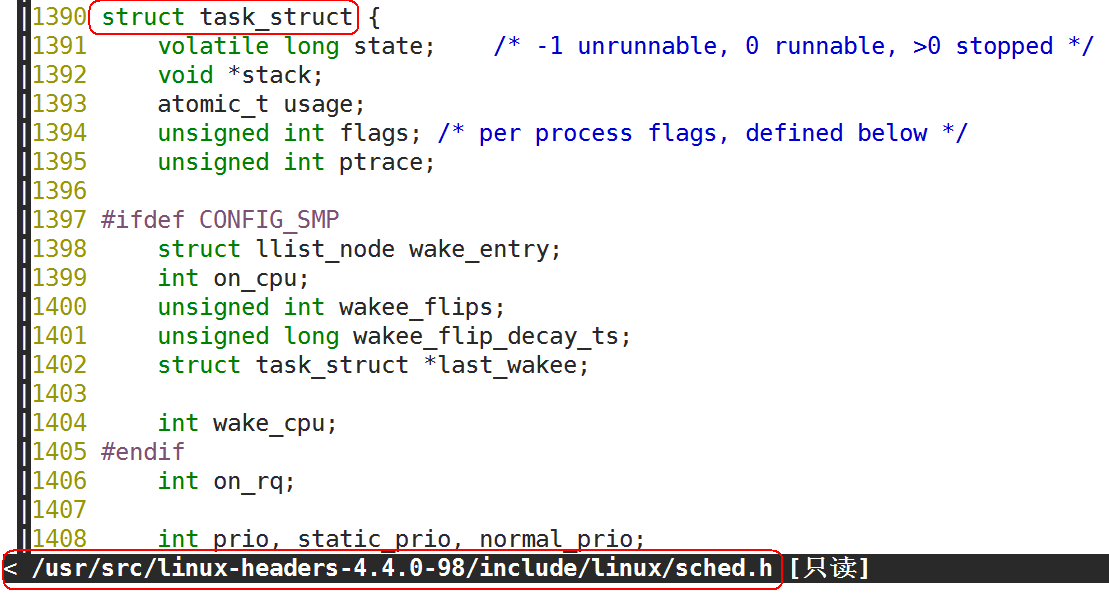
### 2.1.4 MMU

MMU是Memory Management Unit的缩写，中文名是内存管理单元，它是中央处理器（CPU）中用来管理虚拟存储器、物理存储器的控制线路，同时也负责虚拟地址映射为物理地址，以及提供硬件机制的内存访问授权，多用户多进程操作系统。

### 2.1.5 进程控制块PCB

进程运行时，内核为进程每个进程分配一个PCB（进程控制块），维护进程相关的信息，Linux内核的进程控制块是task\_struct结构体。

在 /usr/src/linux-headers-xxx/include/linux/sched.h 文件中可以查看struct task\_struct 结构体定义：



其内部成员有很多，我们掌握以下部分即可：

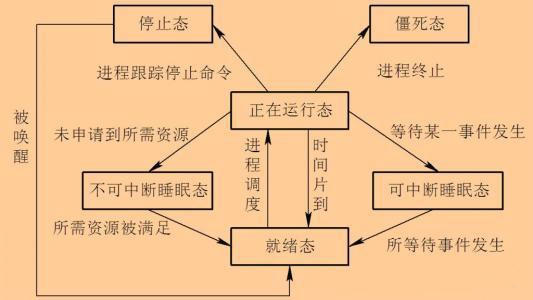
* 进程id。系统中每个进程有唯一的id，在C语言中用pid\_t类型表示，其实就是一个非负整数。
* 进程的状态，有就绪、运行、挂起、停止等状态。
* 进程切换时需要保存和恢复的一些CPU寄存器。
* 描述虚拟地址空间的信息。
* 描述控制终端的信息。
* 当前工作目录（Current Working Directory）。
* umask掩码。
* 文件描述符表，包含很多指向file结构体的指针。
* 和信号相关的信息。
* 用户id和组id。
* 会话（Session）和进程组。
* 进程可以使用的资源上限（Resource Limit）。

### 2.1.6 进程状态

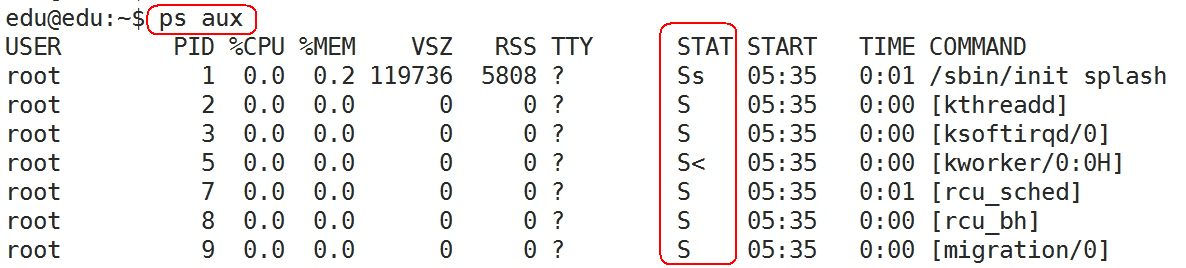
进程状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化而转换。

在三态模型中，进程状态分为三个基本状态，即运行态，就绪态，阻塞态。

在五态模型中，进程分为新建态、终止态，运行态，就绪态，阻塞态。



如何查看进程状态：



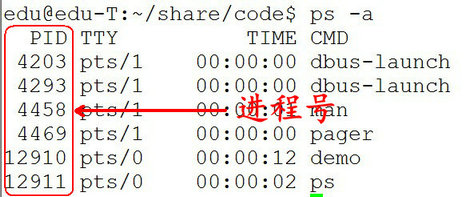
stat中的参数意义如下：

|  |  |
| --- | --- |
| **参数** | **含义** |
| D | 不可中断 Uninterruptible（usually IO） |
| R | 正在运行，或在队列中的进程 |
| S(大写) | 处于休眠状态 |
| T | 停止或被追踪 |
| Z | 僵尸进程 |
| W | 进入内存交换（从内核2.6开始无效） |
| X | 死掉的进程 |
| < | 高优先级 |
| N | 低优先级 |
| s | 包含子进程 |
| + | 位于后台的进程组 |

## 2.2 进程创建

### 2.2.1 进程号

每个进程都由一个进程号来标识，其类型为 pid\_t（无符号整型），进程号的范围：0～32767。进程号总是唯一的，但进程号可以重用。当一个进程终止后，其进程号就可以再次使用。



接下来，再给大家介绍三个不同的进程号。

* **进程号（PID）**：

标识进程的一个非负整型数。

* **父进程号（PPID）**：

任何进程（ 除 init 进程）都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号（PPID）。如，A 进程创建了 B 进程，A 的进程号就是 B 进程的父进程号。

* **进程组号（PGID）**：

进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号（PGID） 。这个过程有点类似于 QQ 群，组相当于 QQ 群，各个进程相当于各个好友，把各个好友都拉入这个 QQ 群里，主要是方便管理，特别是通知某些事时，只要在群里吼一声，所有人都收到，简单粗暴。但是，这个进程组号和 QQ 群号是有点区别的，默认的情况下，当前的进程号会当做当前的进程组号。

#include <sys/types.h>

#include <unistd.h>

**pid\_t getpid(void);**

**功能：**

获取本进程号（PID）

**参数：**

无

**返回值：**

本进程号

**pid\_t getppid(void);**

**功能：**

获取调用此函数的进程的父进程号（PPID）

**参数：**

无

**返回值：**

调用此函数的进程的父进程号（PPID）

**pid\_t getpgid(pid\_t pid);**

**功能：**

获取进程组号（PGID）

**参数：**

pid：进程号

**返回值：**

参数为 0 时返回当前进程组号，否则返回参数指定的进程的进程组号

示例代码：

int main()

{

pid\_t pid, ppid, pgid;

pid = getpid();

printf("pid = %d\n", pid);

ppid = getppid();

printf("ppid = %d\n", ppid);

pgid = getpgid(pid);

printf("pgid = %d\n", pgid);

return 0;

}

### 2.2.2 进程的创建

2.2.2.1 创建进程

系统允许一个进程创建新进程，新进程即为子进程，子进程还可以创建新的子进程，形成进程树结构模型。

#include <sys/types.h>

#include <unistd.h>

**pid\_t fork(void);**

**功能：**

用于从一个已存在的进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

**参数：**

无

**返回值：**

成功：子进程中返回 0，父进程中返回子进程 ID。pid\_t，为无符号整型。

失败：返回-1。

失败的两个主要原因是：

1）当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。

2）系统内存不足，这时 errno 的值被设置为 ENOMEM。

示例代码：

int main()

{

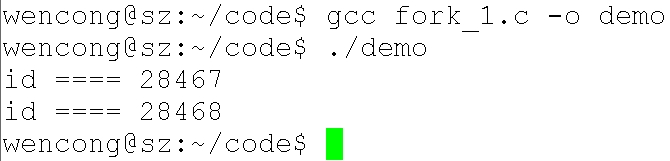
fork();

printf("id ==== %d\n", getpid()); // 获取进程号

return 0;

}

运行结果如下：

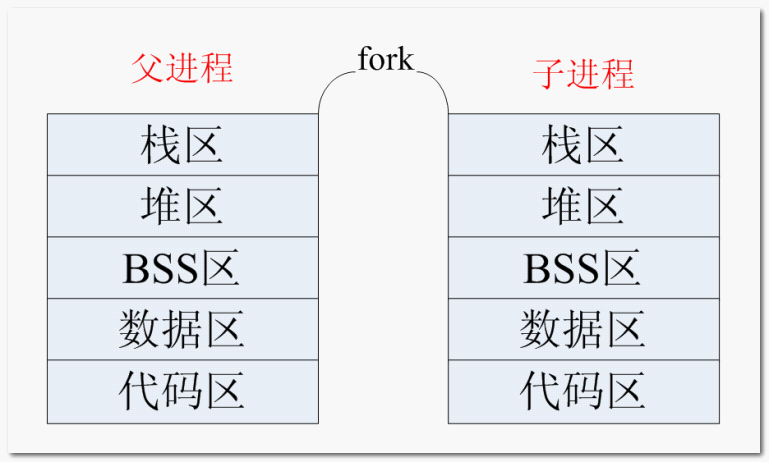


从运行结果，我们可以看出，fork() 之后的打印函数打印了两次，而且打印了两个进程号，这说明，fork() 之后确实创建了一个新的进程，新进程为子进程，原来的进程为父进程。

2.2.2.2 父子进程关系

使用 fork() 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间：包括进程上下文（进程执行活动全过程的静态描述）、进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等。

子进程所独有的只有它的进程号，计时器等（只有小量信息）。因此，使用 fork() 函数的代价是很大的。



简单来说， 一个进程调用 fork() 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

实际上，更准确来说，Linux 的 fork() 使用是通过写时拷贝 (copy- on-write) 实现。写时拷贝是一种可以推迟甚至避免拷贝数据的技术。内核此时并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间。只用在需要写入的时候才会复制地址空间，从而使各个进行拥有各自的地址空间。也就是说，资源的复制是在需要写入的时候才会进行，在此之前，只有以只读方式共享。

注意：fork之后父子进程共享文件，fork产生的子进程与父进程相同的文件文件描述符指向相同的文件表，引用计数增加，共享文件文件偏移指针。

2.2.2.3 区分父子进程

子进程是父进程的一个复制品，可以简单认为父子进程的代码一样的。那大家想过没有，这样的话，父进程做了什么事情，子进程也做什么事情（如上面的例子），是不是不能实现满足我们实现多任务的要求呀，那我们是不是要想个办法区别父子进程呀，这就通过 fork() 的返回值。

fork() 函数被调用一次，但返回两次。两次返回的区别是：子进程的返回值是 0，而父进程的返回值则是新子进程的进程 ID。

示例代码：

int main()

{

pid\_t pid;

pid = fork();

if (pid < 0)

{ // 没有创建成功

perror("fork");

}

if (0 == pid)

{ // 子进程

while (1)

{

printf("I am son\n");

sleep(1);

}

}

else if (pid > 0)

{ // 父进程

while (1)

{

printf("I am father\n");

sleep(1);

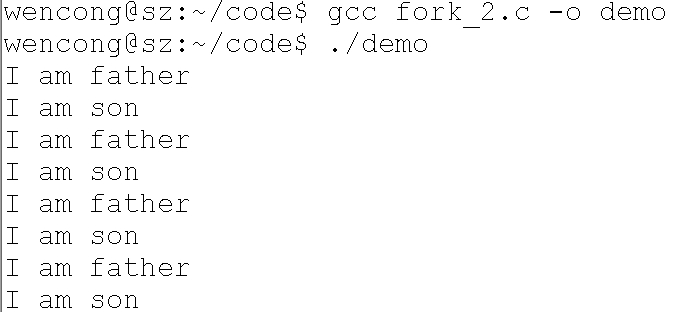
}

}

return 0;

}

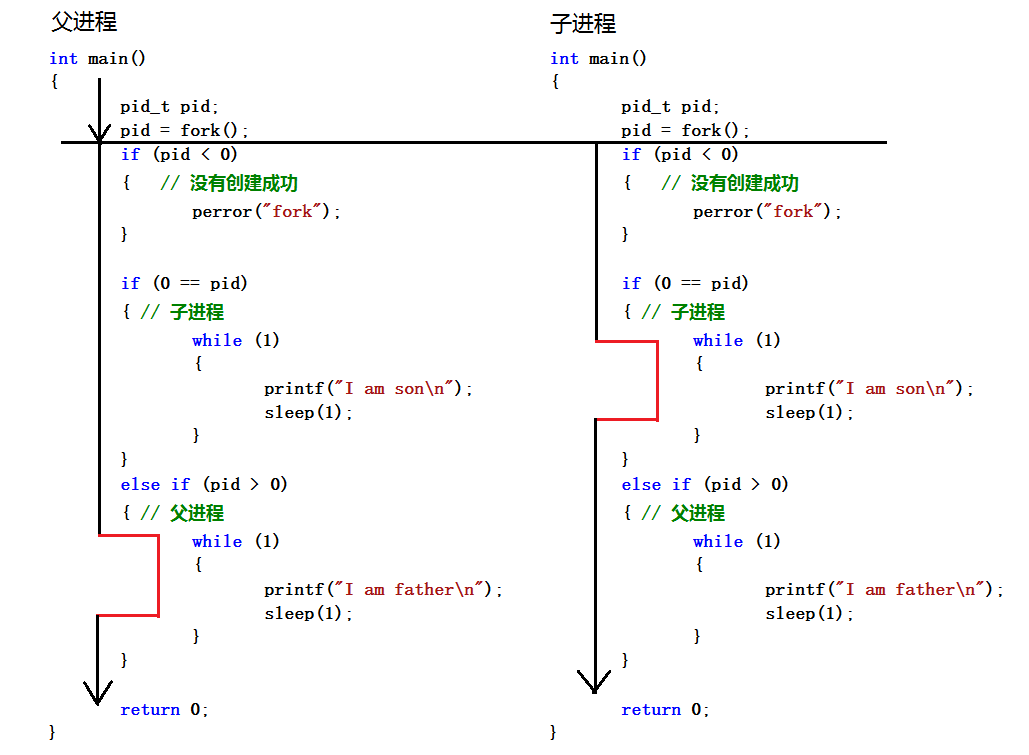
运行结果如下：



通过运行结果，可以看到，父子进程各做一件事（各自打印一句话）。这里，我们只是看到只有一份代码，实际上，fork() 以后，有两个地址空间在独立运行着，有点类似于有两个独立的程序（父子进程）在运行着。

一般来说，在 fork() 之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。

需要注意的是，在子进程的地址空间里，子进程是从 fork() 这个函数后才开始执行代码。



2.2.2.4 父子进程地址空间

下面的例子，为验证父子进程各自的地址空间是独立的：

int a = 10; // 全局变量

int main()

{

int b = 20; //局部变量

pid\_t pid;

pid = fork();

if (pid < 0)

{ // 没有创建成功

perror("fork");

}

if (0 == pid)

{ // 子进程

a = 111;

b = 222; // 子进程修改其值

printf("son: a = %d, b = %d\n", a, b);

}

else if (pid > 0)

{ // 父进程

sleep(1); // 保证子进程先运行

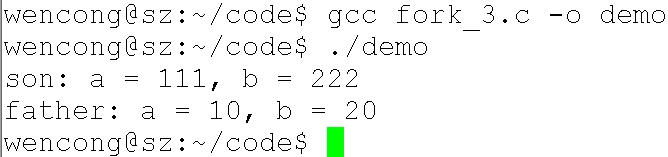
printf("father: a = %d, b = %d\n", a, b);

}

return 0;

}

运行结果如下：



通过得知，在子进程修改变量 a，b 的值，并不影响到父进程 a，b 的值。

## 2.3 进程相关命令

### 2.3.1 ps

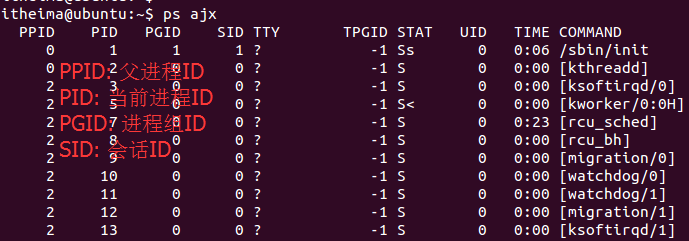
ps命令可以查看进程的详细状况，常用选项(选项可以不加“-”)如下：

|  |  |
| --- | --- |
| **选项** | **含义** |
| -a | 显示终端上的所有进程，包括其他用户的进程 |
| -u | 显示进程的详细状态 |
| -x | 显示没有控制终端的进程 |
| -j | 列出与作业控制相关的信息 |

显示当前用户下所有进程：**ps aux**



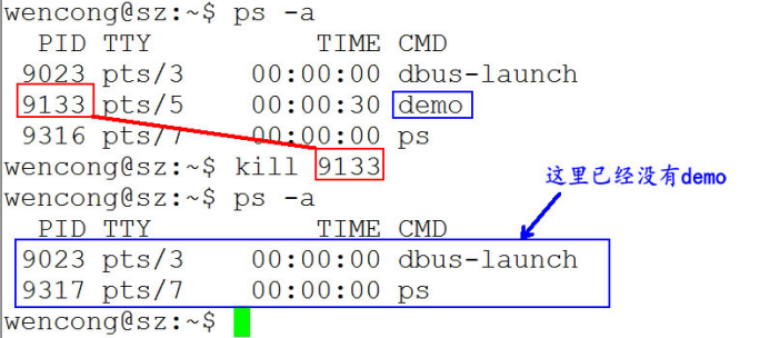
以比较完整的格式显示所有的进程：ps ajx



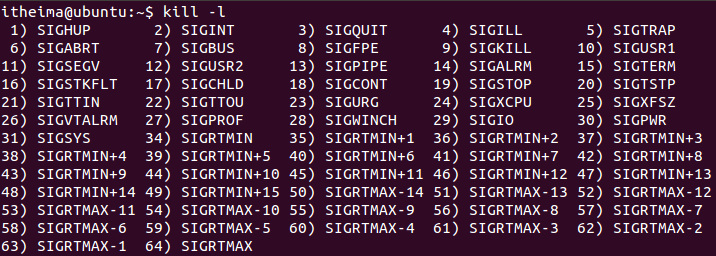
### 2.3.2 kill

命令功能: 发送指定的信号到相应进程。

**kill 9133**：9133 为应用程序所对应的进程号



查看信号编号: kill -l(字母)



有些进程不能直接杀死，这时候我们需要加一个参数“-9”，使用9号信号SIGKILL

杀死进程: kill -SIGKILL/(-9) 89899【进程标识号】

## 2.4 多进程gdb调试

使用gdb调试的时候，gdb只能跟踪一个进程。可以在fork函数调用之前，通过指令设置gdb调试工具跟踪父进程或者是跟踪子进程。默认跟踪父进程。

* set follow-fork-mode child 设置gdb在fork之后跟踪子进程。
* set follow-fork-mode parent 设置跟踪父进程（默认）。

注意，一定要在fork函数调用之前设置才有效。

## 2.5 进程控制

### 2.5.1 结束进程

#include <stdlib.h>

**void exit(int status);**

#include <unistd.h>

**void \_exit(int status);**

**功能：**

结束调用此函数的进程。

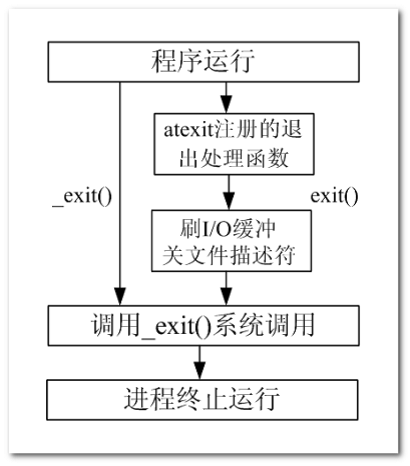
**参数：**

status：返回给父进程的参数（低 8 位有效），至于这个参数是多少根据需要来填写。

**返回值：**

无

exit() 和 \_exit() 函数功能和用法是一样的，无非时所包含的头文件不一样，还有的区别就是：exit()属于标准库函数，\_exit()属于系统调用函数。



### 2.5.2 回收子进程

2.5.2.1 概述

在每个进程退出的时候，内核释放该进程所有的资源、包括打开的文件、占用的内存等。但是仍然为其保留一定的信息，这些信息主要主要指进程控制块PCB的信息（包括进程号、退出状态、运行时间等）。

父进程可以通过调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。

wait() 和 waitpid() 函数的功能一样，区别在于，wait() 函数会阻塞，waitpid() 可以设置不阻塞，waitpid() 还可以指定等待哪个子进程结束。

注意：一次wait或waitpid调用只能清理一个子进程，清理多个子进程应使用循环。

2.5.2.2 wait

#include <sys/types.h>

#include <sys/wait.h>

**pid\_t wait(int \*status);**

**功能：**

等待任意一个子进程结束，如果任意一个子进程结束了，此函数会回收该子进程的资源。

**参数：**

status : 进程退出时的状态信息。

**返回值：**

成功：已经结束子进程的进程号

失败： -1

调用 wait() 函数的进程会挂起（阻塞），直到它的一个子进程退出或收到一个不能被忽视的信号时才被唤醒（相当于继续往下执行）。

若调用进程没有子进程，该函数立即返回；若它的子进程已经结束，该函数同样会立即返回，并且会回收那个早已结束进程的资源。

所以，wait()函数的主要功能为回收已经结束子进程的资源。

如果参数 status 的值不是 NULL，wait() 就会把子进程退出时的状态取出并存入其中，这是一个整数值（int），指出了子进程是正常退出还是被非正常结束的。

这个退出信息在一个 int 中包含了多个字段，直接使用这个值是没有意义的，我们需要用宏定义取出其中的每个字段。

宏函数可分为如下三组：

1. WIFEXITED(status)

* 为非0 → 进程正常结束

WEXITSTATUS(status)

* 如上宏为真，使用此宏 → 获取进程退出状态 (exit的参数)

1. WIFSIGNALED(status)

* 为非0 → 进程异常终止

WTERMSIG(status)

* 如上宏为真，使用此宏 → 取得使进程终止的那个信号的编号。

1. WIFSTOPPED(status)
   * 为非0 → 进程处于暂停状态

WSTOPSIG(status)

* 如上宏为真，使用此宏 → 取得使进程暂停的那个信号的编号。

WIFCONTINUED(status)

* 为真 → 进程暂停后已经继续运行

2.5.2.3 waitpid

#include <sys/types.h>

#include <sys/wait.h>

**pid\_t waitpid(pid\_t pid, int \*status, int options);**

**功能：**

等待子进程终止，如果子进程终止了，此函数会回收子进程的资源。

**参数：**

pid : 参数 pid 的值有以下几种类型：

pid > 0 等待进程 ID 等于 pid 的子进程。

pid = 0 等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid 不会等待它。

pid = -1 等待任一子进程，此时 waitpid 和 wait 作用一样。

pid < -1 等待指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

status : 进程退出时的状态信息。和 wait() 用法一样。

options : options 提供了一些额外的选项来控制 waitpid()。

0：同 wait()，阻塞父进程，等待子进程退出。

WNOHANG：没有任何已经结束的子进程，则立即返回。

WUNTRACED：如果子进程暂停了则此函数马上返回，并且不予以理会子进程的结束状态。（由于涉及到一些跟踪调试方面的知识，加之极少用到）

**返回值：**

waitpid() 的返回值比 wait() 稍微复杂一些，一共有 3 种情况：

1) 当正常返回的时候，waitpid() 返回收集到的已经回收子进程的进程号；

2) 如果设置了选项 WNOHANG，而调用中 waitpid() 发现没有已退出的子进程可等待，则返回 0；

3) 如果调用中出错，则返回-1，这时 errno 会被设置成相应的值以指示错误所在，如：当 pid 所对应的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid() 就会出错返回，这时 errno 被设置为 ECHILD；

2.5.2.4 孤儿进程

进程运行结束，但子进程还在运行（未运行结束）的子进程就称为孤儿进程（Orphan Process）。

每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init ，而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。

因此孤儿进程并不会有什么危害。

2.5.2.5 僵尸进程

进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

这样就会导致一个问题，如果进程不调用wait() 或 waitpid() 的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害，应当避免。

## 2.6 进程替换

### 2.6.1 概述

在 Windows 平台下，我们可以通过双击运行可执行程序，让这个可执行程序成为一个进程；而在 Linux 平台，我们可以通过 ./ 运行，让一个可执行程序成为一个进程。

但是，如果我们本来就运行着一个程序（进程），我们如何在这个进程内部启动一个外部程序，由内核将这个外部程序读入内存，使其执行起来成为一个进程呢？这里我们通过 exec 函数族实现。

exec 函数族，顾名思义，就是一簇函数，在 Linux 中，并不存在 exec() 函数，exec 指的是一组函数，一共有 6 个：

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

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

int execle(const char \*path, const char \*arg, ..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

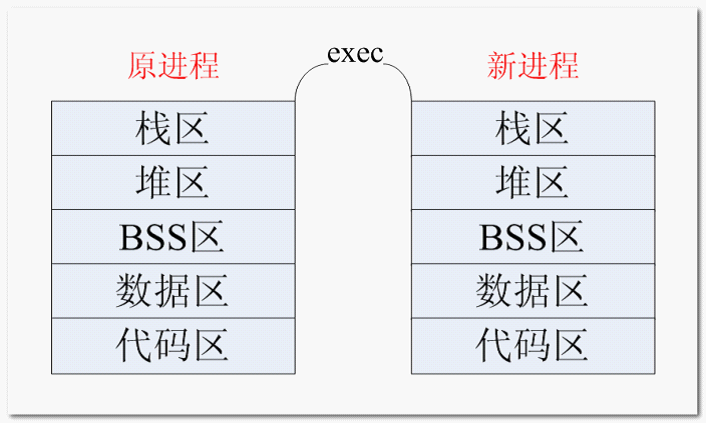
int execvp(const char \*file, char \*const argv[]);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

其中只有 execve() 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

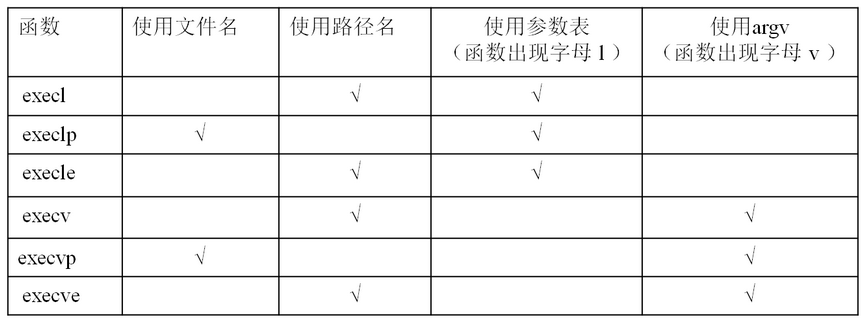
exec 函数族的作用是根据指定的文件名或目录名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。

进程调用一种 exec 函数时，该进程完全由新程序替换，而新程序则从其 main 函数开始执行。因为调用 exec 并不创建新进程，所以前后的进程 ID （当然还有父进程号、进程组号、当前工作目录……）并未改变。exec 只是用另一个新程序替换了当前进程的正文、数据、堆和栈段（进程替换）。



### 2.6.2 exec 函数族使用说明

exec 函数族的 6 个函数看起来似乎很复杂，但实际上无论是作用还是用法都非常相似，只有很微小的差别。



补充说明：

|  |  |
| --- | --- |
| l(list) | 参数地址列表，以空指针结尾 |
| v(vector) | 存有各参数地址的指针数组的地址 |
| p(path) | 按 PATH 环境变量指定的目录搜索可执行文件 |
| e(environment) | 存有环境变量字符串地址的指针数组的地址 |

exec 函数族与一般的函数不同，exec 函数族中的函数执行成功后不会返回，而且，exec 函数族下面的代码执行不到。只有调用失败了，它们才会返回 -1，失败后从原程序的调用点接着往下执行。

# 4. 进程间通信

## 4.1 什么是进程间通信

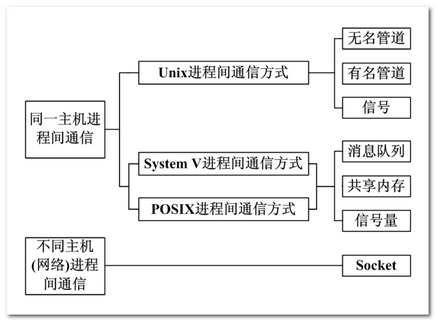
进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。

但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信( IPC：Inter Processes Communication )。

进程间通信的目的：

* 数据传输：一个进程需要将它的数据发送给另一个进程。
* 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
* 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制。
* 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

Linux 操作系统支持的主要进程间通信的通信机制：



## 4.2 管道

### 4.2.1 概述

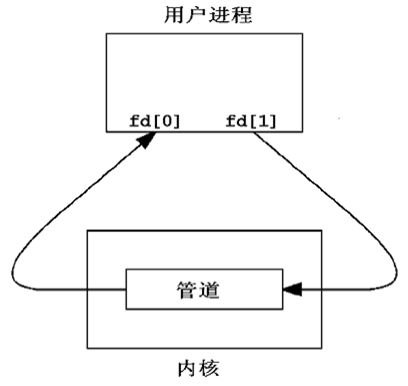
管道也叫无名管道，它是是 UNIX 系统 IPC（进程间通信） 的最古老形式，所有的 UNIX 系统都支持这种通信机制。

管道有如下特点：

1. 半双工，数据在同一时刻只能在一个方向上流动。
2. 数据只能从管道的一端写入，从另一端读出。
3. 写入管道中的数据遵循先入先出的规则。
4. 管道所传送的数据是无格式的，这要求管道的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
5. 管道不是普通的文件，不属于某个文件系统，其只存在于内存中。
6. 管道在内存中对应一个缓冲区。不同的系统其大小不一定相同。
7. 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据。
8. 管道没有名字，只能在具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用。

对于管道特点的理解，我们可以类比现实生活中管子，管子的一端塞东西，管子的另一端取东西。

管道是一种特殊类型的文件，在应用层体现为两个打开的文件描述符。



### 4.2.2 管道的操作

#include <unistd.h>

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

**功能**：创建无名管道。

**参数**：

pipefd : 为 int 型数组的首地址，其存放了管道的文件描述符 pipefd[0]、pipefd[1]。

当一个管道建立时，它会创建两个文件描述符 fd[0] 和 fd[1]。其中 fd[0] 固定用于读管道，而 fd[1] 固定用于写管道。一般文件 I / O 的函数都可以用来操作管道(lseek() 除外)。

**返回值**：

成功：0

失败：-1

下面我们写这个一个例子，子进程通过无名管道给父进程传递一个字符串数据：

int main()

{

int fd\_pipe[2] = { 0 };

pid\_t pid;

if (pipe(fd\_pipe) < 0)

{// 创建管道

perror("pipe");

}

pid = fork(); // 创建进程

if (pid == 0)

{ // 子进程

char buf[] = "I am mike";

// 往管道写端写数据

write(fd\_pipe[1], buf, strlen(buf));

\_exit(0);

}

else if (pid > 0)

{// 父进程

wait(NULL); // 等待子进程结束，回收其资源

char str[50] = { 0 };

// 从管道里读数据

read(fd\_pipe[0], str, sizeof(str));

printf("str=[%s]\n", str); // 打印数据

}

return 0;

}

### 4.2.3 管道的读写特点

4.2.3.1 读写特点

使用管道需要注意以下4种特殊情况（假设都是阻塞I/O操作，没有设置O\_NONBLOCK标志）：

1. 如果所有指向管道写端的文件描述符都关闭了（管道写端引用计数为0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次read会返回0，就像读到文件末尾一样。
2. 如果有指向管道写端的文件描述符没关闭（管道写端引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可读了才读取数据并返回。
3. 如果所有指向管道读端的文件描述符都关闭了（管道读端引用计数为0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。当然也可以对SIGPIPE信号实施捕捉，不终止进程。具体方法信号章节详细介绍。
4. 如果有指向管道读端的文件描述符没关闭（管道读端引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了才写入数据并返回。

**总结：**

* 读管道：
  + 管道中有数据，read返回实际读到的字节数。
  + 管道中无数据：
    - 管道写端被全部关闭，read返回0 (相当于读到文件结尾)
    - 写端没有全部被关闭，read阻塞等待(不久的将来可能有数据递达，此时会让出cpu)
* 写管道：
  + 管道读端全部被关闭， 进程异常终止(也可使用捕捉SIGPIPE信号，使进程不终止)
  + 管道读端没有全部关闭：
    - 管道已满，write阻塞。
    - 管道未满，write将数据写入，并返回实际写入的字节数。

4.2.3.2 设置非阻塞

设置方法：

//获取原来的flags

int flags = fcntl(fd[0], F\_GETFL);

// 设置新的flags

flag |= O\_NONBLOCK;

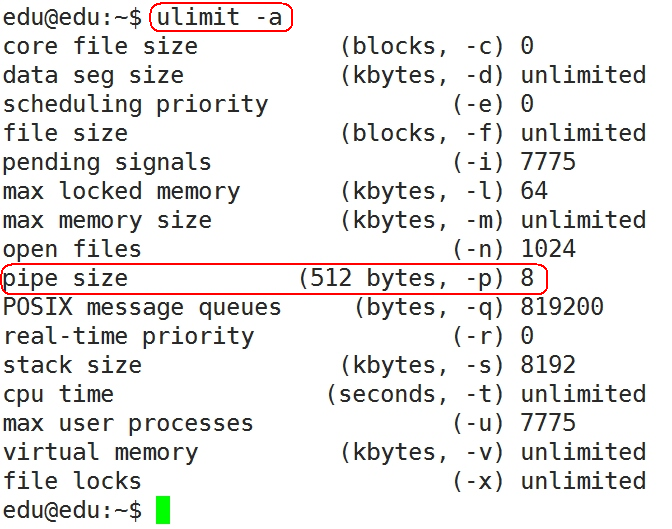
// flags = flags | O\_NONBLOCK;

fcntl(fd[0], F\_SETFL, flags);

### 4.2.4 查看管道缓冲区大小

4.2.4.1 命令

可以使用ulimit -a 命令来查看当前系统中创建管道文件所对应的内核缓冲区大小。



4.2.4.2 函数

#include <unistd.h>

**long fpathconf(int fd, int name);**

**功能：**该函数可以通过name参数查看不同的属性值

**参数：**

fd：文件描述符

name：

\_PC\_PIPE\_BUF，查看管道缓冲区大小

\_PC\_NAME\_MAX，文件名字字节数的上限

**返回值：**

成功：根据name返回的值的意义也不同。

失败： -1

使用示例：

int main()

{

int fd[2];

int ret = pipe(fd);

if (ret == -1)

{

perror("pipe error");

exit(1);

}

long num = fpathconf(fd[0], \_PC\_PIPE\_BUF);

printf("num = %ld\n", num);

return 0;

}

## 4.3 FIFO

### 4.3.1 概述

管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了命名管道（FIFO），也叫有名管道、FIFO文件。

命名管道（FIFO）不同于无名管道之处在于它提供了一个路径名与之关联，以 FIFO 的文件形式存在于文件系统中，这样，即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信，因此，通过 FIFO 不相关的进程也能交换数据。

命名管道（FIFO)和无名管道（pipe）有一些特点是相同的，不一样的地方在于：

1. FIFO 在文件系统中作为一个特殊的文件而存在，但 FIFO 中的内容却存放在内存中。
2. 当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。
3. FIFO 有名字，不相关的进程可以通过打开命名管道进行通信。

### 4.3.2 FIFO的创建

4.3.2.1 通过命令

mkfifo 管道名

4.3.2.2 通过库函数

#include <sys/types.h>

#include <sys/stat.h>

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

**功能：**

命名管道的创建。

**参数：**

pathname : 普通的路径名，也就是创建后 FIFO 的名字。

mode : 文件的权限，与打开普通文件的 open() 函数中的 mode 参数相同。(0666)

**返回值：**

成功：0 状态码

失败：如果文件已经存在，则会出错且返回 -1。

### 4.3.3 FIFO的操作

一旦使用mkfifo创建了一个FIFO，就可以使用open打开它，常见的文件I/O函数都可用于fifo。如：close、read、write、unlink等。

FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。**它们不支持诸如lseek()等文件定位操作。**

//进行1，写操作

int fd = open("my\_fifo", O\_WRONLY);

char send[100] = "Hello Mike";

write(fd, send, strlen(send));

//进程2，读操作

int fd = open("my\_fifo", O\_RDONLY);//等着只写

char recv[100] = { 0 };

//读数据，命名管道没数据时会阻塞，有数据时就取出来

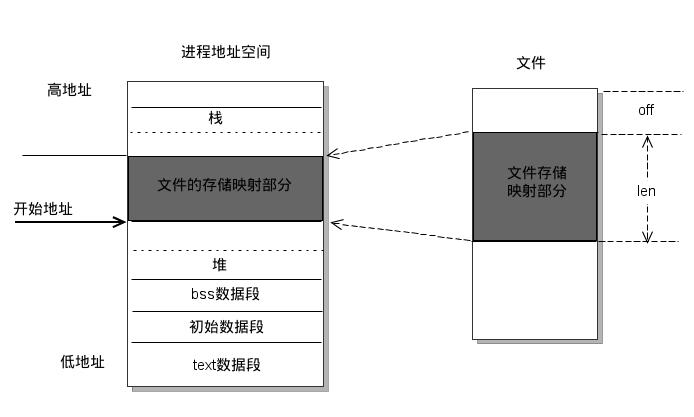
read(fd, recv, sizeof(recv));

printf("read from my\_fifo buf=[%s]\n", recv);

## 4.4 共享存储映射

### 4.4.1 概述

存储映射I/O (Memory-mapped I/O) 使一个磁盘文件与存储空间中的一个缓冲区相映射。



于是当从缓冲区中取数据，就相当于读文件中的相应字节。于此类似，将数据存入缓冲区，则相应的字节就自动写入文件。这样，就可在不适用read和write函数的情况下，使用地址（指针）完成I/O操作。

共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式, 因为进程可以直接读写内存，而不需要任何数据的拷贝。

### 4.4.2 存储映射API说明

4.4.2.1 mmap

#include <sys/mman.h>

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

**功能：**

一个文件或者其它对象映射进内存

**参数：**

addr : 指定映射的起始地址, 通常设为NULL, 由系统指定

length：映射到内存的文件长度

prot： 映射区的保护方式, 最常用的 :

a) 读：PROT\_READ

b) 写：PROT\_WRITE

c) 读写：PROT\_READ | PROT\_WRITE

flags： 映射区的特性, 可以是

a) MAP\_SHARED : 写入映射区的数据会复制回文件, 且允许其他映射该文件的进程共享。

b) MAP\_PRIVATE : 对映射区的写入操作会产生一个映射区的复制(copy - on - write), 对此区域所做的修改不会写回原文件。

fd：由open返回的文件描述符, 代表要映射的文件。

offset：以文件开始处的偏移量, 必须是4k的整数倍, 通常为0, 表示从文件头开始映射

**返回值：**

成功：返回创建的映射区首地址

失败：MAP\_FAILED宏

关于mmap函数的使用总结：

1. 第一个参数写成NULL
2. 第二个参数要映射的文件大小 > 0
3. 第三个参数：PROT\_READ 、PROT\_WRITE
4. 第四个参数：MAP\_SHARED 或者 MAP\_PRIVATE
5. 第五个参数：打开的文件对应的文件描述符
6. 第六个参数：4k的整数倍，通常为0

4.4.2.2 munmap

#include <sys/mman.h>

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

**功能：**

释放内存映射区

**参数：**

addr：使用mmap函数创建的映射区的首地址

length：映射区的大小

**返回值：**

成功：0

失败：-1

4.4.2.3 注意事项

1. 创建映射区的过程中，隐含着一次对映射文件的读操作。
2. 当MAP\_SHARED时，要求：映射区的权限应 <=文件打开的权限(出于对映射区的保护)。而MAP\_PRIVATE则无所谓，因为mmap中的权限是对内存的限制。
3. 映射区的释放与文件关闭无关。只要映射建立成功，文件可以立即关闭。
4. 特别注意，当映射文件大小为0时，不能创建映射区。所以，用于映射的文件必须要有实际大小。mmap使用时常常会出现总线错误，通常是由于共享文件存储空间大小引起的。
5. munmap传入的地址一定是mmap的返回地址。坚决杜绝指针++操作。
6. 如果文件偏移量必须为4K的整数倍。
7. mmap创建映射区出错概率非常高，一定要检查返回值，确保映射区建立成功再进行后续操作。

### 4.4.3 共享存储映射的使用

4.4.2.1 通过共享映射的方式操作文件

int fd = open("xxx.txt", O\_RDWR); //读写文件

int len = lseek(fd, 0, SEEK\_END); //获取文件大小

//一个文件映射到内存，ptr指向此内存

void \* ptr = mmap(NULL, len, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

if (ptr == MAP\_FAILED)

{

perror("mmap error");

exit(1);

}

close(fd); //关闭文件

char buf[4096];

printf("buf = %s\n", (char\*)ptr); // 从内存中读数据，等价于从文件中读取内容

strcpy((char\*)ptr, "this is a test");//写内容

int ret = munmap(ptr, len);

if (ret == -1)

{

perror("munmap error");

exit(1);

}

4.4.2.2 使用共享映射实现两个进程之间的通信

int fd = open("xxx.txt", O\_RDWR);// 打开一个文件

int len = lseek(fd, 0, SEEK\_END);//获取文件大小

// 创建内存映射区

void \*ptr = mmap(NULL, len, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

if (ptr == MAP\_FAILED)

{

perror("mmap error");

exit(1);

}

close(fd); //关闭文件

// 创建子进程

pid\_t pid = fork();

if (pid == 0) //子进程

{

sleep(1); //演示，保证父进程先执行

// 读数据

printf("%s\n", (char\*)ptr);

}

else if (pid > 0) //父进程

{

// 写数据

strcpy((char\*)ptr, "i am u father!!");

// 回收子进程资源

wait(NULL);

}

// 释放内存映射区

int ret = munmap(ptr, len);

if (ret == -1)

{

perror("munmap error");

exit(1);

}

4.4.2.3 通过匿名映射实现父子进程通信

通过使用我们发现，使用映射区来完成文件读写操作十分方便，父子进程间通信也较容易。

但缺陷是，每次创建映射区一定要依赖一个文件才能实现。

通常为了建立映射区要open一个temp文件，创建好了再unlink、close掉，比较麻烦。 可以直接使用匿名映射来代替。

其实Linux系统给我们提供了创建匿名映射区的方法，无需依赖一个文件即可创建映射区。同样需要借助标志位参数flags来指定。

使用MAP\_ANONYMOUS (或MAP\_ANON)， 如:

int \*p = mmap(NULL, 4, PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS, -1, 0);

* 4"随意举例，该位置表示映射区大小，可依实际需要填写。
* MAP\_ANONYMOUS和MAP\_ANON这两个宏是Linux操作系统特有的宏。在类Unix系统中如无该宏定义，可使用如下两步来完成匿名映射区的建立。

// 创建匿名内存映射区

int len = 4096;

void \*ptr = mmap(NULL, len, PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANON, -1, 0);

if (ptr == MAP\_FAILED)

{

perror("mmap error");

exit(1);

}

// 创建子进程

pid\_t pid = fork();

if (pid > 0) //父进程

{

// 写数据

strcpy((char\*)ptr, "hello mike!!");

// 回收

wait(NULL);

}

else if (pid == 0)//子进程

{

sleep(1);

// 读数据

printf("%s\n", (char\*)ptr);

}

// 释放内存映射区

int ret = munmap(ptr, len);

if (ret == -1)

{

perror("munmap error");

exit(1);

}

## 4.5 信号

### 4.5.1 概述

4.5.1.1 什么是信号

信号是 Linux 进程间通信的最古老的方式。信号是软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式 。信号可以导致一个正在运行的进程被另一个正在运行的异步进程中断，转而处理某一个突发事件。

“中断”在我们生活中经常遇到，譬如，我正在房间里打游戏，突然送快递的来了，把正在玩游戏的我给“中断”了，我去签收快递( 处理中断 )，处理完成后，再继续玩我的游戏。

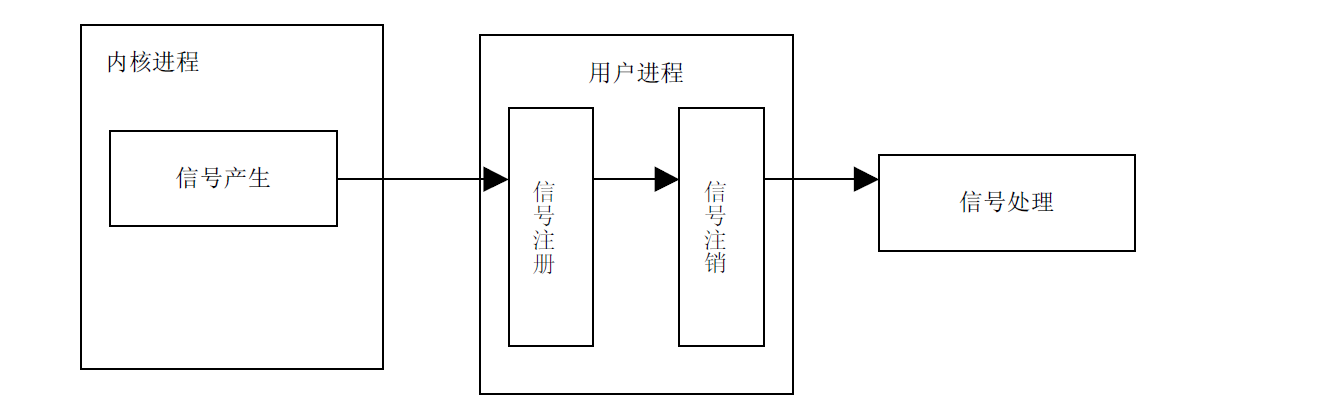
这里我们学习的“信号”就是属于这么一种“中断”。我们在终端上敲“Ctrl+c”，就产生一个“中断”，相当于产生一个信号，接着就会处理这么一个“中断任务”（默认的处理方式为中断当前进程）。

信号特点：

* 简单
* 不能携带大量信息
* 满足某个特设条件才发送

信号可以直接进行用户空间进程和内核空间进程的交互，内核进程可以利用它来通知用户空间进程发生了哪些系统事件。

一个完整的信号周期包括三个部分：信号的产生，信号在进程中的注册，信号在进程中的注销，执行信号处理函数。如下图所示：



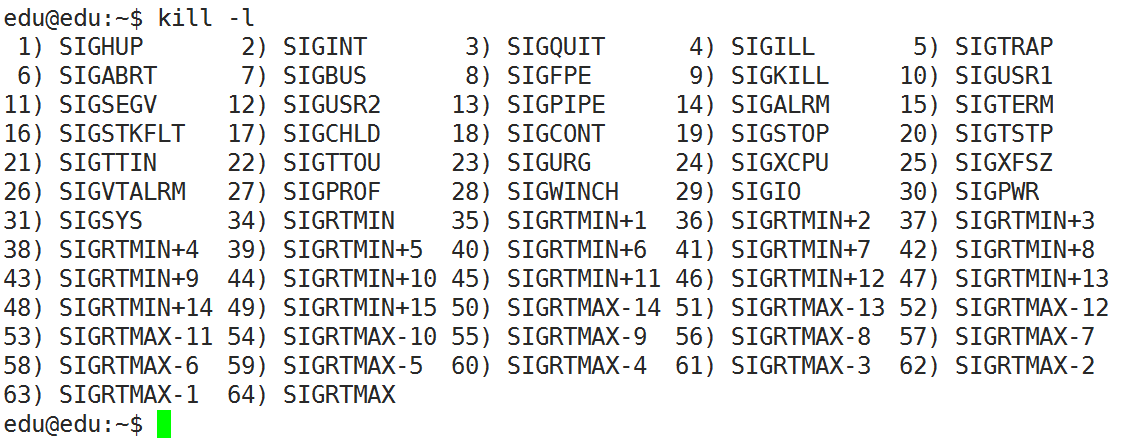
注意：这里信号的产生，注册，注销时信号的内部机制，而不是信号的函数实现。

4.5.1.2 信号的编号

1) 信号编号

Unix早期版本就提供了信号机制，但不可靠，信号可能丢失。Berkeley 和 AT&T都对信号模型做了更改，增加了可靠信号机制。但彼此不兼容。POSIX.1对可靠信号例程进行了标准化。

Linux 可使用命令：kill -l（"l" 为字母），查看相应的信号。



不存在编号为0的信号。其中1-31号信号称之为常规信号（也叫普通信号或标准信号），34-64称之为实时信号，驱动编程与硬件相关。名字上区别不大。而前32个名字各不相同。

2) Linux常规信号一览表

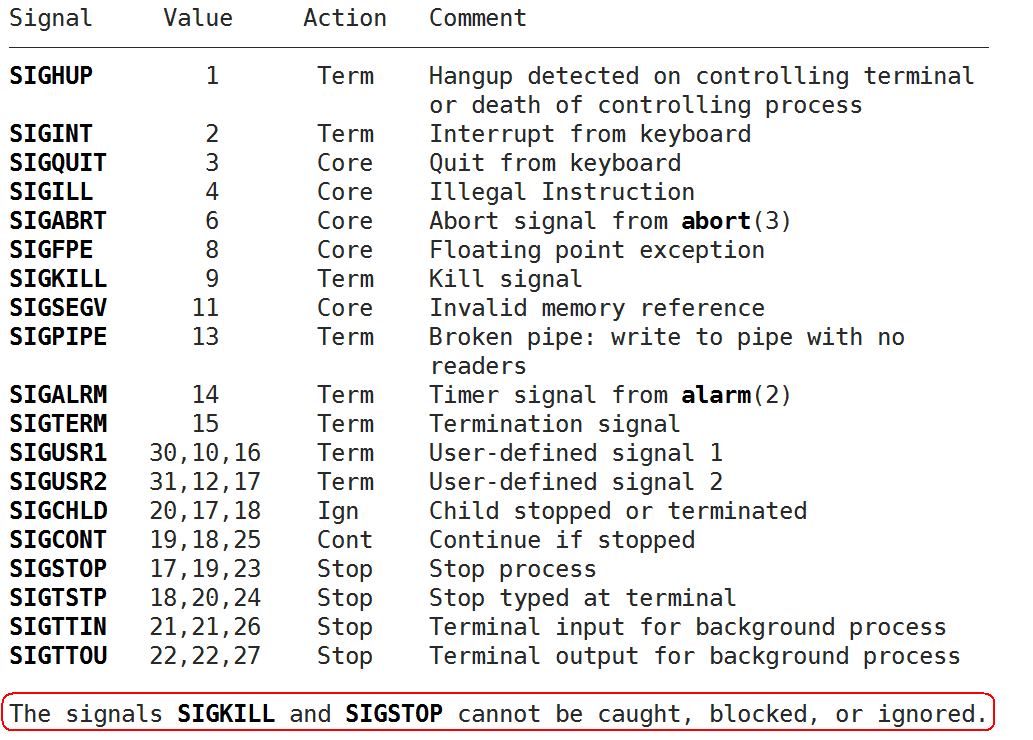
|  |  |  |  |
| --- | --- | --- | --- |
| **编号** | **信号** | **对应事件** | **默认动作** |
| 1 | SIGHUP | 用户退出shell时，由该shell启动的所有进程将收到这个信号 | 终止进程 |
| 2 | SIGINT | 当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号 | 终止进程 |
| 3 | SIGQUIT | 用户按下<ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号 | 终止进程 |
| 4 | SIGILL | CPU检测到某进程执行了非法指令 | 终止进程并产生core文件 |
| 5 | SIGTRAP | 该信号由断点指令或其他 trap指令产生 | 终止进程并产生core文件 |
| 6 | SIGABRT | 调用abort函数时产生该信号 | 终止进程并产生core文件 |
| 7 | SIGBUS | 非法访问内存地址，包括内存对齐出错 | 终止进程并产生core文件 |
| 8 | SIGFPE | 在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误 | 终止进程并产生core文件 |
| 9 | SIGKILL | 无条件终止进程。本信号不能被忽略，处理和阻塞 | 终止进程，可以杀死任何进程 |
| 10 | SIGUSE1 | 用户定义的信号。即程序员可以在程序中定义并使用该信号 | 终止进程 |
| 11 | SIGSEGV | 指示进程进行了无效内存访问(段错误) | 终止进程并产生core文件 |
| 12 | SIGUSR2 | 另外一个用户自定义信号，程序员可以在程序中定义并使用该信号 | 终止进程 |
| 13 | SIGPIPE | Broken pipe向一个没有读端的管道写数据 | 终止进程 |
| 14 | SIGALRM | 定时器超时，超时的时间 由系统调用alarm设置 | 终止进程 |
| 15 | SIGTERM | 程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号 | 终止进程 |
| 16 | SIGSTKFLT | Linux早期版本出现的信号，现仍保留向后兼容 | 终止进程 |
| 17 | SIGCHLD | 子进程结束时，父进程会收到这个信号 | 忽略这个信号 |
| 18 | SIGCONT | 如果进程已停止，则使其继续运行 | 继续/忽略 |
| 19 | SIGSTOP | 停止进程的执行。信号不能被忽略，处理和阻塞 | 为终止进程 |
| 20 | SIGTSTP | 停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号 | 暂停进程 |
| 21 | SIGTTIN | 后台进程读终端控制台 | 暂停进程 |
| 22 | SIGTTOU | 该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生 | 暂停进程 |
| 23 | SIGURG | 套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达 | 忽略该信号 |
| 24 | SIGXCPU | 进程执行时间超过了分配给该进程的CPU时间 ，系统产生该信号并发送给该进程 | 终止进程 |
| 25 | SIGXFSZ | 超过文件的最大长度设置 | 终止进程 |
| 26 | SIGVTALRM | 虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间 | 终止进程 |
| 27 | SGIPROF | 类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间 | 终止进程 |
| 28 | SIGWINCH | 窗口变化大小时发出 | 忽略该信号 |
| 29 | SIGIO | 此信号向进程指示发出了一个异步IO事件 | 忽略该信号 |
| 30 | SIGPWR | 关机 | 终止进程 |
| 31 | SIGSYS | 无效的系统调用 | 终止进程并产生core文件 |
| 34~64 | SIGRTMIN ～ SIGRTMAX | LINUX的实时信号，它们没有固定的含义（可以由用户自定义） | 终止进程 |

4.5.1.3 信号4要素

每个信号必备4要素，分别是：

1）编号 2）名称 3）事件 4）默认处理动作

可通过man 7 signal查看帮助文档获取：



在标准信号中，有一些信号是有三个“Value”，第一个值通常对alpha和sparc架构有效，中间值针对x86、arm和其他架构，最后一个应用于mips架构。一个‘-’表示在对应架构上尚未定义该信号。

不同的操作系统定义了不同的系统信号。因此有些信号出现在Unix系统内，也出现在Linux中，而有的信号出现在FreeBSD或Mac OS中却没有出现在Linux下。这里我们只研究Linux系统中的信号。

Action为默认动作：

* Term：终止进程
* Ign： 忽略信号 (默认即时对该种信号忽略操作)
* Core：终止进程，生成Core文件。(查验死亡原因，用于gdb调试)
* Stop：停止（暂停）进程
* Cont：继续运行进程

注意通过man 7 signal命令查看帮助文档，其中可看到 : The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

这里特别强调了**9) SIGKILL 和19) SIGSTOP信号，不允许忽略和捕捉，只能执行默认动作。甚至不能将其设置为阻塞。**

另外需清楚，只有每个信号所对应的事件发生了，该信号才会被递送(但不一定递达)，不应乱发信号！！

4.5.1.4 信号的状态

1. **产生**
   1. 当用户按某些终端键时，将产生信号。

* 终端上按“Ctrl+c”组合键通常产生中断信号 SIGINT
* 终端上按“Ctrl+\”键通常产生中断信号 SIGQUIT
* 终端上按“Ctrl+z”键通常产生中断信号 SIGSTOP 等。
  1. 硬件异常将产生信号。

除数为 0，无效的内存访问等。这些情况通常由硬件检测到，并通知内核，然后内核产生适当的信号发送给相应的进程。

* 1. 软件异常将产生信号。

当检测到某种软件条件已发生(如：定时器alarm)，并将其通知有关进程时，产生信号。

* 1. 调用系统函数(如：kill、raise、abort)将发送信号。

注意：接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者必须是超级用户。

* 1. 运行 kill 命令将发送信号。

此程序实际上是使用 kill 函数来发送信号。也常用此命令终止一个失控的后台进程。

1. **未决状态：没有被处理**
2. **递达：信号被处理了**

4.5.1.5 阻塞信号集和未决信号集

信号的实现手段导致信号有很强的延时性，但对于用户来说，时间非常短，不易察觉。

Linux内核的进程控制块PCB是一个结构体，task\_struct, 除了包含进程id，状态，工作目录，用户id，组id，文件描述符表，还包含了信号相关的信息，主要指**阻塞信号集和未决信号集**。

1) 阻塞信号集(信号屏蔽字)

将某些信号加入集合，对他们设置屏蔽，当屏蔽x信号后，再收到该信号，该信号的处理将推后(处理发生在解除屏蔽后)。

2) 未决信号集

* 信号产生，未决信号集中描述该信号的位立刻翻转为1，表信号处于未决状态。当信号被处理对应位翻转回为0。这一时刻往往非常短暂。
* 信号产生后由于某些原因(主要是阻塞)不能抵达。这类信号的集合称之为未决信号集。在屏蔽解除前，信号一直处于未决状态。

### 4.5.2 信号的产生

4.5.2.1 kill

#include <sys/types.h>

#include <signal.h>

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

**功能：**给指定进程发送指定信号(不一定杀死)

**参数：**

pid : 取值有 4 种情况 :

pid > 0: 将信号传送给进程 ID 为pid的进程。

pid = 0 : 将信号传送给当前进程所在进程组中的所有进程。

pid = -1 : 将信号传送给系统内所有的进程。

pid < -1 : 将信号传给指定进程组的所有进程。这个进程组号等于 pid 的绝对值。

sig : 信号的编号，这里可以填数字编号，也可以填信号的宏定义，可以通过命令 kill - l("l" 为字母)进行相应查看。不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致。

**返回值：**

成功：0

失败：-1

super用户(root)可以发送信号给任意用户，普通用户是不能向系统用户发送信号的。

kill -9 (root用户的pid) 是不可以的。同样，普通用户也不能向其他普通用户发送信号，终止其进程。 只能向自己创建的进程发送信号。

普通用户基本规则是：发送者实际或有效用户ID == 接收者实际或有效用户ID

示例代码：

int main()

{

pid\_t pid = fork();

if (pid == 0)

{//子进程

int i = 0;

for (i = 0; i<5; i++)

{

printf("in son process\n");

sleep(1);

}

}

else

{//父进程

printf("in father process\n");

sleep(2);

printf("kill sub process now \n");

kill(pid, SIGINT);

}

return 0;

}

4.5.2.2 raise

#include <signal.h>

**int raise(int sig);**

**功能：**给当前进程发送指定信号(自己给自己发)，等价于 kill(getpid(), sig)

**参数：**

sig：信号编号

**返回值：**

成功：0

失败：非0值

4.5.2.3 abort

#include <stdlib.h>

**void abort(void);**

**功能：**给自己发送异常终止信号 6) SIGABRT，并产生core文件，等价于kill(getpid(), SIGABRT);

**参数：**无

**返回值：**无

4.5.2.4 闹钟（定时器）

1) alarm

#include <unistd.h>

**unsigned int alarm(unsigned int seconds);**

**功能：**

设置定时器(闹钟)。在指定seconds后，内核会给当前进程发送14）SIGALRM信号。进程收到该信号，默认动作终止。每个进程都有且只有唯一的一个定时器。

取消定时器alarm(0)，返回旧闹钟余下秒数。

**参数：**

seconds：指定的时间，以秒为单位

**返回值：**

返回0或剩余的秒数

定时，与进程状态无关(自然定时法)！就绪、运行、挂起(阻塞、暂停)、终止、僵尸……无论进程处于何种状态，alarm都计时。

示例代码：

int main()

{

int seconds = 0;

seconds = alarm(5);

printf("seconds = %d\n", seconds);

sleep(2);

seconds = alarm(5);

printf("seconds = %d\n", seconds);

while (1);

return 0;

}

2) setitimer

#include <sys/time.h>

**int setitimer(int which, const struct itimerval \*new\_value, struct itimerval \*old\_value);**

**功能：**

设置定时器(闹钟)。 可代替alarm函数。精度微秒us，可以实现周期定时。

**参数：**

**which：**指定定时方式

a) 自然定时：ITIMER\_REAL → 14）SIGALRM计算自然时间

b) 虚拟空间计时(用户空间)：ITIMER\_VIRTUAL → 26）SIGVTALRM 只计算进程占用cpu的时间

c) 运行时计时(用户 + 内核)：ITIMER\_PROF → 27）SIGPROF计算占用cpu及执行系统调用的时间

**new\_value：**struct itimerval, 负责设定timeout时间

struct itimerval {

struct timerval it\_interval; // 闹钟触发周期

struct timerval it\_value; // 闹钟触发时间

};

struct timeval {

long tv\_sec; // 秒

long tv\_usec; // 微秒

}

itimerval.it\_value： 设定第一次执行function所延迟的秒数

itimerval.it\_interval： 设定以后每几秒执行function

**old\_value**： 存放旧的timeout值，一般指定为NULL

**返回值：**

成功：0

失败：-1

示例代码：

void myfunc(int sig)

{

printf("hello\n");

}

int main()

{

struct itimerval new\_value;

//定时周期

new\_value.it\_interval.tv\_sec = 1;

new\_value.it\_interval.tv\_usec = 0;

//第一次触发的时间

new\_value.it\_value.tv\_sec = 2;

new\_value.it\_value.tv\_usec = 0;

signal(SIGALRM, myfunc); //信号处理

setitimer(ITIMER\_REAL, &new\_value, NULL); //定时器设置

while (1);

return 0;

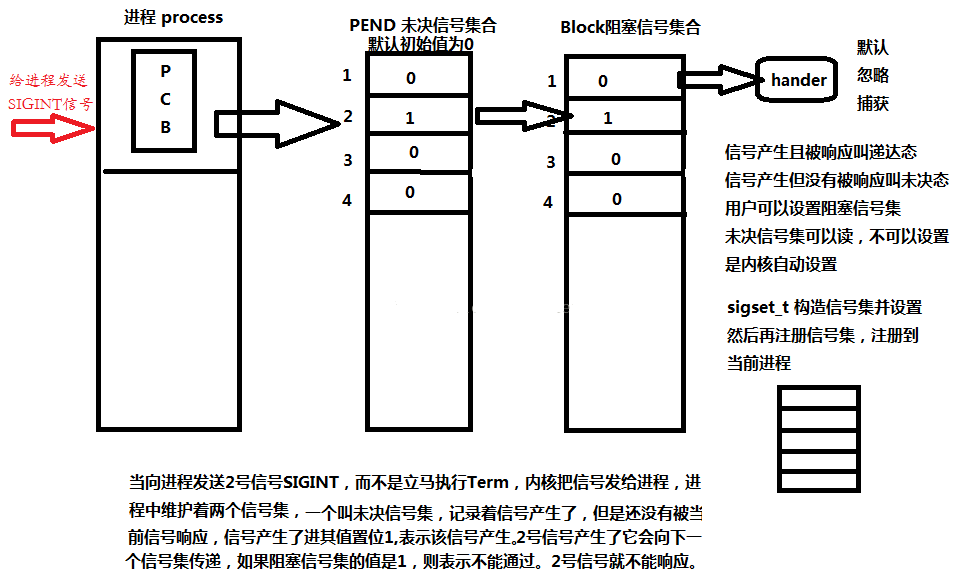
}

### 4.5.3 信号集

4.5.3.1 概述

在PCB中有两个非常重要的信号集。一个称之为“阻塞信号集”，另一个称之为“未决信号集”。

这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对其进行位操作。而需自定义另外一个集合，借助信号集操作函数来对PCB中的这两个信号集进行修改。



4.5.3.2 自定义信号集

为了方便对多个信号进行处理，一个用户进程常常需要对多个信号做出处理，在 Linux 系统中引入了信号集（信号的集合）。

这个信号集有点类似于我们的 QQ 群，一个个的信号相当于 QQ 群里的一个个好友。

信号集是一个能表示多个信号的数据类型，sigset\_t set，set即一个信号集。既然是一个集合，就需要对集合进行添加/删除等操作。

#include <signal.h>

int sigemptyset(sigset\_t \*set); //将set集合置空

int sigfillset(sigset\_t \*set)； //将所有信号加入set集合

int sigaddset(sigset\_t \*set, int signo); //将signo信号加入到set集合

int sigdelset(sigset\_t \*set, int signo); //从set集合中移除signo信号

int sigismember(const sigset\_t \*set, int signo); //判断信号是否存在

除sigismember外，其余操作函数中的set均为传出参数。sigset\_t类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。

示例代码：

int main()

{

sigset\_t set; // 定义一个信号集变量

int ret = 0;

sigemptyset(&set); // 清空信号集的内容

// 判断 SIGINT 是否在信号集 set 里

// 在返回 1， 不在返回 0

ret = sigismember(&set, SIGINT);

if (ret == 0)

{

printf("SIGINT is not a member of set \nret = %d\n", ret);

}

sigaddset(&set, SIGINT); // 把 SIGINT 添加到信号集 set

sigaddset(&set, SIGQUIT);// 把 SIGQUIT 添加到信号集 set

// 判断 SIGINT 是否在信号集 set 里

// 在返回 1， 不在返回 0

ret = sigismember(&set, SIGINT);

if (ret == 1)

{

printf("SIGINT is a member of set \nret = %d\n", ret);

}

sigdelset(&set, SIGQUIT); // 把 SIGQUIT 从信号集 set 移除

// 判断 SIGQUIT 是否在信号集 set 里

// 在返回 1， 不在返回 0

ret = sigismember(&set, SIGQUIT);

if (ret == 0)

{

printf("SIGQUIT is not a member of set \nret = %d\n", ret);

}

return 0;

}

4.5.3.3 sigprocmask

信号阻塞集也称信号屏蔽集、信号掩码。每个进程都有一个阻塞集，创建子进程时子进程将继承父进程的阻塞集。信号阻塞集用来描述哪些信号递送到该进程的时候被阻塞（在信号发生时记住它，直到进程准备好时再将信号通知进程）。

所谓阻塞并不是禁止传送信号, 而是暂缓信号的传送。若将被阻塞的信号从信号阻塞集中删除，且对应的信号在被阻塞时发生了，进程将会收到相应的信号。

我们可以通过 sigprocmask() 修改当前的信号掩码来改变信号的阻塞情况。

#include <signal.h>

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

**功能：**

检查或修改信号阻塞集，根据 how 指定的方法对进程的阻塞集合进行修改，新的信号阻塞集由 set 指定，而原先的信号阻塞集合由 oldset 保存。

**参数：**

**how :** 信号阻塞集合的修改方法，有 3 种情况：

SIG\_BLOCK：向信号阻塞集合中添加 set 信号集，新的信号掩码是set和旧信号掩码的并集。相当于 mask = mask|set。

SIG\_UNBLOCK：从信号阻塞集合中删除 set 信号集，从当前信号掩码中去除 set 中的信号。相当于 mask = mask & ~ set。

SIG\_SETMASK：将信号阻塞集合设为 set 信号集，相当于原来信号阻塞集的内容清空，然后按照 set 中的信号重新设置信号阻塞集。相当于mask = set。

**set :** 要操作的信号集地址。

若 set 为 NULL，则不改变信号阻塞集合，函数只把当前信号阻塞集合保存到 oldset 中。

**oldset :** 保存原先信号阻塞集地址

**返回值**：

成功：0，

失败：-1，失败时错误代码只可能是 EINVAL，表示参数 how 不合法。

4.5.3.4 sigpending

#include <signal.h>

int sigpending(sigset\_t \*set);

功能：读取当前进程的未决信号集

参数：

set：未决信号集

返回值：

成功：0

失败：-1

4.5.3.5 示例代码

int main()

{

// 自定义信号集

sigset\_t myset, old;

sigemptyset(&myset);// 清空　－》　０

// 添加要阻塞的信号

sigaddset(&myset, SIGINT);

sigaddset(&myset, SIGQUIT);

sigaddset(&myset, SIGKILL);

// 自定义信号集设置到内核中的阻塞信号集

sigprocmask(SIG\_BLOCK, &myset, &old);

sigset\_t pend;

int i = 0;

while (1)

{

// 读内核中的未决信号集的状态

sigpending(&pend);

for (int i = 1; i<32; ++i)

{

if (sigismember(&pend, i))

{

printf("1");

}

else if (sigismember(&pend, i) == 0)

{

printf("0");

}

}

printf("\n");

sleep(1);

i++;

// 10s之后解除阻塞

if (i > 10)

{

// sigprocmask(SIG\_UNBLOCK, &myset, NULL);

sigprocmask(SIG\_SETMASK, &old, NULL);

}

}

return 0;

}

### 4.5.4 信号捕捉

4.5.4.1 信号的处理方式

一个进程收到一个信号的时候，可以用如下方法进行处理：

1）执行系统默认动作

对大多数信号来说，系统默认动作是用来终止该进程。

2）忽略此信号(丢弃)

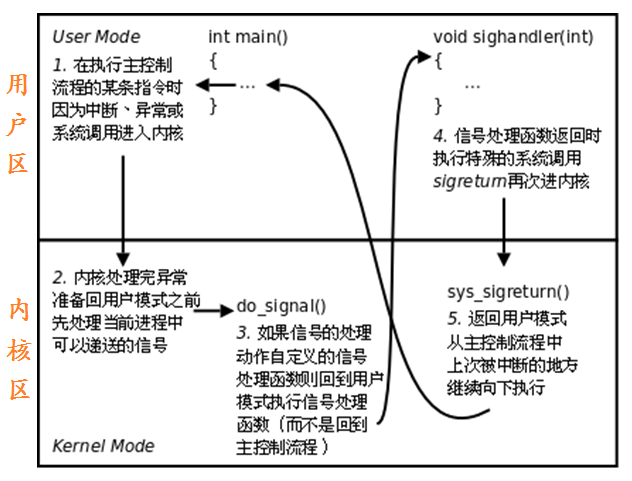
接收到此信号后没有任何动作。

3）执行自定义信号处理函数(捕获)

用用户定义的信号处理函数处理该信号。

注意：SIGKILL 和 SIGSTOP 不能更改信号的处理方式，因为它们向用户提供了一种使进程终止的可靠方法。

内核实现信号捕捉过程：



4.5.4.2 signal

#include <signal.h>

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

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

**功能**：

注册信号处理函数（不可用于 SIGKILL、SIGSTOP 信号），即确定收到信号后处理函数的入口地址。此函数不会阻塞。

**参数**：

signum：信号的编号，这里可以填数字编号，也可以填信号的宏定义，可以通过命令 kill - l("l" 为字母)进行相应查看。

handler : 取值有 3 种情况：

SIG\_IGN：忽略该信号

SIG\_DFL：执行系统默认动作

信号处理函数名：自定义信号处理函数，如：func

回调函数的定义如下：

void func(int signo)

{

// signo 为触发的信号，为 signal() 第一个参数的值

}

**返回值**：

成功：第一次返回 NULL，下一次返回此信号上一次注册的信号处理函数的地址。如果需要使用此返回值，必须在前面先声明此函数指针的类型。

失败：返回 SIG\_ERR

该函数由ANSI定义，由于历史原因在不同版本的Unix和不同版本的Linux中可能有不同的行为。因此应该尽量避免使用它，取而代之使用sigaction函数。

示例代码：

// 信号处理函数

void signal\_handler(int signo)

{

if (signo == SIGINT)

{

printf("recv SIGINT\n");

}

else if (signo == SIGQUIT)

{

printf("recv SIGQUIT\n");

}

}

int main()

{

printf("wait for SIGINT OR SIGQUIT\n");

/\* SIGINT: Ctrl+c ; SIGQUIT: Ctrl+\ \*/

// 信号注册函数

signal(SIGINT, signal\_handler);

signal(SIGQUIT, signal\_handler);

while (1); //不让程序结束

return 0;

}

4.5.4.3 sigaction

1) sigaction函数说明

#include <signal.h>

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

**功能：**

检查或修改指定信号的设置（或同时执行这两种操作）。

**参数：**

signum：要操作的信号。

act： 要设置的对信号的新处理方式（传入参数）。

oldact：原来对信号的处理方式（传出参数）。

如果 act 指针非空，则要改变指定信号的处理方式（设置），如果 oldact 指针非空，则系统将此前指定信号的处理方式存入 oldact。

**返回值：**

成功：0

失败：-1

struct sigaction结构体：

struct sigaction {

void(\*sa\_handler)(int); //旧的信号处理函数指针

void(\*sa\_sigaction)(int, siginfo\_t \*, void \*); //新的信号处理函数指针

sigset\_t sa\_mask; //信号阻塞集

int sa\_flags; //信号处理的方式

void(\*sa\_restorer)(void); //已弃用

};

* sa\_handler、sa\_sigaction：信号处理函数指针，和 signal() 里的函数指针用法一样，应根据情况给 sa\_sigaction、sa\_handler 两者之一赋值，其取值如下：

1. SIG\_IGN：忽略该信号
2. SIG\_DFL：执行系统默认动作
3. 处理函数名：自定义信号处理函数

* sa\_mask：信号阻塞集，在信号处理函数执行过程中，临时屏蔽指定的信号。
* sa\_flags：用于指定信号处理的行为，通常设置为0，表使用默认属性。它可以是一下值的“按位或”组合：
* SA\_RESTART：使被信号打断的系统调用自动重新发起（已经废弃）
* 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。

信号处理函数：

**void(\*sa\_sigaction)(int signum, siginfo\_t \*info, void \*context);**

参数说明：

signum：信号的编号。

info：记录信号发送进程信息的结构体。

context：可以赋给指向 ucontext\_t 类型的一个对象的指针，以引用在传递信号时被中断的接收进程或线程的上下文。

示例代码：

void myfunc(int sig)

{

printf("hello signal: %d\n", sig);

sleep(5);

printf("wake up .....\n");

}

int main()

{

// 注册信号捕捉函数

struct sigaction act;

act.sa\_flags = 0;

act.sa\_handler = myfunc;

// 设置临时屏蔽的信号

sigemptyset(&act.sa\_mask); // 清空

// ctrl + 反斜杠

sigaddset(&act.sa\_mask, SIGQUIT);

sigaction(SIGINT, &act, NULL); //注册信号

while (1);

return 0;

}

2) 信号捕捉特性

1. 进程正常运行时，默认PCB中有一个信号屏蔽字，假定为☆，它决定了进程自动屏蔽哪些信号。

当注册了某个信号捕捉函数，捕捉到该信号以后，要调用该函数。而该函数有可能执行很长时间，在这期间所屏蔽的信号不由☆来指定。而是用sa\_mask来指定。调用完信号处理函数，再恢复为☆。

1. XXX信号捕捉函数执行期间，XXX信号自动被屏蔽。
2. 阻塞的常规信号不支持排队，产生多次只记录一次。（后32个实时信号支持排队）

3) 信号传参

#include <signal.h>

**int sigqueue(pid\_t pid, int sig, const union sigval value);**

**功能：**

给指定进程发送信号。

**参数：**

pid : 进程号。

sig : 信号的编号。

value : 通过信号传递的参数。

union sigval 类型如下：

union sigval

{

int sival\_int;

void \*sival\_ptr;

};

**返回值：**

成功：0

失败：-1

向指定进程发送指定信号的同时，携带数据。但如传地址，需注意，不同进程之间虚拟地址空间各自独立，将当前进程地址传递给另一进程没有实际意义。

下面我们做这么一个例子，一个进程在发送信号，一个进程在接收信号的发送。

发送信号示例代码如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*功能： 发 SIGINT 信号及信号携带的值给指定的进程

\*参数： argv[1]：进程号 argv[2]：待发送的值（默认为100）

\*返回值： 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main()

{

if (argc >= 2)

{

pid\_t pid, pid\_self;

union sigval tmp;

pid = atoi(argv[1]); // 进程号

if (argc >= 3)

{

tmp.sival\_int = atoi(argv[2]);

}

else

{

tmp.sival\_int = 100;

}

// 给进程 pid，发送 SIGINT 信号，并把 tmp 传递过去

sigqueue(pid, SIGINT, tmp);

pid\_self = getpid(); // 进程号

printf("pid = %d, pid\_self = %d\n", pid, pid\_self);

}

return 0;

}

接收信号示例代码如下：

// 信号处理回调函数

void signal\_handler(int signum, siginfo\_t \*info, void \*ptr)

{

printf("signum = %d\n", signum); // 信号编号

printf("info->si\_pid = %d\n", info->si\_pid); // 对方的进程号

printf("info->si\_sigval = %d\n", info->si\_value.sival\_int); // 对方传递过来的信息

}

int main()

{

struct sigaction act, oact;

act.sa\_sigaction = signal\_handler; //指定信号处理回调函数

sigemptyset(&act.sa\_mask); // 阻塞集为空

act.sa\_flags = SA\_SIGINFO; // 指定调用 signal\_handler

// 注册信号 SIGINT

sigaction(SIGINT, &act, &oact);

while (1)

{

printf("pid is %d\n", getpid()); // 进程号

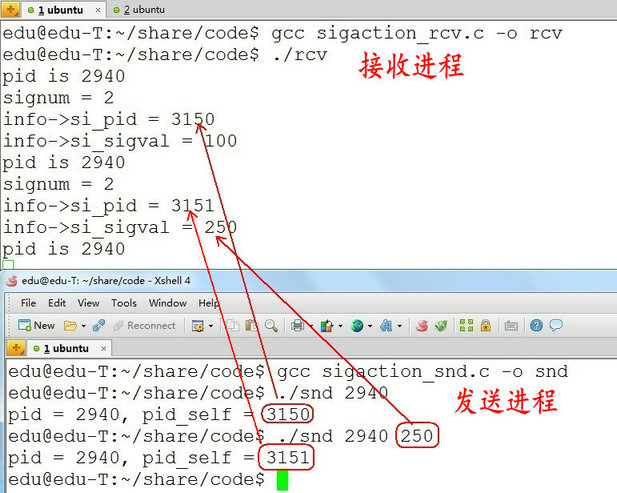
pause(); // 捕获信号，此函数会阻塞

}

return 0;

}

两个终端分别编译代码，一个进程接收，一个进程发送，运行结果如下：



4.5.4.4 不可重入、可重入函数

如果有一个函数不幸被设计成为这样：那么不同任务调用这个函数时可能修改其他任务调用这个函数的数据，从而导致不可预料的后果。这样的函数是不安全的函数，也叫不可重入函数。

满足下列条件的函数多数是不可重入（不安全）的：

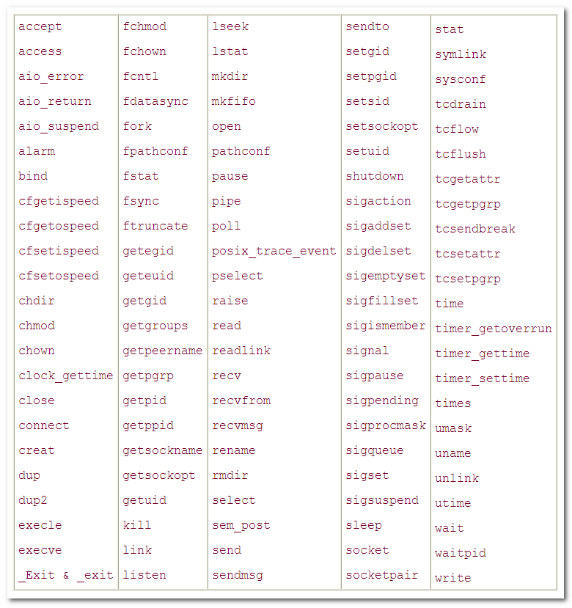
* 函数体内使用了静态的数据结构；
* 函数体内调用了malloc() 或者 free() 函数(谨慎使用堆)；
* 函数体内调用了标准 I/O 函数。

相反，肯定有一个安全的函数，这个安全的函数又叫可重入函数。那么什么是可重入函数呢？所谓可重入是指一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错。

保证函数的可重入性的方法：

* 在写函数时候尽量使用局部变量（例如寄存器、栈中的变量）；
* 对于要使用的全局变量要加以保护（如采取关中断、信号量等互斥方法），这样构成的函数就一定是一个可重入的函数。

Linux常见的可重入函数：



**注意：信号处理函数应该为可重入函数。**

### 4.5.5 SIGCHLD信号

4.5.5.1 SIGCHLD的产生条件

1. 子进程终止时
2. 子进程接收到SIGSTOP信号停止时
3. 子进程处在停止态，接受到SIGCONT后唤醒时

4.5.5.2 如何避免僵尸进程

1. 最简单的方法，父进程通过 wait() 和 waitpid() 等函数等待子进程结束，但是，这会导致父进程挂起。
2. 如果父进程要处理的事情很多，不能够挂起，通过 signal() 函数人为处理信号 SIGCHLD ， 只要有子进程退出自动调用指定好的回调函数，因为子进程结束后， 父进程会收到该信号 SIGCHLD ，可以在其回调函数里调用 wait() 或 waitpid() 回收。

示例代码：

void sig\_child(int signo)

{

pid\_t pid;

//处理僵尸进程, -1 代表等待任意一个子进程, WNOHANG代表不阻塞

while ((pid = waitpid(-1, NULL, WNOHANG)) > 0)

{

printf("child %d terminated.\n", pid);

}

}

int main()

{

pid\_t pid;

// 创建捕捉子进程退出信号

// 只要子进程退出，触发SIGCHLD，自动调用sig\_child()

signal(SIGCHLD, sig\_child);

pid = fork(); // 创建进程

if (pid < 0)

{ // 出错

perror("fork error:");

exit(1);

}

else if (pid == 0)

{ // 子进程

printf("I am child process,pid id %d.I am exiting.\n", getpid());

exit(0);

}

else if (pid > 0)

{ // 父进程

sleep(2); // 保证子进程先运行

printf("I am father, i am exited\n\n");

system("ps -ef | grep defunct"); // 查看有没有僵尸进程

}

return 0;

}

3）如果父进程不关心子进程什么时候结束，那么可以用signal（SIGCHLD, SIG\_IGN）通知内核，自己对子进程的结束不感兴趣，父进程忽略此信号，那么子进程结束后，内核会回收， 并不再给父进程发送信号。

示例代码：

int main()

{

pid\_t pid;

// 忽略子进程退出信号的信号

// 那么子进程结束后，内核会回收， 并不再给父进程发送信号

signal(SIGCHLD, SIG\_IGN);

pid = fork(); // 创建进程

if (pid < 0)

{ // 出错

perror("fork error:");

exit(1);

}

else if (pid == 0)

{ // 子进程

printf("I am child process,pid id %d.I am exiting.\n", getpid());

exit(0);

}

else if (pid > 0)

{ // 父进程

sleep(2); // 保证子进程先运行

printf("I am father, i am exited\n\n");

system("ps -ef | grep defunct"); // 查看有没有僵尸进程

}

return 0;

}

# 5. 守护进程

## 5.1 终端

在UNIX系统中，用户通过终端登录系统后得到一个Shell进程，这个终端成为Shell进程的控制终端（Controlling Terminal），进程中，控制终端是保存在PCB中的信息，而fork会复制PCB中的信息，因此由Shell进程启动的其它进程的控制终端也是这个终端。

默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。

信号中还讲过，在控制终端输入一些特殊的控制键可以给前台进程发信号，例如Ctrl+C表示SIGINT，Ctrl+\表示SIGQUIT。

#include <unistd.h>

**char \*ttyname(int fd);**

**功能：**由文件描述符查出对应的文件名

**参数：**

fd：文件描述符

**返回值：**

成功：终端名

失败：NULL

下面我们借助ttyname函数，通过实验看一下各种不同的终端所对应的设备文件名：

int main()

{

printf("fd 0: %s\n", ttyname(0));

printf("fd 1: %s\n", ttyname(1));

printf("fd 2: %s\n", ttyname(2));

return 0;

}

## 5.2 进程组

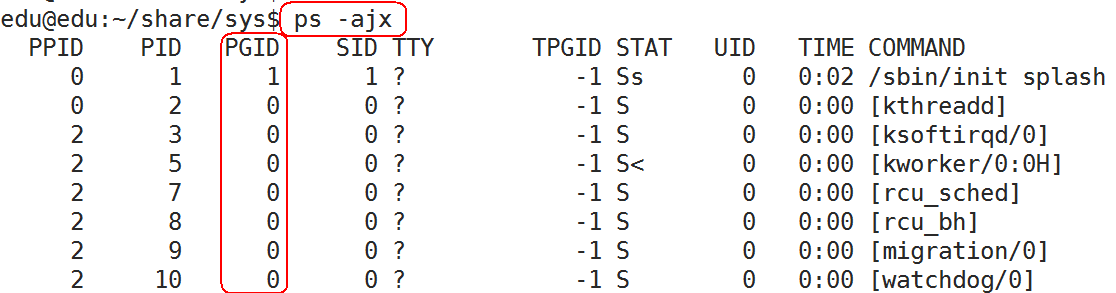
### 5.2.1 概述

进程组，也称之为作业。BSD于1980年前后向Unix中增加的一个新特性。代表一个或多个进程的集合。

每个进程都属于一个进程组。在waitpid函数和kill函数的参数中都曾使用到。操作系统设计的进程组的概念，是为了简化对多个进程的管理。

当父进程，创建子进程的时候，默认子进程与父进程属于同一进程组。进程组ID为第一个进程ID(组长进程)。所以，组长进程标识：其进程组ID为其进程ID

可以使用**kill -SIGKILL -进程组ID**(负的)来将整个进程组内的进程全部杀死：



组长进程可以创建一个进程组，创建该进程组中的进程，然后终止。只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关。

进程组生存期：进程组创建到最后一个进程离开(终止或转移到另一个进程组)。

一个进程可以为自己或子进程设置进程组ID

### 5.2.2 API介绍

#include <unistd.h>

**pid\_t getpgrp(void);**  /\* POSIX.1 version \*/

**功能：**获取当前进程的进程组ID

**参数：**无

**返回值**：总是返回调用者的进程组ID

**pid\_t getpgid(pid\_t pid);**

**功能**：获取指定进程的进程组ID

**参数**：

pid：进程号，如果pid = 0，那么该函数作用和getpgrp一样

**返回值**：

成功：进程组ID

失败：-1

**int setpgid(pid\_t pid, pid\_t pgid);**

**功能**：

改变进程默认所属的进程组。通常可用来加入一个现有的进程组或创建一个新进程组。

**参数**：

将参1对应的进程，加入参2对应的进程组中

**返回值**：

成功：0

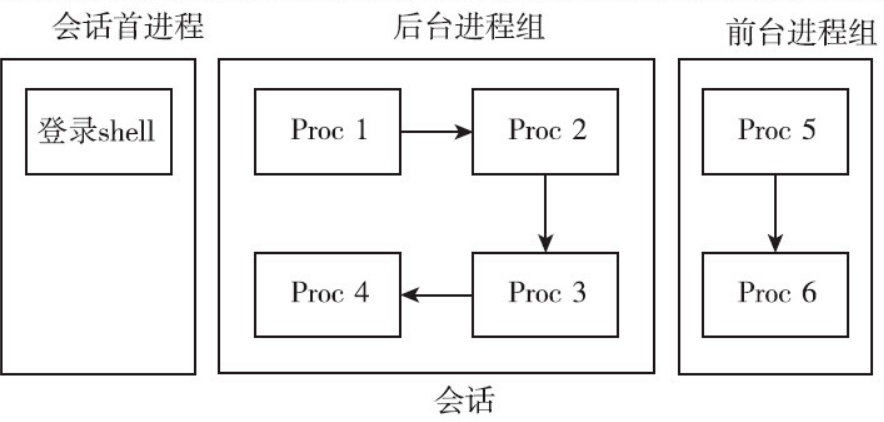
失败：-1

## 5.3 会话

### 5.3.1 概述

会话是一个或多个进程组的集合。

* 一个会话可以有一个控制终端。这通常是终端设备或伪终端设备；
* 建立与控制终端连接的会话首进程被称为控制进程；
* 一个会话中的几个进程组可被分为一个前台进程组以及一个或多个后台进程组；
* 如果一个会话有一个控制终端，则它有一个前台进程组，其它进程组为后台进程组；
* 如果终端接口检测到断开连接，则将挂断信号发送至控制进程（会话首进程）。



### 5.3.2 创建会话注意事项

1. 调用进程不能是进程组组长，该进程变成新会话首进程(session header)
2. 该调用进程是组长进程，则出错返回
3. 该进程成为一个新进程组的组长进程
4. 需有root权限(ubuntu不需要)
5. 新会话丢弃原有的控制终端，该会话没有控制终端
6. 建立新会话时，先调用fork, 父进程终止，子进程调用setsid

### 5.3.3 API介绍

5.3.3.1 getsid

#include <unistd.h>

**pid\_t getsid(pid\_t pid);**

**功能**：获取进程所属的会话ID

**参数**：

pid：进程号，pid为0表示查看当前进程session ID

**返回值**：

成功：返回调用进程的会话ID

失败：-1

组长进程不能成为新会话首进程，新会话首进程必定会成为组长进程。

5.3.3.2 setsid

#include <unistd.h>

**pid\_t setsid(void);**

**功能：**

创建一个会话，并以自己的ID设置进程组ID，同时也是新会话的ID。调用了setsid函数的进程，既是新的会长，也是新的组长。

**参数：**无

**返回值：**

成功：返回调用进程的会话ID

失败：-1

## 5.4 守护进程

### 5.4.1 概述

守护进程（Daemon Process），也就是通常说的 Daemon 进程（精灵进程），是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以d结尾的名字。

守护进程是个特殊的孤儿进程，这种进程脱离终端，为什么要脱离终端呢？之所以脱离于终端是为了避免进程被任何终端所产生的信息所打断，其在执行过程中的信息也不在任何终端上显示。由于在 Linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。

Linux 的大多数服务器就是用守护进程实现的。比如，Internet 服务器 inetd，Web 服务器 httpd 等。

### 5.4.2 创建守护进程模型

1. 创建子进程，父进程退出(必须)

* 所有工作在子进程中进行形式上脱离了控制终端

1. 在子进程中创建新会话(必须)

* setsid()函数
* 使子进程完全独立出来，脱离控制

1. 改变当前目录为根目录(不是必须)

* chdir()函数
* 防止占用可卸载的文件系统
* 也可以换成其它路径

1. 重设文件权限掩码(不是必须)

* umask()函数
* 防止继承的文件创建屏蔽字拒绝某些权限
* 增加守护进程灵活性

1. 关闭文件描述符(不是必须)

* 继承的打开文件不会用到，浪费系统资源，无法卸载

1. 开始执行守护进程核心工作(必须)

* 守护进程退出处理程序模型

### 5.4.3 示例代码

写一个守护进程, 每隔2s获取一次系统时间, 将这个时间写入到磁盘文件：

/\*

\* time\_t rawtime;

\* time ( &rawtime ); --- 获取时间，以秒计，从1970年1月一日起算，存于rawtime

\* localtime ( &rawtime ); //转为当地时间，tm 时间结构

\* asctime() // 转为标准ASCII时间格式：

\*/

void write\_time(int num)

{

time\_t rawtime;

struct tm \* timeinfo;

// 获取时间

time(&rawtime);

#if 0

// 转为本地时间

timeinfo = localtime(&rawtime);

// 转为标准ASCII时间格式

char \*cur = asctime(timeinfo);

#else

char\* cur = ctime(&rawtime);

#endif

// 将得到的时间写入文件中

int fd = open("/home/edu/timelog.txt", O\_RDWR | O\_CREAT | O\_APPEND, 0664);

if (fd == -1)

{

perror("open error");

exit(1);

}

// 写文件

int ret = write(fd, cur, strlen(cur) + 1);

if (ret == -1)

{

perror("write error");

exit(1);

}

// 关闭文件

close(fd);

}

int main(int argc, const char\* argv[])

{

pid\_t pid = fork();

if (pid == -1)

{

perror("fork error");

exit(1);

}

if (pid > 0)

{

// 父进程退出

exit(1);

}

else if (pid == 0)

{

// 子进程

// 提升为会长，同时也是新进程组的组长

setsid();

// 更改进程的执行目录

chdir("/home/edu");

// 更改掩码

umask(0022);

// 关闭文件描述符

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

// 注册信号捕捉函数

//先注册，再定时

struct sigaction sigact;

sigact.sa\_flags = 0;

sigemptyset(&sigact.sa\_mask);

sigact.sa\_handler = write\_time;

sigaction(SIGALRM, &sigact, NULL);

// 设置定时器

struct itimerval act;

// 定时周期

act.it\_interval.tv\_sec = 2;

act.it\_interval.tv\_usec = 0;

// 设置第一次触发定时器时间

act.it\_value.tv\_sec = 2;

act.it\_value.tv\_usec = 0;

// 开始计时

setitimer(ITIMER\_REAL, &act, NULL);

// 防止子进程退出

while (1);

}

return 0;

}

# 6. 线程

## 6.1 概述

### 6.1.1 简介

6.1.1.1 什么是线程

在许多经典的操作系统教科书中，总是把进程定义为程序的执行实例，它并不执行什么, 只是维护应用程序所需的各种资源，而线程则是真正的执行实体。

所以，线程是轻量级的进程（LWP：light weight process），在Linux环境下线程的本质仍是进程。

为了让进程完成一定的工作，进程必须至少包含一个线程。



进程，直观点说，保存在硬盘上的程序运行以后，会在内存空间里形成一个独立的内存体，这个内存体有自己的地址空间，有自己的堆，上级挂靠单位是操作系统。操作系统会以进程为单位，分配系统资源，所以我们也说，进程是资源分配的最小单位。

线程存在与进程当中(进程可以认为是线程的容器)，是操作系统调度执行的最小单位。说通俗点，线程就是干活的。

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

如果说进程是一个资源管家，负责从主人那里要资源的话，那么线程就是干活的苦力。一个管家必须完成一项工作，就需要最少一个苦力，也就是说，一个进程最少包含一个线程，也可以包含多个线程。苦力要干活，就需要依托于管家，所以说一个线程，必须属于某一个进程。

进程有自己的地址空间，线程使用进程的地址空间，也就是说，进程里的资源，线程都是有权访问的，比如说堆啊，栈啊，静态存储区什么的。

6.1.1.2 线程函数列表安装

* 命令：sudo apt-get install manpages-posix-dev

manpages-posix-dev 包含 POSIX 的 header files 和 library calls 的用法

* 查看：man -k pthread

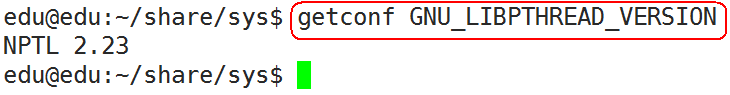
6.1.1.3 NPTL

当 Linux 最初开发时，在内核中并不能真正支持线程。但是它的确可以通过 clone() 系统调用将进程作为可调度的实体。这个调用创建了调用进程（calling process）的一个拷贝，这个拷贝与调用进程共享相同的地址空间。LinuxThreads 项目使用这个调用来完全在用户空间模拟对线程的支持。不幸的是，这种方法有一些缺点，尤其是在信号处理、调度和进程间同步原语方面都存在问题。另外，这个线程模型也不符合 POSIX 的要求。

要改进 LinuxThreads，非常明显我们需要内核的支持，并且需要重写线程库。有两个相互竞争的项目开始来满足这些要求。一个包括 IBM 的开发人员的团队开展了 NGPT（Next-Generation POSIX Threads）项目。同时，Red Hat 的一些开发人员开展了 NPTL 项目。NGPT 在 2003 年中期被放弃了，把这个领域完全留给了 NPTL。

NPTL，或称为 Native POSIX Thread Library，是 Linux 线程的一个新实现，它克服了 LinuxThreads 的缺点，同时也符合 POSIX 的需求。与 LinuxThreads 相比，它在性能和稳定性方面都提供了重大的改进。

查看当前pthread库版本：getconf GNU\_LIBPTHREAD\_VERSION

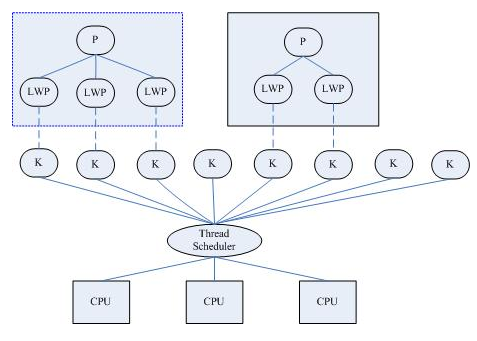


### 6.1.2 线程的特点

类Unix系统中，早期是没有“线程”概念的，80年代才引入，借助进程机制实现出了线程的概念。

因此在这类系统中，进程和线程关系密切：

1. 线程是轻量级进程(light-weight process)，也有PCB，创建线程使用的底层函数和进程一样，都是clone
2. 从内核里看进程和线程是一样的，都有各自不同的PCB.
3. 进程可以蜕变成线程
4. 在linux下，线程最是小的执行单位；进程是最小的分配资源单位



查看指定进程的LWP号：ps -Lf pid

实际上，无论是创建进程的fork，还是创建线程的pthread\_create，底层实现都是调用同一个内核函数 clone 。

* 如果复制对方的地址空间，那么就产出一个“进程”；
* 如果共享对方的地址空间，就产生一个“线程”。

Linux内核是不区分进程和线程的, 只在用户层面上进行区分。所以，线程所有操作函数 pthread\_\* 是库函数，而非系统调用。

### 6.1.3 线程共享资源、非共享资源

6.1.3.1 共享资源

* 1. 文件描述符表
  2. 每种信号的处理方式
  3. 当前工作目录
  4. 用户ID和组ID
  5. 内存地址空间 (.text/.data/.bss/heap/共享库)

6.1.3.2 非共享资源

* 1. 线程id
  2. 处理器现场和栈指针(内核栈)
  3. 独立的栈空间(用户空间栈)
  4. errno变量
  5. 信号屏蔽字
  6. 调度优先级

### 6.1.4 线程优、缺点

优点：

* + 提高程序并发性
  + 开销小
  + 数据通信、共享数据方便

缺点：

* + 库函数，不稳定
  + 调试、编写困难、gdb不支持
  + 对信号支持不好

优点相对突出，缺点均不是硬伤。Linux下由于实现方法导致进程、线程差别不是很大。

## 6.2 线程的常用操作

### 6.2.1 线程号

就像每个进程都有一个进程号一样，每个线程也有一个线程号。进程号在整个系统中是唯一的，但线程号不同，线程号只在它所属的进程环境中有效。

进程号用 pid\_t 数据类型表示，是一个非负整数。线程号则用 pthread\_t 数据类型来表示，Linux 使用无符号长整数表示。

有的系统在实现 pthread\_t 的时候，用一个结构体来表示，所以在可移植的操作系统实现不能把它做为整数处理。

#include <pthread.h>

**pthread\_t pthread\_self(void);**

**功能：**

获取线程号。

**参数：**

无

**返回值：**

调用线程的线程 ID 。

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

**功能：**

判断线程号 t1 和 t2 是否相等。为了方便移植，尽量使用函数来比较线程 ID。

**参数：**

t1，t2：待判断的线程号。

**返回值：**

相等： 非 0

不相等：0

示例代码：

int main()

{

pthread\_t thread\_id;

thread\_id = pthread\_self(); // 返回调用线程的线程ID

printf("Thread ID = %lu \n", thread\_id);

if (0 != pthread\_equal(thread\_id, pthread\_self()))

{

printf("Equal!\n");

}

else

{

printf("Not equal!\n");

}

return 0;

}

线程函数的程序在 pthread 库中，故链接时要加上参数 -lpthread。

### 6.2.2 线程的创建

#include <pthread.h>

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

**const pthread\_attr\_t \*attr,**

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

**void \*arg );**

**功能：**

创建一个线程。

**参数：**

thread：线程标识符地址。

attr：线程属性结构体地址，通常设置为 NULL。

start\_routine：线程函数的入口地址。

arg：传给线程函数的参数。

**返回值：**

成功：0

失败：非 0

在一个线程中调用pthread\_create()创建新的线程后，当前线程从pthread\_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread\_create的函数指针start\_routine决定。

由于pthread\_create的错误码不保存在errno中，因此不能直接用perror()打印错误信息，可以先用strerror()把错误码转换成错误信息再打印。

示例代码：

// 回调函数

void \*thread\_fun(void \* arg)

{

sleep(1);

int num = \*((int \*)arg);

printf("int the new thread: num = %d\n", num);

return NULL;

}

int main()

{

pthread\_t tid;

int test = 100;

// 返回错误号

int ret = pthread\_create(&tid, NULL, thread\_fun, (void \*)&test);

if (ret != 0)

{

printf("error number: %d\n", ret);

// 根据错误号打印错误信息

printf("error information: %s\n", strerror(ret));

}

while (1);

return 0;

}

### 6.2.3 回收线程资源

6.2.3.1 pthread\_join

#include <pthread.h>

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

**功能：**

等待线程结束（此函数会阻塞），并回收线程资源，类似进程的 wait() 函数。如果线程已经结束，那么该函数会立即返回。

**参数：**

thread：被等待的线程号。

retval：用来存储线程退出状态的指针的地址。

**返回值：**

成功：0

失败：非 0

示例代码：

void \*thead(void \*arg)

{

static int num = 123; //静态变量

printf("after 2 seceonds, thread will return\n");

sleep(2);

return &num;

}

int main()

{

pthread\_t tid;

int ret = 0;

void \*value = NULL;

// 创建线程

pthread\_create(&tid, NULL, thead, NULL);

// 等待线程号为 tid 的线程，如果此线程结束就回收其资源

// &value保存线程退出的返回值

pthread\_join(tid, &value);

printf("value = %d\n", \*((int \*)value));

return 0;

}

调用该函数的线程将挂起等待，直到id为thread的线程终止。thread线程以不同的方法终止，通过pthread\_join得到的终止状态是不同的，总结如下：

1. 如果thread线程通过return返回，retval所指向的单元里存放的是thread线程函数的返回值。
2. 如果thread线程被别的线程调用pthread\_cancel异常终止掉，retval所指向的单元里存放的是常数PTHREAD\_CANCELED。
3. 如果thread线程是自己调用pthread\_exit终止的，retval所指向的单元存放的是传给pthread\_exit的参数。

6.2.3.2 pthread\_detach

一般情况下，线程终止后，其终止状态一直保留到其它线程调用pthread\_join获取它的状态为止。但是线程也可以被置为detach状态，这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。

不能对一个已经处于detach状态的线程调用pthread\_join，这样的调用将返回EINVAL错误。也就是说，如果已经对一个线程调用了pthread\_detach就不能再调用pthread\_join了。

#include <pthread.h>

**int pthread\_detach(pthread\_t thread);**

**功能：**

使调用线程与当前进程分离，分离后不代表此线程不依赖与当前进程，线程分离的目的是将线程资源的回收工作交由系统自动来完成，也就是说当被分离的线程结束之后，系统会自动回收它的资源。所以，此函数不会阻塞。

**参数：**

thread：线程号。

**返回值**：

成功：0

失败：非0

### 6.2.4 线程退出

在进程中我们可以调用exit函数或\_exit函数来结束进程，在一个线程中我们可以通过以下三种在不终止整个进程的情况下停止它的控制流。

* 线程从执行函数中返回。
* 线程调用pthread\_exit退出线程。
* 线程可以被同一进程中的其它线程取消。

6.2.4.1 pthread\_exit

#include <pthread.h>

**void pthread\_exit(void \*retval);**

**功能：**

退出调用线程。一个进程中的多个线程是共享该进程的数据段，因此，通常线程退出后所占用的资源并不会释放。

**参数：**

retval：存储线程退出状态的指针。

**返回值：**无

示例代码：

void \*thread(void \*arg)

{

static int num = 123; //静态变量

int i = 0;

while (1)

{

printf("I am runing\n");

sleep(1);

i++;

if (i == 3)

{

pthread\_exit((void \*)&num);

// return &num;

}

}

return NULL;

}

int main(int argc, char \*argv[])

{

int ret = 0;

pthread\_t tid;

void \*value = NULL;

pthread\_create(&tid, NULL, thread, NULL);

pthread\_join(tid, &value);

printf("value = %d\n", \*(int \*)value);

return 0;

}

6.2.4.1 pthread\_cancel

#include <pthread.h>

**int pthread\_cancel(pthread\_t thread);**

**功能：**

杀死(取消)线程

**参数**：

thread : 目标线程ID。

**返回值：**

成功：0

失败：出错编号

注意：线程的取消并不是实时的，而又一定的延时。需要等待线程到达某个取消点(检查点)。

类似于玩游戏存档，必须到达指定的场所(存档点，如：客栈、仓库、城里等)才能存储进度。

杀死线程也不是立刻就能完成，必须要到达取消点。

取消点：是线程检查是否被取消，并按请求进行动作的一个位置。通常是一些系统调用creat，open，pause，close，read，write..... 执行命令man 7 pthreads可以查看具备这些取消点的系统调用列表。

可粗略认为一个系统调用(进入内核)即为一个取消点。

示例代码：

void \*thread\_cancel(void \*arg)

{

while (1)

{

pthread\_testcancel(); //设置取消点

}

return NULL;

}

int main()

{

pthread\_t tid;

pthread\_create(&tid, NULL, thread\_cancel, NULL); //创建线程

sleep(3); //3秒后

pthread\_cancel(tid); //取消tid线程

pthread\_join(tid, NULL);

return 0;

}

## 6.3 线程属性（了解）

### 6.3.1 概述

Linux下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。

如我们对程序的性能提出更高的要求那么需要设置线程属性，比如可以通过设置线程栈的大小来降低内存的使用，增加最大线程个数。

typedef struct

{

int etachstate; //线程的分离状态

int schedpolicy; //线程调度策略

struct sched\_param schedparam; //线程的调度参数

int inheritsched; //线程的继承性

int scope; //线程的作用域

size\_t guardsize; //线程栈末尾的警戒缓冲区大小

int stackaddr\_set; //线程的栈设置

void\* stackaddr; //线程栈的位置

size\_t stacksize; //线程栈的大小

} pthread\_attr\_t;

主要结构体成员：

1. 线程分离状态
2. 线程栈大小（默认平均分配）
3. 线程栈警戒缓冲区大小（位于栈末尾）
4. 线程栈最低地址

属性值不能直接设置，须使用相关函数进行操作，初始化的函数为pthread\_attr\_init，这个函数必须在pthread\_create函数之前调用。之后须用pthread\_attr\_destroy函数来释放资源。

线程属性主要包括如下属性：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parameters）。默认的属性为非绑定、非分离、缺省的堆栈、与父进程同样级别的优先级。

### 6.3.2 线程属性初始化

#include <pthread.h>

**int pthread\_attr\_init(pthread\_attr\_t \*attr);**

**功能**：

初始化线程属性函数，注意：应先初始化线程属性，再pthread\_create创建线程

**参数**：

attr：线程属性结构体

**返回值**：

成功：0

失败：错误号

**int pthread\_attr\_destroy(pthread\_attr\_t \*attr);**

**功能**：

销毁线程属性所占用的资源函数

**参数**：

attr：线程属性结构体

**返回值**：

成功：0

失败：错误号

### 6.3.3 线程的分离状态

线程的分离状态决定一个线程以什么样的方式来终止自己。

* 非分离状态：线程的默认属性是非分离状态，这种情况下，原有的线程等待创建的线程结束。只有当pthread\_join()函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。
* 分离状态：分离线程没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。应该根据自己的需要，选择适当的分离状态。

#include <pthread.h>

**int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);**

**功能**：设置线程分离状态

**参数**：

attr：已初始化的线程属性

detachstate： 分离状态

PTHREAD\_CREATE\_DETACHED（分离线程）

PTHREAD \_CREATE\_JOINABLE（非分离线程）

**返回值**：

成功：0

失败：非0

**int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int \*detachstate);**

**功能**：获取线程分离状态

**参数**：

attr：已初始化的线程属性

detachstate： 分离状态

PTHREAD\_CREATE\_DETACHED（分离线程）

PTHREAD \_CREATE\_JOINABLE（非分离线程）

**返回值**：

成功：0

失败：非0

这里要注意的一点是，如果设置一个线程为分离线程，而这个线程运行又非常快，它很可能在pthread\_create函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用pthread\_create的线程就得到了错误的线程号。

要避免这种情况可以采取一定的同步措施，最简单的方法之一是可以在被创建的线程里调用pthread\_cond\_timedwait函数，让这个线程等待一会儿，留出足够的时间让函数pthread\_create返回。

设置一段等待时间，是在多线程编程里常用的方法。但是注意不要使用诸如wait()之类的函数，它们是使整个进程睡眠，并不能解决线程同步的问题。

### 6.3.4 线程的栈地址

POSIX.1定义了两个常量来检测系统是否支持栈属性：

* \_POSIX\_THREAD\_ATTR\_STACKADDR
* \_POSIX\_THREAD\_ATTR\_STACKSIZE

也可以给sysconf函数传递来进行检测：

* \_SC\_THREAD\_ATTR\_STACKADDR
* \_SC\_THREAD\_ATTR\_STACKSIZE

当进程栈地址空间不够用时，指定新建线程使用由malloc分配的空间作为自己的栈空间。通过pthread\_attr\_setstack和pthread\_attr\_getstack两个函数分别设置和获取线程的栈地址。

#include <pthread.h>

**int pthread\_attr\_setstack(pthread\_attr\_t \*attr, void \*stackaddr, size\_t stacksize);**

**功能：**设置线程的栈地址

**参数**：

attr：指向一个线程属性的指针

stackaddr：内存首地址

stacksize：返回线程的堆栈大小

**返回值**：

成功：0

失败：错误号

**int pthread\_attr\_getstack(const pthread\_attr\_t \*attr, void \*\*stackaddr, size\_t \*stacksize);**

**功能**：获取线程的栈地址

**参数**：

attr：指向一个线程属性的指针

stackaddr：返回获取的栈地址

stacksize：返回获取的栈大小

**返回值**：

成功：0

失败：错误号

### 6.3.5 线程的栈大小

当系统中有很多线程时，可能需要减小每个线程栈的默认大小，防止进程的地址空间不够用,当线程调用的函数会分配很大的局部变量或者函数调用层次很深时，可能需要增大线程栈的默认大小。

#include <pthread.h>

**int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, size\_t stacksize);**

**功能**：设置线程的栈大小

**参数**：

attr：指向一个线程属性的指针

stacksize：线程的堆栈大小

**返回值**：

成功：0

失败：错误号

**int pthread\_attr\_getstacksize(const pthread\_attr\_t \*attr, size\_t \*stacksize);**

**功能**：获取线程的栈大小

**参数**：

attr：指向一个线程属性的指针

stacksize：返回线程的堆栈大小

**返回值**：

成功：0

失败：错误号

### 6.3.6 示例代码

#define SIZE 0x100000

void \*th\_fun(void \*arg)

{

while (1)

{

sleep(1);

}

}

int main()

{

pthread\_t tid;

int err, detachstate, i = 1;

pthread\_attr\_t attr;

size\_t stacksize;

void \*stackaddr;

pthread\_attr\_init(&attr); //线程属性初始化

pthread\_attr\_getstack(&attr, &stackaddr, &stacksize); //获取线程的栈地址

pthread\_attr\_getdetachstate(&attr, &detachstate); //获取线程分离状态

if (detachstate == PTHREAD\_CREATE\_DETACHED)

{

printf("thread detached\n");

}

else if (detachstate == PTHREAD\_CREATE\_JOINABLE)

{

printf("thread join\n");

}

else

{

printf("thread unknown\n");

}

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED); //设置分离状态

while (1)

{

stackaddr = malloc(SIZE);

if (stackaddr == NULL)

{

perror("malloc");

exit(1);

}

stacksize = SIZE;

pthread\_attr\_setstack(&attr, stackaddr, stacksize); //设置线程的栈地址

err = pthread\_create(&tid, &attr, th\_fun, NULL); //创建线程

if (err != 0)

{

printf("%s\n", strerror(err));

exit(1);

}

printf("%d\n", i++);

}

pthread\_attr\_destroy(&attr); //销毁线程属性所占用的资源函数

return 0;

}

## 6.4 线程使用注意事项

1. 主线程退出其他线程不退出，主线程应调用pthread\_exit
2. 避免僵尸线程
   1. pthread\_join
   2. pthread\_detach
   3. pthread\_create指定分离属性

被join线程可能在join函数返回前就释放完自己的所有内存资源，所以不应当返回被回收线程栈中的值;

1. malloc和mmap申请的内存可以被其他线程释放
2. 应避免在多线程模型中调用fork，除非马上exec，子进程中只有调用fork的线程存在，其他线程在子进程中均pthread\_exit
3. 信号的复杂语义很难和多线程共存，应避免在多线程引入信号机制

# 7. 线程的同步与互斥

## 7.1 概述

现代操作系统基本都是多任务操作系统，即同时有大量可调度实体在运行。在多任务操作系统中，同时运行的多个任务可能：

* 都需要访问/使用同一种资源
* 多个任务之间有依赖关系，某个任务的运行依赖于另一个任务

这两种情形是多任务编程中遇到的最基本的问题，也是多任务编程中的核心问题，同步和互斥就是用于解决这两个问题的。

**互斥：**是指散步在不同任务之间的若干程序片断，当某个任务运行其中一个程序片段时，其它任务就不能运行它们之中的任一程序片段，只能等到该任务运行完这个程序片段后才可以运行。最基本的场景就是：一个公共资源同一时刻只能被一个进程或线程使用，多个进程或线程不能同时使用公共资源。

**同步：**是指散步在不同任务之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。最基本的场景就是：两个或两个以上的进程或线程在运行过程中协同步调，按预定的先后次序运行。比如 A 任务的运行依赖于 B 任务产生的数据。

显然，同步是一种更为复杂的互斥，而互斥是一种特殊的同步。也就是说互斥是两个任务之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步也是不能同时运行，但他是必须要安照某种次序来运行相应的线程（也是一种互斥）！因此互斥具有唯一性和排它性，但互斥并不限制任务的运行顺序，即任务是无序的，而同步的任务之间则有顺序关系。

## 7.2 互斥锁mutex

### 7.2.1 为什么需要互斥锁？

在多任务操作系统中，同时运行的多个任务可能都需要使用同一种资源。这个过程有点类似于，公司部门里，我在使用着打印机打印东西的同时（还没有打印完），别人刚好也在此刻使用打印机打印东西，如果不做任何处理的话，打印出来的东西肯定是错乱的。

下面我们用程序模拟一下这个过程，线程一需要打印“ hello ”，线程二需要打印“ world ”，不加任何处理的话，打印出来的内容会错乱：

// 打印机，公共资源

void printer(char \*str)

{

while (\*str != '\0')

{

putchar(\*str);

fflush(stdout);

str++;

sleep(1);

}

printf("\n");

}

// 线程一

void \*thread\_fun\_1(void \*arg)

{

char \*str = "hello";

printer(str); //打印

}

// 线程二

void \*thread\_fun\_2(void \*arg)

{

char \*str = "world";

printer(str); //打印

}

int main()

{

pthread\_t tid1, tid2;

// 创建 2 个线程

pthread\_create(&tid1, NULL, thread\_fun\_1, NULL);

pthread\_create(&tid2, NULL, thread\_fun\_2, NULL);

// 等待线程结束，回收其资源

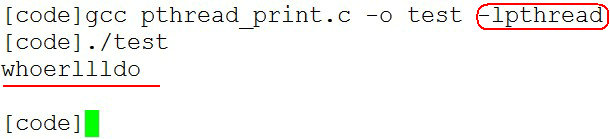
pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

return 0;

}

运行结果如下：



实际上，打印机是有做处理的，我在打印着的时候别人是不允许打印的，只有等我打印结束后别人才允许打印。这个过程有点类似于，把打印机放在一个房间里，给这个房间安把锁，这个锁默认是打开的。当 A 需要打印时，他先过来检查这把锁有没有锁着，没有的话就进去，同时上锁在房间里打印。而在这时，刚好 B 也需要打印，B 同样先检查锁，发现锁是锁住的，他就在门外等着。而当 A 打印结束后，他会开锁出来，这时候 B 才进去上锁打印。

### 7.2.2 互斥锁mutex

而在线程里也有这么一把锁：互斥锁（mutex），也叫互斥量，互斥锁是一种简单的加锁的方法来控制对共享资源的访问，互斥锁只有两种状态,即上锁( lock )和解锁( unlock )。

互斥锁的操作流程如下：

1）在访问共享资源后临界区域前，对互斥锁进行加锁。

2）在访问完成后释放互斥锁导上的锁。

3）对互斥锁进行加锁后，任何其他试图再次对互斥锁加锁的线程将会被阻塞，直到锁被释放。

互斥锁的数据类型是： pthread\_mutex\_t。

### 7.2.3 互斥锁基本操作

7.2.3.1 初始化互斥锁

#include <pthread.h>

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

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

**功能**：

初始化一个互斥锁。

**参数**：

mutex：互斥锁地址。类型是 pthread\_mutex\_t 。

attr：设置互斥量的属性，通常可采用默认属性，即可将 attr 设为 NULL。

可以使用宏 PTHREAD\_MUTEX\_INITIALIZER 静态初始化互斥锁，比如：

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

这种方法等价于使用 NULL 指定的 attr 参数调用 pthread\_mutex\_init() 来完成动态初始化，不同之处在于 PTHREAD\_MUTEX\_INITIALIZER 宏不进行错误检查。

**返回值**：

成功：0，成功申请的锁默认是打开的。

失败：非 0 错误码

restrict关键字：只用于限制指针，告诉编译器，所有修改该指针指向内存中内容的操作，只能通过本指针完成。不能通过除本指针以外的其他变量或指针修改。

7.2.3.2 销毁互斥锁

#include <pthread.h>

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

**功能**：

销毁指定的一个互斥锁。互斥锁在使用完毕后，必须要对互斥锁进行销毁，以释放资源。

**参数**：

mutex：互斥锁地址。

**返回值**：

成功：0

失败：非 0 错误码

7.2.3.3 上锁

#include <pthread.h>

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

**功能：**

对互斥锁上锁，若互斥锁已经上锁，则调用者一直阻塞，直到互斥锁解锁后再上锁。

**参数**：

mutex：互斥锁地址。

**返回值**：

成功：0

失败：非 0 错误码

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

* 调用该函数时，若互斥锁未加锁，则上锁，返回 0；
* 若互斥锁已加锁，则函数直接返回失败，即 EBUSY。

7.2.3.4 解锁

#include <pthread.h>

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

**功能**：

对指定的互斥锁解锁。

**参数**：

mutex：互斥锁地址。

**返回值**：

成功：0

失败：非0错误码

### 7.2.4 示例代码

pthread\_mutex\_t mutex; //互斥锁

// 打印机

void printer(char \*str)

{

pthread\_mutex\_lock(&mutex); //上锁

while (\*str != '\0')

{

putchar(\*str);

fflush(stdout);

str++;

sleep(1);

}

printf("\n");

pthread\_mutex\_unlock(&mutex); //解锁

}

// 线程一

void \*thread\_fun\_1(void \*arg)

{

char \*str = "hello";

printer(str); //打印

}

// 线程二

void \*thread\_fun\_2(void \*arg)

{

char \*str = "world";

printer(str); //打印

}

int main(void)

{

pthread\_t tid1, tid2;

pthread\_mutex\_init(&mutex, NULL); //初始化互斥锁

// 创建 2 个线程

pthread\_create(&tid1, NULL, thread\_fun\_1, NULL);

pthread\_create(&tid2, NULL, thread\_fun\_2, NULL);

// 等待线程结束，回收其资源

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

pthread\_mutex\_destroy(&mutex); //销毁互斥锁

return 0;

}

### 7.2.5 死锁

死锁问题：

* 忘了解锁会产生死锁
* 重复加锁会导致死锁

## 7.3 读写锁

### 7.3.1 概述

当有一个线程已经持有互斥锁时，互斥锁将所有试图进入临界区的线程都阻塞住。但是考虑一种情形，当前持有互斥锁的线程只是要读访问共享资源，而同时有其它几个线程也想读取这个共享资源，但是由于互斥锁的排它性，所有其它线程都无法获取锁，也就无法读访问共享资源了，但是实际上多个线程同时读访问共享资源并不会导致问题。

在对数据的读写操作中，更多的是读操作，写操作较少，例如对数据库数据的读写应用。为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了读写锁来实现。

读写锁的特点如下：

1）如果有其它线程读数据，则允许其它线程执行读操作，但不允许写操作。

2）如果有其它线程写数据，则其它线程都不允许读、写操作。

读写锁分为读锁和写锁，规则如下：

1）如果某线程申请了读锁，其它线程可以再申请读锁，但不能申请写锁。

2）如果某线程申请了写锁，其它线程不能申请读锁，也不能申请写锁。

POSIX 定义的读写锁的数据类型是： pthread\_rwlock\_t。

### 7.3.2 基本操作

7.3.2.1 初始化互斥锁

#include <pthread.h>

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

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

**功能：**

用来初始化 rwlock 所指向的读写锁。

**参数**：

rwlock：指向要初始化的读写锁指针。

attr：读写锁的属性指针。如果 attr 为 NULL 则会使用默认的属性初始化读写锁，否则使用指定的 attr 初始化读写锁。

可以使用宏 PTHREAD\_RWLOCK\_INITIALIZER 静态初始化读写锁，比如：

pthread\_rwlock\_t my\_rwlock = PTHREAD\_RWLOCK\_INITIALIZER;

这种方法等价于使用 NULL 指定的 attr 参数调用 pthread\_rwlock\_init() 来完成动态初始化，不同之处在于PTHREAD\_RWLOCK\_INITIALIZER 宏不进行错误检查。

**返回值**：

成功：0，读写锁的状态将成为已初始化和已解锁。

失败：非 0 错误码。

7.3.2.2 销毁读写锁

#include <pthread.h>

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

**功能**：

用于销毁一个读写锁，并释放所有相关联的资源（所谓的所有指的是由 pthread\_rwlock\_init() 自动申请的资源） 。

**参数**：

rwlock：读写锁指针。

**返回值**：

成功：0

失败：非 0 错误码

7.3.2.3 申请读锁

#include <pthread.h>

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

**功能：**

以阻塞方式在读写锁上获取读锁（读锁定）。

如果没有写者持有该锁，并且没有写者阻塞在该锁上，则调用线程会获取读锁。

如果调用线程未获取读锁，则它将阻塞直到它获取了该锁。一个线程可以在一个读写锁上多次执行读锁定。

线程可以成功调用 pthread\_rwlock\_rdlock() 函数 n 次，但是之后该线程必须调用 pthread\_rwlock\_unlock() 函数 n 次才能解除锁定。

**参数：**

rwlock：读写锁指针。

**返回值：**

成功：0

失败：非 0 错误码

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

* 用于尝试以非阻塞的方式来在读写锁上获取读锁。
* 如果有任何的写者持有该锁或有写者阻塞在该读写锁上，则立即失败返回。

7.3.2.4 申请写锁

#include <pthread.h>

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

**功能**：

在读写锁上获取写锁（写锁定）。

如果没有写者持有该锁，并且没有写者读者持有该锁，则调用线程会获取写锁。

如果调用线程未获取写锁，则它将阻塞直到它获取了该锁。

**参数**：

rwlock：读写锁指针。

**返回值**：

成功：0

失败：非 0 错误码

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

* 用于尝试以非阻塞的方式来在读写锁上获取写锁。
* 如果有任何的读者或写者持有该锁，则立即失败返回。

7.3.2.5 解锁

#include <pthread.h>

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

**功能**：

无论是读锁或写锁，都可以通过此函数解锁。

**参数**：

rwlock：读写锁指针。

**返回值**：

成功：0

失败：非 0 错误码

### 7.3.3 示例代码

下面是一个使用读写锁来实现 4 个线程读写一段数据是实例。

在此示例程序中，共创建了 4 个线程，其中两个线程用来写入数据，两个线程用来读取数据。当某个线程读操作时，其他线程允许读操作，却不允许写操作；当某个线程写操作时，其它线程都不允许读或写操作。

pthread\_rwlock\_t rwlock; //读写锁

int num = 1;

//读操作，其他线程允许读操作，却不允许写操作

void \*fun1(void \*arg)

{

while (1)

{

pthread\_rwlock\_rdlock(&rwlock);

printf("read num first===%d\n", num);

pthread\_rwlock\_unlock(&rwlock);

sleep(1);

}

}

//读操作，其他线程允许读操作，却不允许写操作

void \*fun2(void \*arg)

{

while (1)

{

pthread\_rwlock\_rdlock(&rwlock);

printf("read num second===%d\n", num);

pthread\_rwlock\_unlock(&rwlock);

sleep(2);

}

}

//写操作，其它线程都不允许读或写操作

void \*fun3(void \*arg)

{

while (1)

{

pthread\_rwlock\_wrlock(&rwlock);

num++;

printf("write thread first\n");

pthread\_rwlock\_unlock(&rwlock);

sleep(2);

}

}

//写操作，其它线程都不允许读或写操作

void \*fun4(void \*arg)

{

while (1)

{

pthread\_rwlock\_wrlock(&rwlock);

num++;

printf("write thread second\n");

pthread\_rwlock\_unlock(&rwlock);

sleep(1);

}

}

int main()

{

pthread\_t ptd1, ptd2, ptd3, ptd4;

pthread\_rwlock\_init(&rwlock, NULL);//初始化一个读写锁

//创建线程

pthread\_create(&ptd1, NULL, fun1, NULL);

pthread\_create(&ptd2, NULL, fun2, NULL);

pthread\_create(&ptd3, NULL, fun3, NULL);

pthread\_create(&ptd4, NULL, fun4, NULL);

//等待线程结束，回收其资源

pthread\_join(ptd1, NULL);

pthread\_join(ptd2, NULL);

pthread\_join(ptd3, NULL);

pthread\_join(ptd4, NULL);

pthread\_rwlock\_destroy(&rwlock);//销毁读写锁

return 0;

}

## 7.4 条件变量

### 7.4.1 概述

与互斥锁不同，条件变量是用来等待而不是用来上锁的，条件变量本身不是锁！

条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。

条件变量的两个动作：

* 条件不满, 阻塞线程
* 当条件满足, 通知阻塞的线程开始工作

条件变量的类型: pthread\_cond\_t。

### 7.4.2 基本操作

7.4.2.1 初始化条件变量

#include <pthread.h>

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

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

**功能**：

初始化一个条件变量

**参数**：

cond：指向要初始化的条件变量指针。

attr：条件变量属性，通常为默认值，传NULL即可

也可以使用静态初始化的方法，初始化条件变量：

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

**返回值**：

成功：0

失败：非0错误号

7.4.2.2 销毁条件变量

#include <pthread.h>

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

**功能**：

销毁一个条件变量

**参数**：

cond：指向要初始化的条件变量指针

**返回值**：

成功：0

失败：非0错误号

7.4.2.3 阻塞等待条件变量

#include <pthread.h>

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

**pthread\_mutex\_t \*restrict mutex);**

**功能：**

阻塞等待一个条件变量

a) 阻塞等待条件变量cond（参1）满足

b) 释放已掌握的互斥锁（解锁互斥量）相当于pthread\_mutex\_unlock(&mutex);

a) b) 两步为一个原子操作。

c) 当被唤醒，pthread\_cond\_wait函数返回时，解除阻塞并重新申请获取互斥锁pthread\_mutex\_lock(&mutex);

**参数**：

cond：指向要初始化的条件变量指针

mutex：互斥锁

**返回值**：

成功：0

失败：非0错误号

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

**pthread\_mutex\_t \*restrict mutex,**

**const struct timespec \*restrict abstime);**

**功能**：

限时等待一个条件变量

**参数**：

cond：指向要初始化的条件变量指针

mutex：互斥锁

abstime：绝对时间

**返回值**：

成功：0

失败：非0错误号

abstime补充说明：

struct timespec {

time\_t tv\_sec; /\* seconds \*/ // 秒

long tv\_nsec; /\* nanosecondes\*/ // 纳秒

}

time\_t cur = time(NULL); //获取当前时间。

struct timespec t; //定义timespec 结构体变量t

t.tv\_sec = cur + 1; // 定时1秒

pthread\_cond\_timedwait(&cond, &t);

7.4.2.4 唤醒至阻塞在条件变量上的线程

#include <pthread.h>

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

**功能**：

唤醒至少一个阻塞在条件变量上的线程

**参数**：

cond：指向要初始化的条件变量指针

**返回值**：

成功：0

失败：非0错误号

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

**功能**：

唤醒全部阻塞在条件变量上的线程

**参数**：

cond：指向要初始化的条件变量指针

**返回值**：

成功：0

失败：非0错误号

### 7.4.3 生产者消费者条件变量模型

7.4.3.1 示例代码

线程同步典型的案例即为生产者消费者模型，而借助条件变量来实现这一模型，是比较常见的一种方法。

假定有两个线程，一个模拟生产者行为，一个模拟消费者行为。两个线程同时操作一个共享资源（一般称之为汇聚），生产向其中添加产品，消费者从中消费掉产品。

// 节点结构

typedef struct node

{

int data;

struct node\* next;

}Node;

// 永远指向链表头部的指针

Node\* head = NULL;

// 线程同步 - 互斥锁

pthread\_mutex\_t mutex;

// 阻塞线程 - 条件变量类型的变量

pthread\_cond\_t cond;

// 生产者

void\* producer(void\* arg)

{

while (1)

{

// 创建一个链表的节点

Node\* pnew = (Node\*)malloc(sizeof(Node));

// 节点的初始化

pnew->data = rand() % 1000; // 0-999

// 使用互斥锁保护共享数据

pthread\_mutex\_lock(&mutex);

// 指针域

pnew->next = head;

head = pnew;

printf("====== produce: %lu, %d\n", pthread\_self(), pnew->data);

pthread\_mutex\_unlock(&mutex);

// 通知阻塞的消费者线程，解除阻塞

pthread\_cond\_signal(&cond);

sleep(rand() % 3);

}

return NULL;

}

void\* customer(void\* arg)

{

while (1)

{

pthread\_mutex\_lock(&mutex);

// 判断链表是否为空

if (head == NULL)

{

// 线程阻塞

// 该函数会对互斥锁解锁

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

// 解除阻塞之后，对互斥锁做加锁操作

}

// 链表不为空 - 删掉一个节点 - 删除头结点

Node\* pdel = head;

head = head->next;

printf("------ customer: %lu, %d\n", pthread\_self(), pdel->data);

free(pdel);

pthread\_mutex\_unlock(&mutex);

}

return NULL;

}

int main(int argc, const char\* argv[])

{

pthread\_t p1, p2;

// init

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&cond, NULL);

// 创建生产者线程

pthread\_create(&p1, NULL, producer, NULL);

// 创建消费者线程

pthread\_create(&p2, NULL, customer, NULL);

// 阻塞回收子线程

pthread\_join(p1, NULL);

pthread\_join(p2, NULL);

pthread\_mutex\_destroy(&mutex);

pthread\_cond\_destroy(&cond);

return 0;

}

7.4.3.2 条件变量的优点

相较于mutex而言，条件变量可以减少竞争。

如直接使用mutex，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚（链表）中没有数据，消费者之间竞争互斥锁是无意义的。

有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

## 7.5 信号量

### 7.5.1 概述

信号量广泛用于进程或线程间的同步和互斥，信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。

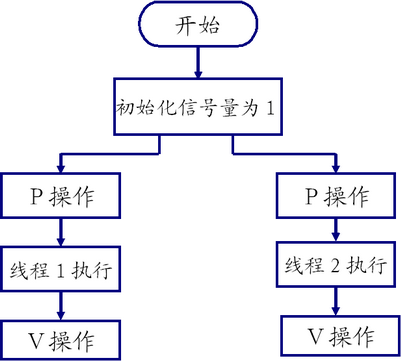
编程时可根据操作信号量值的结果判断是否对公共资源具有访问的权限，当信号量值大于 0 时，则可以访问，否则将阻塞。

PV 原语是对信号量的操作，一次 P 操作使信号量减１，一次 V 操作使信号量加１。

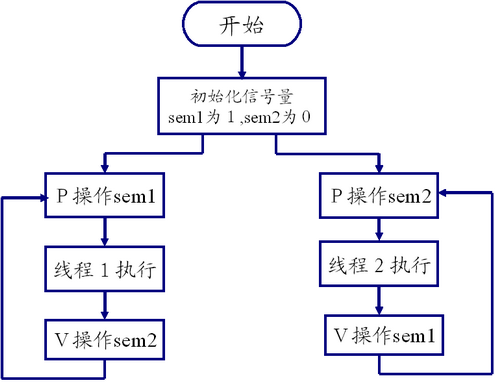
信号量主要用于进程或线程间的同步和互斥这两种典型情况。

信号量数据类型为：sem\_t。

信号量用于互斥：



信号量用于同步：



### 7.5.2 基本操作

7.5.2.1 初始化信号量

#include <semaphore.h>

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

**功能**：

创建一个信号量并初始化它的值。一个无名信号量在被使用前必须先初始化。

**参数**：

sem：信号量的地址。

pshared：等于 0，信号量在线程间共享（常用）；不等于0，信号量在进程间共享。

value：信号量的初始值。

**返回值**：

成功：0

失败： - 1

7.5.2.2 销毁信号量

#include <semaphore.h>

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

**功能**：

删除 sem 标识的信号量。

**参数**：

sem：信号量地址。

**返回值**：

成功：0

失败： - 1

7.5.2.3 信号量 P 操作（减 1）

#include <semaphore.h>

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

**功能**：

将信号量的值减 1。操作前，先检查信号量（sem）的值是否为 0，若信号量为 0，此函数会阻塞，直到信号量大于 0 时才进行减 1 操作。

**参数**：

sem：信号量的地址。

**返回值**：

成功：0

失败： - 1

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

* 以非阻塞的方式来对信号量进行减 1 操作。
* 若操作前，信号量的值等于 0，则对信号量的操作失败，函数立即返回。

**int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);**

* 限时尝试将信号量的值减 1
* abs\_timeout：绝对时间

abs\_timeout补充说明：

struct timespec {

time\_t tv\_sec; /\* seconds \*/ // 秒

long tv\_nsec; /\* nanosecondes\*/ // 纳秒

}

time\_t cur = time(NULL); //获取当前时间。

struct timespec t; //定义timespec 结构体变量t

t.tv\_sec = cur + 1; // 定时1秒

sem\_timedwait(&cond, &t);

7.5.2.4 信号量 V 操作（加 1）

#include <semaphore.h>

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

**功能**：

将信号量的值加 1 并发出信号唤醒等待线程（sem\_wait()）。

**参数**：

sem：信号量的地址。

**返回值**：

成功：0

失败：-1

7.5.2.5 获取信号量的值

#include <semaphore.h>

**int sem\_getvalue(sem\_t \*sem, int \*sval);**

**功能**：

获取 sem 标识的信号量的值，保存在 sval 中。

**参数**：

sem：信号量地址。

sval：保存信号量值的地址。

**返回值**：

成功：0

失败：-1

### 7.5.3 示例代码

sem\_t sem; //信号量

void printer(char \*str)

{

sem\_wait(&sem);//减一

while (\*str)

{

putchar(\*str);

fflush(stdout);

str++;

sleep(1);

}

printf("\n");

sem\_post(&sem);//加一

}

void \*thread\_fun1(void \*arg)

{

char \*str1 = "hello";

printer(str1);

}

void \*thread\_fun2(void \*arg)

{

char \*str2 = "world";

printer(str2);

}

int main(void)

{

pthread\_t tid1, tid2;

sem\_init(&sem, 0, 1); //初始化信号量，初始值为 1

//创建 2 个线程

pthread\_create(&tid1, NULL, thread\_fun1, NULL);

pthread\_create(&tid2, NULL, thread\_fun2, NULL);

//等待线程结束，回收其资源

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

sem\_destroy(&sem); //销毁信号量

return 0;

}