第一章 Linux基本文件IO

对普通计算机用户来说，文件就是存储在永久性存储器上的一段数据流，通常是可执行程序或者是某种格式的数据。文件放置于文件夹，文件夹放置于某个磁盘分区中，这是从普通计算机用户眼里看到的文件。

但linux操作系统中文件的概念，却远远不局限与此，文件是linux对大多数系统资源访问的接口。linux常见的文件类型：普通文件、目录文件、设备文件、管道文件、套接字和链接文件等等。

在linux中所有的进程，在内核中都有一个对应的结构体来描述这个进程task_struct,也叫做进程管理块PCB（process control block），这个结构体中有一个文件描述符表files_struct，用来保存该进程对应的所有文件描述符。

1 linux文件

1. 普通文件：普通计算机用户看到的文件，仅仅是linux文件类型中的一种，我们称之为普通文件，它们通常驻留在磁盘上的某处。普通文件按照信息存储方式来划分，可以分为文本文件和二进制文件：

   文本文件：这类文件以文本的某种编码（比如 ASCII码）形式存储在存储器中，它是以“行”为基础结构的一种信息组织和存储方式。

   二进制文件：这类文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂它们，二进制文件一般是可执行程序、图像、声音等等。

2. 目录文件：主要目的是用于管理和组织系统中的大量文件。它存储一组相关文件的位置、大小等文件有关的信息。

3. 设备文件：linux操作系统把每一个I/O设备都看成一个文件，与普通文件一样处理，这样可以使文件与设备的操作尽可能统一，对I/O设备的使用和一般文件的使用一样不必了解I/O设备的细节。

4. 管道文件：主要用于在进程间传递数据。

5. 套接字文件：用于网络上的通信。

6. 符号链接文件：这个文件包含了另一个文件的路径名。被链接的文件可以是任意文件或目录。

   上述是linux丰富的文件类型，包括除了普通文件和目录文件之外的几种“特殊文件”，正是由于这些特殊文件的存在，linux程序员可以按照统一的接口来实现基本文件读写、设备访问、硬盘读写、网络通信、系统终端，甚至内核状态信息的访问等等。无论是哪种类型的文件，linux都把他们看作是无结构的流式文件，把文件的内容看作是一系列有序的字符流。

2 文件访问

程序要访问一个文件，首先需要通过一个文件路径名来打开文件，当进程打开一个文件的时候，进程将获得一个非负整数标识，即"文件描述符`file description`。通过文件描述符，可以对文件进行I/O处理。

对文件执行I/O操作，有两种基本方式：一种是系统调用的I/O方法，另一种是标准C的文件I/O方法。

系统调用的I/O方法和标准C的I/O方法的区别是：

1、基于标准C的文件操作函数的名字都是以字母“f”开头，而系统调用函数则不用，例如`fopen()`对应于系统调用的`open()`;

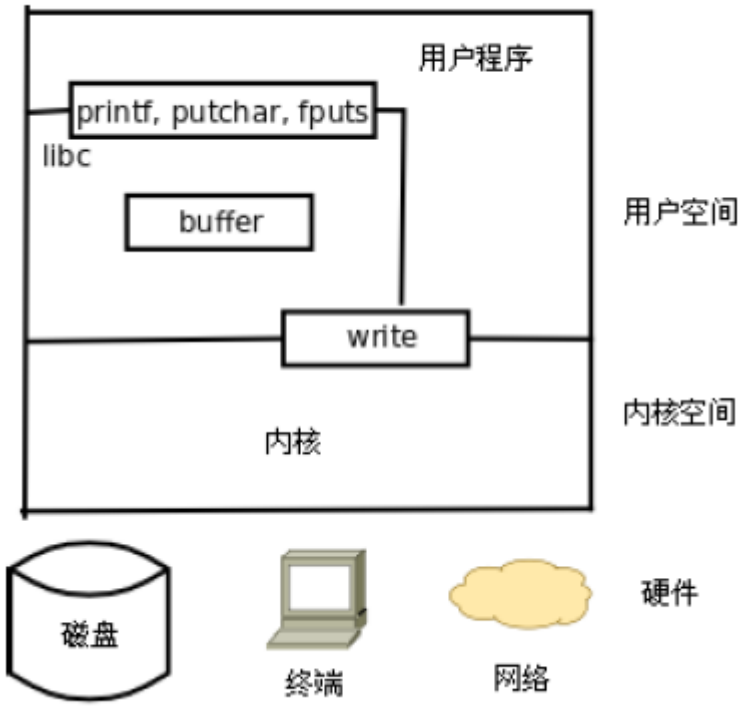
2、系统调用I/O方法是更低一级的接口，通常完成相同的任务是，比使用基于标准c的I/O方法需要更多编码的工作量。

3、系统调用直接处理文件描述符，而标准C函数则处理`FILE*`类型的文件句柄。

4、基于标准C的I/O方法其实就是对系统调用方法的封装，标准C的I/O方法使用自动缓冲技术，使程序能减少系统调用，从而提高程序的性能。

5、基于标准C的I/O方法替用户处理有关系统调用的细节，比如系统调用被信号中断的处理等等。

基于标准C的I/O方法显然给程序员提供了极大的方便，但是有些程序却不能使用基于标准C的I/O方法。比如使用缓冲技术使得网络通信陷入困境，因为它将干扰网络通讯所使用的通信协议。考虑到这两种I/O方法的不同，在使用终端或者通过文件交换信息时，通常采用基于标准C的I/O方法。而使用网络或者管道通信时，通常采用系统调用的I/O方法。



3 文件访问的系统调用API

linux最常用的文件操作系统调用包括：打开、创建文件的open()和creat(), 关闭文件close(), 读取文件read(),写入文件write(),移动文件指针lseek(),文件控制fcntl()和文件权限access()等。

3.1 文件的创建、打开和关闭

通过`open()`和`creat()`系统调用，都可以创建一个并打开一个文件，系统调用`open()`和`creat()`成功时，都会返回一个非负数的文件描述符，使用`close()`函数可以关闭指定的文件描述符的文件。

在使用任何与文件相关的系统调用之前，程序应该包含fcntl.h和unistd.h头文件，它们为最普遍的文件例程提供了函数原型和常数。

- open()函数原型

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

当open()调用成功后，它会返回一个新的独一无二的文件描述符。

open()函数必须指定打开文件的flags标志，其中必须指定标志O_RDONLY、O_WRONLY和O_RDWR中的一个，其他标志都是可选的，可以将其于前面的三种标志之一进行或 运算以生成最终的标志。

flag	功能
O_RDONLY	以只读方式打开文件
O_WRONLY	以只读方式打开文件
O_RDWR	以读写方式打开文件
O_APPEND	以追加模式打开文件，在每次写入操作指向之前，自动将文件指针定位到文件末尾，并不是只在打开文件的时候定位到文件末尾，但在网络文件系统进行操作时不一定有用。
O_DIRECTORY	假如参数path那么不是一个目录，那么open将失败。
O_CREAT	如果文件不存在就创建，使用此选项时需要提供第三个参数mode，文件的访问权限。
O_EXCL	如果使用了这个标志，则使用O_CREAT标志来打开一个文件时，如果文件已经存在，open将返回失败，但在网络文件系统进行操作时不一定有用。
O_NOFOLLOW	强制参数pathname所指的文件不能是符号链接。
O_NONBLOCK	打开文件后，对这个文件描述符的所有的操作都以非阻塞方式进行。
O_NDELAY	和O_NONBLOCK完全一样。
O_SYNC	当把数据写入到这个文件描述符时，强制立即输出到物理设备。
O_TRUNC	如果打开的文件是一个已经存在的普通文件，并且指定了可写标志(O_WRONLY、O_RDWR),那么在打开时就消除原文件的所有内容。但打开的文件是一个FIFO或者终端设备时，这个标志将不起作用。

在目录`/usr/include/asm-generic/fcntl.h`可以查看到。

- 例如：

源文件：open_test.c

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

int main(int argc, char** argv)
{
    int fd;
    fd = open("/tmp/open_test", O_CREAT|O_WRONLY|O_TRUNC, 0640);
    if(fd == -1)
    {
        printf("open failed fd = %d\n", fd);
    }
    else
    {
        printf("/tmp/open_test created, fd = [%d]\n", fd);
    }
    return 0;
}
```

代码中open()函数传入了3个参数:

- 第一个参数是一个字符串参数，创建或者打开文件的路径。
- 第二个参数指明了open操作需要创建一个新文件。
- 第三个参数指明了新建文件的访问权限。

  整数类型的变量fd记录了open()函数的返回值。如果fd等于-1，那么表示open()函数返回失败，否则fd记录了系统返回的所新建并打开的文件描述符。

  在程序退出之前，使用close()来关闭文件。

- 思考问题

一个进程默认打开3个文件描述符

```
STDIN_FILENO 0   标准输入文件描述符

STDOUT_FILENO 1  标准输出文件描述符

STDERR_FILENO 2  标准错误文件描述符
```

新打开文件返回文件描述符表中未使用的最小文件描述符。

- 使用creat()函数来实现

  creat()函数原型

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

  使用creat替换掉open函数

```
fd = creat("/tmp/open_test", 0640);
```

- 使用close函数来关闭文件

  close()函数原型

```
int close(int fd);
```

close()系统调用关闭并释放一个文件描述符。close()成功时返回值等于0，错误时返回-1。但是程序一般不需要检查close系统调用的返回值。除非发生严重的程序错误。当进程退出的时候文件描述符会自动关闭。

- 错误值

open()和creat()系统调用成功时，都会返回一个新的文件描述符，当返回失败时，将返回-1。通过errno以及使用perror(), 以及strerror()可以查看错误信息。

```
void perror(const char *s);
char *strerror(int errnum);
```

可以在`/usr/include/asm-generic/errno.h`以及`/usr/include/asm-generic/errno-base.h`中去查看errno的具体定义值。

open()和creat()常见错误信息

错误值	返回值含义
EACCESS	访问请求不允许(权限不够)，在参数pathname中有目录不允许搜索，或者文件不存在且对上层的写操作又不允许。
EEXIST	使用了O_CREAT和O_EXCL标志，但文件已经存在。
EFAULT	文中在一个不能访问的地址空间。
EISDIR	参数pathname是一个目录，而又涉及到写操作。
ELOOP	在分解pathname时，遇到太多符号链接或者指明O_NOFOLLOW但是pathname是一个符号链接。
EMFILE	程序打开的文件数已经达到最大值了。
ENAMETOOLONG	文件名超长。
ENOENT	文件不存在
ENFILE	打开的总文件数已经达到上限了。
ENOSPC	文件将要被创建，但是设备存储没有空间了。
ENOTDIR	参数pathname不是一个目录。

- 刷新缓冲区

  由于linux内核对物理存储可能会有写延时，所以就算成功的关闭了一个文件，也不能保证数据都被成功写入物理存储。当文件关闭时，对文件系统来说一般不去刷新缓冲区。

  如果你要保证数据写入磁盘等物理存储设备中就使用fsync()。fsync()函数原型是：

```
int fsync(int fd);
```

- 文件umask

  用touch命令创建一个文件时，创建权限是0666，而touch进程继承了Shell进程的umask掩码，所以最终的文件权限是0666&022=0644。

  ```
  touch file123
  ls -l file123
  -rw-rw-r-- 1 where where 0 9月11 23:48 file123
  ```

  同样道理，用gcc编译生成一个可执行文件时，创建权限是0777，而最终的文件权限是0777 & 022 = 0755。

- 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2048, 命令设置的方式不能超过10000，并且只是临时生效，重启失效。

```
ulimit -n 2048
```

- 写文件/etc/security/limits.conf，永久生效, 内容如下：

```
*       soft nofile 65535
*       hard nofile 65535
```

其中，“*”表示所有用户都生效，重启后，在任何地方执行ulimit -n就会显示65535。

3.2 文件读

文件打开后，我们可以使用read()来进行文件的读操作。这个系统调用的函数原型是：

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

三个参数分别是：

`fd`: 要进行读写操作的文件描述符。

`buf`:要写入文件内容或读出文件内容的内存地址。

`count`:要读写的字节数。

read()是从文件描述符fd所引用的文件中读取count字节到buf缓冲区中。

如果read()成功读取了数据，就返回所读取的字节数目，否则返回-1。如果read()读到了文件的结尾或者被一个信号所中断，返回值会小于count。当文件指针已经为于文件结尾，read()操作将返回0。

- 用read()函数从文件读取数据

  源文件：read_test.c

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

int main(int argc, char** argv)
{
    int fd_read;
    char buf[100];
    int ret;

  /*判断是否传入文件名*/
    if(argc < 2)
    {
        printf("Usage read_test FILENAME\n");
        return -1;
    }
    fd_read = open(argv[1], O_RDONLY);
    if(fd_read == -1)
    {
        perror("open error");
        return -1;
    }

    /*读取数据*/
    ret = read(fd_read, buf, sizeof(buf));
    printf("read return = [%d]\n", ret);
    buf[ret]='\0';
    /*如果读出来数据，则打印数据*/
    if(ret >= 0)
    {
        printf("========= buf =========\n");
        printf("%s\n", buf);
        printf("======================\n");
    }
    else
    {
        perror("read error");
    }
    close(fd_read);
    return 0;

}
```

当read()返回-1的时候，errno以及使用perror()可以查看读文件的错误信息。

我们经常需要检查的是EINTR错误，产生这个错误是由于read()系统调用在读取任何数据前被信号所中断。发生这个错误的时候，文件指针并没有移动，我们需要重新读取数据。

- 错误值

错误值	含义
EAGAIN	使用O_NONBLOCK标志 指定了非阻塞输入输出，但当前没有数据可读。
EBADF	fd不是一个合法的文件描述符，或者不是为了读操作而打开。
EINTR	在读取到数据以前调用被信号中断。
EINVAL	fd所指向的对象不合适读，或者是文件打开时指定了O_DIRECT标志。
EISDIR	fd指向一个目录。

read封装，对中断进行处理

```
ssize_t readn(int fd,void *usrbuf,size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while(nleft > 0)
    {
        if((nread = read(fd,bufp,nleft)) == -1)
        {
            if(errno == EINTR) {/*中断继续 */  continue; }
            else { return -1; }/*error*/
        }
        else if(nread == 0)  { break;  /*读到文件末尾EOF*/  }
        else
        {/*读内容成功*/
            nleft -= nread;
            bufp += nread;
        }
    }
    return (n - nleft);
}
```

3.3 文件写

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

write()从buf中写count字节到文件描述符fd所引用的文件中，成功时返回实际所写的字节数。

在实际的写入过程中，可能会出现写入的字节数少于count。这时返回的是实际写入的字节数，所以调用write()后都必须检查返回值是否与要写入的相同，如果不同就要采取相应的设施。

- 使用write()往文件里面写数据

源代码：write_test.c

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int fd;
    char buf[] = "helloworld";
    int bytes_write;
    if(argc < 2)
    {
        printf("Usage: write_test FILEWRITE\n");
        return -1;
    }
    fd = open(argv[1], O_CREAT|O_WRONLY|O_TRUNC, 0640);
    if(fd == -1)
    {
        perror("create error");
        return -2;
    }
    int bytes_write = write(fd, buf, sizeof(buf));
    printf("bytes_write: %d \n", bytes_write);

    return 0;
}
```

- 错误值

错误值	含义
EBADF	fd不是一个合法的文件描述符，或者不是为写 操作而打开。
EINTR	系统调用在写入任何数据之前调用被信号所中断。
EINVAL	fd所有指向的对象不合适写，或者是文件打开时指定了O_DIRECT标志。
ENOSPC	fd指向的文件所在的设备无可用空间。
EPIPE	fd连接到一个管道，或者套接字的读方向一端已经关闭。

write封装，对中断返回进行了处理

```
ssize_t writen(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nwrite;
    char *bufp = usrbuf;

    while(nleft > 0)
    {
        if((nwrite = write(fd,bufp,nleft)) == -1)
        {
            if(errno == EINTR) {/*中断继续 */  continue; }
            else { return -1; }/*error*/
        }
        else
        {/*写内容成功*/
            nleft -= nwrite;
            bufp += nwrite;
        }
    }
    return (n - nleft);
}
```

3.4 文件的读写位置

每一个被打开的文件，都有一个文件指针表明当前的存取位置。一般文件被新建或者打开的时候，文件指针都位于文件头，除非在打开的时候指定了O_APPEND标志。文件的读写操作都是从文件指针位置开始的，每次文件的读取和写入，文件指针都会根据读写的字节数向后移动，直到文件结尾。

如果需要从文件的随机位置读写数据，那么就需要先移动文件指针。lseek()系统调用可以使文件指针移动到文件中的指定位置。下面是lseek()的函数原型。

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

lseek()的三个参数分别是:

- fd: 文件描述符
- offset：移动的偏移量，单位为字节数
- whence: 文件指针移动偏移量的解释，有三个选项，如下表：

宏名称	含义
SEEK_SET	从文件头开始计算，文件指针移动到offset个字节位置。
SEEK_CUR	从文件指针当前位置开始计算，向后移动offset个字节的位置。
SEEK_END	文件指针移动到文件结尾

lseek()移动文件指针成功时，将返回文件指针的当前位置。失败时返回-1。

下面的代码，通过文件指针移动到文件结尾，lseek()将返回文件指针的位置，这个指针偏移量的就是文件大小。

源代码 lssek_test.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int fd_lseek;
    char buf[100];
    int file_size;
    /*判断是否传入文件名*/
    if(argc < 2)
    {
        printf("Usage: lseek_test FILENAME\n");
        return -1;
    }
    /*打开要读取的文件*/
    fd_lseek = open(argv[1], O_RDONLY);
    if( fd_lseek == -1)
    {
        perror("open error:");
        return -1;
    }
    /*移动指针到文件尾*/
    file_size = lseek(fd_lseek, 0, SEEK_END);
    if(file_size >= 0)
    {
        printf("size of [%s] = %d\n", argv[1], file_size);
    }
    else
    {
        perror("lseek error");
    }
    close(fd_lseek);
    return 0;
}
```

创建4g空洞文件源码lseek_creat.c

```
#ifndef _FILE_OFFSET_BITS
#define _FILE_OFFSET_BITS 64
#endif
#include<stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    if(argc < 2)
    {
        printf("Usage: %s [file]\n", argv[0]);
        return -1;
    }
    int fd = open(argv[1], O_WRONLY | O_CREAT, 0755);
    if(fd ==  -1)
    {
        perror("open");
        return -1;
    }
    off_t ret = lseek(fd, 0xFFFFFFFF - 1, SEEK_SET);
    if(ret == -1)
    {
        perror("lseek");
        return -1;
    }
    write(fd, "\0", 1);
    close(fd);

    return 0;
}
```

如果想要在32位系统上操作大于2g的文件，你需要在包含任何头文件前添加宏#define _FILE_OFFSET_BITS 64。

3.5 文件的访问权限

linux有严格的文件权限控制，当我们需要在程序中判断一个文件时否具有读、写等权限的时候，可以使用access()系统调用。

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

参数pathname时要判断的文件路径名。参数mode可以是以下值或者是他们的组合：

- R_OK: 判断文件是否有读权限
- W_OK: 判断文件是否有写权限
- X_OK: 判断文件是否有可执行权限
- F_OK:判断文件是否存在

当access()系统调用对文件的测试成功时返回0，只要有其中一个条件不符，则返回-1。下面是一个简单的例子，检查一个文件是否存在并具有可执行权限并输出检查结果。

源代码：access_test.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    if(argc < 2)
    {
        printf("Usage : access [filename]\n");
        exit(-1);
    }
    if (access(argv[1], F_OK | R_OK | W_OK | X_OK) == 0) {
        printf("access all\n");
    } else {
        perror("access");
    }
    return 0;
}
```

3.6 修改文件属性

当文件被打开之后，进程会获取一个文件描述符，文件描述符包含了文件描述符标志以及当前进程对文件的访问权限等信息状态标志。当我们需要获取或者修改文件描述符中包含的标志时，可以使用fcntl()系统调用。

其函数原型为：

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

fcntl()是一个参数fd是文件描述符，第二个参数指定了函数的操作，fcntl()函数常用的功能有:

cmd	功能
F_GETFL	获取文件描述符对应的文件标志，文件标志为open()时的那些标志。
F_SETFL	设置文件描述符对应的文件标志，可以更改的标志有:O_APPEND、O_ASYNC、O_DIRECT、O_NOATIME和O_NONBLOCK，通过第三个参数arg来传递标志。

下面的函数是一个修改文件标志的例子，函数set_fl可以对文件描述符的文件标志进行打开或者关闭操作。

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
void set_fl(int fd, int flags, int on)
{
    int fl;
    if(fl = fcntl(fd, F_GETFL, 0) == -1)
    {
        perror("fcntl");
    }
    if( on )
    {
        /*打开标志*/
        fl |= flags;
    }
    else
    {
        /*关闭标志*/
        fl &= ~flags;
    }
    if(fcntl(fd, F_SETFL, fl) == -1)
    {
        perror("fcntl");
    }
}

int main(int argc, char** argv)
{
    int fd;
    fd = creat("./fcntl_test", 0640);
    if(fd == -1)
    {
        perror("create fcntl_test error:");
    }
    printf("create fcntl_test, fd = [%d]\n", fd);
    set_fl(fd, O_APPEND, 1);
    close(fd);
    return 0;
}
```

4 阻塞与非阻塞

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。

下面这个小程序从终端读数据再写回终端。

4.1 阻塞读终端

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    char buf[10];
    int n;
    n = read(STDIN_FILENO, buf, 9);
    if (n < 0)
    {
        perror("read STDIN_FILENO");
        exit(1);
    }
    buf[n] = '\0';
    write(STDOUT_FILENO, buf, n + 1);
    return 0;
}
```

4.2 非阻塞终端

```c
#include <errno.h>
#include<stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    char buf[10];
    int ret;
    int flags = fcntl(STDIN_FILENO, F_GETFL);
    if(flags == -1)
    {
        perror("fcntl");
        return -1;
    }
    flags |= O_NONBLOCK;
    if(fcntl(STDIN_FILENO, F_SETFL, flags) == -1)
    {
        perror("fcntl");
        return -1;
    }
AGAIN:
    ret = read(STDIN_FILENO, buf, sizeof(buf));
    if(ret == -1)
    {
        if(errno == EAGAIN)
        {
            printf("#####read again#####\n");
            sleep(1);
            goto AGAIN;
        }
        perror("read");
        return -1;
    }
    ret = write(STDOUT_FILENO, buf, ret);
    if(ret == -1)
    {
        perror("write");
        return -1;
    }
    return 0;
}
```

第二章 Linux 文件系统

文件系统的功能包括：管理和调度文件的存储空间，提供文件的逻辑结构、物理结构和存储方法，实现文件从标识到实际地址的映射，实现文件的控制操作和存取操作，实现文件信息的共享并提供可靠的文件保密和保护措施，提供文件的安全措施。

1 文件系统格式

一般Windows常用的分区格式有三种，分别是FAT16、FAT32、NTFS格式。

而在Linux操作系统里常见的主要有Ext2、Ext3、Ext4、Linuxswap和VFAT几种格式:

首先说说Ext2：Ext2是GNU/Linux系统中标准的文件系统。

这个可以说是Linux系统中使用最多的一种文件系统，它是专门为Linux设计的，拥有极快的速度和极小CPU占用率。Ext2既可以用于标准的块设备(如硬盘)，也被应用在软盘等移动存储设备上。

关于Ext3和Ext4：是Ext2的升级版本，也就是说它在保有Ext2的格式之下进行了扩展，比如最大存储文件大小得到提高。

关于Linuxswap:是Linux中一种专门用于交换分区的swap文件系统。Linux系统主要是使用这一整个分区作为交换空间。

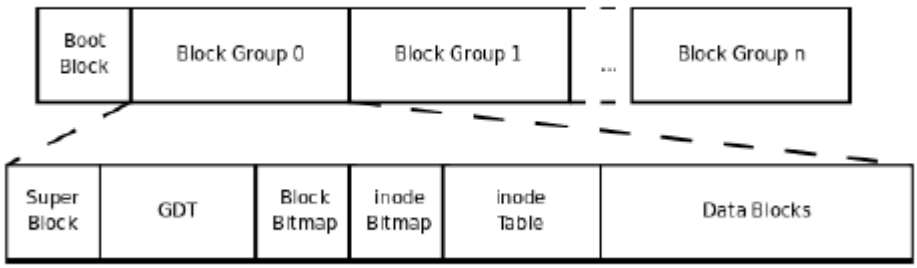
一般情况下，这个swap格式的交换分区是主内存的2倍。在内存不够时，Linux会将部分数据写到交换分区上。

VFAT也叫长文件名系统，这是一个与Windows系统兼容的Linux文件系统，支持长文件名，可以作为Windows与Linux交换文件的分区。

2 ext2

我们知道，一个磁盘可以划分成多个分区，每个分区必须先用格式化工具（例如mkfs命令）格式化成某种格式的文件系统，然后才能存储文件，格式化的过程会在磁盘上写一些管理存储布局的信息。

下图是一个磁盘分区格式化成ext2文件系统后的存储布局。文件系统中存储的最小单位是块（Block），一个块究竟多大是在格式化时确定的，例如mke2fs的-b选项可以设定块大小为1024、2048或4096字节。而上图中启动块（BootBlock）的大小是确定的，就是1KB，启动块是由PC标准规定的，用来存储磁盘分区信息和启动信息，任何文件系统都不能使启动块。启动块之后才是 ext2文件系统的开始。一块磁盘只有一个BootBlock。



启动块之后才是ext2文件系统的开始，ext2文件系统将整个分区划成若干个同样大小的块组（Block Group），每个块组都由以下部分组成。

超级块（Super Block） 描述整个分区的文件系统信息，例如块大小、文件系统版本号、上次mount的时间等等。超级块在每个块组的开头都有一份拷贝。

块组描述符表（GDT，Group Descriptor Table） 由很多块组描述符组成，整个分区分成多少个块组就对应有多少个块组描述符。每个块组描述符（Group Descriptor）存储一个块组的描述信息，例如在这个块组中从哪里开始是inode表，从哪里开始是数据块，空闲的inode和数据块还有多少个等等。和超级块类似，块组描述符表在每个块组的开头也都有一份拷贝，这些信息是非常重要的，一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。通常内核只用到第0个块组中的拷贝，当执行e2fsck检查文件系统一致性时，第0个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第0个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。

块位图（Block Bitmap） 一个块组中的块是这样利用的：数据块存储所有文件的数据，比如某个分区的块大小是1024字节，某个文件是2049字节，那么就需要三个数据块来存，即使第三个块只存了一个字节也需要占用一个整块。超级块、块组描述符表、块位图、inode位图、inode表这几部分存储该块组的描述信息。那么如何知道哪些块已经用来存储文件数据或其它描述信息，哪些块仍然空闲可用呢？块位图就是用来描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个bit代表本块组中的一个块，这个bit为1表示该块已用，这个bit为0表示该块空闲可用。

为什么用df命令统计整个磁盘的已用空间非常快呢？因为只需要查看每个块组的块位图即可，而不需要搜遍整个分区。相反，用du命令查看一个较大目录的已用空间就非常慢，因为不可避免地要搜遍整个目录的所有文件。

与此相联系的另一个问题是：在格式化一个分区时究竟会划出多少个块组呢？主要的限制在于块位图本身必须只占一个块。用mke2fs格式化时默认块大小是1024字节，可以用-b参数指定块大小，现在设块大小指定为b字节，那么一个块可以有8b个bit，这样大小的一个块位图就可以表示8b个块的占用情况，因此一个块组最多可以有8b个块，如果整个分区有s个块，那么就可以有s/(8b)个块组。格式化时可以用-g参数指定一个块组有多少个块，但是通常不需要手动指定，mke2fs工具会计算出最优的数值。

inode位图（inode Bitmap） 和块位图类似，本身占一个块，其中每个bit表示一个inode是否空闲可用。

inode表（inode Table） 我们知道，一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类型（常规、目录、符号链接等），权限，文件大小，创建/修改/访问时间等，也就是stat命令看到的那些信息，这些信息存在inode中而不是数据块中。每个文件都有一个inode，一个块组中的所有inode组成了inode表。

inode表占多少个块在格式化时就要决定并写入块组描述符中，mke2fs格式化工具的默认策略是一个块组有多少个8KB就分配多少个inode。由于数据块占了整个块组的绝大部分，也可以近似认为数据块有多少个8KB就分配多少个inode，换句话说，如果平均每个文件的大小是8KB，当分区存满的时候inode表会得到比较充分的利用，数据块也不浪费。如果这个分区存的都是很大的文件（比如电影），则数据块用完的时候inode会有一些浪费，如果这个分区存的都是很小的文件（比如源代码），则有可能数据块还没用完inode就已经用完了，数据块可能有很大的浪费。如果用户在格式化时能够对这个分区以后要存储的文件大小做一个预测，也可以用mke2fs的-i参数手动指定每多少个字节分配一个inode。

数据块（Data Block） 根据不同的文件类型有以下几种情况

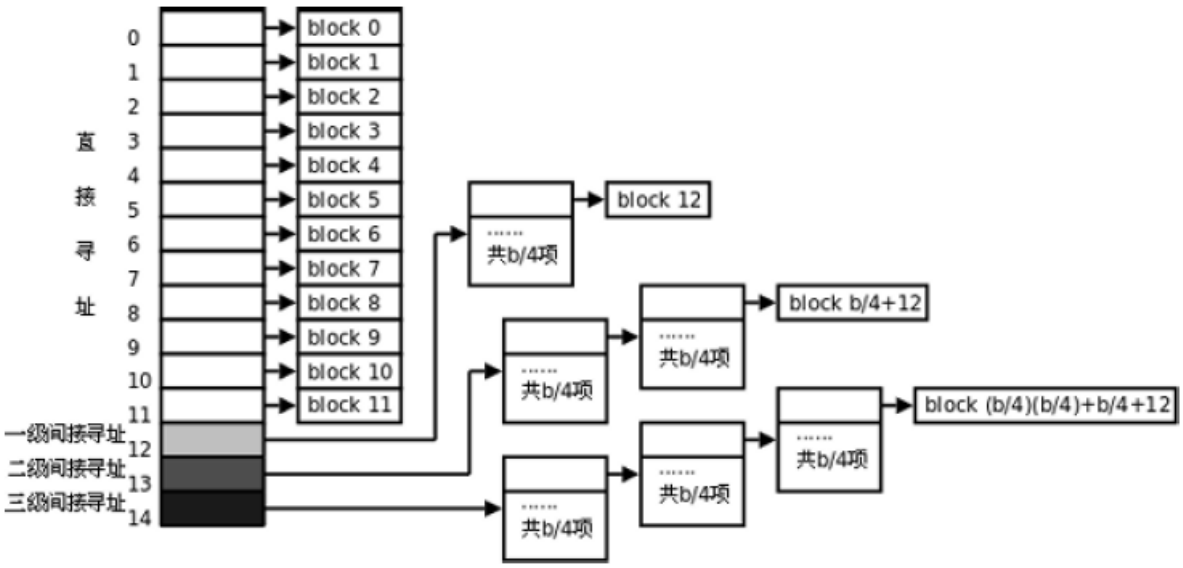
对于常规文件，文件的数据存储在数据块中。

对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外，ls -l命令看到的其它信息都保存在该文件的inode中。注意这个概念：目录也是一种文件，是一种特殊类型的文件，目录中保存记录项，记录项中记录对应文件的名字、inode、记录项长度、文件类型。

对于符号链接，如果目标路径名较短则直接保存在inode中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。

设备文件、FIFO等特殊文件没有数据块，设备文件的主设备号和次设备号保存在inode中。

- 数据块寻址:



从上图可以看出，索引项Blocks[13]指向两级的间接寻址块，最多可表示(b/4)[^2]+b/4+12个数据块，对于1K的块大小最大可表示64.26MB的文件。索引项Blocks[14]指向三级的间接寻址块，最多可表示(b/4)[^3]+(b/4)[^2]+b/4+12个数据块，对于1K的块大小最大可表示16.06GB的文件。

可见，这种寻址方式对于访问不超过12个数据块的小文件是非常快的，访问文件中的任意数据只需要两次读盘操作，一次读inode（也就是读索引项）一次读数据块。而访问大文件中的数据则需要最多五次读盘操作：inode、一级间接寻址块、二级间接寻址块、三级间接寻址块、数据块。实际上，磁盘中的inode和数据块往往经被内核缓存了，读大文件的效率也不会太低。

3 获取文件属性

3.1 stat/lstat

struct stat这个结构体是用来描述一个linux系统文件系统中的文件属性的结构。

可以有两种方法来获取一个文件的属性：

- 通过路径：

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *struct_stat);
int lstat(const char *path,struct stat *struct_stat);
```

如果是链接文件需要使用lstat。

两个函数的第一个参数都是文件的路径，第二个参数是struct stat的指针。返回值为0，表示成功执行。

执行失败是，error被自动设置为下面的值：

```
EBADF:  文件描述词无效
EFAULT:  地址空间不可访问
ELOOP:  遍历路径时遇到太多的符号连接
ENAMETOOLONG:文件路径名太长
ENOENT:路径名的部分组件不存在，或路径名是空字串
ENOMEM:内存不足
ENOTDIR:路径名的部分组件不是目录
```

这两个方法区别在于stat没有处理字符链接(软链接）的能力，如果一个文件是符号链接，stat会直接返回它所指向的文件的属性；而lstat返回的就是这个符号链接的内容。这里需要说明一下的是软链接和硬链接的含义。我们知道目录在linux中也是一个文件，文件的内容就是这这个目录下面所有文件与inode的对应关系。那么所谓的硬链接就是在某一个目录下面将一个文件名与一个inode关联起来，其实就是添加一条记录！而软链接也叫符号链接更加简单了，这个文件的内容就是一个字符串，这个字符串就是它所链接的文件的绝对或者相对地址。

- 通过文件描述符

```
int fstat(int fdp, struct stat *struct_stat);    //通过文件描述符获取文件对应的属性。fdp为文件描述符
```

- stat结构的结构

```
struct stat {
        mode_t     st_mode;       //文件对应的模式，文件，目录等
        ino_t      st_ino;       //inode节点号
        dev_t      st_dev;        //设备号码
        dev_t      st_rdev;       //特殊设备号码
        nlink_t    st_nlink;      //文件的硬链接数连接数
        uid_t      st_uid;        //文件所有者
        gid_t      st_gid;        //文件所有者对应的组
        off_t      st_size;       //普通文件，对应的文件字节数
        time_t     st_atime;      //文件最后被访问的时间
        time_t     st_mtime;      //文件内容最后被修改的时间
        time_t     st_ctime;      //文件属性inode改变时间
        blksize_t st_blksize;    //文件内容对应的块大小
        blkcnt_t   st_blocks;     //文件内容对应的块数量
      };
```

stat结构体中的`st_mode`则定义了下列数种情况：

```
S_IFMT  0170000     文件类型的位遮罩
S_IFLNK 0120000     符号连接
S_IFREG 0100000     一般文件
S_IFBLK 0060000     区块装置
S_IFDIR 0040000     目录
S_IFCHR 0020000     字符装置
S_IFIFO 0010000     先进先出
S_ISUID 04000     文件的(set user-id on execution)位
S_ISGID 02000     文件的(set group-id on execution)位
S_ISVTX 01000     文件的sticky位
S_IRUSR 00400     文件所有者具可读取权限
S_IWUSR 00200     文件所有者具可写入权限
S_IXUSR 00100     文件所有者具可执行权限
S_IRGRP 00040             用户组具可读取权限
S_IWGRP 00020             用户组具可写入权限
S_IXGRP 00010             用户组具可执行权限
S_IROTH 00004             其他用户具可读取权限
S_IWOTH 00002             其他用户具可写入权限
S_IXOTH 00001             其他用户具可执行权限
```

上述的文件类型在POSIX中定义了检查这些类型的宏定义：

```
例如:
S_ISLNK (st_mode)    判断是否为符号连接
S_ISREG (st_mode)    是否为一般文件
S_ISDIR (st_mode)    是否为目录
S_ISCHR (st_mode)    是否为字符设备文件
S_ISSOCK (st_mode)   是否为socket
```

源码: stat_test1.c

```c
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    struct stat stat_buf;
    if(argc < 2)
    {
        printf("Usage: %s [filename]", argv[0]);
        return -1;
    }
    lstat(argv[1], &stat_buf);
    if(S_ISLNK(stat_buf.st_mode))
        printf("%s is link file\n", argv[1]);
  /*
    也可以这么写
    if(S_IFLNK == (stat_buf.st_mode & S_IFMT))
        printf("%s is link file\n", argv[1]);
  */
    if(S_ISREG (stat_buf.st_mode))
        printf("%s is regular file\n", argv[1]);
    if(S_ISDIR (stat_buf.st_mode))
        printf("%s is directory file\n", argv[1]);
    return 0;
}
```

源码： stat_test2.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
int main(int argc, char** argv)
{
    struct stat buf;
    if(argc < 2)
    {
        printf("Usage: stat [filename]\n");
        return -1;
    }
    if(stat(argv[1], &buf) == -1)
    {
        perror("stat");
        return -1;
    }
    printf("access time %s\n", ctime(&buf.st_atime));
    printf("modify time %s\n", ctime(&buf.st_mtime));
    printf("change inode time %s\n", ctime(&buf.st_ctime));

    return 0;
```

```
#include <time.h>
char *ctime(const time_t *timep);   //将时间转换成字符串
```

3.2 粘住位S_ISVTX

粘住位S_ISVTX，如果在一个执行文件没有设置该位，则执行该文件，且进程结束后，系统会把该进程的正文部分放置磁盘的交换区中，在交换区中文件是连续存放的，不像非交换区一样，一个文件的内容分散在磁盘的几个块中。如果不想被交换出去，那么可以设置粘住位。

现今的系统扩展了该位的使用范围，比如说针对目录设置该位，则只有对该用户具有写权限的用户在满足以下条件时才能删除或更名该目录下的文件：

（1）拥有此文件；

（2）拥有此目录；

（3）超级用户权限

在linux中，此扩展功能的应用在/tmp目录下有所体现，我们可以在该目录下任意创建自己的文件，但就是不能删除或更名其他用户的文件。

3.3 SUID/SGID位

SUID位的功能简单的说就是让组用户或其他用户在执行该文件时拥有文件所有者(own)权限。

SGID位的功能简单的说就是让组用户或其他用户在执行该文件时拥有文件所有组(group)权限

由于SUID和SGID是在执行程序（程序的可执行位被设置）时起作用，而可执行位只对普通文件和目录文件有意义，所以设置其他种类文件的SUID和SGID位是没有多大意义的。

如果一个文件被设置了SUID或SGID位，会分别表现在所有者或同组用户的权限的可执行位上。例如：

```
1、-rwsr-xr-x 表示SUID和所有者权限中可执行位被设置
2、-rwSr--r-- 表示SUID被设置，但所有者权限中可执行位没有被设置
3、-rwxr-sr-x 表示SGID和同组用户权限中可执行位被设置
4、-rw-r-Sr-- 表示SGID被设置，但同组用户权限中可执行位没有被设置
```

当一个程序设置了为SUID位时，内核就知道了运行这个程序的时候，应该认为是文件的所有者在运行这个程序。即该程序运行的时候，有效用户ID是该程序的所有者。举个例子：

```
where@ubuntu:~$ ls -l /etc/passwd
-rw-r--r-- 1 root root 2254  8月 19 15:53 /etc/passwd
where@ubuntu:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 45420  1月 27  2016 /usr/bin/passwd
```

虽然你以where登陆系统，但是当你输入passwd命令来更改密码的时候，由于 passwd设置了SUID位，因此虽然进程的实际用户ID是where对应的ID，但是进程的有效用户ID则是passwd文件的所有者root的ID, 因此可以修改/etc/passwd文件。

4 改变文件属性

4.1 chmod更改文件权限

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

```
参数 mode 有下列数种组合：
1、S_ISUID 04000 文件的 (set user-id on execution)位
2、S_ISGID 02000 文件的 (set group-id on execution)位
3、S_ISVTX 01000 文件的sticky 位
4、S_IRUSR 00400 文件所有者具可读取权限
5、S_IWUSR 00200 文件所有者具可写入权限
6、S_IXUSR 00100 文件所有者具可执行权限
7、S_IRGRP 00040 用户组具可读取权限
8、S_IWGRP 00020 用户组具可写入权限
9、S_IXGRP 00010 用户组具可执行权限
10、S_IROTH 00004 其他用户具可读取权限
11、S_IWOTH 00002 其他用户具可写入权限
12、S_IXOTH 00001 其他用户具可执行权限
```

返回值：权限改变成功返回0, 失败返回-1, 错误原因存于errno.

源码：chmod.c

```c
#include<stdio.h>
#include <sys/stat.h>
int main(int argc, char** argv)
{
    if(argc < 3)
    {
        printf("Usage  %s [mode] [filepath]\n", argv[0]);
        return -1;
    }
    int mode;
    sscanf(argv[1], "%o", &mode);
    chmod(argv[2], mode);
    return 0;
}
```

4.2 chown更改所属用户

```c
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
/*lchown是改变符号链接本身的所有者，而不是该符号链接所指向的文件。*/
```

源码chown.c

```c
#include<stdio.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    int uid, gid;
    if(argc < 4)
    {
        printf("usage: %s [uid] [gid] [file]\n", argv[0]);
        return -1;
    }
    sscanf(argv[1], "%d", &uid);
    sscanf(argv[2], "%d", &gid);
    chown(argv[3], uid, gid);
    perror("chown");
    return 0;
}
```

4.3 utime修改文件的访问修改时间

```
#include <sys/types.h>
#include <utime.h>
int utime(const char * filename,struct utimbuf * buf);
```

函数说明： utime()用来修改参数filename文件所属的inode访问修改时间。
结构utimbuf定义如下

```
struct utimbuf{
    time_t actime;
    time_t modtime;
};
```

返回值：

如果参数buf为空指针(NULL)，则该文件的存取时间和更改时间全部会设为目前时间。执行成功则返回0，失败返回-1，错误代码存于errno。错误代码 EACCESS 存取文件时被拒绝，权限不足ENOENT 指定的文件不存在。

源码：utime_test.c

```
#include<stdio.h>
#include <utime.h>
#include <time.h>
int main(int argc, char** argv)
{
    struct utimbuf time_buf;
    time_t new_time = time(NULL);
    if(argc < 2)
    {
        printf("usage: %s \n", argv[0]);
    }
    time_buf.actime = new_time;
    time_buf.modtime = new_time;
    utime(argv[1], &time_buf);
    perror("utime");
    return 0;
}
```

4.4 truncate 截断文件

函数说明： truncate()会将参数path 指定的文件大小改为参数length 指定的大小。如果原来的文件大小比参数length大，则超过的部分会被删去。

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

返回值

执行成功则返回0，失败返回-1，错误原因存于errno。错误代码:

```
EACCESS #参数path所指定的文件无法存取。
EROFS   #欲写入的文件存在于只读文件系统内
EFAULT  #参数path指针超出可存取内存空间
EINVAL  #参数path包含不合法字符
ENAMETOOLONG #参数path太长
ENOTDIR #参数path路径并非一目录
EISDIR  #参数path 指向一目录
ETXTBUSY #参数path所指的文件为共享程序，而且正被执行中
```

4.5 link文件连接

4.5.1 link

创建一个硬链接，当rm删除文件时，只是删除了目录下的记录项和把inode硬链接计数减1，当硬链接计数减为0时，才会真正的删除文件。

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

- 硬链接通常要求位于同一文件系统中,POSIX允许跨文件系统
- 符号链接没有文件系统限制
- 通常不允许创建目录的硬链接，某些unix系统下超级用户可以创建目录的硬链
- 创建目录项以及增加硬链接计数应当是一个原子操作

4.5.2 symlink 建立文件符号连接

```
#include<unistd.h>
int symlink( const char * oldpath,const char * newpath);
```

函数说明：　symlink()以参数`newpath`指定的名称来建立一个新的连接(符号连接)到参数`oldpath`所指定的已存在文件。参数`oldpath`指定的文件不一定要存在，如果参数`newpath`指定的名称为一已存在的文件则不会建立连接。

返回值:

成功则返回0，失败返回-1，错误原因存于errno。

```
EPERM #参数oldpath与newpath所指的文件系统不支持符号连接
EROFS #欲测试写入权限的文件存在于只读文件系统内
EFAULT #参数oldpath或newpath指针超出可存取内存空间。
ENAMETOOLONG #参数oldpath或newpath太长
ENOMEM #核心内存不足
EEXIST #参数newpath所指的文件名已存在。
EMLINK #参数oldpath所指的文件已达到最大连接数目
ELOOP #参数pathname有过多符号连接问题
ENOSPC #文件系统的剩余空间不足
EIO I/O #存取错误
```

4.5.3 readlink取得符号连接所指的文件

```
#include<unistd.h>
int readlink(const char * path ,char * buf, size_t bufsiz);
```

函数说明： readlink()会将参数path的符号连接内容存到参数buf所指的内存空间，返回的内容不是以NULL作字符串结尾，但会将字符串的字符数返回。若参数bufsiz小于符号连接的内容长度，过长的内容会被截断。

返回值：

执行成功则传符号连接所指的文件路径字符串，失败则返回-1，错误代码存于errno。

```
EACCESS #取文件时被拒绝，权限不够
EINVAL #参数bufsiz 为负数
EIO I/O #存取错误。
ELOOP #欲打开的文件有过多符号连接问题。
ENAMETOOLONG #参数path的路径名称太长
ENOENT #参数path所指定的文件不存在
ENOMEM #核心内存不足
ENOTDIR #参数path路径中的目录存在但却非真正的目录。
```

4.5.4 unlink删除文件

```
#include<unistd.h>
int unlink(const char * pathname);
```

函数说明：unlink()会删除参数pathname指定的文件。

1. 如果是符号链接，删除符号链接
2. 如果是硬链接，硬链接数减1，当减为0时，释放数据块和inode
3. 如果文件硬链接数为0，但有进程已打开该文件，并持有文件描述符，则等该进程关闭该文件时，kernel才真正去删除该文件

返回值 ：成功则返回0，失败返回-1，错误原因存于errno

```
EROFS #文件存在于只读文件系统内
EFAULT #参数pathname指针超出可存取内存空间
ENAMETOOLONG #参数pathname太长
ENOMEM #核心内存不足
ELOOP #参数pathname 有过多符号连接问题
EIO I/O #存取错误
```

4.5.5 rename 更改文件名称或位置

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

函数说明： rename()会将参数oldpath 所指定的文件名称改为参数newpath所指的文件名称。若newpath所指定的文件已存在，则会被删除。
返回值：执行成功则返回0，失败返回-1，错误原因存于errno

4.5.6 getcwd获取当前的工作目录

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

4.5.7 chdir改变当前的工作目录

```
#include<unistd.h>
int chdir(const char* path);
int fchdir(int fd);
```

函数说明： chidr、fchdir用来将当前的工作目录改变成以参数path指定的路径、或者fd 所指的文件描述词。
返回值： 执行成功则返回0，失败返回-1，errno为错误代码。

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include <stdio.h>
int main()
{
    int fd;
    fd = open("/home/where", O_RDONLY);
    fchdir(fd);
    printf("current working directory : %s \n", getcwd(NULL, NULL));
    close(fd);
}
```

5.文件夹操作

5.1 mkdir 创建文件夹

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

递归创建文件夹

```
#include<stdio.h>
#include<unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
int main(int argc, char** argv)
{
    if(argc < 2)
    {
        printf("usage: %s [dir]\n", argv[0]);
        return -1;
    }
    int ret;
    char *buf, *p;
    buf = (char*)malloc(strlen(argv[1]) + 1);
    p = strtok(argv[1], "/");
    strcpy(buf, p);
    while(1)
    {
        printf("%s\n", buf);
        ret = mkdir(buf, 0755);
        if(ret == -1)
        {
            if(EEXIST != errno)
            {
                perror("mkdir");
                return -1;
            }
        }
        p = strtok(NULL, "/");
        if(p == NULL)
            break;
        strcat(buf, "/");
        strcat(buf, p);
    }
    return 0;

}
```

5.2 rmdir 删除文件夹

```
#include <unistd.h>
int rmdir(const char *pathname);
```

5.3 opendir打开文件夹

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

函数说明： opendir()用来打开参数name指定的目录，并返回DIR*形态的目录流，和open()类似，接下来对目录的读取和搜索都要使用此返回值。
返回值： 成功则返回DIR* 型态的目录流，打开失败则返回NULL。

```
EACCESS #权限不足
EMFILE #已达到进程可同时打开的文件数上限。
ENFILE #已达到系统可同时打开的文件数上限。
ENOTDIR #参数name非真正的目录
ENOENT #参数name 指定的目录不存在，或是参数name 为一空字符串。
ENOMEM #核心内存不足。
```

5.4 readdir读文件夹

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
struct dirent {
    ino_t d_ino; /* 此目录进入点的inode */
    off_t d_off; /* 目录文件开头至此目录进入点的位移 */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* 所指的文件类型*/
    char d_name[256]; /* 文件名 */
};
```

函数说明： readdir()返回参数dir目录流的下个目录进入点。

返回值： 成功则返回下个目录进入点。有错误发生或读取到目录文件尾则返回NULL。EBADF参数dir为无效的目录流。

```
DT_BLK      #This is a block device.
DT_CHR      #This is a character device.
DT_DIR      #This is a directory.
DT_FIFO     #This is a named pipe (FIFO).
DT_LNK      #This is a symbolic link.
DT_REG      #This is a regular file.
DT_SOCK     #This is a UNIX domain socket.
DT_UNKNOWN  #The file type could not be determined
```

```
#include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
#include <stdio.h>
int main()
{
    DIR *dir;
    struct dirent *ptr;
    int i;
    dir = opendir("/home/");
    while((ptr = readdir(dir))! = NULL)
    {
        printf("d_name: %s\n", ptr->d_name);
    }
    closedir(dir);
}
```

输出如下：

```
d_name: .
d_name: ..
d_name: xing
d_name: where
```

5.5 rewinddir 重置读取目录的位置

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirp);
```

函数说明： rewinddir()用来设置参数dir 目录流目前的读取位置为原来开头的读取位置。
错误码： EBADF参数dir为无效的目录流。

源码：rewinddir_test.c

```c
#include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
#include <stdio.h>
int main(void)
{
    DIR * dir;
    struct dirent *ptr;
    dir = opendir("/home/");
    while((ptr = readdir(dir))!=NULL)
    {
        printf("d_name :%s\n",ptr->d_name);
    }
    /*重新遍历一次*/
    rewinddir(dir);

    printf("readdir again!\n");
    while((ptr = readdir(dir))!=NULL)
    {
        printf("d_name: %s\n",ptr->d_name);
    }
    closedir(dir);
}
```

运行结果：

```
d_name :.
d_name :..
d_name :xing
d_name :where
readdir again!
d_name: .
d_name: ..
d_name: xing
d_name: where
```

5.6 telldir、seekdir

```c
#include <dirent.h>
long telldir(DIR *dirp);
void seekdir(DIR *dirp, long offset);
```

函数说明： seekdir()用来设置参数dir目录流目前的读取位置，在调用readdir()时便从此新位置开始读取。参数offset 代表距离目录文件开头的偏移量。

telldir()返回参数dir目录流目前的读取位置。此返回值代表距离目录文件开头的偏移量返回值返回下个读取位置，有错误发生时返回-1。

源码：tellseek_test.c

```
#include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
#include <stdio.h>
int main(int argc, char**argv)
{
    DIR * dir;
    struct dirent * ptr;
    long offset,i = 0;
    if(argc < 2)
    {
        printf("Usage %s [file] \n", argv[0]);
        return -1;
    }
    dir = opendir(argv[1]);

    while((ptr = readdir(dir)) != NULL)
    {
      /*保存第二个记录项的偏移*/
        if(++i == 2)
            offset = telldir(dir);
        printf("d_name :%s\n", ptr->d_name);
    }

    /*重置目录指针到第二个记录项*/
    seekdir(dir, offset);
    printf("Readdir again!\n");
    while((ptr = readdir(dir))!=NULL)
    {
        printf("d_name :%s\n", ptr->d_name);
    }
    closedir(dir);
}
```

5.7 closedir

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

5.8 递归遍历目录

递归列出目录中的文件列表

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_PATH 1024
void fsize(char *name);
void Perror(char * title)
{
    perror(title);
    exit(-1);
}
void dirwalk(char *dir)
{
    char name[MAX_PATH];
    struct dirent *dp;
    DIR *dfd;
    if ((dfd = opendir(dir)) == NULL) { Perror("opendir"); }

    while ((dp = readdir(dfd)) != NULL)
    {
        if (strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
            continue;
        /*计算子目录路径字符串长度是否超长了*/
        if (strlen(dir)+strlen(dp->d_name)+2 > sizeof(name))
            printf("name %s %s too long\n",dir, dp->d_name);
        else
        {
            sprintf(name, "%s/%s", dir, dp->d_name);
            fsize(name);
        }
    }
    closedir(dfd);
}
void fsize(char *name)
{
    struct stat stbuf;
    if (lstat(name, &stbuf) == -1) { Perror("stat"); }
    if (S_ISDIR(stbuf.st_mode))
        dirwalk(name);
    printf("%8ld %s\n", stbuf.st_size, name);
}

int main(int argc, char **argv)
{
    if(argc < 2)
    {
        printf("usage %s [file]\n", argv[0]);
        return -1;
    }

    if(argv[1][strlen(argv[1]) - 1] == '/')
```

```
        argv[1][strlen(argv[1]) - 1] = '\0';
    fsize(argv[1]);
    return 0;
}
```

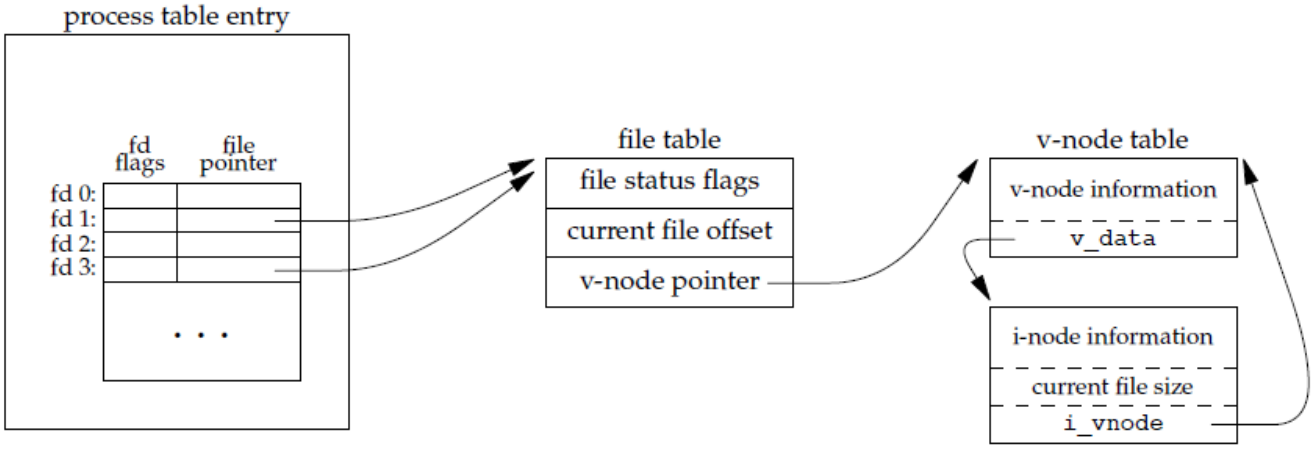
6 VFS虚拟文件系统

Linux支持各种各样的文件系统格式，如ext2、ext3、reiserfs、FAT、NTFS、iso9660等等，不同的磁盘分区、光盘或其它存储设备都有不同的文件系统格式，然而这些文件系统都可以mount到某个目录下，使我们看到一个统一的目录树，各种文件系统上的目录和文件我们用ls命令看起来是一样的，读写操作用起来也都是一样的，这是怎么做到的呢？Linux内核在各种不同的文件系统格式之上做了一个抽象层，使得文件、目录、读写访问等概念成为抽象层的概念，因此各种文件系统看起来用起来都一样，这个抽象层称为虚拟文件系统（VFS，Virtual Filesystem）。

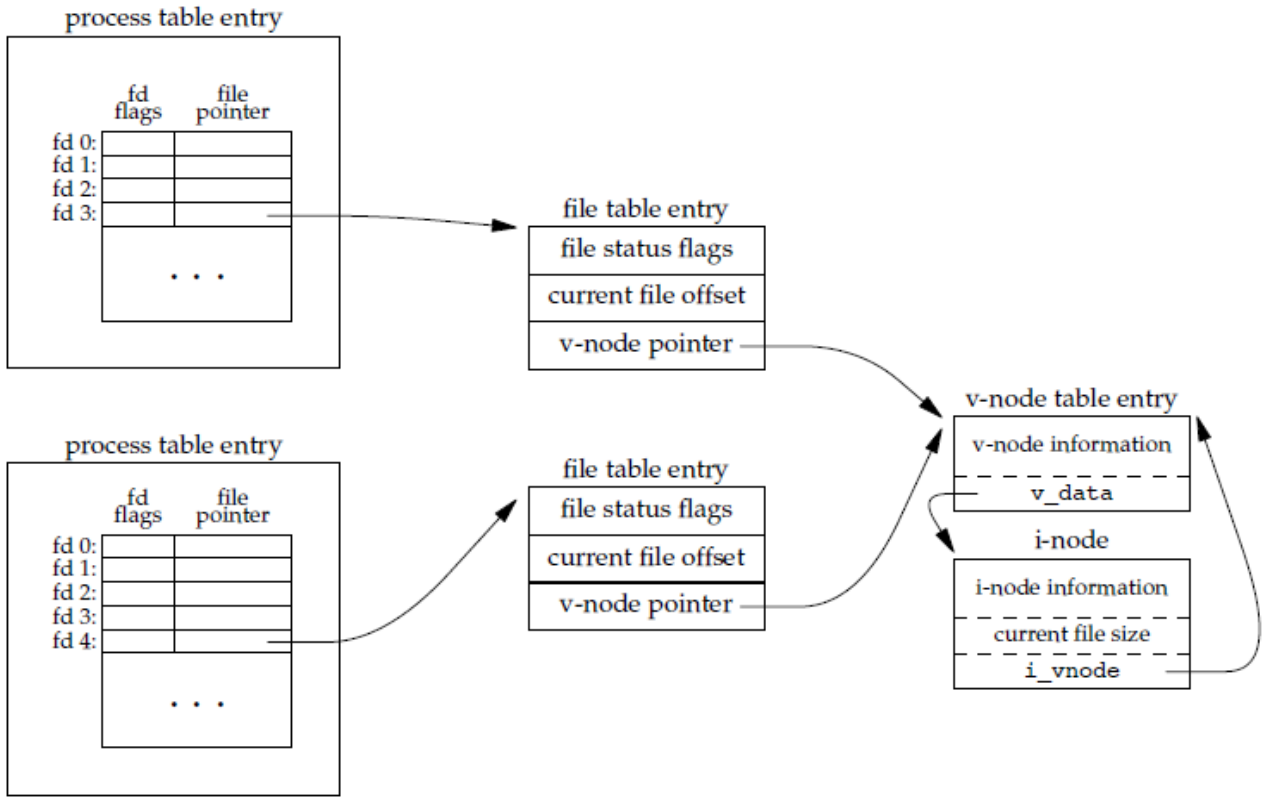
- dup/dup2

```
#include <unistd.h>
int dup(int oldfd); //复制一个文件描述符指向oldfd的file结构体
int dup2(int oldfd, int newfd); //复制newfd文件描述符指向oldfd的file结构体,newfd如果已经被使用，则先关闭
```

dup和dup2都可用来复制一个现存的文件描述符，使两个文件描述符指向同一个file结构体。如果两个文件描述符指向同一个file结构体，File Status Flag和读写位置只保存一份在file结构体中，并且file结构体的引用计数是2，引用计数为0的时候file结构体释放。如下图：



如果两次open同一文件得到两个文件描述符，则每个描述符对应一个不同的file结构体，可以有不同的File Status Flag和读写位置。请注意区分这两种情况。如下图：



源码：dup_test.c

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

int main(int argc, char *argv[])
{
    int fd, copyfd;

    fd = open("./log.txt", O_RDWR);
    /*复制fd*/
    copyfd = dup(fd);

    char buf1[] = "hello ";
    char buf2[] = "world!";

    /*往fd文件写入内容*/
    if (write(fd, buf1, 6) != 6) {
        printf("write error!");
    }

    /*打印出fd和copyfd的偏移量，经过上面的写操作，都变成6了*/
    printf("%d\n", lseek(fd, 0, SEEK_CUR));
    printf("%d\n", lseek(copyfd, 0, SEEK_CUR));

    /*往copyfd写入内容*/
    if (write(copyfd, buf2, 6) != 6) {
        printf("write error!");
    }

    return 0;
}
```

重定向标准输出标准错误

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char**argv)
{
    int fd = open("./log", O_WRONLY | O_CREAT, 0755);
    dup2(fd, 1);
    dup2(fd, 2);
    /*这里记住一定要先重定向再关闭*/
    close(fd);

    printf("helloworld\n");
    puts("helloworld\n");

}
```

第三章 Linux进程

1 什么是进程

程序是一个包含可执行指令的文件，而进程是一个开始执行但还没有结束的程序的实例。进程的标准定义是：“进程是一个具有独立功能的程序关于某个数据集合的一次可以并发执行的运行活动，是处于活动状态的计算机程序”。

在进程的整个运行期间，它将会用到各种系统资源，会用到CPU运行它的指令，需要物理内存保存它的数据，它可能打开和使用各种文件，直接或间接地使用系统中的各种物理设备。系统中最为宝贵的资源是CPU，Linux是一个多进程的操作系统，Linux内核为每个进程指派一定的运行时间，这个时间通常很短，短到毫秒为单位，然后依照某种规则，从多个进程中挑选一个运行，其他的进程暂时等待，当正在运行的那个进程时间耗尽，或执行完毕退出。当正在运行的进程等待其他的系统资源时，Linux 内核将取得CPU的控制权，并将CPU分配给其他正在等待的进程。因为每个进程占用的时间很短，从使用者的角度来看，就好像多个进程同时运行一样了。Linux

系统内核必须了解进程本身的情况和进程所用到的各种资源，以便在多个进程之间合理地分配系统资源。

Linux是一个多处理操作系统，进程具有独立的权限与职责。如果系统中某个进程崩溃，它不会影响到其余的进程。每个进程运行在其各自的虚拟地址空间中，进程之间可以通过由内核控制的机制互相通讯。

2 tast struct简介

为了让linux来管理系统中的进程，每个进程用一个task_struct数据结构来表示。队列task包含指向系统中所有task_struct结构的指针。创建新进程时，Linux将从系统内存中分配一个task_struct结构并将其加入task队列中。系统中还有一个当前进程的指针，用来指向当前运行进程的结构。

task_struct数据结构庞大而复杂，但它可以分成一些功能组成部分：

（1）进程状态

进程在执行过程中会根据环境来改变状态，Linux进程有以下状态: 运行状态、等待状态、停止状态和僵尸状态。

（2）进程调度信息

调度器需要这些信息以便判定系统中哪个进程最迫切需要运行。

（3）标识符

系统中每个进程都有进程标志。进程标志并不是task队列的索引，它仅仅是个数字。每个进程还有一个用户与组标志，它们用来控制进程对系统中文件和设备的存取权限。

（4）进程间通信

Linux支持经典的Unix IPC机制。

（5）进程关系

Linux系统中所有进程都是相互联系的。除了初始化进程外，所有进程都有一个父进程。新进程不是被创建，而是被复制，或者从以前的进程克隆而来。每个进程对应的task_struct结构中包含有指向其父进程和兄弟进程(具有相同父进程的进程)以及子进程的指针。

（6）时间和定时器

核心需要记录进程的创建时间以及在其生命期间消耗的CPU时间。

（7）文件系统

进程可以自由地打开或者关闭文件，进程的task_struct结构中包含一个指向每个打开文件描述符的指针。

两个指向VFS索引节点的指针。每个VFS索引节点唯一地表示文件中的一个目录或者文件，同时还对底层文件系统提供了统一的接口。第一个索引节点是进程的根目录，第二个节点是当前的工作目录。两个VFS索引节点都有一个计数字段用来表明指向节点的进程数，当多个进程引用它们时，它的值就增加。

（8）虚拟内存

多数进程都有一些虚拟内存，linux核心必须跟踪虚拟内存与系统物理内存的映射关系。

（9）处理器的内容

进程可以认为是系统当前状态的总和。进程运行时，它将使用处理器的寄存器以及堆栈等等。进程被挂起时，进程的上下文，所有的CPU相关的状态必须保存在它的task_struct结构中。当调度器重新调度该进程时，所有上下文被重新设定。

（10）绑定的终端

进程是否需要控制终端，也记录在task_struct.

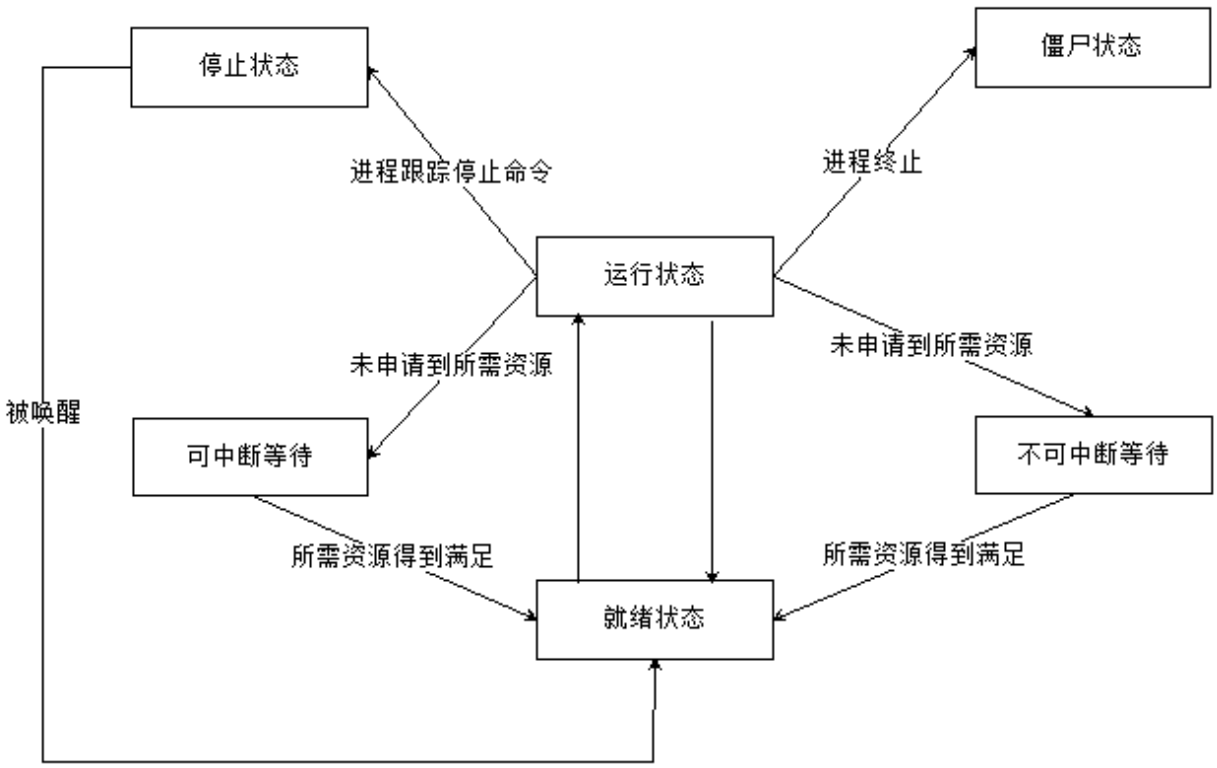
3 进程状态以及状态切换

进程在生存周期中的其状态时变化的。下面是Linux操作系统的进程的几种常见的状态：

- 运行状态，此状态下进程正在运行或者是准备运行状态(即就绪状态)。在ps命令列出的状态列Stat列中，字母R表示运行状态。
- 等待状态，进程正在等待事件的发生或者等待某种系统资源。Linux操作系统中的等待进程分为可中断等待和不可中断等待。可中断等待进程可以被信号(signal)中断, ps命令看到的字母S表示程序处于可中断等待状态。不可中断状态进程不受信号干扰，直到硬件状态改变，通常是处于I/O操作过程中，字母D表示进程处于不可中断等待状态。
- 停止状态，进程被停止，通常是收到了一个控制信号或者正在被跟踪调试。字母T表示进程处于停止状态，只有被唤醒才能分配到cpu时间片。
- 僵尸状态，进程由于某种原因已经终止或结束，但在进程表项中仍有记录，该记录可由父进程收集。字母Z表示进程处于僵尸状态。

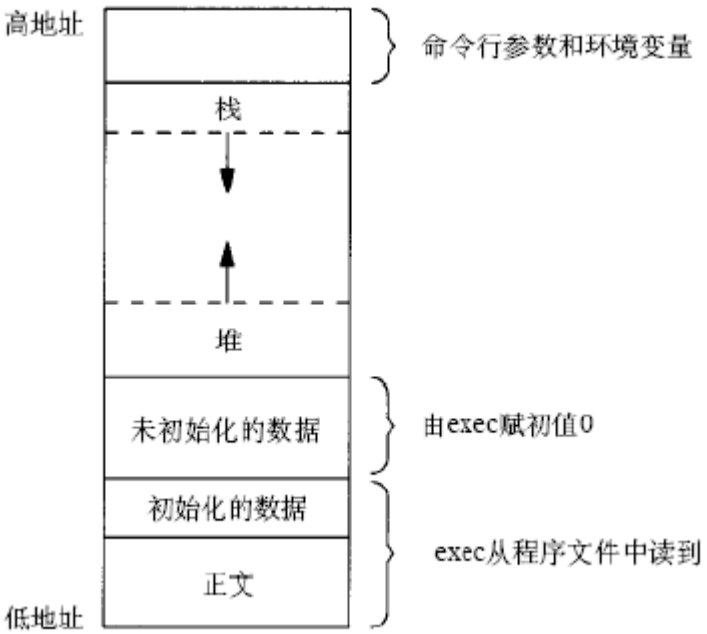
  在linux系统中，进程的执行模式划分为用户模式和内核模式。如果当前运行的是应用程序或者内核之外的系统程序，那么对应进程就在用户模式下运行：如果在用户程序执行过程中出现系统调用或者发生中断事件，就要运行操作系统核心程序，进程模式就变成了内核模式。

  按照进程的功能和运行的程序分类，进程可以划分为两大类：一类是系统进程，只运行在内核模式，执行操作系统代码，完成一些管理性的工作，例如内存分配和进程切换，另外一个类是用户进程，通常在用户模式中执行,并通过系统调用或在出现中断、异常时进入内核模式。用户进程既可以在用户模式下运行，也可以在内核模式下运行。



4 进程环境变量

每个进程都有它所运行的的一个环境变量，环境变量一般是存放在内存的用户空间的一个环境变量表中，这个环境变量表是在进程生成时，从父进程的环境变量表中拷贝一份。



libc中定义的全局变量`char** environ`指向环境变量表，environ没有包含在任何头文件中，所以在使用时要用`extern`

声明。例如：

```
#include <stdio.h>
int main(int argc, char** argv)
{
    extern char** environ;
    int i = 0;
    while(environ[i]!=NULL)
    {
        printf("%s\n", environ[i++]);
    }
    return 0;
}
```

4.1 获取环境变量

使用getenv获取某个环境变量的值。

函数原型为:

```
char * getenv(const char* name);
```

函数的参数`name`为环境变量的键值，找到就返回指向value的指针, 否则返回NULL。

4.2 设置环境变量

- 使用setenv来设置某个环境变量的值。

  函数原型为:

```
int setenv(const char* name, const char* value, int rewrite);
```

  函数的参数`name`为环境变量的键值，`value`为环境变量的值；

  参数rewrite为1，则覆盖原来的值;

  参数rewrite为0，则不覆盖原来的值 ;

- 使用unset来清除某个环境变量。

```
 int unsetenv (const char *name)
```

源码：test_setenv

```
#include <stdio.h>
int main(int argc, char** argv)
{
    printf("PATH=%s\n", getenv("PATH"));
    setenv("PATH", "/", 1);
    printf("PATH=%s\n", getenv("PATH"));
    return 0;
}
```

5 进程控制

在Linux系统中，用户创建一个进程的唯一方法就是使用系统调用fork()。内核为完成系统调用fork()要进行几步操作。

第一步，为新进程在进程表中分配一个表项task_struct。系统对一个用户可以同时运行的进程数是有限制的，对超级 用户没有该限制，但也不能超过进程表的最大表项的数目。

第二步，给子进程一个唯一的进程标识号(PID)。该进程标识号其实就是该表项在进程表中的索引号。

第三步，复制一个父进程的进程表项的副本给子进程。内核初始化子进程的进程表项时，是从父进程处拷贝的。所以子进程拥有父进程一样的uid、euid、gid，用于计算优先权的nice值、当前目录、当前根、用户文件描述符表等。

第四步，把与父进程相连的文件表和索引节点表的引用数加1，这些文件自动地与该子进程相连。

Linux系统的系统调用exit()是进程用来终止执行的。进程发出该调用，内核就会释放该进程所有的资源，释放进程上下文所占的内存空间，但进程表项还被保留，内核将进程表项中记录进程状态的字段设为僵尸状态。内核在进程收到不可捕捉的信号时，会从内核内部调用exit(),使得进程退出。父进程通过wait()得到其子进程的进程表项中记录的计时数据，并释放进程表项。最后，内核让进程1(init进程)接收终止执行的进程的所有子进程。如果有子进程僵尸，就向init进程发出一个SIGCHLD的中断信号。

一个进程通过调用wait()来与它的子进程同步，如果发出wait()调用的进程没有子进程则返回一个错误，如果找到一个僵尸的子进程就取子进程的PID以及子进程退出时的退出参数。如果有子进程，但没有僵尸的子进程，发出wait()调用的进程就将进入可中断的睡眠状态，直到收到一个子进程僵尸(SIGCHLD)的信号或其他信号。

进程控制的另一个主要内容就是对其他程序引用。该功能是通过系统调用exec()来实现的，exec()调用将一个可执行的程序文件读入并执行。内核读入程序文件的正文，清除原先进程的数据区，清除原先进程的中断信号处理函数的地址，当exec()调用返回时，进程执行新的正文。

6 进程调度

CPU资源是有限的，那么在调度进程时，每个进程只允许运行很短的时间，当这个时间用完之后，系统将选择另一个进程来运行，原来的进程必须等待一段时间以继续运行，这段时间称为时间片。

Linux使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后，系统必须将当前进程的状态，处理器中的寄存器以及上下文状态保存到task_struct结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将CPU时间合理的分配给系统中每个可执行进程，调度管理器必须将这些时间信息也保存在task_struct中。

7 获得进程有关的ID

- 用户标识号UID(user ID):用于标识正在运行进程的用户。
- 用户组标识号GID(group ID): 用于标识正在运行进程的用户所属的组ID。
- 进程标识号PID(process ID):用于标识进程。

```
int getuid(), 获取进程的实际用户ID。
int geteuid(),获取进程的有效用户ID。
int getgid(), 获取进程的用户所属的实际用户组ID。
int getegid(), 获取进程的用户所属的有效用户组ID。

int getpid(), 获取当前进程的ID。
int getppid(), 获取父进程的ID。
```

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    printf("Current process's UID = [%d]\n", getuid());
    printf("Current process's GID = [%d]\n", getgid());
    printf("Current process's PID = [%d]\n", getpid());
    printf("Current process's PPID = [%d]\n", getppid());
    return 0;
}
```

注意:

1、实际用户ID和实际用户组ID：标识我是谁。也就是登录用户的uid和gid，比如我的Linux以where登录，在Linux运行的所有的命令的实际用户ID都是where的uid，实际用户组ID都是where的gid（可以用id命令查看）。

2、有效用户ID和有效用户组ID：进程用来决定我们对资源的访问权限。一般情况下，有效用户ID等于实际用户ID，有效用户组ID等于实际用户组ID。当设置SUID位设置，则有效用户ID等于文件的所有者的uid，而不是实际用户ID；同样，如果设置了SGID位，则有效用户组ID等于文件所有者的gid，而不是实际用户组ID。

8 fork函数创建进程

fork叉子，西方人吃饭用的东西，经常用作刀叉。

一个现有进程可以调用fork函数创建一个新进程。包括代码、数据和分配给进程的资源全部都复制一份，除了进程标识pid不同，其他基本都跟父进程一样，由fork创建的新进程被称为子进程（child process）。fork函数被调用一次但返回两次。

fork函数原型

```
pid_t fork();
```

1）在父进程中，fork返回新创建子进程的进程ID；
2）在子进程中，fork返回0；
3）创建子进程失败，fork返回-1；

源代码: fork_test.c

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv)
{
    pid_t pid;
    printf("Current process PID = [%d]\n", getpid());
    pid = fork();
    if(pid == 0)
    {
        printf("child process PID = [%d], parent PID = [%d]\n", getpid(), getppid());
    }
    else if(pid != -1)
    {
        printf("parent process, PID = [%d]\n", getpid());
    }
    else
    {
        perror("fork");
    }
    return 0;
}
```

程序非常简单，调用了fork()函数并把返回值赋值给pid变量。分别在pid等于0、pid不等于-1的时候输出当前进程的PID。

编译运行这个程序，结果如下：

```
where@ubuntu:~$ ./fork_test
Current process's PID = [18662] ppid = [18377]
parent process, PID = [18662]
child process,  PID = [18663], parent PID = [18662]
```

可见，当pid等于0的时候，表明是当前进程派生出来的子进程。如果返回的pid不等于-1，表明派生操作成功并且返回值就是新的子进程PID。如果返回值等于-1，那么操作失败。

在代码pid=fork()之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在执行了，这两个进程的代码部分完全相同。

源码：fork_test2.c

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv)
{
    pid_t pid;
    int i;
    printf("Current process PID = [%d]\n", getpid());
    pid = fork();
    if(pid == 0)
    {
        for(i = 0; i < 10; i++)
        {
            printf("child process, PID=[%d], my parent PID=[%d]\n", getpid(), getppid());
            sleep(1);
        }
    }
    else if(pid != -1)
    {
        for(i = 0; i < 10; i++)
        {
            printf("parent process, PID=[%d], child PID=[%d]\n", getppid(), pid);
            sleep(1);
        }
    }
    else
    {
        printf("fork error\n");
    }
    return 0;
}
```

我们的父进程和子进程函数中都加入了一个for循环，循环输出10次，并在每次输出之后睡眠1秒。下面时程序的执行结果：

输出如图:

```
where@ubuntu:~$ ./fork_test2
Current process PID = [18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
parent process, PID=[18377], child PID=[18774]
child process, PID=[18774], my parent PID=[18773]
```

从输出的结果，可以看到，子进程的输出和父进程的输出是交替输出的，而不是由某个一个进程执行完10次循环之后，才能执行另外一个进程，当进程进入睡眠模式的时候，内核会调度新的进程进入运行模式，这样使得两个进程看上去像是同步执行。

调用fork的时候直接从父进程复制一份PCB到子进程，除了PID不同以外，其他的都一样，包括文件描述符表中的文件指针也指向同一个地址。就相当于是调用了dup复制了文件描述符。于是父子进程其实是共享同一个文件读写偏移量。

测试例子如下：

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

int main()
{
    int fd,nr;
    char buf[20];
    pid_t pid;
    fd = open("log",O_RDONLY);
    pid = fork();
    if(pid == 0)
    {
        nr = read(fd,buf,10);
        buf[nr]='\0';
        printf("pid %d buf %s\n",getpid(),buf);
        exit(0);
    }
    nr = read(fd,buf,10);
    buf[nr]='\0';
    printf("pid %d buf %s\n",getpid(),buf);
    return 0;
}
```

在log文件中写入helloworld1234567890，运行程序后输出：

```
where@ubuntu:~$ ./a.out
pid 55020 buf helloworld
pid 55021 buf 1234567890
where@ubuntu:~$
```

足够说明fork的时候，复制出来的文件描述符是共享文件偏移量的。

9 终止进程以及进程返回值

`exit()`函数的功能是终止调用的进程。它的函数原型如下：

```
#include <stdllib.h>
void exit(int status);
```

exit()函数会调用系统调用_exit, 立即终止调用的进程。所有属于该进程的文件描述符都 关闭。该进程的所有子进程由进程init接收，并对该进程的父进程发出一个SIGCHLD（子进程僵死）的信号。参数status作为退出的状态值返回父进程，该值可以通过系统调用wait()来收集。

注意：子进程被SIGSTOP或者SIGTSTP挂起的时候，父进程正常退出，这个时候，父进程将发送SIGHUP信号给所有被挂起的子进程。

10 等待进程

子进程终止的时候，必须由父进程回收其进程表task_struct，否则进程将处于僵尸状态直到被回收。如果父进程在子进程终止前已经终止，那么该进程的所有子进程由init进程回收。

下面我们来看一个例子，如果子进程终止而没有被回收的状态。

源代码： zombi_test.c

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int ret;
    pid_t pid;
    pid = fork();
    if(pid == 0)
    {
        printf("child PID [%d] will exit now.\n", getpid());
        exit(0);
    }
    else if(pid != -1)
    {
        sleep(60);
    }
    else
    {
        printf("fork error\n");
    }
    return 0;
}
```

程序执行屏幕输出：

```
where@ubuntu:~$ ./zombie_test
child PID [19833] will exit now.
```

然后到另外一个终端查看进程：

```
where@ubuntu:~$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
where     7362  0.0  0.5   8736  5132 pts/10   Ss    8月04   0:02 bash
where    15545  0.0  0.4   8560  5032 pts/0    Ss    8月05   0:00 bash
where    18377  0.0  0.4   8668  5112 pts/10   S     8月05   0:00 bash
where    19495  0.0  0.4   8560  4920 pts/10   S    01:52   0:00 /bin/bash
where    19832  0.0  0.0   2032   552 pts/10   S+   03:00   0:00 ./zombie_test
where    19833  0.0  0.0      0     0 pts/10   Z+   03:00   0:00 [./zombie_test] <defunc
where    19835  0.0  0.2   6656  2380 pts/0    R+   03:00   0:00 ps -u
```

可见这时zombie_test处于“Z”的僵尸状态。当过30秒后，父进程退出，我们再查看子进程时，发现子进程已经被回收了。原来当父进程退出的时候，子进程将被移交给init，init进程回收了这个僵尸进程。

- 系统调用wait()的功能是发出调用的进程只要有子进程，就睡眠直到有一个子进程终止。它的函数原型是：

```c
#include <sys/wait.h>
pid_t wait(int *status);
```

wait()函数将使进程进入睡眠直到它的一个子进程退出时或收到一个不能被忽略的信号被唤醒。如果调用时，已经有退出的子进程，该调用立即返回。其中调用返回时参数status中包含子进程退出时的状态信息，返回值是退出的子进程的PID。如果参数status是NULL，那么返回值将被忽略。

检查子程序返回状态的宏定义。

宏定义	含义
WIFEXITED(status)	如果进程通过系统调用_exit或函数调用exit正常退出，该宏值为真。
WIFSIGNALED(status)	如果子进程由于得到信号(signal)导致退出时，该宏的值为真。
WEXITSTATUS(status)	如果WIFEXITED返回真，该宏返回由子进程调用_exit(status)或exit(status)时设置的调用参数status值。
WTERMSIG(status)	如果WIFSIGNAL(status)返回为真，该宏返回导致子进程退出的信号(signal)的值。

源代码： wait_test.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
int main(int argc, char** argv)
{
    int ret;
    pid_t pid;
    pid_t pid_c;
    pid = fork();
    if(pid == 0)
    {
        printf("child PID [%d] will exit with status 0.\n", getpid());
        exit(0);
    }
    else if(pid == -1)
    {
        pid_c = wait(&ret);
        printf("child prosss PID [%d] return [%d].\n", pid_c, ret);
    }
    else
    {
        printf("error fork\n");
    }
    return 0;
}
```

执行这个函数，结果如下：

```
where@ubuntu:~$ ./wait_test
child PID [20846] will exit with status 0.
child prosss PID [20846] return [0].
```

- 另外一个等待函数waitpid，函数原型如下：

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

waitpid()函数与调用wait()的区别时waitpid()等待由参数pid指定的子进程退出。其中参数pid的含义与取值方法如下表:

waitpid() pid参数

参数	含义
pid < -1时	等待进程组ID等于pid的绝对值的进程组子进程退出
pid=0时	等待进程组ID等于当前进程的进程组ID的子进程退出
pid>0时	等待进程ID等于参数pid的子进程退出。
pid=-1时	等待任何子进程退出，相当于调用wait()。

waitpid() options参数可以使用一下的宏或运算获得，如果都不要这些标志可以直接填0

宏	含义
WNOHANG	该选项要求如果没有子进程退出就立即返回。
WUNTRACED	子进程变为停止状态，该调用也从等待中返回和报告状态，如果status不是空，调用将使status指向该信息。
WCONTINUED	如果子进程由于被SIGCONT唤醒， waitpid则立即返回

waitpid()将返回退出的子进程的PID。如果设置了WNOHANG选项没有子进程退出就返回0。如果发生错误时返回-1，发生错误时，可能设置的错误代码如下。

返回值	含义
ECHILD	该调用指定的子进程pid不存在，或者不是发出调用进程的子进程。
EINVAL	参数options无效。
ERESTARTSYS	WONHANG没有设置并且捕获到SIGCHLD或其他未屏蔽信号。

检查子程序返回状态的宏定义。

宏定义	含义
WIFEXITED(status)	如果进程通过系统调用_exit或函数调用exit正常退出，该宏值为真。
WIFSIGNALED(status)	如果子进程由于得到信号(signal)导致退出时，该宏的值为真。
WIFSTOPPED(status)	如果子进程没有终止，但停止了并可以重新执行时，返回该值。这种情况仅仅出现在waitpid调用中使用了WUNTRACED选项。
WIFCONTINUED(status)	子进程由停止态转为就绪态，返回真。这种情况仅仅出现在waitpid调用中使用了WCONTINUED选项。
WEXITSTATUS(status)	如果WIFEXITED返回真，该宏返回由子进程调用_exit(status)或exit(status)时设置的调用参数status值。
WTERMSIG(status)	如果WIFSIGNALED(status)返回为真，该宏返回导致子进程退出的信号(signal)的值。
WSTOPSIG(status)	如果WIFSTOPPED(status)返回为真，该宏返回导致子进程停止的信号(signal)的值。

下面的例子用了waitpid()来等待子进程，并用宏定义来判断返回值。

源代码: waitpid_test.c

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int status;
    pid_t pid;
    pid_t pid_c;
    pid = fork();
    if(pid == 0)
    {
        printf("child process\n");
        sleep(5);
        printf("child PID = [%d] \n", getpid());
        exit(3);
    }
    else if(pid != -1)
    {
        do
        {
            pid_c = waitpid(pid, &status, WNOHANG);
            sleep(1);
        }
        while(pid_c == 0); /*如果返回0代表没有子进程退出*/
        if(WIFEXITED(status))
          printf("child process exit code = [%d]\n", WEXITSTATUS(status));
    }
    else
    {
        printf("fork error\n");
    }
    return 0;
}
```

这个例子中，我们让子进程先睡眠5秒然后退出，父进程等待函数waitpid()的option使用了WNOHANG选项，是的父进程不会被阻塞在waitpid()函数。用一个while循环没间隔1秒钟就等待一次子进程的退出。

僵尸进程: 子进程退出，父进程没有回收子进程资源（PCB），则子进程变成僵尸进程。

孤儿进程: 父进程先于子进程结束，则子进程成为孤儿进程,子进程的父进程成为1号进程init进程，称为init进程领养孤儿进程。

11 执行其他程序

我们派生一个子进程来完成某项工作的时候，经常需要让另外一个程序来完成。函数exec()可以用来执行一个可执行文件来代替当前进程的执行映像。需要注意的是，该调用并没有生成新的进程，而是在原有进程的基础上，替换原有进程的正文，调用前后是同一个进程，进程号PID不变，但执行的程序变了。在Linux中，有6种调用的形式，他们的函数原型如下：

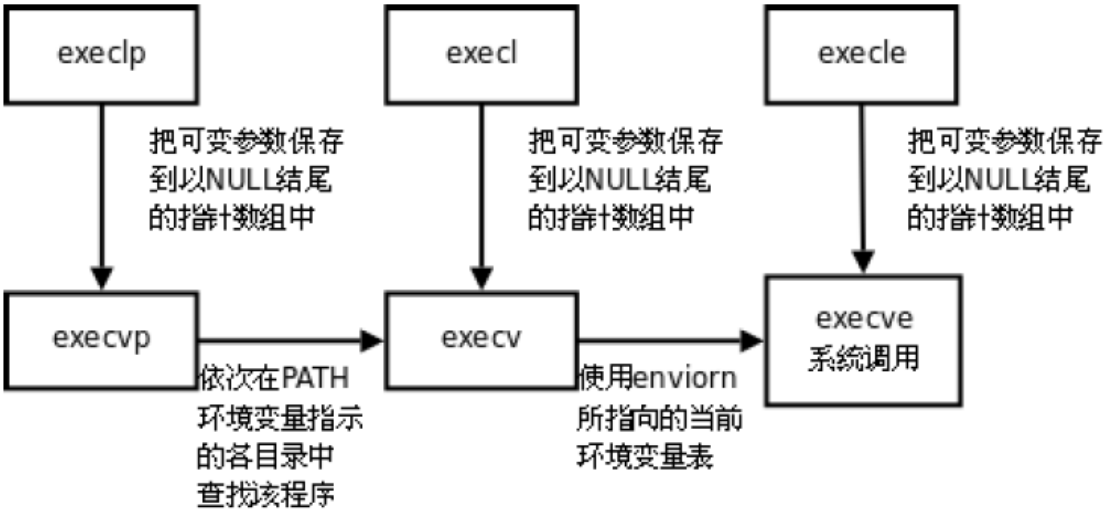
```
#include <unistd.h>
int execl(const char* path, const char *arg, ...);
int execlp(const char* file, const char *arg, ...);
int execle(const char* path, const char *arg, ..., char * const envp[]);
int execv(const char* path, char* const argv[]);
int execvp(const char* file, char * const argv[]);
int execve(const char* filename, char* const argv[], char* const envp[]);
```

exec()的六个函数实现的功能是一样的，只是在传递参数和设置环境变量方面提供了不同的方式。6个函数都是以exec四个字母开头的，后面的字母表示了其用法上的区别：

表明后面的参数列表是要传递给程序的参数列表，参数列表的第一个参数

区分	含义
带有字母`l`的函数	表明后面的参数列表是要传递给程序的参数列表，参数列表的第一个参数必须是要执行程序，最后一个参数必须是NULL
带有字母`p`的函数	第一个参数可以是相对路径或程序名，如果无法立即找到要执行的程序，那么就在环境变量PATH指定的路径中搜索。其他函数的第一个参数则必须是绝对路径。
带有字母`v`函数	表明程序的参数列表通过一个字符串数组来传递。这个数组和最后传递给程序的main函数的字符串数组argv完全相同。第一个参数必须是程序名，最后一个参数也必须是NULL。
带有字母`e`的函数	用户可以自己设置程序接收一个设置环境变量的数组。

事实上，只有execve是真正的系统调用，其它五个函数最终都调用execve，所以execve在man手册第2节，其它函数在man手册第3节。这些函数之间的关系如下图所示。



- 用execl()执行一个程序

  源代码execl_test.c

```
#include <stdio.h>
#include <unistd.h>
extern char **environ;
int main(int argc, char** argv)
{
    char* arg[3];
    arg[0] = "/bin/ls";
    arg[1] = "/";
    arg[2] = NULL;
    printf("this is execl test\n");
    execl(arg[0], arg[1], arg[2]);
    printf("running after the execvp() call");
    return 0;
}
```

从输出来看，execl()执行了新的程序，代替了当前进程的程序，因此除非execl()调用失败，否则后面的代码永远不会被调用。

```
where@ubuntu:~$ ./execl_test
this is execvp test
a.out          execl_test.c  Music      test_env test_setenv
creat          fcntl_test    myls       test_env.c   test_setenv.c
creat.c        fcntl_test.c  myls.c     test_exit    test_struct.c
Desktop        filanem   Pictures   test_exit.c  test.txt
Documents      filename  Public     test_fork    Videos
Downloads      fork_test2    Templates  test_fork.c  write
examples.desktop  helloworld    test       test_id  write.c
execl_test     include   test.c     test_id.c
where@ubuntu:~$
```

- 用execvp执行一个程序

  源代码:execvp_test.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    char* arg[3];
    arg[0] = "ls";
    arg[1] = "/";
    arg[2] = NULL;
    printf("this is execvp test\n");
    execvp("ls", arg);
    return 0;
}
```

  第一个参数由`/bin/ls`变成了`ls`,第二个参数是一个字符串数组。运行结果与execl_test的一模一样。

- exec函数通常和fork()一起配合使用，由fork派生一个子进程，在子进程执行新的程序。例如：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
int main(int argc, char** argv)
{
    int ret;
    pid_t pid;
    pid_t pid_c;
    pid = fork();
    if(pid == 0)
    {
        printf("child PID=[%d]\n", getpid());
        execlp("ls", "ls", "-ls", NULL);
    }
    else if(pid != -1)
    {
        pid_c = wait(&ret)
        printf("child PID=[%d] wait return = [%d]\n", pid_c, ret);
    }
    return 0;
}
```

实现一个自己的shell

```c
#include <sys/wait.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char** argv)
{
    pid_t pid;
    char buf[1024];
    char * arg[10];
    int i, ret;
AGAIN:
 /*读取命令行*/
    printf("myshell:%s$", getcwd(NULL, 1024));
    fflush(stdout);
    ret = read(STDIN_FILENO, buf, sizeof(buf) - 1);
    if(ret == -1)
    {
        perror("read");
        exit(1);
    }
    buf[ret - 1] = '\0';
/*处理exit命令*/
    if(strcmp(buf, "exit") == 0)
    {
        exit(0);
    }
/*分割参数*/
    i = 0;
    arg[i] = strtok(buf, " ");
    while(arg[i] !=  NULL)
    {
        if(++i >= sizeof(arg))
            break;
        arg[i] = strtok(NULL, " ");
    }
/*处理cd命令*/
    if(i ==  2 && strcmp(arg[0], "cd") == 0)
    {
        chdir(arg[1]);
        goto AGAIN;
    }
/*处理其他命令*/
    pid = fork();
    if(pid == 0)
    {
        execvp(arg[0], arg);
        return 0;
    }
    else if(pid != -1)
    {
        wait(NULL);
        goto AGAIN;
```

```
    }
    return 0;
}
```

第四章 Linux 进程间通信

1 信号

1.1 什么是信号机制

信号(signal)机制是Linux系统中最为古老的进程之间的通信机制。Linux信号也可以称为软中断，是在软件层次上对中断机制的一种模拟。在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程发生了什么。

- 信号时间的发生有两个来源：

  （1）硬件来源，比如我们按下了键盘或者其他硬件信号触发，硬件异常如除以零运算，内存非法访问。

（2）软件来源，最常见发送信号的系统函数`kill()` `raise()` `alarm()` 和 `setitimer()`等函数以及ctl+c发出SIGINT、ctl+z SIGTSTP、ctl+\ SIGQUIT。

- Linux系统中定义了一系列的信号，可以使用`kill -l`命令列出所有的信号。

  Linux的信号机制是从Unix继承下来的，早期Unix系统只定义了32种信号，现在Linux支持64种信号。

```
1) SIGHUP: #当用户退出shell时，由该shell启动的所有进程将收到这个信号，默认动作为终止进程
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。默认动作为终止里程。
3）SIGQUIT: #当用户按下<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。默认动作为终止进程。
4）SIGILL: #CPU检测到某进程执行了非法指令。默认动作为终止进程并产生core文件
5）SIGTRAP: #该信号由断点指令或其他trap指令产生。默认动作为终止里程并产生core文件。
6 ) SIGABRT:#调用abort函数时产生该信号。默认动作为终止进程并产生core文件。
7）SIGBUS: #非法访问内存地址，包括内存对齐出错，默认动作为终止进程并产生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。默认动作为终止进程并产生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。默认动作为终止进程。它向系统管理员提供了可以杀死任何进程的方法。
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。默认动作为终止进程。
11）SIGSEGV: #指示进程进行了无效内存访问。默认动作为终止进程并产生core文件。
12）SIGUSR2: #这是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。默认动作为终止进程。1
13）SIGPIPE: #Broken pipe向一个没有读端的管道写数据。默认动作为终止进程。
14) SIGALRM:#定时器超时，超时的时间由系统调用alarm设置。默认动作为终止进程。
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。默认动作为终止进程。
16）SIGCHLD: #子进程结束时，父进程会收到这个信号。默认动作为忽略这个信号。
17）SIGCONT: #停止进程的执行。信号不能被忽略，处理和阻塞。默认动作为终止进程。
18）SIGTTIN: #后台进程读终端控制台。默认动作为暂停进程。
19）SIGTSTP: #停止进程的运行。按下<ctrl+z>组合键时发出这个信号。默认动作为暂停进程。
21）SIGTTOU:#该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生。默认动作为暂停进程。
22）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，默认动作为忽略该信号。
23）SIGXFSZ: #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。默认动作为终止进程。
24）SIGXFSZ: #超过文件的最大长度设置。默认动作为终止进程。
25）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。默认动作为终止进程。
26）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。默认动作为终止进程。
27）SIGWINCH: #窗口变化大小时发出。默认动作为忽略该信号。
28）SIGIO: #此信号向进程指示发出了一个异步IO事件。默认动作为忽略。
29）SIGPWR: #关机。默认动作为终止进程。
30）SIGSYS: #无效的系统调用。默认动作为终止进程并产生core文件。
31）SIGRTMIN～（64）SIGRTMAX: #LINUX的实时信号，它们没有固定的含义（可以由用户自定义）。所有的实时信号的默认动作都为终止进程
```

1.2 进程对信号的响应和处理

- 进程可以通过三种方式来响应和处理一个信号：

  （1）忽略信号：即对信号不做任何处理，但是两个信号不能忽略:`SIGKILL`以及`SIGSTOP`.

  （2）捕捉信号：当信号发生时，执行用户定义的信号处理函数。

  （3）执行默认操作: Linux对每种信号都规定了默认操作，man 7 signal查看。

```
Term Default action is to terminate the process.

Ign Default action is to ignore the signal.

Core Default action is to terminate the process and dump core (see core(5)).

Stop Default action is to stop the process.

Cont Default action is to continue the process if it is currently stopped.
```

- 信号之间不存在相对的优先权。信号在产生时也并不马上送给进程，信号必须等待直到进程再一次被调度运行。
- 进程对收到的信号有对应的缺省操作，如果进程要自定义处理某个信号，那么就要在进程中安装该信号。

  安装函数原型是:

```
#include <signal.h>
void ( *signal(int sig, void (*func)(int)) )(int);
```

  `signal()`主要用于前32种非实时信号的安装

  `sig`：是要安装的信号值。

  `func`：是指定信号处理函数，如果想忽略信号可以填`SIG_IGN`（函数地址宏）,如果想采用系统默认的处理函数`SIG_DFL`（函数地址宏）。

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

  sigaction用于后32种实时信号的安装。

1.3 信号的发送

并不是每个进程都可以向其他的进程发送信号。一般的进程只能向具有相同uid和gid的进程发送信号，或向相同进程组中的其他进程发送信号。常用的发送信号的函数有kill()、raise()、alarm()、setitimer()、abort()等。

- kill()函数可以给指定的进程发送某一个信号，其函数原型是:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

参数`pid`含义：

pid值	含义
pid > 0	给PID为pid的进程发送信号
pid = 0	给同一个进程组的所有进程发送信号
pid < -1	给进程组ID为pid的所有进程发送信号
pid = -1	给除了自身之外的PID大于1的进程发送信号

参数`sig`为具体要发送的信号值，为0时什么信号都不发送。

返回值，kill()发送成功，返回0，否则返回-1。

通过errno以及perror可以查看错误信息。

宏	含义
EINVAL	所发送的信号无效。
EPERM	没有向目标进程发送信号的权利。
ESRCH	目标进程不存在或进程已经终止，处于僵尸状态。

源代码:kill_test.c

```c
#include <unistd.h>
#include <signal.h>

void handle_sig()
{
    printf("SIGBUS\n");
}

int main(int argc, char** argv)
{
   pid_t pid;
   pid = fork();
   signal(SIGBUS, handle_sig);
   if(pid == 0)
   {
       printf("child process\n");
       kill(getppid(), SIGBUS);
       printf("child send SIGBUS\n");
   }
   else if(pid != -1)
   {
       while(1)
          sleep(1);
   }
   else
   {
        perror("fork:");
   }

   return 0;
}
```

- raise()函数用户给进程本身发送一个信号，其函数原型为：

```c
#include <signal.h>
int raise(int sig);
```

源代码：raise_test.c

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void handle_sig(int sig)
{
    if(sig == SIGBUS)
    {
        printf("get SIGBUS\n");
    }
}
int main(int argc, char** argv)
{
    signal(SIGBUS, handle_sig);
    raise(SIGBUS);
    return 0;
}
```

- 定时器函数，alarm()专为SIGALRM信号设计，其函数原型是：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

参数`seconds`是定时器的时间，单位为秒，当设置了alarm()之后，在指定的`seconds`秒之后，将给自己发送一个SIGALRM信号，当参数`seconds`为0的时候，将清除当前进程的alrm设置。

调用alarm()函数时，如果进程已经有一个未结束的alarm，那么旧的alarm将被删除，并返回旧的alarm的剩余时间。否则alarm()函数返回0。

源代码: alarm_test.c

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void handle_sig(int sig)
{
        static int i = 0;
        printf("signal alarm %d \n", i++);
        alarm(1);
}
int main(int argc, char** argv)
{

        signal(SIGALRM, handle_sig);
        alarm(1);
        while(1)
        {
             sleep(1);
        }
        return 0;
}
```

- 更强大的定时器函数setitimer()，其函数原型如下：

```
int setitimer(int which, const struct itimerval* value, struct itimerval * ovalue);
```

（1）which参数指定定时器类型

which	含义
ITIMER_REAL	以系统真实的时间来计算，它送出SIGALRM信号。
ITIMER_VIRTUAL	以该进程在用户态下花费的时间来计算，它送出SIGVTALRM信号。
ITIMER_PROF	以该进程在用户态下和内核态下所费的时间来计算，它送出SIGPROF信号。

（2）value参数指定定时时间：

```
struct itimerval {
    struct timeval it_interval; /*每隔多少秒发送一次信号 */
    struct timeval it_value;    /* 第一次定时时间 */
};

struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
};
```

(3)ovalue，old value会存放旧的定时值，一般可以忽略，可以直接填NULL。

源代码：setitimer_test.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void handle_sig(int sig)
{
    static int i = 0;
    printf("signal alarm [%d]\n", i++);
}

int main(int argc, char** argv)
{
    struct itimerval timval;
    timval.it_interval.tv_sec = 1;
    timval.it_interval.tv_usec = 0;
    timval.it_value.tv_sec = 5;
    timval.it_value.tv_usec = 0;
    signal(SIGALRM, handle_sig);
    setitimer(ITIMER_REAL, &timval, NULL);
    while(1)
    {
        sleep(1);
    }
    return 0;

}
```

注意：定时器有且只能有一个，多次设置会覆盖之前的定时设置。

- abort()向进程发送SIGABORT信号，默认情况下进程会异常退出。

1.4 等待信号

```
#include <unistd.h>
int pause(void);
/*
* 阻塞等待某个信号的递达，然后再继续往下，返回对应的信号值，如果递达信号是忽略，则继续挂起。
*/
```

示例代码:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void sig_handle(int sig)
{
    printf("sig %d\n", sig);
}

int main(int argc, char** argv)
{
    signal(SIGINT, sig_handle);
    int sig = pause();
    puts("end pause");
    return 0;
}
```

1.5 信号集

信号集顾名思义就是信号的集合，信号集的类型为sigset_t，每一种信号用1bit来表示，前面我们提到信号有64种，那么这个sigset_t类型至少占64bit,可以通过sizeof(sigset_t)来查看。

每个进程的PCB进程控制块中都有两个信号集，一个叫作未决信号集，一个叫作信号屏蔽字，信号集的每一位不是0就是1，初始状态下，两个信号集的值都为0。

当有信号传递到该进程的时候，未决信号集的对应位设置为1，其他位不变，这个时候信号只是传递到进程，并未被处理，叫作未决状态。常规信号在递达之前产生多次只计一次，而实时信号在递达之前产生多次可以依次放在一个队列里。用户只能获取未决信号集值，无法改变其值。

未决信号想要递达程序的信号处理函数(默认、忽略、自定义)，还要经过信号屏蔽字的过滤，一旦该信号对应bit为1，则该信号将阻塞，不能传递到信号的处理函数。用户可以设置获取信号屏蔽字的值。

- 信号集处理函数

```
#include <signal.h>
int sigemptyset(sigset_t *set) //将set每一位都置0
int sigfillset(sigset_t *set) //将set每一位都置1
int sigaddset(sigset_t *set, int signo) //将set中signo信号对应的bit置1
int sigdelset(sigset_t *set, int signo) //将set中signo信号对应的bit置0
int sigismember(const sigset_t *set, int signo) //判断set中signo信号对应bit是否为1,返回1或者0。
```

- sigprocmask

  调用函数sigprocmask可以读取或更改进程的信号屏蔽字。

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
/*
how参数:
    SIG_BLOCK set包含了我们希望添加到当前信号屏蔽字的信号，相当于mask=mask|set
    SIG_UNBLOCK set包含了我们希望从当前信号屏蔽字中解除阻塞的信号，相当于mask=mask&~set
    SIG_SETMASK 设置当前信号屏蔽字为set所指向的值，相当于mask=set
set参数:传入参数
oset参数:传出参数，获取原先的信号屏蔽字，一般可以填0
返回值：若成功则为0，若出错则为-1
*/
```

如果调用sigprocmask解除了对当前若干个未决信号的阻塞，则在sigprocmask返回前，至少将其中一个信号递达。

- sigpending

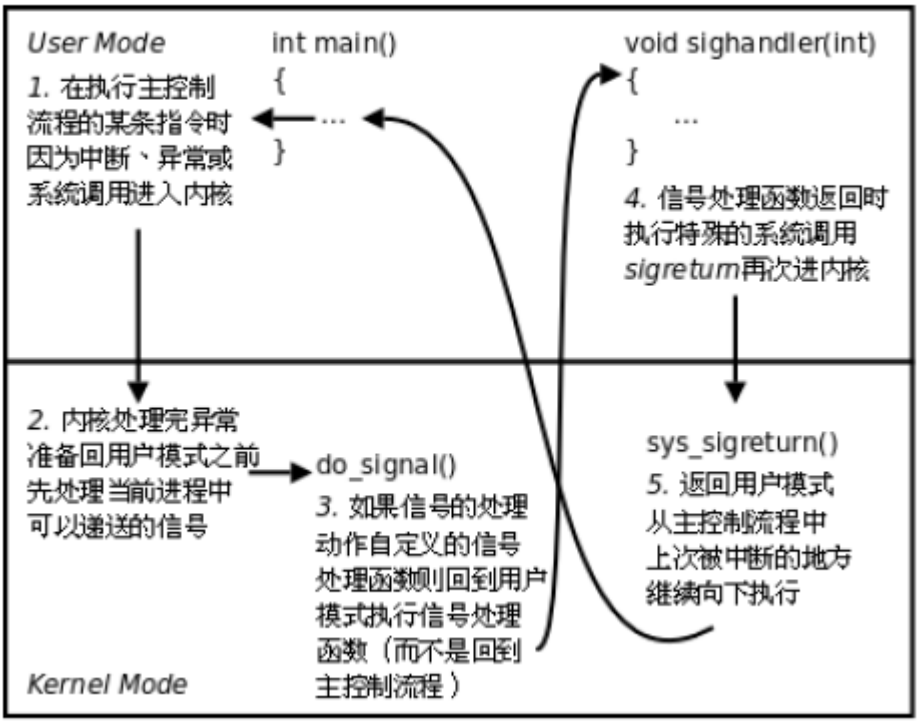
读取当前进程的未决信号集

```
#include <signal.h>
int sigpending(sigset_t *set);
/*
通过set参数传出。调用成功则返回0，出错则返回-1。
*/
```

- 综合事例

```c
#include <signal.h>
#include <stdio.h>
void printsigset(const sigset_t *set)
{
    int i;
    for (i = 1; i < 32; i++)
        if (sigismember(set, i) == 1)
            putchar('1');
        else
            putchar('0');
    puts("");
}
int main(void)
{
    sigset_t s, p;
    sigemptyset(&s);
    sigaddset(&s, SIGINT);  //将信号集s中的SIGINT置为1

    sigprocmask(SIG_BLOCK, &s, NULL); //设置信号屏蔽字
    while (1)
    {
        sigpending(&p);  //获取未决信号集
        printsigset(&p); //打印未决信号集
        sleep(1);
    }
    return 0;
}
```

1.6 信号传递过程



1.7 sigaction函数

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

struct sigaction
{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
/*
sa_handler : 早期的捕捉函数
sa_sigaction : 新添加的捕捉函数，可以传参，和sa_handler互斥，两者通过sa_flags选择采用哪种捕捉函数
sa_mask : 在执行捕捉函数时，设置阻塞其它信号，sa_mask | 进程阻塞信号集，退出捕捉函数后，还原回原有的阻塞信号集
sa_flags : SA_SIGINFO 或者0用来制定调用sa_handler还是sa_sigaction,SA_SIGINFO时为调用sa_sigaction,
           SA_RESTART 让被打断的系统调用重新开始
sa_restorer : 保留，已过时
*/
```

- sigaction例子

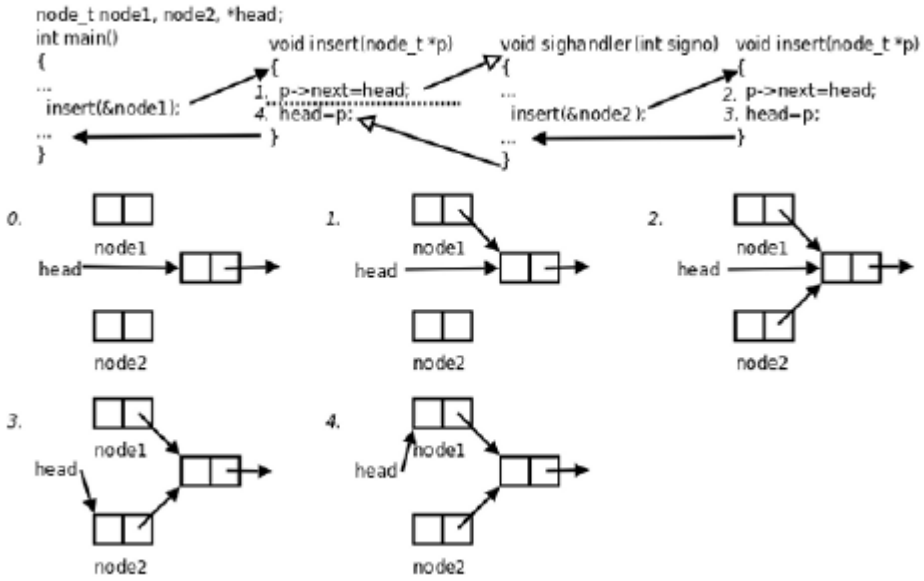
```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void sig_handle(int sig)
{
    puts("recv SIGINT");
    sleep(5);
    puts("end");
}

int main(int argc, char** argv)
{
    struct sigaction act;
    act.sa_handler = sig_handle;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGQUIT); //当进入信号处理函数的时候，屏蔽掉SIGQUIT的递达

    sigaction(SIGINT, &act, NULL);
    while(1)
        sleep(1);
    return 0;
}
```

1.8 read函数的EINTR错误

```c
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
void sig_handle(int sig)
{
    printf("SIGINT\n");
}

int main(int argc, char** argv)
{
    char buf[10];
    struct sigaction act;
    act.sa_handler = sig_handle;
    act.sa_flags = 0;
    //act.sa_flags = SA_RESTART;  //先试一试sa_flags为0时，然后再试一试SA_RESTART的情况
    sigemptyset(&act.sa_mask);

    sigaction(SIGINT, &act, NULL);

    puts("read stdio:");
    int ret = read(STDIN_FILENO,  buf, 9);
    if(ret == -1)
    {
        if(errno == EINTR)
            perror("read:");
    }
    else
    {
        buf[ret] = '\0';
        printf("read %d : %s", ret,  buf);
    }
    return 0;
}
```

以上的运行结果的是，act.sa_flags = 0时一旦发送了一个信号，read将被打断，返回-1，并且设置errno为EINTR。如果想打断后继续运行read，可以设置一下act.sa_flags = SA_RESTART。或者也可以使用signal函数来处理信号，signal相当于默认设置了SA_RESTART标志。

1.9 可重入函数



不含全局变量和静态变量是可重入函数的一个要素, 可重入函数见man 7 signal, 在信号捕捉函数里应使用可重入函数,在信号捕捉函数里禁止调用不可重入函数。

例如：strtok就是一个不可重入函数，因为strtok内部维护了一个内部静态指针，保存上一次切割到的位置，如果信号的捕捉函数中也去调用strtok函数，则会造成切割字符串混乱，应用strtok_r版本，r表示可重入。

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>

static char buf[] = "hello world good book";
void sig_handle(int sig)
{
    strtok(NULL, " ");
}

int main(int argc, char** argv)
{

    signal(SIGINT, sig_handle);

    printf("%s\n", strtok(buf, " "));
    printf("%s\n", strtok(NULL, " "));
    sleep(5);   //可以被信号打断，返回剩余的时间，想想看这个函数应该怎么调用
    printf("%s\n", strtok(NULL, " "));

    return 0;
}
```

运行的时候，发现，通过Ctrl+c发射信号与没发射信号的结果不一样，可以改用strtok_r函数。

1.10 SIGCHLD信号处理

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
```

源代码signal_test.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handle_sig_child()
{
    int status;
    pid_t pid;
    pid = waitpid(-1, &status, WUNTRACED | WCONTINUED);
    printf("recv child pid %d \n", pid);
    if(WIFEXITED(status))
        printf("child process exited with %d\n", WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
        printf("child process signaled with %d\n", WTERMSIG(status));
    else if(WIFSTOPPED(status))
        printf("child process stoped\n");
    else if(WIFCONTINUED(status))
        printf("child process continued\n");
}
int main(int argc, char** argv)
{
    pid_t pid;
    /*捕捉SIGCHLD信号*/
    signal(SIGCHLD, handle_sig_child);
    pid = fork();
    if(pid == 0)
    {
        sleep(5);
        printf("child PID [%d]\n", getpid());
        exit(0);
    }
    else if(pid != -1)
    {
        while(1)
            sleep(1);
    }
    else
    {
        printf("fork error\n");
    }

}
```

运行结果:

```
where@ubuntu:~$ ./sigal_child_test
child PID [9881]
child PID [9881] return [0]
```

2 管道FIFO

2.1 管道基本概念

管道是针对于本地计算机的两个进程之间的通信而设计的通信方法，管道建立后，实际上是获得两个文件描述符：一个用与读取而另一个用于写入。任何从管道写入端写入的数据，可以从管道读取端读出。

管道通信具有以下特点：

- 管道是半双工的，数据只能向一个方向流动，需要双方通信时，要建立起两个管道。

- 管道存放在内存中，是一种独立的文件系统。

2.2 无名管道的创建与读写

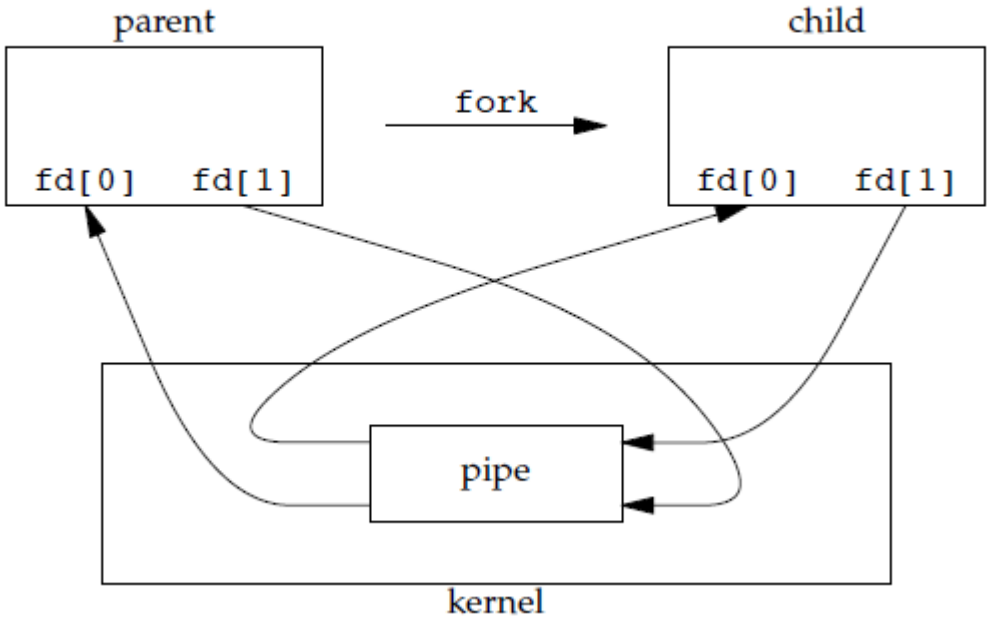
系统调用pipe()用于创建一个管道，其函数原型如下：

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

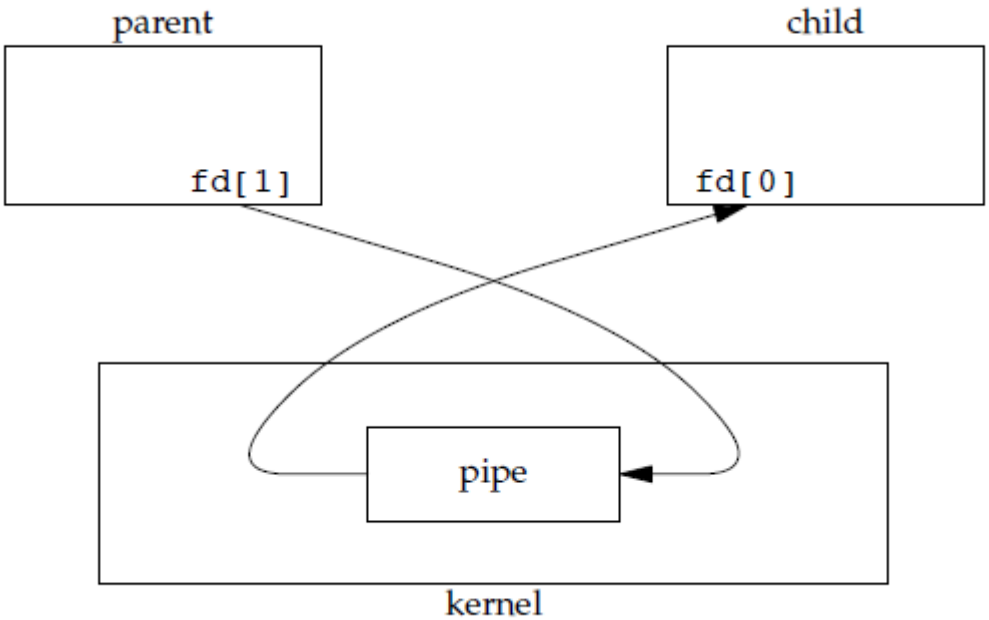
pipe()将建立一对文件描述符，放到参数`pipefd`中。`pipefd[0]`文件描述符用来从管道中读取数据，`pipefd[1]`用于写入数据到管道。

单个进程中的管道几乎没有任何意义，通常，进程会先调用pipe，接着调用fork，从而创建从父进程到子进程的通道。

fork出子进程后，父进程的文件描述表也复制到子进程，于是如下图：



关闭掉父进程的fd[0], 以及子进程的fd[1]，则父进程可以发送消息给子进程，反之亦然。



源代码：pipe_test.c

```c
#include <stdio.h>
#include <unistd.h>

void child_process(int pipefd[])
{
        int i = 0;
        char writebuf[128] = {0};

        /*子进程关闭读文件描述符*/
        close(pipefd[0]);
        while(1)
        {
                sprintf(writebuf, "write pipe : %d", i);

                /*子进程往管道写入数据*/
                write(pipefd[1], writebuf, strlen(writebuf));
                printf("write pipe: %s\n", writebuf);
                i = (i + 1) % 10;
                sleep(1);
        }
}

void father_process(int pipefd[])
{
        char readbuf[128] = {0};

         /*父进程关闭写文件描述符*/
        close(pipefd[1]);
        while(1)
        {
                /*父进程从管道中读取数据*/
                read(pipefd[0], readbuf, sizeof(readbuf));
                printf("read pipe: %s\n", readbuf);
        }
}

int main(int argc, char** argv)
{
        int pipefd[2];
        pid_t pid;

        if( pipe(pipefd) == -1)
        {
                perror("pipe:");
                return -1;
        }

        pid = fork();
        if(pid == 0)
        {
                child_process(pipefd);
        }
        else if(pid != -1)
```

```
        {
                father_process(pipefd);
        }
        else
        {
                perror("fork:");
        }
        return 0;
}
```

运行结果：子进程写数据，父进程读出数据并且打印。

```
where@ubuntu:~$ ./pipe_test
write pipe: write pipe : 0
read pipe: write pipe : 0
write pipe: write pipe : 1
read pipe: write pipe : 1
write pipe: write pipe : 2
read pipe: write pipe : 2
write pipe: write pipe : 3
```

注意：匿名管道只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）。

如果管道读端被关闭，那么这个时候如果继续`write()`管道，会发出信号`SIGPIPE`。如果管道写端被关闭，那么这个时候如果继续`read()`管道直接返回0。

2.3 命名管道FIFO

FIFO是first in first out（先进先出）的缩写，FIFO也称为“命名管道”。FIFO是一种特殊类型的管道，它在文件系统中有一个相应的文件，称为管道文件。

FIFO文件可以通过`mkfifo()`函数创建。在FIFO文件创建之后，任何一个具有适当权限的进程都可以打开FIFO文件。

- mkfifo()函数原型为：

```
#include <unistd.h>
int mkfifo(const char* pathname, mode_t mode);
```

参数：

`pathname`一个FIFO文件的路径名。

`mode`与普通文件`creat()`函数中的mode参数相同。

返回值：

如果要创建的文件已经存在，返回-1， `errno`为`EEXIST`错误, 成功返回0。

一般文件的I/O函数都可以用于FIFO，如open()、read()、write()、close()等等。

- 源代码fifo_write_test.c

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
void handle_sig(int sig)
{
    printf("signal pipe\n");
    exit(-1);
}

int main(int argc, char** argv)
{
    int fd;
    int ret;
    char buf[128];
    int i =  0;

    /*处理管道信号*/
    signal(SIGPIPE, handle_sig);
    if(mkfifo("./fifo", 0640) == -1)
    {
        if(errno !=  EEXIST)
        {
            perror("mkfifo");
            return -1;
        }
    }
    /*只写方式打开管道*/
    fd = open("./fifo", O_WRONLY);
    if(fd == -1)
    {
        perror("open");
        return -1;
    }

    while(1)
    {
        sprintf(buf, "data %d", i++);
        /*往管道写数据*/
        ret = write(fd, buf, strlen(buf));

        printf("write fifo [%d] %s\n", ret, buf);

        sleep(1);
    }
    return 0;
}
```

源码中，通过mkfifo来创建FIFO文件，并且以只写 的方式打开，只有当两边的管道都打开的时候才能写进去，否则阻塞在`write()`函数上，如果管道另一端打开后被关闭，那么这个时候如果继续`write()`FIFO管道，会发出信号`SIGPIPE`.

- 源代码fifo_read_test.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char** argv)
{
    int fd;
    int ret;
    char buf[128];
    if(mkfifo("./fifo", 0640) == -1)
    {
        if(errno !=  EEXIST) /*如果错误类型是fifo文件已经存在，则继续执行*/
        {
            perror("mkfifo");
            return -1;
        }
    }

    /*以只读方式打开管道*/
    fd = open("./fifo", O_RDONLY);
    if(fd == -1)
    {
        perror("open");
        return -1;
    }

    while(1)
    {
        memset(buf, 0, sizeof(buf));
        /*读管道*/
        ret = read(fd, buf, sizeof(buf) - 1);

        printf("read fifo [%d] : %s\n", ret, buf);

        sleep(1);
    }
    return 0;
}
```

源代码中，通过`mkfifo()`函数创建FIFO管道，如果已经存在那么就直接只读方式打开，如果另一端没有被打开，则阻塞在`read()`函数上，如果另一端打开后关闭，则`read()`一直读到EOF也就是0个字节。

2.4 管道容量

管道有它的容量大小，通过man 7 pipe来查看。默认情况下为64k字节，也可以通过fcntl函数来查看：

```
#define _GNU_SOURCE //在任何头文件包含之前添加这个宏定义
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
/*
cmd:F_GETPIPE_SZ 获取管道容量大小，设置管道容量大小F_SETPIPE_SZ
*/
```

例如：

```
#define _GNU_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
    if(mkfifo("./fifo", 0640) == -1)
    {
        if(errno !=  EEXIST)
        {
            perror("mkfifo");
            return -1;
        }
    }
    /*只写方式打开管道*/
    int fd = open("./fifo", O_WRONLY);
    if(fd == -1)
    {
        perror("open");
        return -1;
    }

    printf("pipe size %d\n", fcntl(fd, F_GETPIPE_SZ));
    fcntl(fd, F_SETPIPE_SZ, 4096 * 3); /*必须填写4096的整数倍*/
    printf("pipe size %d\n", fcntl(fd, F_GETPIPE_SZ));
    return 0;
}
```

2.5 注意事项

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（管道写端的引用计数等于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（管道写端的引用计数大于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（管道读端的引用计数等于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。讲信号时会讲到怎样使SIGPIPE信号不终止进程。

4.如果有指向管道读端的文件描述符没关闭（管道读端的引用计数大于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。

2.6 查看系统限制

```
#include <unistd.h>
long pathconf( const char* path, int name );
long fpathconf(int fd, int name);
```

pathconf函数返回配置文件的限制值，是与文件或目录相关联的运行时限制。path参数是你想得到限制值的路径，name是想得到限制值的名称，name的取值主要有以下几个取值：

限制名	说明	name参数
FILESIZEBITS	在指定目录中允许的普通文件最大长度所需的最少位数	_PC_FILESIZEBITS
LINK_MAX	文件链接数的最大值	_PC_LINK_MAX
MAX_CANON	终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名的最大字节数	_PC_NAME_MAX
PATH_MAX	相对路径名的最大字节数，包括null	_PC_PATH_MAX
PIPE_BUF	能原子的写到管道的最大字节数	_PC_PIPE_BUF
SYMLINK_MAX	符号链接中的字节数	_PC_SYMLINK_MAX

3 POSIX IPC

POSIX(Portable Operating System Interface)可移植操作系统接口，IPC机制(Inter-Process Communication,进程间通信)，POSIX IPC主要包括消息队列、信号量、共享内存。

3.1 POSIX信号量

POSIX信号量有两种：有名信号量和无名信号量，无名信号量也被称作基于内存的信号量。有名信号量通过IPC名字进行进程间的同步，而无名信号量如果不是放在进程间的共享内存区中，是不能用来进行进程间同步的，只能用来进行线程同步，线程同步将在后面讲到，这里我们重点研究有名信号量。

3.1.1 创建一个信号量

创建的过程还要求初始化信号量的值。根据信号量取值（代表可用资源的数目）的不同，POSIX信号量还可以分为：

- 二值信号量：信号量的值只有0和1，这和互斥量很类型，若资源被锁住，信号量的值为0，若资源可用，则信号量的值为1；
- 计数信号量：信号量的值在0到一个大于1的限制值（POSIX指出系统的最大限制值至少要为32767）。该计数表示可用的资源的个数。

```
#include <semaphore.h>
sem_t *sem_open(const char *name,int oflag, mode_t mode , int value);
 /*
返回：若成功则为指向信号量的指针，有名字的信号量作为文件被创建在/dev/shm里。若出错则为SEM_FAILED,

name参数：mq_open、sem_open、shm_open这三个函数的name参数都使用“Posix IPC名字”，它可能某个文件系统中的一个真正的路径名，也可能不是。

oflag参数：可以是0、O_CREAT或O_CREAT|O_EXCL。如果指定O_CREAT标志而没有指定O_EXCL，那么只有当所需的信号量尚未存在时才初始化它。但是如果在所需的信号量存在的情况下指定O_CREAT|O_EXCL却会报错。

mode参数：指定权限位。

value参数：指定信号量的初始值。该初始值不能超过SEM_VALUE_MAX（这个常值必须至少为32767）。二值信号量的初始值通常为1，计数信号量的初始值则往往大于1。
*/

int sem_close(sem_t *sem);
/*
 一个进程终止时，内核会对其上打开着的所有有名信号量自动执行关闭操作，不论该进程是自愿终止还是非自愿终止。
  关闭一个信号量并没有将它从系统中删除。Posix有名信号量是随内核持续的。即使当前没有进程打开着某个信号量，他的值仍然保持。
*/

int sem_unlink(const char *name);
/*
    删除创建的信号量文件
*/
```

3.1.2 等待一个信号量

该操作会检查信号量的值，如果其值小于或等于0，那就阻塞，直到该值变成大于0，然后等待进程将信号量的值减1，进程获得共享资源的访问权限。

- 函数`sem_wait()`用来阻塞当前进程直到信号量sem的值大于0，解除阻塞后将sem的值减去1，表明公共资源经使用后减少，其函数原型是：

```
int sem_wait(sem_t *sem);
```

成功返回0，错误返回-1，设置errno

- 函数sem_trywait()与函数sem_wait()的区别是当信号量的值等于0时，sem_trywait()函数不会阻塞当前线程。

```
int sem_trywait(sem_t *sem);
```

成功返回0，错误返回-1，设置errno，当没等到资源的时候，errno为EAGAIN

3.1.3 释放一个信号量

该操作将信号量的值加1，如果有进程阻塞着等待该信号量，那么其中一个进程将被唤醒。

- 函数`sem_post()`用来增加信号量的值，其函数原型是：

```
int sem_post(sem_t* sem);
```

3.1.4 综合例子

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
void error_print(const char* msg)
{
    perror(msg);
    exit(-1);
}

int main(int argc, char** argv)
{
    int ret;

    sem_t* sem = sem_open("/mysem", O_CREAT,  0666, 1);
    if(sem == NULL)
        error_print("sem_open");

    puts("wait semahpore...\n");

    sem_wait(sem);
    puts("get");
    sleep(5);
    puts("post");
    sem_post(sem);

    sem_close(sem);
    return 0;
}
```

编译时需要加上-lpthread：

```
gcc semtest.c -o semtest -lpthread
```

运行两次改程序，观察现象。

3.2 POSIX消息队列

消息队列可以认为是一个链表。进程（线程）可以往里写消息，也可以从里面取出消息。一个进程可以往某个消息队列里写消息，然后终止，另一个进程随时可以从消息队列里取走这些消息。这里也说明了，消息队列具有随内核的持续性，也就是系统不重启，消息队列永久存在。

3.2.1 创建（并打开）、关闭、删除一个消息队列

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
                    struct mq_attr *attr);

/*
函数说明：函数创建或打开一个消息队列，创建后，在/dev/mqueque下可以看到对应文件
返回值：成功返回消息队列描述符，失败返回-1，错误原因存于errno中
name：文件路径，必须以/开头，而且有且只有一个/
oflag：与open的打开标志一样，必须要指定O_RDONLY、O_WRONLY或者O_RDWR其中一个，加上O_CREAT标志，能够创建消息队列
mode：与open的打开mode一样
attr：创建消息队列的时候需要，指定消息队列大小消息个数等，为NULL表示使用默认参数
*/
/*消息队列属性结构体*/
struct mq_attr {
   long mq_flags;       /* Flags: 0 or O_NONBLOCK */
   long mq_maxmsg;      /* Max. # of messages on queue */
   long mq_msgsize;     /* Max. message size (bytes) */
   long mq_curmsgs;     /* # of messages currently in queue */
};

mqd_t mq_close(mqd_t mqdes);
/*
函数说明：关闭一个打开的消息队列，表示本进程不再对该消息队列读写
返回值：成功返回0，失败返回-1，错误原因存于errno中
*/

int mq_unlink(const char *name);
/*
函数说明：删除一个消息队列，好比删除一个文件，其他进程再也无法访问
返回值：成功返回0，失败返回-1，错误原因存于errno中
*/
```

需要注意：

消息队列的名字最好使用“/”打头，并且只有一个“/”的名字。否则可能出现移植性问题；
它必须符合已有的路径名规则（最多由PATH_MAX个字节构成，包括结尾的空字节）。

3.2.2 Posix消息队列读写

消息队列的读写主要使用下面两个函数：

```
#include <mqueue.h>


int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
/*
功能：发送一条消息到队列
mqdes:消息队列文件描述符
msg_ptr: 消息内容
msg_len: 内容长度
msg_prio: 消息优先级,它是一个小于MQ_PRIO_MAX的数，数值越大，优先级越高
返回：若成功则为消息中字节数，若出错则为-1
*/


ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
/*
功能：从队列接收一条消息
mqdes:消息队列文件描述符
msg_ptr: 消息内容
msg_len: 内容长度
msg_prio: 消息优先级
返回：若成功则为0，  若出错则为-1
*/
```

源码sendmq.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

void error_print(const char* msg)
{
     perror(msg);
     exit(-1);
}

/*向消息队列发送消息，消息队列名及发送的信息通过参数传递*/
int main(int argc, char *argv[])
{
    const char* mqname = "/mymq";
    char buf[] = "helloworld";
    mqd_t mqd;
    int ret;

    //只写模式找开消息队列，不存在就创建
    mqd = mq_open(mqname, O_WRONLY | O_CREAT, 0666, NULL);
    if(mqd < 0)
        error_print("mq_open");

    /*向消息队列写入消息，如消息队列满则阻塞，直到消息队列有空闲时再写入*/
    ret = mq_send(mqd, buf, strlen(buf) + 1, 10);
    if(ret < 0)
        error_print("mq_send");

    ret = mq_close(mqd);
    if(ret < 0)
        error_print("mq_close");
    return 0;
}
```

编译：

```
gcc -o sendmq sendmq.c -lrt
```

运行后可以在/dev/mqueue下看到mymq文件。

源码recvmq.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
void error_print(const char* msg)
{
     perror(msg);
     exit(-1);
}
/*读取某消息队列,消息队列名通过参数传递*/
int main(int argc, char *argv[])
{
     const char* mqname = "/mymq";
     mqd_t mqd;
     struct mq_attr attr;
     char *buf;
     unsigned int prio;
     int ret;

     /*只读模式打开消息队列*/
     mqd = mq_open(mqname, O_RDONLY);
     if(mqd < 0)
          error_print("mq_open");

     /*取得消息队列属性，根据mq_msgsize动态申请内存*/
     ret = mq_getattr(mqd, &attr);
     if(ret < 0)
          error_print("mq_getattr");

     /*动态申请保证能存放单条消息的内存*/
     buf = (char*)malloc(attr.mq_msgsize);
     if(NULL == buf)
          error_print("malloc");

     /*接收一条消息*/
     ret = mq_receive(mqd, buf, attr.mq_msgsize, &prio);
     if(ret < 0)
          error_print("receive");

     printf("read data %s  priority %u\n", buf, prio);

     ret = mq_close(mqd);
     if(ret < 0)
         error_print("mq_close");

 /*消息队列使用完后就可以删除
     ret = mq_unlink(mqname);
     if(ret < 0)
         error_print("mq_unlink");
```

```
  */
    free(buf);
    return 0;
}
```

编译并执行：

```
gcc -o recvmq recvmq.c -lrt
```

需要注意的是：当消息不断发送，达到消息队列容量最大值的时候，mq_send将阻塞，直到消息队列被接收走，如果消息还未接收，就把消息队列文件删除，则消息丢失。

3.2.3 消息队列的属性

```
Posix消息队列的属性使用如下结构存放：
struct mq_attr
{
    long mq_flags; /*阻塞标志位，0为非阻塞(O_NONBLOCK)*/
    long mq_maxmsg; /*队列所允许的最大消息条数*/
    long mq_msgsize; /*每条消息的最大字节数*/
    long mq_curmsgs; /*队列当前的消息条数*/
};
/*
队列可以在创建时由mq_open()函数的第四个参数指定mq_maxmsg，mq_msgsize。 如创建时没有指定则使用默认值，一旦创建，则不可再改变。
队列可以在创建后由mq_setattr()函数设置mq_flags
*/

#include <mqueue.h>

/*取得消息队列属性，放到mqstat中*/
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

/*设置消息队列属性，设置值由mqstat提供，原先值写入omqstat中，只是用来改变O_NONBLOCK标志*/
int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat);

均返回：若成功则为0，若出错为-1
```

获取消息队列的属性：

attrmq.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

void error_print(const char* msg)
{
     perror(msg);
     exit(-1);
}
int main()
{
    mqd_t mqd;
    int ret;
    struct mq_attr mqattr;
    const char* mqname = "/mymq";
    /*只读模式打开消息队列*/
    mqd = mq_open(mqname, O_CREAT | O_RDONLY, 0666, NULL);
    if(mqd < 0)
          error_print("mq_open");

    /*取得消息队列属性*/
    ret = mq_getattr(mqd, &attr);
    if(ret < 0)
         error_print("mq_getattr");

    printf("nonblock flag: %ld\n", attr.mq_flags);
    printf("max msgs: %ld\n", attr.mq_maxmsg);
    printf("max msg size: %ld\n", attr.mq_msgsize);
    printf("current msg count: %ld\n", attr.mq_curmsgs);

    ret = mq_close(mqd);
    if(ret < 0)
        error_print("mq_close");

    ret = mq_unlink(mqname);
    if(ret < 0)
        error_print("mq_unlink");

    return 0;
}
```

编译并执行：

```
gcc -o attrmq attrmq.c -lrt
```

设置消息队列的属性：

源码setarrtmq.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

void error_print(const char* msg)
{
     perror(msg);
     exit(-1);
}
int main()
{
    mqd_t mqd;
    int ret;
     const char * mqname = "/mymq";
    struct mq_attr mqattr;
    mqattr.mq_maxmsg = 10; // 注意不能超过系统最大限制，最大限制通过ulimit -a查看
    mqattr.mq_msgsize = 8192;

    mqd = mq_open(mqname, O_RDWR | O_CREAT, 0666, &mqattr);

    if(mqd < 0)
    {
        perror("mq_open");
        exit(1);
    }

    mqattr.mq_flags = O_NONBLOCK;
    mq_setattr(mqd, &mqattr, NULL);// mq_setattr()只关注mq_flags

    /*取得消息队列属性*/
    ret = mq_getattr(mqd, &mqattr);
    if(ret < 0)
         error_print("mq_getattr");

    printf("nonblock flag: %ld\n", mqattr.mq_flags);
    printf("max msgs: %ld\n", mqattr.mq_maxmsg);
    printf("max msg size: %ld\n", mqattr.mq_msgsize);
    printf("current msg count: %ld\n", mqattr.mq_curmsgs);

    ret = mq_close(mqd);
    if(ret < 0)
        error_print("mq_close");

    return 0;
}
```

编译运行：

```
gcc -o setattrmq setattrmq.c -lrt
```

消息队列系统限制查看：

```
cat /proc/sys/fs/mqueue/msg_max #查看消息队列的消息最大长度
cat /proc/sys/fs/mqueue/msgsize_max #查看消息队列的消息最大个数
```

3.3 POSIX共享内存

在Linux中，POSIX共享内存对象驻留在tmpfs伪文件系统中。系统默认挂载在/dev/shm目录下。当调用shm_open函数创建或打开POSIX共享内存对象时，系统会将创建/打开的共享内存文件放到/dev/shm目录下。

创建共享内存的基本步骤是：

1. 程序执行shm_open函数创建了共享内存区域,此时会在/dev/shm/创建共享内存文件.

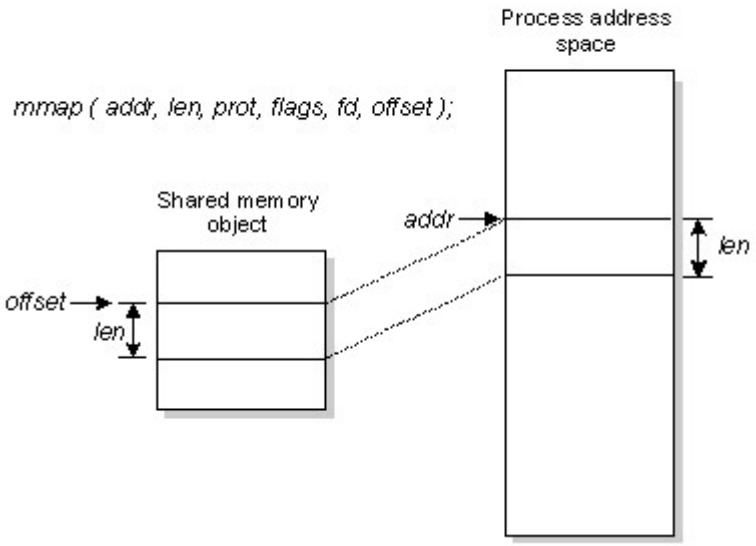
```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
/*
创建或打开一个共享内存,成功返回一个整数的文件描述符，错误返回-1。
1.name:共享内存区的名字;
2.oflag标志位: open的标志一样，一般填写O_CREAT|O_TRUNC|O_RDWR。
3.mode权限位:  open的mode一样
*/
```

2. 通过ftruncate函数改变shm_open创建共享内存的大小为页大小sysconf(_SC_PAGE_SIZE)整数倍,如果不执ftruncate函数的话，会报Bus error的错误。

```
#include <unistd.h>
int ftruncate(int fd, off_t length);

/*函数说明：ftruncate()会将参数fd指定的文件大小改为参数length指定的大小。参数fd为已打开的文件描述词，而且必须是以写入模式打开的文件。如果原来的文件大小比参数length大，则超过的部分会被删去。
返 回  值：0、-1
错误原因：errno
          EBADF      参数fd文件描述词为无效的或该文件已关闭
          EINVAL     参数fd为一socket并非文件，或是该文件并非以写入模式打开
 */
```

1. 通过mmap函数将创建的共享内存文件映射到内存。

```
#include <sys/mman.h>
void* mmap(void* start,size_t length,int prot,int flags,int fd,off_t offset);

/*函数说明：mmap()必须以PAGE_SIZE为单位进行映射。
  start:      映射区的开始地址，设置为0时表示由系统决定映射区的起始地址。
  length:     映射区的长度，长度单位是以字节为单位，不足一内存页按一内存页处理
  prot:       期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过或运算合理地组合在一起
              PROT_EXEC  页内容可以被执行
              PROT_READ  页内容可以被读取
              PROT_WRITE 页可以被写入
              PROT_NONE  页不可访问
  flags:     指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体
              MAP_SHARED 与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。直到msync()或者munmap()被调用，文件实际上不会被更新。
              MAP_PRIVATE 建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。
              MAP_LOCKED 锁定映射区的页面，从而防止页面被交换出内存。
  fd:        有效的文件描述符。
  offset:    被映射对象内容的起点。

  成功返回共享内存地址，失败返回MAP_FAILED
*/
```



2. 通过munmap卸载共享内存

```
int munmap(void* start,size_t length);
/*
* start:共享内存地址
* length:共享内存大小
*/
```

3. 通过shm_unlink删除内存共享文件

```
int shm_unlink(const char *name);
/*
*name:内存共享文件
*/
```

我们用下面的源程序对POSIX共享内存进行测试,如下shmen_write.c:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/wait.h>
void error_print(char *msg)
{
        perror(msg);
        exit(-1);
}

int main (int argc, char *argv[])
{
        int ret, i;
        const char *memname = "/mymem";
        size_t mem_size = sysconf(_SC_PAGE_SIZE);
        int fd = shm_open(memname, O_CREAT|O_RDWR, 0666);
        if (fd == -1)
            error_print("shm_open");
        /*映射之前必须要截断*/
        ret = ftruncate(fd, mem_size);
        if (ret != 0)
             error_print("ftruncate");

        void *ptr = mmap(0, mem_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        if (ptr == MAP_FAILED)
             error_print("MMAP");
        close(fd);
        for(i = 0; i < 20; i++)
        {
            sprintf((char*)ptr, "data %d", i);
            printf("write data %s\n", (char*)ptr);
        }
        ret = munmap(ptr, mem_size);
        if (ret != 0)
             error_print("munmap");
        ret = shm_unlink(memname);
        if (ret != 0)
             error_print("shm_unlink");
        return 0;
}
```

编译程序shmen_write.c

```
gcc shmen_write.c -o shmen_write -lrt
```

shmen_read.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/wait.h>
void error_print(char *msg)
{
        perror(msg);
        exit(-1);
}

int main (int argc, char *argv[])
{
        int ret, i;
        const char *memname = "/mymem"; //共享内存文件以/开头，可以更好兼容不同系统
         struct stat statbuf;
        size_t mem_size = 2 * sysconf(_SC_PAGE_SIZE);
        int fd = shm_open(memname, O_CREAT|O_RDWR, 0666);
        if (fd == -1)
            error_print("shm_open");
        ret = ftruncate(fd, mem_size);
        if (ret != 0)
             error_print("ftruncate");

        void *ptr = mmap(0, mem_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        if (ptr == MAP_FAILED)
             error_print("MMAP");
        close(fd);

         for(i = 0; i < 20; i++)
        {
            printf("read data %s\n", (char*)ptr);
             sleep(1);
        }
        ret = munmap(ptr, mem_size);
        if (ret != 0)
             error_print("munmap");
         ret = shm_unlink(memname);
        if (ret != 0)
             error_print("shm_unlink");
        return 0;
}
```

编译程序shmen_read.c

```
gcc shmen_read.c -o shmen_read -lrt
```

先运行shmem_write，再运行shmem_read.

第五章 Linux 进程关系

1 进程组

进程组就是一个或多个进程的集合,进程组ID是一个正整数，每个进程组都有一个组长进程，组长进程的进程号等于进程组ID。组长进程可以创建一个进程组、创建该组中的进程。

进程组生存期：进程组创建，到最后一个进程离开进程组(终止或转移到另一个进程组)。

一个进程可以为自己或为其子进程设置进程组ID。

进程组可被分为一个前台进程组和一个或多个后台进程组。为什么要这么分呢？前台进程组是指需要与终端进行交互的进程组（只能有一个）。

比如有些进程是需要完成IO操作的，那么这个进程就会被设置为前台进程组.当我们键入终端的中断键和退出键时，就会将信号发送到前台进程组中的所有进程。而后台进程组是指不需要与终端进程交互的进程组，比如：一些进程不需要完成IO 操作，或者一些守护进程就会被设置为后台进程组。

再比如，可以直接向一个进程组发送信号，例如：

pkill_test.c源码：

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    int pid = fork();
    if(pid == 0)
    {
        while(1)
            sleep(1);
    }
    else if(pid > 0)
    {
        while(1)
            sleep(1);
    }
    return 0;

}
```

```
where@ubuntu:~$ ./pkill_test &
[1] 26327
where@ubuntu:~$ ps -j
  PID  PGID   SID TTY          TIME CMD
25165 25165 25165 pts/0    00:00:00 bash
26327 26327 25165 pts/0    00:00:00 pkill_test
26328 26327 25165 pts/0    00:00:00 pkill_test
26330 26330 25165 pts/0    00:00:00 ps
where@ubuntu:~$ kill -9 -26327
[1]+  已终止               ./pkill_test
where@ubuntu:~$ ps -j
  PID  PGID   SID TTY          TIME CMD
25165 25165 25165 pts/0    00:00:00 bash
26333 26333 25165 pts/0    00:00:00 ps
```

运行pkill_test，pkill_test通过fork创建了一个子进程，父子进程在同一个组内，这时可以通过pkill -g把这一个小组的进程全部杀死。

- 进程组标识PGID(process group ID):用于进程所属的进程组ID。

```
pid_t getpgrp(), 获取当前进程所在的进程组的ID
pid_t getpgid(pid_t pid), pid=0 获取代表当前进程的组ID，pid>0时，获取指定进程pid的组id
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    pid_t pid;
    if ((pid = fork()) < 0)
    {
        perror("fork");
        exit(1);
    }
    else if (pid == 0)
    {
        printf("child process PID is %d\n",getpid());
        printf("Group ID is %d\n",getpgrp());
        printf("Group ID is %d\n",getpgid(0));
        printf("Group ID is %d\n",getpgid(getpid()));
        exit(0);
    }
    sleep(3);
    printf("parent process PID is %d\n",getpid());
    printf("Group ID is %d\n",getpgrp());
    return 0;
}
```

- 设置进程组

  使用`setpgid()`加入一个现有的进程组或创建一个新进程组

```
int setpgid(pid_t pid, pid_t pgid)
```

setpgid将pid进程所属的进程组ID设为参数pgid指定的进程组ID。

如果参数pid 为0，则会设置当前进程的进程组ID。

如果参数pgid为0，则由pid指定的进程ID将用作进程组ID。

一个进程只能为它自己或它的子进程设置进程组ID。

如改变子进程为新的组，应在fork后，exec前使用。

源码：setpgid_test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char** argv)
{
    pid_t pid;
    if ((pid = fork()) < 0)
    {
        perror("fork");
        exit(-1);
    }
    else if (pid == 0)
    {
        printf("child process PID  [%d]\n",getpid());
        printf("child group ID  [%d]\n",getpgid(0)); // 返回组id
        sleep(5);
        printf("child group ID  [%d]\n",getpgid(0));
        exit(0);
    }
    sleep(1);
    setpgid(pid,pid); // 父进程改变子进程的组id为子进程本身
    sleep(5);
    printf("parent process PID  [%d]\n", getpid());
    printf("parent's parent process PID is [%d]\n", getppid());
    printf("parent group ID [%d]\n",getpgid(0));
    setpgid(getpid(),getppid()); // 改变父进程的组id为父进程的父进程
    printf("parent group ID [%d]\n",getpgid(0));
    return 0;
}
```

2 终端

linux 的终端就是控制台， 是用户与内核交互的平台， 通过输入指令来控制内核完成任务操作。外形是一个方框，有光标在闪烁。

在Linux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），控制终端是保存在task_struct中的信息，而我们知道fork会复制task_struct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，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。

```
init-->fork-->exec-->agetty-->login-->用户输入帐号-->输入密码-->fork-->shell
```

ttyname函数可以由文件描述符查出当前进程对应的终端设备文件名，该文件描述符必须指向一个终端设备而不能是任意文件。下面我们通过实验看一下各种不同的终端所对应的设备文件名。

```
#include <unistd.h>
#include <stdio.h>
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;
}
```

登录后的shell其输入输出是连接到用户使用的终端的，不管是本地登录的tty，还是远程登录的pty。但是，为什么要有终端呢？shell的输入直接接到键盘、输出直接接到显示器，这样不行么？尤其是远程的情况，shell的输入输出为什么不能直接接到网络，而非要弄一个pty出来呢？　　最容易想到的一点，终端能够使得上层不必关心输入输出设备本身的细节，只管对其读写就行了。不过这一点似乎并不是终端所特有的，因为VFS已经能够胜任了。应用程序open设备文件，得到fd，然后同样只用管对其读写就行了，而不用关心这个fd代表的是键盘、还是普通文件，具体的细节已经被隐藏在设备驱动程序之中。

不过，相比于普通的读写，终端还实现了很多对输入输出的处理逻辑。如：

1、回车换行的转换：定义输入输出如何映射回车换行符。比如：回车键是\r、还是\n、还是\r\n；再如：\n应该如何打印到屏幕上，是回车+换行、还是只换行不回车、等等；

2、行编辑：允许让输入字符不是立马送到应用程序，而是在换行以后才能被读取到。未换行的输入字符可以通过退格键进行编辑（比如在你密码输错的时候，是可以用CTRL+退格来进行编辑的）；

3、回显：可以让输入字符自动被回显到终端的输出上。于是，键盘每输入一个字符都能在显示器上看到它，而这些字符其实很可能是还没被应用程序读取到的（因为有行编辑）；

4、功能键：允许定义功能键。比如最常用的Ctrl+C，杀死前台进程，就是由终端来触发的。终端检测到Ctrl+C输入，会向前台进程组发送SIGTERM信号

5、输入输出流向控制，只有前台进程组能够从终端中读数据、而前台后台程序都能向终端写数据。这点也是必须的，跟用户进程交互的是前台进程，用户的输入当然不能被其他后台进程抢走。但是一个进程是前台还是后台，是它自己所不知道的，没法靠进程自己来判断什么时候可以读终端、什么时候不能读。所以需要终端来提供支持，如果后台进程读这个终端，终端的驱动程序将向其发送SIGTTIN信号，从而将其挂起。直到shell将其重新置为前台进程时（通过fg命令），该进程才会继续执行；

可以说，终端是人机交互时，应用程序与用户之间的一个中间层。如果应用程序是在跟人交互，使用终端是其不二的选择；否则则没有必要使用终端。

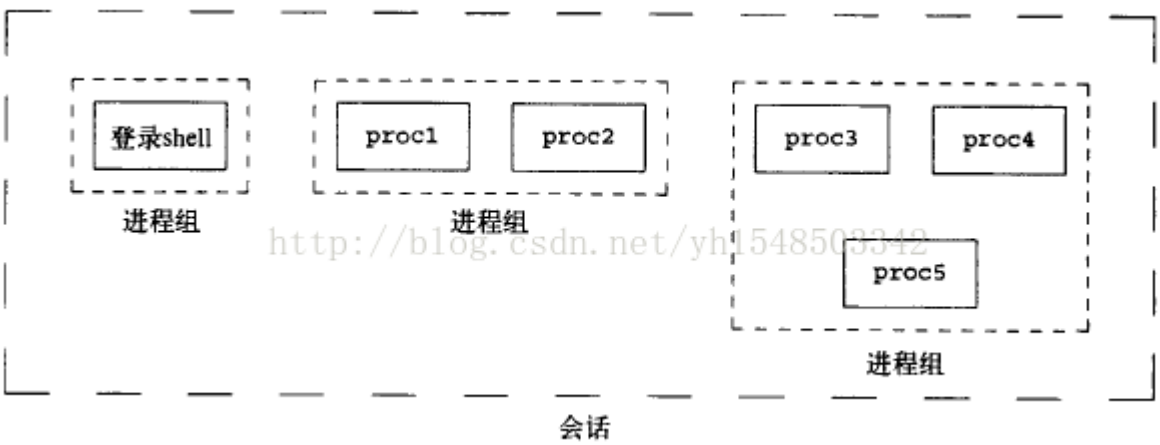
3 会话

- 会话是基于连接的。会话的源头，就是用户与系统之间连接的启用。

  用户登录是一个会话的开始。登录之后，用户会得到一个跟用户使用的终端相连的进程，这个进程被称作是这个会话的leader，会话的id就等于该进程的pid。由该进程fork出来的子进程都是这个会话的成员。leader进程的退出，将导致它所连接的终端被hangup，这意味着会话结束。并向所有的会话中的进程发射信号SIGHUP挂起信号。默认信号处理的进程退出。

  一个会话(session)只能有一个终端，称为controlling terminal，此外，建立或者改变终端与会话的联系只能由会话领导者(session leader)来进行。会话可以有终端也可以没有终端。

  一个会话又可以包含多个进程组。一个会话对应一个控制终端



- 查看会话ID

```
pid_t getsid(pid_t pid)
```

pid为0表示察看当前进程session ID，`ps -ajx`命令查看系统中的进程。参数a表示不仅列当前用户的进程，也列出所有其他用户的进程，参数x表示不仅列有控制终端的进程，也列出所有无控制终端的进程，参数j表示列出与作业控制相关的信息。

组长进程不能成为新会话leader进程，新会话leader进程必定会成为组长进程。

- 设置会话ID

```
pid_t setsid(void)
```

1. 当进程是会话的领头进程时setsid()调用失败并返回（-1）,setsid()调用成功后，返回新的会话的ID。

2.调用setsid函数的进程成为新的会话的领头进程，

3.该进程会成为一个新进程组的组长进程

4.与其父进程的会话组和进程组脱离。

5.由于会话对控制终端的独占性，进程同时与控制终端脱离。

想要满足以上条件，建立新会话时，先调用fork, 父进程终止，子进程调用setsid。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;
    if ((pid = fork())<0)
    {
        perror("fork");
        exit(-1);
    }
    else if (pid == 0)
    {
        printf("child process PID [%d]\n", getpid());
        printf("child group ID [%d]\n", getpgid(0));
        printf("child session ID [%d]\n", getsid(0));
        sleep(5);
        setsid(); // 子进程非组长进程，故其成为新会话leader进程，且成为组长进程。该进程组id即为会话进程
        printf("changed:\n");
        printf("child process PID [%d]\n", getpid());
        printf("child group ID [%d]\n", getpgid(0));
        printf("child session ID [%d]\n", getsid(0));
        while(1)
        {
            sleep(1);
        }
    }
    return 0;
}
```

4 守护进程

4.1 概念

守护进程（Daemon）是运行在后台的一种特殊进程。它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。守护进程是一种很有用的进程。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。同时，守护进程完成许多系统任务。比如，作业规划进程crond，打印进程lpd等。（这里的结尾字母d就是Daemon的意思）。

4.2 如何创建守护进程

守护进程编程步骤

1. 创建子进程，父进程退出

   为避免挂起控制终端将Daemon放入后台执行。方法是在进程中调用fork使父进程终止，让Daemon在子进程中后台执行。并且获得一个进程组ID不是自己的进程。如：在前台会收到SIGSTOP信号。

2. 在子进程中创建新会话

   setsid()函数，使子进程完全独立出来，脱离控制。

3. 改变当前目录为根目录

   chdir()函数，防止占用可卸载的文件系统，也可以换成其它路径。进程活动时，其工作目录所在的文件系统不能卸下。一般需要将工作目录改变到根目录。

4. 重设文件权限掩码

   进程从创建它的父进程那里继承了文件创建掩码，调用umask()函数设置为0,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。

5. 关闭文件描述符

   继承的打开文件不会用到，浪费系统资源，无法卸载。

6. 开始执行守护进程核心工作

7. 守护进程退出处理

   释放内存申请，占用的系统资源的释放。

源码：定时写log的守护进程log_test.c

```c
#include<unistd.h>
#include<signal.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/param.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<time.h>
void init_daemon()
{
    int pid;
    int i;
    pid=fork();
    if(pid < 0)
    {
        perror("fork");
        exit(-1);  //创建错误，退出
    }
    else if(pid>0) //父进程退出
    {
        exit(0);
    }
    /* 子进程创建新的会话 */
    setsid();
    /* 改变当前工作目录到/目录下.*/
    if (chdir("/") < 0)
    {
        perror("chdir");
        exit(1);
    }
    /* 设置umask为0 */
    umask(0);
    /*关闭进程打开的文件句柄*/
    for(i=0;i<NOFILE;i++)
    {
        close(i);
    }
    return;
}

int main(int argc, char** argv)
{
    FILE *fp;
    time_t t;
    init_daemon();
    while(1)
    {
        sleep(60); //等待一分钟再写入
        fp=fopen("/home/where/test.log","a+");
        if(fp>=0)
        {
            time(&t);
            fprintf(fp,"current time is:%s\n",ctime(&t));  //转换为本地时间输出
```

```
            fclose(fp);
        }
    }
    return 0;
}
```

在用户目录里面创建一个test.log文件，并且写日志，test.log内容为

```
 current time is:Sun Aug 14 01:26:41 2016

current time is:Sun Aug 14 01:26:42 2016

current time is:Sun Aug 14 01:26:43 2016

current time is:Sun Aug 14 01:26:44 2016

current time is:Sun Aug 14 01:26:45 2016
```

第六章 Linux多线程

多线程是在同一时间需要完成多项任务的时候实现的。

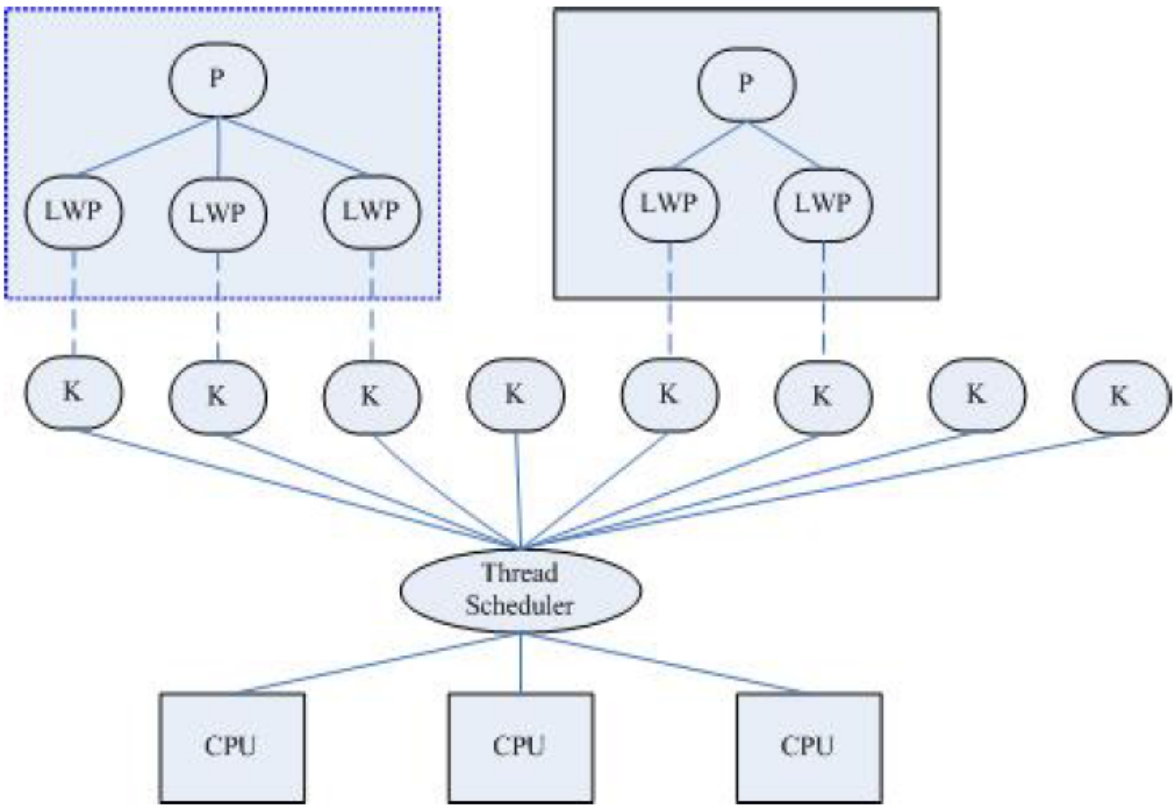
1.轻量级进程(light-weight process)，每个线程也有自己与进程控制表和 PCB 相似的线程控制表 TCB ，而这个 TCB 中所保存的线程状态信息则要比 PCB 表少得多，这些信息主要是相关栈指针（系统栈和用户栈），寄存器中的状态数据。

2.从内核里看进程和线程是一样的，都有各自不同的PCB或者TCB，但是PCB或者TCB中指向内存资源的三级页表是相同的。

3.进程可以蜕变成线程。

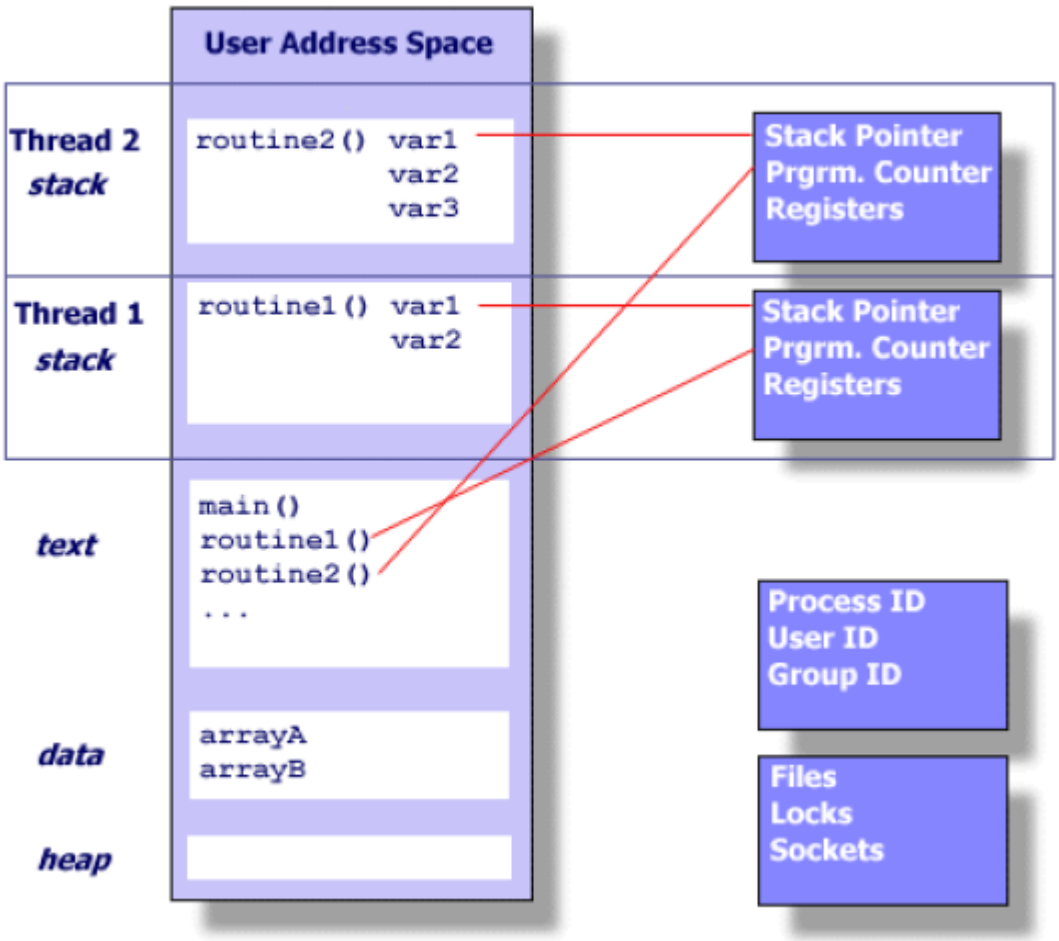
4.在linux下，线程最是小的执行单位；进程是最小的分配资源单位。

```
ps -eLf  查看线程
```



- 线程间共享资源

```
1.文件描述符表
2.当前工作目录
3.用户ID和组ID
4.内存地址空间
    Text
    data
    bss
    堆
    共享库
```



- 线程间非共享资源

```
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
```

- 线程优点

```
1.提高程序的并发性
2.开销小，不用重新分配内存
3.通信和共享数据方便
```

- 线程缺点

```
1.线程不稳定（库函数实现）
2.线程调试比较困难（gdb支持不好）
3.线程无法使用unix经典事件，例如信号
```

- man帮助

查看manpage关于pthread的函数

```
man -k pthread
```

安装pthread相关manpage

```
sudo apt-get install manpages-posix manpages-posix-dev
```

1 线程的创建和使用

1.1 创建线程

用pthread_create()函数，其函数原型是：

```
#include <pthread.h>
int pthread_create(pthread_t*  thread,
                   const pthread_attr_t * attr,
                   void* (*start_routine)(void*), void*  arg);
```

第一个参数为指向线程标识符的指针，第二个参数用到设置线程属性，分配多少的栈空间，以及运行优先级，大多数情况下，我们不需要设置线程的属性，那么空指针NULL就可以了，第三个参数是线程运行函数的起始地址，最后一个参数是要运行在线程中的函数指针。

当创建线程成功时，函数返回0，失败返回错误码。不会设置errno。

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。

start_routine函数接收一个参数，是通过pthread_create的arg参数传递给它的，该参数的类型为void *，这个指针按什么类型解释由调用者自己定义。start_routine的返回值类型也是void *，这个指针的含义同样由调用者自己定义。

start_routine返回时，这个线程就退出了，其它线程可以调用pthread_join得到start_routine的返回值，类似于父进程调用wait(2)得到子进程的退出状态。

pthread_create成功返回后，新创建的线程的id被填写到thread参数所指向的内存单元。我们知道进程id的类型是pid_t，每个进程的id在整个系统中是唯一的，调用getpid(2)可以获得当前进程的id，是一个正整数值。线程id的类型是thread_t，它只在当前进程中保证是唯一的，在不同的系统中thread_t这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用printf打印，调pthread_self(3)可以获得当前线程的id。

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Linux中多线程的实现可以使用pthread线程模型的编程。pthread线程通过libpthread线程函数库来实现。所以在编译多线程程序的时候，需要链接libpthread，比如

```
gcc -o pthread_test -lpthread
```

源代码: pthread_test.c

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
void* mythread(void * arg)
{
    int i;
    for(i = 0; i < 5; i++)
    {
        printf("pthread\n");
        sleep(1);
    }
    return (void*)0;
}
int main(int argc, char** argv)
{
    pthread_t pth;
    int i, ret;
    ret = pthread_create(&pth, NULL, mythread, NULL);
    if(ret != 0)
    {
        printf("create pthread error\n");
        exit(1);
    }

    for(i = 0; i < 5; i++)
    {
        printf("main process\n");
        sleep(1);
    }
    return 0;
}
```

上面的代码，第二个参数我们设置为空指针，这样将生成默认属性的线程。函数mythread如果不需要参数，那么最后一个参数也可以设置为空指针。

运行结果：

```
where@ubuntu:~$ ./pthread_test
main process
pthread
main process
pthread
main process
pthread
main process
pthread
main process
pthread
```

两个线程的输出是交替的。

1.2 错误打印

```
/*不可重入错误信息打印*/
char *strerror(int errnum);
/*可以在多线程中使用，可以重入*/
int strerror_r(int errnum, char *buf, size_t buflen);
char *strerror_r(int errnum, char *buf, size_t buflen);
```

由于标准错误也只有一个，当多个线程同时往标准错误输出信息时，会造成竞争冒险。strerror打印信息紊乱，这个使用需要使用strerror_r 来打印出错信息。

1.3 线程的退出

一种是子线程程序运行完自动退出或者直接return，一种是调用pthread_exit(),其函数原型为：

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

函数的参数时线程退出时的返回码，只要pthread_join中的第二个参数value_ptr不是NULL，这个值将被传递给主线程。

调用线程退出函数，注意和exit函数的区别，任何线程里exit导致进程退出，其他线程未工作结束，主控线程退出时不能return或exit。

需要注意，pthread_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

源代码:pthread_test2.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
void* mythread(void * arg)
{
    int i;
    for(i = 0; i < 5; i++)
    {
        printf("pthread\n");
        sleep(1);
    }
    pthread_exit((void *)i);
}
int main(int argc, char** argv)
{
    pthread_t pth;
    int i, ret;
    void* result;
    ret = pthread_create(&pth, NULL, mythread, NULL);
    if(ret != 0)
    {
        printf("create pthread error\n");
        exit(1);
    }

    for(i = 0; i < 5; i++)
    {
        printf("main process\n");
        sleep(1);
    }
    pthread_join(pth, &result);
    printf("pthread exit code = [%d]\n", (int)result);
    return 0;
}
```

运行结果：

```
where@ubuntu:~$ ./pthread_test
main process
pthread
main process
pthread
main process
pthread
main process
pthread
main process
pthread
pthread exit code = [5]
```

1.4 线程回收

linux线程执行有两种状态joinable状态和unjoinable状态。

1、如果线程是joinable状态，当线程函数自己返回退出时或pthread_exit时都不会释放线程所占用堆栈和线程描述符。只有当你调用了pthread_join之后这些资源才会被释放。

函数pthread_join(),这个函数是主线程用于等待子线程结束，其函数原型是：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用该函数将一直等待到被等待的线程结束为止，这函数返回时，被等待线程的资源被回收。

2、若是unjoinable状态的线程，这些资源在线程函数退出时或`pthread_exit`时自动会被释放。

unjoinable属性可以在pthread_create时指定，或在线程创建后在线程中pthread_detach自己,如：`pthread_detach(pthread_self())`，将状态改为unjoinable状态，确保资源的释放。其实简单的说就是在线程函数头加上`pthread_detach(pthread_self())`的话，线程状态改变，在函数尾部直接pthread_exit线程就会自动退出。

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

pthread_t tid : 分离线程tid
返回值：成功返回0，失败返回错误号。

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
void *my_thread(void *arg)
{
    int i = 3;
    while (i--)
    {
        printf("thread count %d\n", n);
        sleep(1);
    }
    return (void *)1;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int err;
    pthread_create(&tid, NULL, my_thread, NULL);
    /*第一次运行时注释掉下面这行，第二次再打开，分析两次结果*/
    pthread_detach(tid);
    err = pthread_join(tid, &tret);
    if (err != 0)
        printf("thread %s\n", strerror_r(err));
    else
        printf("thread exit code %d\n", (int)tret);
    sleep(1);
    return 0;
}
```

1.5 pthread_cancel

在进程内某个线程可以取消另一个线程。

```c
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_testcancel();
```

被取消的线程，退出值，定义在Linux的pthread库中常数PTHREAD_CANCELED的值是-1。可以在头文件pthread.h中找到它的定义：

```c
#define PTHREAD_CANCELED ((void *) -1)
```

源码：pthread_cancel.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void *th_func(void *arg)
{
    while(1)
    {
        printf("thread run\n");
        sleep(1);
    }
}
int main(void)
{
    pthread_t tid;
    void *ret;
    pthread_create(&tid, NULL, th_func, NULL);
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, &ret);
    printf("thread  exit code %d\n", (int)ret);
    return 0;
}
```

同一进程的线程间，pthread_cancel向另一线程发终止信号。系统并不会马上关闭被取消线程，只有在被取消线程下次系统调用时，才会真正结束线程。或调用pthread_testcancel，让内核去检测是否需要取消当前线程。

1.6 pthread_equal

比较两个线程是否相等

```c
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

2 线程属性

本节作为指引性介绍，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 detachstate; //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;
```

注：目前线程属性在内核中不是直接这么定义的，抽象太深不宜拿出讲课，为方便大家理解，使用早期的线程属性定义，两者之间定义的主要元素差别不大。

2.1 线程属性初始化

先初始化线程属性，再pthread_create创建线程。

```
#include <pthread.h>
pthread_attr_t attr;
int pthread_attr_init(pthread_attr_t *attr); //初始化线程属性
int pthread_attr_destroy(pthread_attr_t *attr); //销毁线程属性所占用的资源
```

2.2 线程的分离状态

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);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

`pthread_attr_t *attr`：被已初始化的线程属性。

`int *detachstate`： 可选为`PTHREAD_CREATE_DETACHED`（分离线程）和`PTHREAD _CREATE_JOINABLE`（非分离线程）。

2.3 线程的栈大小（stack size）

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。

函数pthread_attr_getstacksize和pthread_attr_setstacksize提供设置。

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
```

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：若是成功返回0,否则返回错误的编号。

除上述对栈设置的函数外，还有以下两个函数可以获取和设置线程栈属性,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。传给pthread_attr_setstack函数的地址是缓冲区的低地址（不一定

是栈的开始地址，栈可能从高地址往低地址增长）。

```
#include <pthread.h>
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);
int pthread_attr_getstack(pthread_attr_t *attr, void **stackaddr, size_t *stacksize);
```

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：若是成功返回0,否则返回错误的编号

3 线程同步-互斥锁

当我们需要控制对共享资源的存取的时候，可以用一种简单的加锁的方法来控制。下面来看一个造成值不可知的情况。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NLOOP 5000
static int counter;

void *mythread(void *vptr)
{
    int i, val;
    for (i = 0; i < NLOOP; i++)
    {
        val = counter;
        printf("%x: %d\n", (unsigned int)pthread_self(), ++val);
        counter = val;
    }
    return NULL;
}


int main(int argc, char **argv)
{
    pthread_t tidA, tidB;
    pthread_create(&tidA, NULL, &mythread, NULL);
    pthread_create(&tidB, NULL, &mythread, NULL);
    /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);
    return 0;
}
```

在pthread.h中，互斥锁为结构体`pthread_mutex_t`，并且可以使用动态初始化`pthread_mutex_init()`来初始化一个锁，其函数原型为:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t* attr);
```

第一个参数是互斥锁的地址，第二个参数设置互斥锁的属性，大多数情况下，选择默认属性，传递NULL。

也可以使用静态初始化一把锁。

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- 分别使用`pthread_mutex_lock()`上锁，`pthread_mutex_unlock()`解锁。这个两个函数的原型是:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`pthread_mutex_lock()`函数声明开始用互斥锁上锁，此后代码直至调用pthread_mutex_unlock()为止，均被上锁，即同一时间只能被一个线程调用执行，当一个线程执行到pthread_mutex_lock()处时，如果该锁此时被另一个线程使用，那么此线程被阻塞，线程一直阻塞直到另一个线程释放此互斥锁。

两个函数的参数都是互斥变量的地址，函数执行成功返回0，否则返回错误值。

源代码: mutex_test.c

```c
#include <stdio.h>
#include <pthread.h>
static value = 0;
static pthread_mutex_t mutex;
void* mythread(void* arg)
{
    /*上锁*/
    printf("mythread begin lock\n");
    pthread_mutex_lock(&mutex);

    printf("mythread locked\n");
    sleep(5);
    printf("mythread get value = [%d]\n", value++);

    /*解锁*/
    pthread_mutex_unlock(&mutex);
    printf("mythread unlocked\n");
    pthread_exit((void*)0);
}


int main(int argc, char** argv)
{
    int ret;
    pthread_t pth;
    pthread_mutex_init(&mutex, NULL);
    ret =pthread_create(&pth, NULL, mythread, NULL);
    if(ret != 0)
    {
        printf("create thread:%s\n", strerror_r(ret));
        return -1;
    }

    /*主线程上锁*/
    printf("main thread begin lock\n");
    pthread_mutex_lock(&mutex);

    printf("main thread locked\n");
    sleep(5);
    printf("main thread get value = [%d]\n", value++);

    /*主线程解锁*/
    pthread_mutex_unlock(&mutex);
    printf("main thread unlocked\n");


    pthread_join(pth, NULL);
    return 0;
}
```

执行后输出为:

```
where@ubuntu:~$ ./pthread_test
main thread begin lock
main thread locked
mythread begin lock
main thread get value = [0]
main thread unlocked
mythread locked
mythread get value = [1]
mythread unlocked
```

主线程先抢到锁，然后子线程阻塞等待直到获取到锁资源。

- 非阻塞上锁`pthread_mutex_trylock`，成功获得锁返回0，其他任何值都是没获取到锁。

其函数原型为：

```
int pthread_mutex_trylock(pthread_mutex_t* mutex);
```

将源代码中的`mythread`更改如下：

```
void* mythread(void* arg)
{
        /*尝试获取锁*/
        printf("mythread begin lock\n");
        while(pthread_mutex_trylock(&mutex))
        {
                printf("try lock again\n");
                sleep(1);
        }

        printf("mythread locked\n");
        sleep(5);
        printf("mythread get value = [%d]\n", value++);

        /*解锁*/
        pthread_mutex_unlock(&mutex);
        printf("mythread unlocked\n");
        pthread_exit((void*)0);
}
```

每过一秒钟尝试获取锁。

- 销毁锁，当互斥锁不再使用的时候，应当调用`pthread_mutex_destroy()`来销毁锁，其函数原型为：

```
int pthread_mutex_destroy(pthread_mutex * mutex);
```

保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。

临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。临界区的选定因尽可能小，如果选定太大会影响程序的并行处理性能。

4 读写锁

读写锁是用来解决读者写者问题的，读操作可以共享，写操作是独占的，读可以有多个在读，写只有唯一个在写，同时写的时候不允许读。

```
pthread_rwlock_t
int pthread_rwlock_init(pthread_rwlock_t * ,pthread_rwattr_t *);
int pthread_rwlock_destroy(pthread_rwlock_t* );
int pthread_rwlock_rdlock(pthread_rwlock_t*);
int pthread_rwlock_tryrdlock(pthread_rwlock_t*);
int pthread_rwlock_tryrdlock(pthread_rwlock_t*);
int pthread_rwlock_trywrlock(pthread_rwlock_t*);
int pthread_rwlock_unlock(pthread_rwlock_t*);
```

```c
#include <stdio.h>
#include <pthread.h>
int count;
pthread_rwlock_t rwlock;
/*3个线程不定时写同一全局资源，5个线程不定时读同一全局资源*/
void *th_write(void *arg)
{
    while (1)
    {
        pthread_rwlock_wrlock(&rwlock);
        printf("%x get rwlock\n", (int)pthread_self());
        sleep(5);
        printf("write %x : count=%d \n", (int)pthread_self(), ++count);
        pthread_rwlock_unlock(&rwlock);
        sleep(5);
    }
}
void *th_read(void *arg)
{
    while (1)
    {
        pthread_rwlock_rdlock(&rwlock);
        printf("read %x get rdlock\n", (int)pthread_self());
        sleep(1);
        printf("read %x : %d\n", (int)pthread_self(), count);
        pthread_rwlock_unlock(&rwlock);
        sleep(1);
    }
}
int main(void)
{
    int i;
    pthread_t tid[6];
    pthread_rwlock_init(&rwlock, NULL);



    for (i = 0; i < 5; i++)
        pthread_create(&tid[i], NULL, th_read, NULL);

    pthread_create(&tid[5], NULL, th_write, NULL);

    for (i = 0; i < 6; i++)
        pthread_join(tid[i], NULL);
    pthread_rwlock_destroy(&rwlock);
    return 0;
}
```

5 线程同步-条件变量

互斥锁只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，条件变量通常和互斥锁一起使用。

条件变量被用来阻塞一个线程，当条件不足时，线程通常先解开相应的互斥锁进入阻塞状态，等待条件发生变化。一旦其他的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个 或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。

5.1 条件变量初始化

- 条件变量的结构为`pthread_cond_t`,函数`pthread_cond_init()`被用来初始化一个条件变量。它的函数原型为:

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t* attr);
```

`cond`：是一个指向结构`pthread_cond_t`的指针，

`cond_attr`：是一个指向结构`pthread_condattr_t`的指针。结构`pthread_condattr_t`是条件变量的属性结构，和互斥锁一样我们可以用它来设置

条件变量是进程内可用还是进程间可用，默认值是`PTHREAD_PROCESS_PRIVATE`，即此条件变量被同一进程内的各个线程使用。

5.2 条件变量销毁

- 当条件变量不在使用的时候，应当释放，释放一个条件变量的函数：

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

5.3 等待条件变量

- 当我们需要用一个条件变量来阻塞线程的时候，调用函数pthread_cond_wait().其函数原型为：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

函数的第一个参数是条件变量，第二个参数是线程阻塞的时候要解开的互斥锁。

5.4 唤醒等待的线程

线程可以被函数`pthread_cond_signal`和函数`pthread_cond_broadcast`唤醒，其函数原型是:

```
int pthread_cond_broadcast(pthread_cond_t* cond);
int pthread_cond_signal(pthread_cond_t* cond);
```

他们用来释放被阻塞的条件变量cond上的线程，这两个函数区别在于当有多个线程都被同一个条件变量所阻塞时：

用`pthread_cond_broadcast()`函数可以使所有线程都被唤醒。

用`pthread_cond_signal()`时哪一个线程被唤醒是由线程的调度策略所决定。

源代码：cond_test.c

```c
#include <stdio.h>
#include <pthread.h>
static int condition = 0;
static value = 0;
static pthread_mutex_t mutex;
static pthread_cond_t cond;
void* mythread(void* arg)
{
  /*上锁*/
    pthread_mutex_lock(&mutex);
    printf("mythread(%d) locked\n", (int)arg);

    while(condition != 1)
    {
        /*条件不满足，等待条件*/
        printf("mythread(%d) wait cond\n", (int)arg);
        pthread_cond_wait(&cond, &mutex);
    }
    printf("mythread(%d) get value = [%d]\n", (int)arg, value++);

  /*解锁退出*/
    pthread_mutex_unlock(&mutex);
    printf("mythread(%d) unlocked\n", (int)arg);
    pthread_exit((void*)0);
}


int main(int argc, char** argv)
{
    int ret;
    pthread_t pth0;
    pthread_t pth1;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    ret =pthread_create(&pth, NULL, mythread, (void*)0);
    if(ret != 0)
    {
        printf("create thread:%s\n", strerror_r(ret));
        return -1;
    }
    ret =pthread_create(&pth, NULL, mythread, (void*)1);
    if(ret != 0)
    {
        printf("create thread:%s\n", strerror_r(ret));
        return -1;
    }


    sleep(5);
    /*主线程获取锁*/
    pthread_mutex_lock(&mutex);
    printf("main thread locked\n");

    condition = 1;
```

```
    /*主线程解锁*/
    pthread_mutex_unlock(&mutex);
    printf("main thread unlocked\n");

    /*主线程唤醒在条件上等待的线程*/
    pthread_cond_signal(&cond);

    pthread_join(pth0, NULL);
    pthread_join(pth1, NULL);
    return 0;
}
```

进程开了两个子线程，两个子线程通过上锁`mutex`来访问`value`变量，但是在这访问变量之前需要`condition`这个条件为1，不成立解锁`mutex`阻塞等待条件成立。

主线程通过上锁`mutex`来将`condition`设置为1，然后通过函数`pthread_cond_signal()`唤醒在这个cond条件上等待的两个线程的其中一个。

如果想要唤醒全部在cond条件上等待的线程，则可以使用`pthread_cond_broadcast()`。

`pthread_cond_test`输出如下图：

```
where@ubuntu:~$ ./pthread_cond_test
mythread1 locked
mythread1 wait cond
mythread0 locked
mythread0 wait cond
main thread locked
main thread unlocked
mythread1 get value = [0]
mythread1 unlocked
```

注意: 广播的时候，唤醒所有等待在当前条件变量的线程，谁抢到锁，谁就从等待变量返回，其他线程继续等待在当前的条件变量上。

6 线程同步-信号量

信号量从本质上是一个非负整数计数器，通常被用来控制对公共资源的访问。当可用的公共资源增加时，调用函数`sem_post()`增加信号量。只有当信号量大于0时，函数`sem_wait()`才能返回，并将信号量的值减1，当信号量等于0时，`sem_wait()`将被阻塞直到信号量的值大于0.函数`sem_trywait()`是函数`sem_wait()`的非阻塞版本。

- 信号量的数据类型为结构`sem_t`.函数`sem_init()`用来初始化一个信号量。其的函数原型为：

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned value);
```

```
1. sem：指向信号量结构的一个指针
2. pshared：不为0时此信号量在进程间共享，否则只能为当前进程的所有线程共享。
3. value：给出了信号量的初始值。
```

- 函数`sem_post()`用来增加信号量的值，其函数原型是：

```
int sem_post(sem_t* sem);
```

- 函数`sem_wait()`用来阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减去1，表明公共资源经使用后减少，其函数原型是：

```
int sem_wait(sem_t *sem);
```

- 函数sem_trywait()与函数sem_wait()的区别是当信号量的值等于0时，sem_trywait()函数不会阻塞当前线程。

```
int sem_trywait(sem_t *sem);
```

- 当信号量不再使用的时候，要用函数sem_destroy()来释放信号量。其函数原型是：

```
int sem_destroy(sem_t *sem);
```

- 在下面的例子中，我们创建三个线程a、b、c。但是要求线程按照c->b->a的顺序执行 。

源代码：sem_test.c

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
static sem_t sem1;
static sem_t sem2;

void *thread_a(void *arg)
{
    /*等待信号量sem1*/
    sem_wait(&sem1);
    printf("Thread_a running\n");
}

void *thread_b(void *arg)
{
    /*等待信号量sem2*/
    sem_wait(&sem2);
    printf("thread_b running\n");

    /*增加信号量sem1，使得thread_a可以执行*/
    sem_post(&sem1);
}

void *thread_c(void *arg)
{
    printf("thread_c running\n");

    /*增加信号量sem2，使得thread_b可以执行*/
    sem_post(&sem2);
}

int main(int argc, char** argv)
{
    pthread_t a,b,c;
    /*增加信号量*/
    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, 0);

    /*创建线程a, b, c*/
    pthread_create(&a, NULL, thread_a, NULL);
    pthread_create(&b, NULL, thread_b, NULL);
    pthread_create(&c, NULL, thread_c, NULL);
    /*等待线程退出*/
    pthread_join(a, NULL);
    pthread_join(b, NULL);
    pthread_join(c, NULL);


    /*删除信号量*/
    sem_destroy(&sem1);
    sem_destroy(&sem2);

    return 0;
```

```
}
```

- 生产者与消费者模型

```c
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <time.h>
sem_t sem_empty;
sem_t sem_full;

static int tmp[10];


void * thread_consumer(void *arg)
{
    static int count = 0 ;
    while(1)
    {
        puts("consumer wait");
        sem_wait(&sem_full);
        printf("consumer %d\n", tmp[count]);
        count = ++count % (sizeof(tmp) / sizeof(tmp[0]));
        sem_post(&sem_empty);
        sleep(5);
    }
}
void * thread_productor(void *arg)
{
    srand(time(NULL));
    static int count = 0 ;
    while(1)
    {
        puts("productor wait");
        sem_wait(&sem_empty);
        tmp[count] = rand()  % 100;
        printf("productor %d\n", tmp[count]);
        count = ++count % (sizeof(tmp) / sizeof(tmp[0]));
        sem_post(&sem_full);
        sleep(1);
    }
}


int main(int argc, char** argv)
{
    sem_init(&sem_empty, 0, sizeof(tmp) / sizeof(tmp[0]));
    sem_init(&sem_full, 0, 0);
    pthread_t thA, thB;
    pthread_create(&thA, NULL, thread_consumer, NULL);
    pthread_create(&thB, NULL, thread_productor, NULL);

    pthread_join(thA, NULL);

    pthread_join(thB, NULL);
```

```
    return 0;
}
```

6 文件锁

```
#include<sys/file.h>
int flock(int fd,int operation);
```

参数 operation有下列四种情况:

LOCK_SH 建立共享锁定。多个进程可同时对同一个文件作共享锁定。

LOCK_EX 建立互斥锁定。一个文件同时只有一个互斥锁定。

LOCK_UN 解除文件锁定状态。

LOCK_NB 无法建立锁定时，此操作可不被阻断，马上返回进程。通常与LOCK_SH或LOCK_EX 做OR(|)组合。

例子：

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char** argv)
{
    char buf[] = "helloworld";

    if(argc < 2)
    {
        printf("usage: %s [file]\n", argv[0]);
        exit(1);
    }
    int fd = open(argv[1], O_WRONLY);
    if(fd == -1)
    {
        perror("open");
        exit(1);
    }
    printf("lock file %s\n", argv[1]);
    flock(fd, LOCK_EX);
    printf("get lock\n");
    write(fd, buf, strlen(buf));
    sleep(10);

    flock(fd, LOCK_UN);
    printf("unlock file %s\n", argv[1]);

    close(fd);
    return 0;
}
```